# Matplotlib

***Release 3.9.0***

**John Hunter**
**Darren Dale**
**Eric Firing**
**Michael Droettboom**
**and the matplotlib development team**

**May 16, 2024**

# CONTENTS


I Install 3

1 Installation 7

II Learn 23

2 Quick start guide 27

3 Using Matplotlib 47

4 Tutorials 493

5 Plot types 551

6 Examples 581

7 API Reference 1809

III Community 4053

8 External resources 4055

IV What's new 4057

9 Release notes 4061

V Contribute 4635

10 Contribute 4639

VI About us 4799

11 Project information 4803

VII Appendices 4835

Bibliography 4837

Python Module Index 4839

Index 4843

Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations.

# Part I

# Install

### pip

```
pip install matplotlib
```

### conda

```
conda install -c conda-forge matplotlib
```

### other

CHAPTER

ONE

# INSTALLATION

## 1.1 Install an official release

Matplotlib releases are available as wheel packages for macOS, Windows and Linux on PyPI. Install it using `pip`:

```
python -m pip install -U pip
python -m pip install -U matplotlib
```

If this command results in Matplotlib being compiled from source and there's trouble with the compilation, you can add `--prefer-binary` to select the newest version of Matplotlib for which there is a precompiled wheel for your OS and Python.

---

**Note:** The following backends work out of the box: Agg, ps, pdf, svg

Python is typically shipped with tk bindings which are used by TkAgg.

For support of other GUI frameworks, LaTeX rendering, saving animations and a larger selection of file formats, you can install *optional dependencies*.

---

## 1.2 Third-party distributions

Various third-parties provide Matplotlib for their environments.

### 1.2.1 Conda packages

Matplotlib is available both via the *anaconda main channel*

```
conda install matplotlib
```

as well as via the *conda-forge community channel*

```
conda install -c conda-forge matplotlib
```

### 1.2.2 Python distributions

Matplotlib is part of major Python distributions:

- Anaconda
- ActiveState ActivePython
- WinPython

### 1.2.3 Linux package manager

If you are using the Python version that comes with your Linux distribution, you can install Matplotlib via your package manager, e.g.:

- Debian / Ubuntu: `sudo apt-get install python3-matplotlib`
- Fedora: `sudo dnf install python3-matplotlib`
- Red Hat: `sudo yum install python3-matplotlib`
- Arch: `sudo pacman -S python-matplotlib`

## 1.3 Install a nightly build

Matplotlib makes nightly development build wheels available on the scientific-python-nightly-wheels Anaconda Cloud organization. These wheels can be installed with `pip` by specifying scientific-python-nightly-wheels as the package index to query:

```
python -m pip install \
  --upgrade \
  --pre \
  --index-url https://pypi.anaconda.org/scientific-python-nightly-wheels/
↪simple \
  --extra-index-url https://pypi.org/simple \
  matplotlib
```

## 1.4 Install from source

**Installing for Development**

If you would like to contribute to Matplotlib or otherwise need to install the latest development code, please follow the instructions in *Setting up Matplotlib for development*.

The following instructions are for installing from source for production use. This is generally *not* recommended; please use prebuilt packages when possible. Proceed with caution because these instructions may result in your build producing unexpected behavior and/or causing local testing to fail.

Before trying to install Matplotlib, please install the *Dependencies*.

To build from a tarball, download the latest *tar.gz* release file from the PyPI files page.

We provide a mplsetup.cfg file which you can use to customize the build process. For example, which default backend to use, whether some of the optional libraries that Matplotlib ships with are installed, and so on. This file will be particularly useful to those packaging Matplotlib.

If you are building your own Matplotlib wheels (or sdists) on Windows, note that any DLLs that you copy into the source tree will be packaged too.

## 1.5 Configure build and behavior defaults

Aspects of the build and install process and some behaviorial defaults of the library can be configured via:

### 1.5.1 Environment variables

**HOME**

The user's home directory. On Linux, `~` is shorthand for *HOME*.

**MPLBACKEND**

This optional variable can be set to choose the Matplotlib backend. See *What is a backend?*.

**MPLCONFIGDIR**

This is the directory used to store user customizations to Matplotlib, as well as some caches to improve performance. If *MPLCONFIGDIR* is not defined, *HOME*`/.config/matplotlib` and *HOME*`/.cache/matplotlib` are used on Linux, and *HOME*`/.matplotlib` on other platforms, if they are writable. Otherwise, the Python standard library's `tempfile.gettempdir` is used to find a base directory in which the `matplotlib` subdirectory is created.

**MPLSETUPCFG**

This optional variable can be set to the full path of a `mplsetup.cfg` configuration file used to customize the Matplotlib build. By default, a `mplsetup.cfg` file in the root of the Matplotlib source tree will be read. Supported build options are listed in `mplsetup.cfg.template`.

**PATH**

The list of directories searched to find executable programs.

**PYTHONPATH**

The list of directories that are added to Python's standard search list when importing packages and modules.

**QT_API**

The Python Qt wrapper to prefer when using Qt-based backends. See *the entry in the usage guide* for more information.

### Setting environment variables in Linux and macOS

To list the current value of *PYTHONPATH*, which may be empty, try:

```
echo $PYTHONPATH
```

The procedure for setting environment variables in depends on what your default shell is. Common shells include **bash** and **csh**. You should be able to determine which by running at the command prompt:

```
echo $SHELL
```

To create a new environment variable:

```
export PYTHONPATH=~/Python  # bash/ksh
setenv PYTHONPATH ~/Python  # csh/tcsh
```

To prepend to an existing environment variable:

```
export PATH=~/bin:${PATH}  # bash/ksh
setenv PATH ~/bin:${PATH}  # csh/tcsh
```

The search order may be important to you, do you want `~/bin` to be searched first or last? To append to an existing environment variable:

```
export PATH=${PATH}:~/bin  # bash/ksh
setenv PATH ${PATH}:~/bin  # csh/tcsh
```

To make your changes available in the future, add the commands to your `~/.bashrc` or `~/.cshrc` file.

### Setting environment variables in Windows

Open the **Control Panel** (*Start* □ *Control Panel*), start the **System** program. Click the *Advanced* tab and select the *Environment Variables* button. You can edit or add to the *User Variables*.

Default plotting appearance and behavior can be configured via the *rcParams file*

## 1.6 Dependencies

Mandatory dependencies should be installed automatically if you install Matplotlib using a package manager such as `pip` or `conda`; therefore this list is primarily for reference and troubleshooting.

### 1.6.1 Dependencies

#### Runtime dependencies

#### Required

When installing through a package manager like `pip` or `conda`, the mandatory dependencies are automatically installed. This list is mainly for reference.

- Python (>= 3.9)
- contourpy (>= 1.0.1)
- cycler (>= 0.10.0)
- dateutil (>= 2.7)
- fontTools (>= 4.22.0)
- kiwisolver (>= 1.3.1)
- NumPy (>= 1.23)
- packaging (>= 20.0)
- Pillow (>= 8.0)
- pyparsing (>= 2.3.1)
- importlib-resources (>= 3.2.0; only required on Python < 3.10)

#### Optional

The following packages and tools are not required but extend the capabilities of Matplotlib.

#### Backends

Matplotlib figures can be rendered to various user interfaces. See *What is a backend?* for more details on the optional Matplotlib backends and the capabilities they provide.

- Tk (>= 8.5, != 8.6.0 or 8.6.1): for the Tk-based backends. Tk is part of most standard Python installations, but it's not part of Python itself and thus may not be present in rare cases.
- PyQt6 (>= 6.1), PySide6, PyQt5 (>= 5.12), or PySide2: for the Qt-based backends.
- PyGObject and pycairo (>= 1.14.0): for the GTK-based backends. If using pip (but not conda or system package manager) PyGObject must be built from source; see pygobject documentation.
- pycairo (>= 1.14.0) or cairocffi (>= 0.8): for cairo-based backends.
- wxPython (>= 4): for the wx-based backends. If using pip (but not conda or system package manager) on Linux wxPython wheels must be manually downloaded from https://wxpython.org/pages/downloads/.

- Tornado (>= 5): for the WebAgg backend.
- ipykernel: for the nbagg backend.
- macOS (>= 10.12): for the macosx backend.

### Animations

- ffmpeg: for saving movies.
- ImageMagick: for saving animated gifs.

### Font handling and rendering

- LaTeX (with cm-super and underscore) and GhostScript (>= 9.0): for rendering text with LaTeX.
- fontconfig (>= 2.7): for detection of system fonts on Linux.

### C libraries

Matplotlib brings its own copies of the following libraries:

- `Agg`: the Anti-Grain Geometry C++ rendering engine
- `ttconv`: a TrueType font utility

Additionally, Matplotlib depends on:

- FreeType (>= 2.3): a font rendering library
- QHull (>= 8.0.2): a library for computing triangulations (note that this version is also known as 2020.2)

### Download during install

By default, Matplotlib downloads and builds its own copies of Qhull and FreeType. The vendored version of FreeType is necessary to run the test suite, because different versions of FreeType rasterize characters differently.

### Use system libraries

To force Matplotlib to use a copy of FreeType or Qhull already installed in your system, you must pass configuration settings to Meson via meson-python:

```
python -m pip install \
  --config-settings=setup-args="-Dsystem-freetype=true" \
  --config-settings=setup-args="-Dsystem-qhull=true" \
  .
```

In this case, you need to install the FreeType and Qhull library and headers. This can be achieved using a package manager, e.g. for FreeType:

```
# Pick ONE of the following:
sudo apt install libfreetype6-dev  # Debian/Ubuntu
sudo dnf install freetype-devel    # Fedora
brew install freetype              # macOS with Homebrew
conda install freetype             # conda, any OS
```

(adapt accordingly for Qhull).

On Linux and macOS, it is also recommended to install pkg-config, a helper tool for locating FreeType:

```
# Pick ONE of the following:
sudo apt install pkg-config  # Debian/Ubuntu
sudo dnf install pkgconf     # Fedora
brew install pkg-config      # macOS with Homebrew
conda install pkg-config     # conda
# Or point the PKG_CONFIG environment variable to the path to pkg-config:
export PKG_CONFIG=...
```

If not using pkg-config (in particular on Windows), you may need to set the include path (to the library headers) and link path (to the libraries) explicitly, if they are not in standard locations. This can be done using standard environment variables -- on Linux and macOS:

```
export CFLAGS='-I/directory/containing/ft2build.h'
export LDFLAGS='-L/directory/containing/libfreetype.so'
```

and on Windows:

```
set CL=/IC:\directory\containing\ft2build.h
set LINK=/LIBPATH:C:\directory\containing\freetype.lib
```

If you go this route but need to reset and rebuild to change your settings, remember to clear your artifacts before re-building:

```
git clean -xfd
```

### From source files

If the automatic download does not work (for example, on air-gapped systems) it is preferable to instead use system libraries. However you can manually download the tarballs into `subprojects/packagecache` at the top level of the checkout repository. The expected SHA256 hashes of the downloaded tarballs are in `subprojects/*.wrap` if you wish to verify them, but they will also be checked by the build system before unpacking.

### Minimum pip / manylinux support (linux)

Matplotlib publishes manylinux wheels which have a minimum version of pip which will recognize the wheels

- Python 3.9+: `manylinux2014` / pip >= 19.3

In all cases the required version of pip is embedded in the CPython source.

### Build dependencies

### Python

By default, `pip` will build packages using build isolation, meaning that these build dependencies are temporally installed by pip for the duration of the Matplotlib build process. However, build isolation is disabled when *installing Matplotlib for development*; therefore we recommend using one of our *virtual environment configurations* to create a development environment in which these packages are automatically installed.

If you are developing Matplotlib and unable to use our environment configurations, then you must manually install the following packages into your development environment:

- meson-python (>= 0.13.1).
- ninja (>= 1.8.2). This may be available in your package manager or bundled with Meson, but may be installed via `pip` if otherwise not available.
- PyBind11 (>= 2.6). Used to connect C/C++ code with Python.
- setuptools_scm (>= 7). Used to update the reported `mpl.__version__` based on the current git commit. Also a runtime dependency for editable installs.
- NumPy (>= 1.22). Also a runtime dependency.

### Compiled extensions

Matplotlib requires a C++ compiler that supports C++17, and each platform has a development environment that must be installed before a compiler can be installed. You may also need to install headers for various libraries used in the compiled extension source files.

### Linux

On some Linux systems, you can install a meta-build package. For example, on Ubuntu `apt install build-essential`

Otherwise, use the system distribution's package manager to install *gcc*.

### macOS

Install Xcode for Apple platform development.

### Windows

Install Visual Studio Build Tools

Make sure "Desktop development with C++" is selected, and that the latest MSVC, "C++ CMake tools for Windows," and a Windows SDK compatible with your version of Windows are selected and installed. They should be selected by default under the "Optional" subheading, but are required to build Matplotlib from source.

Alternatively, you can install a Linux-like environment such as CygWin or Windows Subsystem for Linux. If using MinGW-64, we require **v6** of the `Mingw-w64-x86_64-headers`.

We highly recommend that you install a compiler using your platform tool, i.e., Xcode, VS Code or Linux package manager. Choose **one** compiler from this list:

| compiler | minimum version | platforms | notes |
|---|---|---|---|
| GCC | **7.2** | Linux, macOS, Windows | gcc 7.2, GCC: Binaries, |
| Clang (LLVM) | **5** | Linux, macOS | clang 5, LLVM |
| MSVC++ | **16.0** | Windows | Visual Studio 2019 C++ |

### Test dependencies

This section lists the additional software required for *running the tests*.

### Required

- pytest (>= 7.0.0)

### Optional

In addition to all of the optional dependencies on the main library, for testing the following will be used if they are installed.

- Ghostscript (>= 9.0, to render PDF files)
- Inkscape (to render SVG files)
- nbformat and nbconvert used to test the notebook backend
- pandas used to test compatibility with Pandas
- pikepdf used in some tests for the pgf and pdf backends

- psutil used in testing the interactive backends
- pytest-cov (>= 2.3.1) to collect coverage information
- pytest-flake8 to test coding standards using flake8
- pytest-timeout to limit runtime in case of stuck tests
- pytest-xdist to run tests in parallel
- pytest-xvfb to run tests without windows popping up (Linux)
- pytz used to test pytz int
- sphinx used to test our sphinx extensions
- WenQuanYi Zen Hei and Noto Sans CJK fonts for testing font fallback and non-Western fonts
- xarray used to test compatibility with xarray

If any of these dependencies are not discovered, then the tests that rely on them will be skipped by pytest.

---

**Note:** When installing Inkscape on Windows, make sure that you select "Add Inkscape to system PATH", either for all users or current user, or the tests will not find it.

---

### Documentation dependencies

#### Python

The additional Python packages required to build the *documentation* are listed in `doc-requirements.txt` and can be installed using

```
pip install -r requirements/doc/doc-requirements.txt
```

The content of `doc-requirements.txt` is also shown below:

```
# Requirements for building docs
#
# You will first need a matching Matplotlib installation
# e.g (from the Matplotlib root directory)
#     pip install --no-build-isolation --editable .[dev]
#
# Install the documentation requirements with:
#     pip install -r requirements/doc/doc-requirements.txt
#
sphinx>=3.0.0,!=6.1.2
colorspacious
ipython
ipywidgets
ipykernel
numpydoc>=1.0
packaging>=20
```

(continues on next page)

(continued from previous page)

```
pydata-sphinx-theme~=0.15.0
mpl-sphinx-theme~=3.9.0
pyyaml
sphinxcontrib-svg2pdfconverter>=1.1.0
sphinx-copybutton
sphinx-design
sphinx-gallery>=0.12.0
sphinx-tags>=0.3.0
```

### External tools

The documentation requires LaTeX and Graphviz. These are not Python packages and must be installed separately.

### Required

- Graphviz
- a minimal working LaTeX distribution, e.g. TeX Live or MikTeX

The following LaTeX packages:

- dvipng
- underscore
- cm-super
- `collection-fontsrecommended`

The complete version of many LaTex distribution installers, e.g. "texlive-full" or "texlive-all", will often automatically include these packages.

### Optional

The documentation can be built without Inkscape and optipng, but the build process will raise various warnings.

- Inkscape
- optipng
- the font xkcd script or Comic Neue
- the font "Times New Roman"

## 1.7 Frequently asked questions

### 1.7.1 Report a compilation problem

See *Get help*.

### 1.7.2 Matplotlib compiled fine, but nothing shows up when I use it

The first thing to try is a *clean install* and see if that helps. If not, the best way to test your install is by running a script, rather than working interactively from a python shell or an integrated development environment such as **IDLE** which add additional complexities. Open up a UNIX shell or a DOS command prompt and run, for example:

```
python -c "from pylab import *; set_loglevel('debug'); plot(); show()"
```

This will give you additional information about which backends Matplotlib is loading, version information, and more. At this point you might want to make sure you understand Matplotlib's *configuration* process, governed by the `matplotlibrc` configuration file which contains instructions within and the concept of the Matplotlib backend.

If you are still having trouble, see *Get help*.

### 1.7.3 How to completely remove Matplotlib

Occasionally, problems with Matplotlib can be solved with a clean installation of the package. In order to fully remove an installed Matplotlib:

1. Delete the caches from your *Matplotlib configuration directory*.
2. Delete any Matplotlib directories or eggs from your *installation directory*.

### 1.7.4 macOS Notes

#### Which python for macOS?

Apple ships macOS with its own Python, in `/usr/bin/python`, and its own copy of Matplotlib. Unfortunately, the way Apple currently installs its own copies of NumPy, Scipy and Matplotlib means that these packages are difficult to upgrade (see system python packages). For that reason we strongly suggest that you install a fresh version of Python and use that as the basis for installing libraries such as NumPy and Matplotlib. One convenient way to install Matplotlib with other useful Python software is to use the Anaconda Python scientific software collection, which includes Python itself and a wide range of libraries; if you need a library that is not available from the collection, you can install it yourself using standard methods such as *pip*. See the Anaconda web page for installation support.

Other options for a fresh Python install are the standard installer from python.org, or installing Python using a general macOS package management system such as homebrew or macports. Power users on macOS will likely want one of homebrew or macports on their system to install open source software packages, but it

is perfectly possible to use these systems with another source for your Python binary, such as Anaconda or Python.org Python.

### Installing macOS binary wheels

If you are using Python from https://www.python.org, Homebrew, or Macports, then you can use the standard pip installer to install Matplotlib binaries in the form of wheels.

pip is installed by default with python.org and Homebrew Python, but needs to be manually installed on Macports with

```
sudo port install py38-pip
```

Once pip is installed, you can install Matplotlib and all its dependencies with from the Terminal.app command line:

```
python3 -m pip install matplotlib
```

You might also want to install IPython or the Jupyter notebook (`python3 -m pip install ipython notebook`).

### Checking your installation

The new version of Matplotlib should now be on your Python "path". Check this at the Terminal.app command line:

```
python3 -c 'import matplotlib; print(matplotlib.__version__, matplotlib.__
 ↪file__)'
```

You should see something like

```
3.6.0 /Library/Frameworks/Python.framework/Versions/3.9/lib/python3.9/site-
 ↪packages/matplotlib/__init__.py
```

where `3.6.0` is the Matplotlib version you just installed, and the path following depends on whether you are using Python.org Python, Homebrew or Macports. If you see another version, or you get an error like

```
Traceback (most recent call last):
  File "<string>", line 1, in <module>
ImportError: No module named matplotlib
```

then check that the Python binary is the one you expected by running

```
which python3
```

If you get a result like `/usr/bin/python...`, then you are getting the Python installed with macOS, which is probably not what you want. Try closing and restarting Terminal.app before running the check again. If that doesn't fix the problem, depending on which Python you wanted to use, consider reinstalling Python.org Python, or check your homebrew or macports setup. Remember that the disk image installer only works for Python.org Python, and will not get picked up by other Pythons. If all these fail, please *let us know*.

## 1.8 Troubleshooting

### 1.8.1 Obtaining Matplotlib version

To find out your Matplotlib version number, import it and print the `__version__` attribute:

```
>>> import matplotlib
>>> matplotlib.__version__
'0.98.0'
```

### 1.8.2 `matplotlib` install location

You can find what directory Matplotlib is installed in by importing it and printing the `__file__` attribute:

```
>>> import matplotlib
>>> matplotlib.__file__
'/home/jdhunter/dev/lib64/python2.5/site-packages/matplotlib/__init__.pyc'
```

### 1.8.3 `matplotlib` configuration and cache directory locations

Each user has a Matplotlib configuration directory which may contain a *matplotlibrc* file. To locate your `matplotlib/` configuration directory, use *matplotlib.get_configdir()*:

```
>>> import matplotlib as mpl
>>> mpl.get_configdir()
'/home/darren/.config/matplotlib'
```

On Unix-like systems, this directory is generally located in your *HOME* directory under the `.config/` directory.

In addition, users have a cache directory. On Unix-like systems, this is separate from the configuration directory by default. To locate your `.cache/` directory, use *matplotlib.get_cachedir()*:

```
>>> import matplotlib as mpl
>>> mpl.get_cachedir()
'/home/darren/.cache/matplotlib'
```

On Windows, both the config directory and the cache directory are the same and are in your `Documents and Settings` or `Users` directory by default:

```
>>> import matplotlib as mpl
>>> mpl.get_configdir()
'C:\\Documents and Settings\\jdhunter\\.matplotlib'
>>> mpl.get_cachedir()
'C:\\Documents and Settings\\jdhunter\\.matplotlib'
```

If you would like to use a different configuration directory, you can do so by specifying the location in your `MPLCONFIGDIR` environment variable -- see *Setting environment variables in Linux and macOS*. Note that `MPLCONFIGDIR` sets the location of both the configuration directory and the cache directory.

For more detailed instructions, see the *installation guide*.

# Part II

# Learn

**How to use Matplotlib?**

CHAPTER

TWO

# QUICK START GUIDE

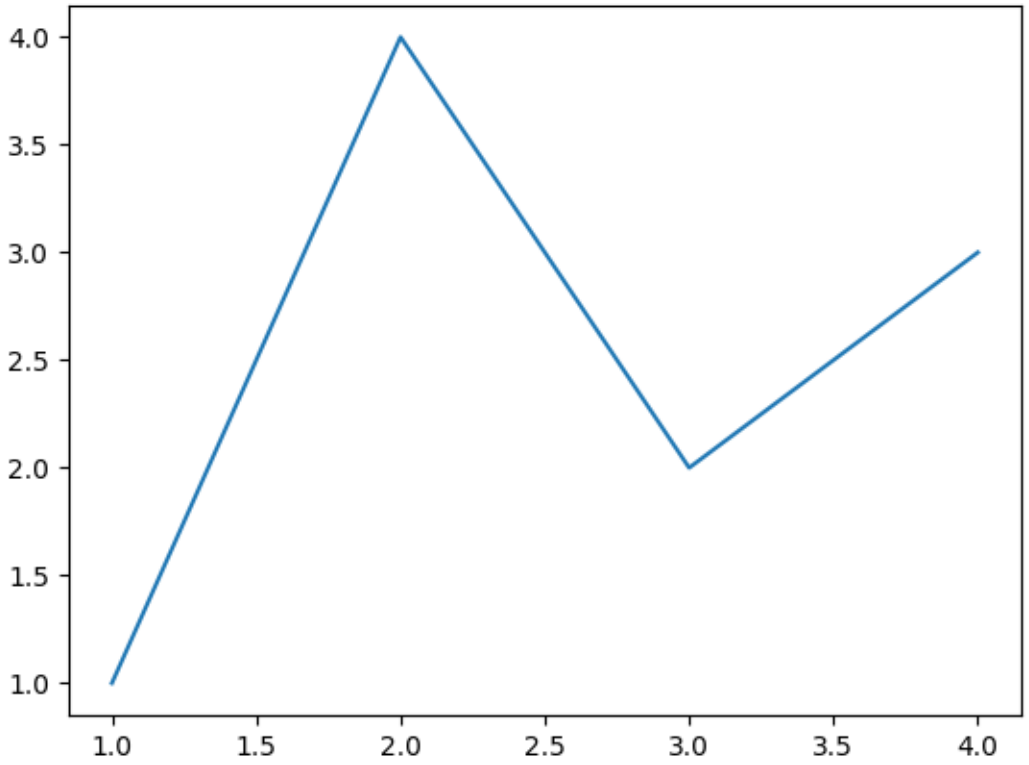
This tutorial covers some basic usage patterns and best practices to help you get started with Matplotlib.

```
import matplotlib.pyplot as plt
import numpy as np
```

## 2.1 A simple example

Matplotlib graphs your data on *Figure*s (e.g., windows, Jupyter widgets, etc.), each of which can contain one or more *Axes*, an area where points can be specified in terms of x-y coordinates (or theta-r in a polar plot, x-y-z in a 3D plot, etc.). The simplest way of creating a Figure with an Axes is using *pyplot.subplots*. We can then use *Axes.plot* to draw some data on the Axes, and *show* to display the figure:

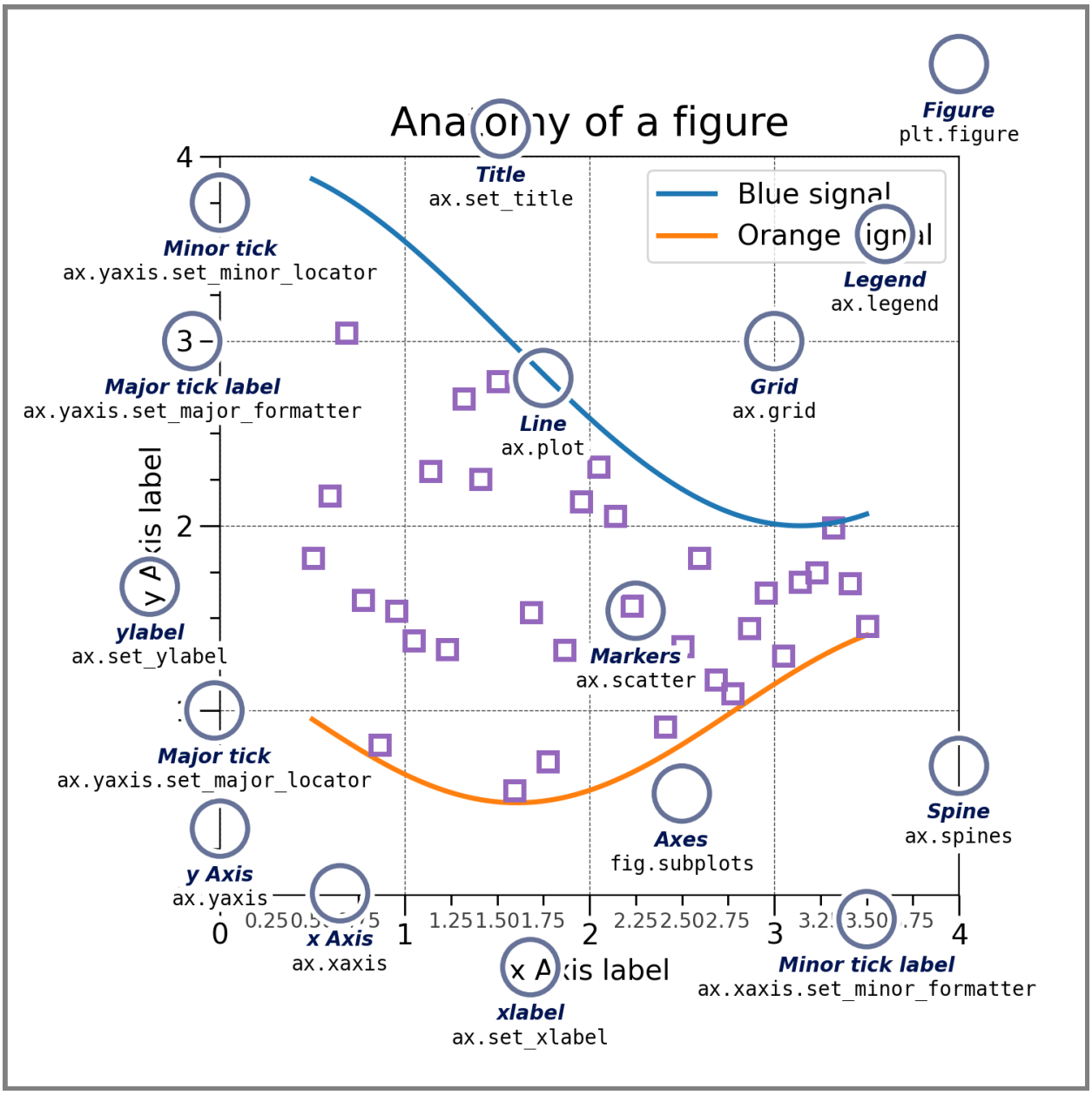
```
fig, ax = plt.subplots()             # Create a figure containing a single␣
 ↪Axes.
ax.plot([1, 2, 3, 4], [1, 4, 2, 3])  # Plot some data on the Axes.
plt.show()                           # Show the figure.
```




Depending on the environment you are working in, `plt.show()` can be left out. This is for example the case with Jupyter notebooks, which automatically show all figures created in a code cell.

## 2.2 Parts of a Figure

Here are the components of a Matplotlib Figure.




### 2.2.1 Figure

The **whole** figure. The Figure keeps track of all the child *Axes*, a group of 'special' Artists (titles, figure legends, colorbars, etc.), and even nested subfigures.

Typically, you'll create a new Figure through one of the following functions:

```
fig = plt.figure()              # an empty figure with no Axes
fig, ax = plt.subplots()        # a figure with a single Axes
fig, axs = plt.subplots(2, 2)   # a figure with a 2x2 grid of Axes
# a figure with one Axes on the left, and two on the right:
```

(continues on next page)

(continued from previous page)

```
fig, axs = plt.subplot_mosaic([['left', 'right_top'],
                               ['left', 'right_bottom']])
```

*subplots()* and *subplot_mosaic* are convenience functions that additionally create Axes objects inside the Figure, but you can also manually add Axes later on.

For more on Figures, including panning and zooming, see *Introduction to Figures*.

### 2.2.2 Axes

An Axes is an Artist attached to a Figure that contains a region for plotting data, and usually includes two (or three in the case of 3D) *Axis* objects (be aware of the difference between **Axes** and **Axis**) that provide ticks and tick labels to provide scales for the data in the Axes. Each *Axes* also has a title (set via *set_title()*), an x-label (set via *set_xlabel()*), and a y-label set via *set_ylabel()*).

The *Axes* methods are the primary interface for configuring most parts of your plot (adding data, controlling axis scales and limits, adding labels etc.).

### 2.2.3 Axis

These objects set the scale and limits and generate ticks (the marks on the Axis) and ticklabels (strings labeling the ticks). The location of the ticks is determined by a *Locator* object and the ticklabel strings are formatted by a *Formatter*. The combination of the correct *Locator* and *Formatter* gives very fine control over the tick locations and labels.

### 2.2.4 Artist

Basically, everything visible on the Figure is an Artist (even *Figure*, *Axes*, and *Axis* objects). This includes *Text* objects, *Line2D* objects, *collections* objects, *Patch* objects, etc. When the Figure is rendered, all of the Artists are drawn to the **canvas**. Most Artists are tied to an Axes; such an Artist cannot be shared by multiple Axes, or moved from one to another.

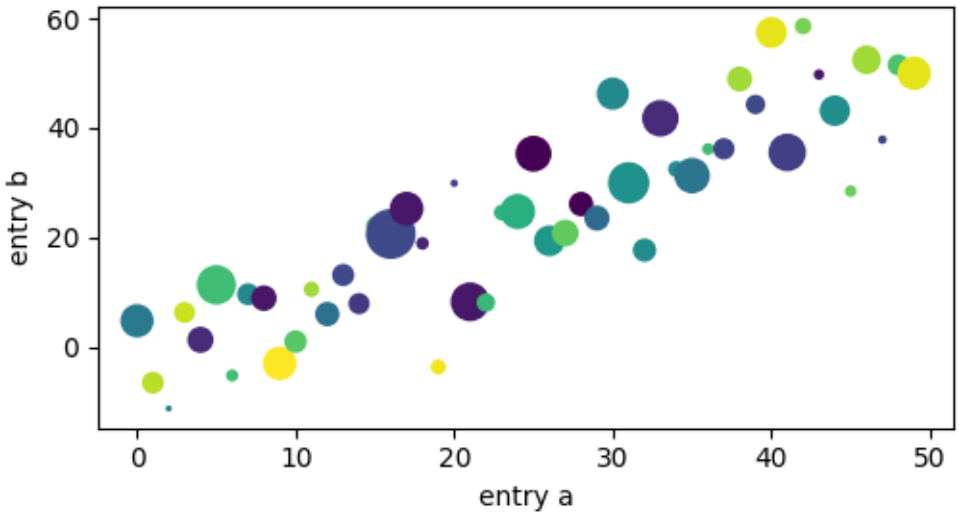
## 2.3 Types of inputs to plotting functions

Plotting functions expect `numpy.array` or `numpy.ma.masked_array` as input, or objects that can be passed to `numpy.asarray`. Classes that are similar to arrays ('array-like') such as `pandas` data objects and `numpy.matrix` may not work as intended. Common convention is to convert these to `numpy.array` objects prior to plotting. For example, to convert a `numpy.matrix`

```
b = np.matrix([[1, 2], [3, 4]])
b_asarray = np.asarray(b)
```

Most methods will also parse a string-indexable object like a *dict*, a structured numpy array, or a `pandas.DataFrame`. Matplotlib allows you to provide the `data` keyword argument and generate plots passing the strings corresponding to the *x* and *y* variables.

```
np.random.seed(19680801)  # seed the random number generator.
data = {'a': np.arange(50),
        'c': np.random.randint(0, 50, 50),
        'd': np.random.randn(50)}
data['b'] = data['a'] + 10 * np.random.randn(50)
data['d'] = np.abs(data['d']) * 100

fig, ax = plt.subplots(figsize=(5, 2.7), layout='constrained')
ax.scatter('a', 'b', c='c', s='d', data=data)
ax.set_xlabel('entry a')
ax.set_ylabel('entry b')
```




## 2.4 Coding styles

### 2.4.1 The explicit and the implicit interfaces

As noted above, there are essentially two ways to use Matplotlib:

- Explicitly create Figures and Axes, and call methods on them (the "object-oriented (OO) style").
- Rely on pyplot to implicitly create and manage the Figures and Axes, and use pyplot functions for plotting.

See *Matplotlib Application Interfaces (APIs)* for an explanation of the tradeoffs between the implicit and explicit interfaces.

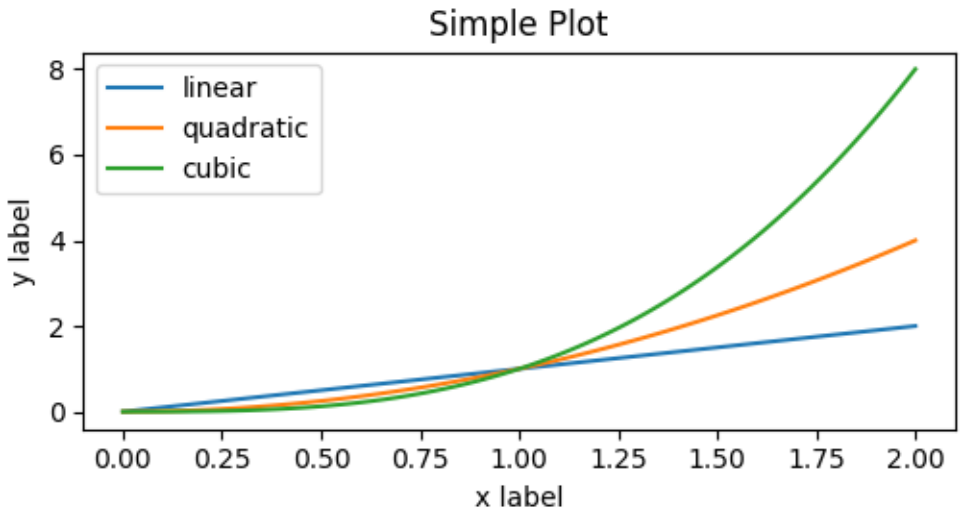
So one can use the OO-style

```
x = np.linspace(0, 2, 100)  # Sample data.

# Note that even in the OO-style, we use `.pyplot.figure` to create the␣
↪Figure.
fig, ax = plt.subplots(figsize=(5, 2.7), layout='constrained')
```

(continues on next page)

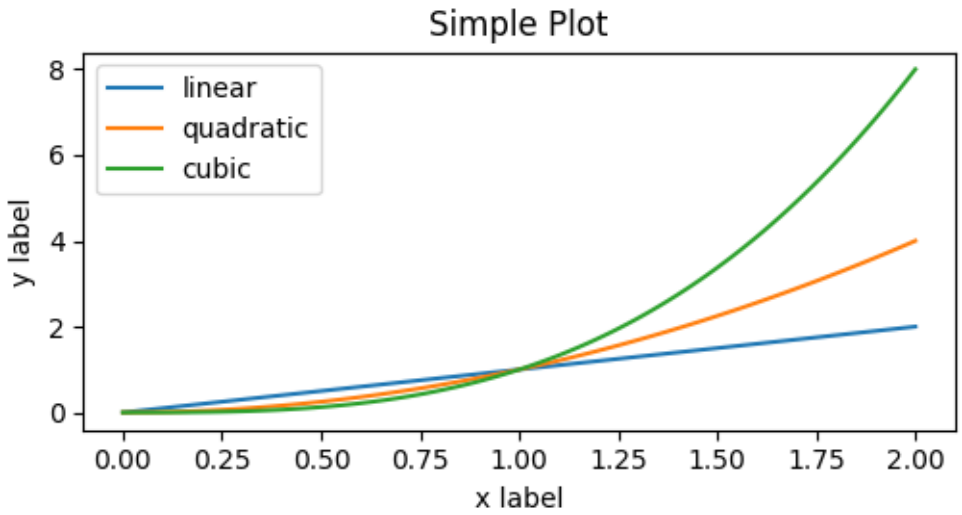
(continued from previous page)

```
ax.plot(x, x, label='linear')  # Plot some data on the Axes.
ax.plot(x, x**2, label='quadratic')  # Plot more data on the Axes...
ax.plot(x, x**3, label='cubic')  # ... and some more.
ax.set_xlabel('x label')  # Add an x-label to the Axes.
ax.set_ylabel('y label')  # Add a y-label to the Axes.
ax.set_title("Simple Plot")  # Add a title to the Axes.
ax.legend()  # Add a legend.
```




or the pyplot-style:

```
x = np.linspace(0, 2, 100)  # Sample data.

plt.figure(figsize=(5, 2.7), layout='constrained')
plt.plot(x, x, label='linear')  # Plot some data on the (implicit) Axes.
plt.plot(x, x**2, label='quadratic')  # etc.
plt.plot(x, x**3, label='cubic')
plt.xlabel('x label')
plt.ylabel('y label')
plt.title("Simple Plot")
plt.legend()
```




(In addition, there is a third approach, for the case when embedding Matplotlib in a GUI application, which completely drops pyplot, even for figure creation. See the corresponding section in the gallery for more info: *Embedding Matplotlib in graphical user interfaces*.)

Matplotlib's documentation and examples use both the OO and the pyplot styles. In general, we suggest using the OO style, particularly for complicated plots, and functions and scripts that are intended to be reused as part of a larger project. However, the pyplot style can be very convenient for quick interactive work.

---

**Note:** You may find older examples that use the `pylab` interface, via `from pylab import *`. This approach is strongly deprecated.

---

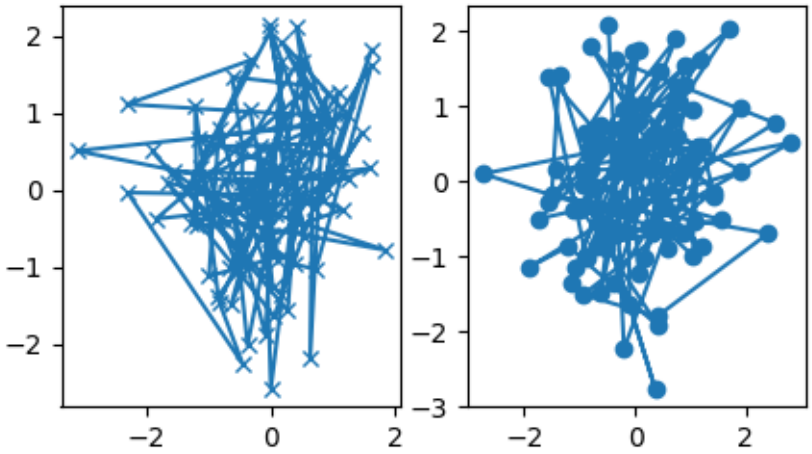
### 2.4.2 Making a helper functions

If you need to make the same plots over and over again with different data sets, or want to easily wrap Matplotlib methods, use the recommended signature function below.

```
def my_plotter(ax, data1, data2, param_dict):
    """
    A helper function to make a graph.
    """
    out = ax.plot(data1, data2, **param_dict)
    return out
```

which you would then use twice to populate two subplots:

```
data1, data2, data3, data4 = np.random.randn(4, 100)  # make 4 random data␣
↪sets
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(5, 2.7))
my_plotter(ax1, data1, data2, {'marker': 'x'})
my_plotter(ax2, data3, data4, {'marker': 'o'})
```

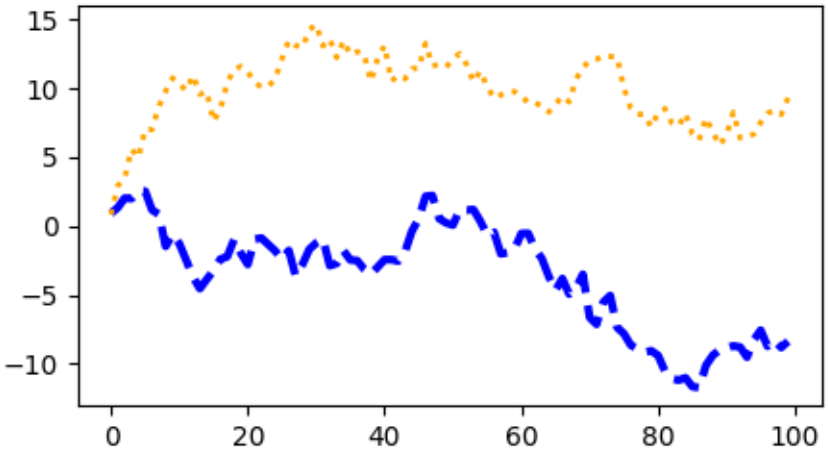



Note that if you want to install these as a python package, or any other customizations you could use one of the many templates on the web; Matplotlib has one at mpl-cookiecutter

## 2.5 Styling Artists

Most plotting methods have styling options for the Artists, accessible either when a plotting method is called, or from a "setter" on the Artist. In the plot below we manually set the *color*, *linewidth*, and *linestyle* of the Artists created by `plot`, and we set the linestyle of the second line after the fact with `set_linestyle`.

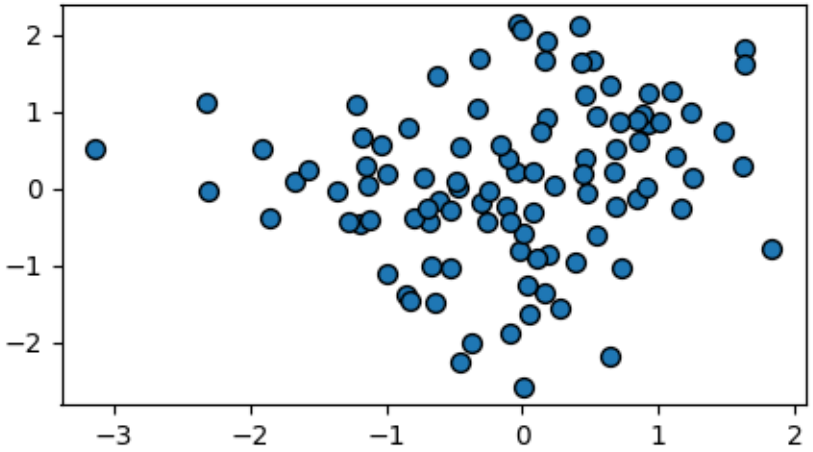
```
fig, ax = plt.subplots(figsize=(5, 2.7))
x = np.arange(len(data1))
ax.plot(x, np.cumsum(data1), color='blue', linewidth=3, linestyle='--')
l, = ax.plot(x, np.cumsum(data2), color='orange', linewidth=2)
l.set_linestyle(':')
```



### 2.5.1 Colors

Matplotlib has a very flexible array of colors that are accepted for most Artists; see *allowable color definitions* for a list of specifications. Some Artists will take multiple colors. i.e. for a `scatter` plot, the edge of the markers can be different colors from the interior:

```
fig, ax = plt.subplots(figsize=(5, 2.7))
ax.scatter(data1, data2, s=50, facecolor='C0', edgecolor='k')
```

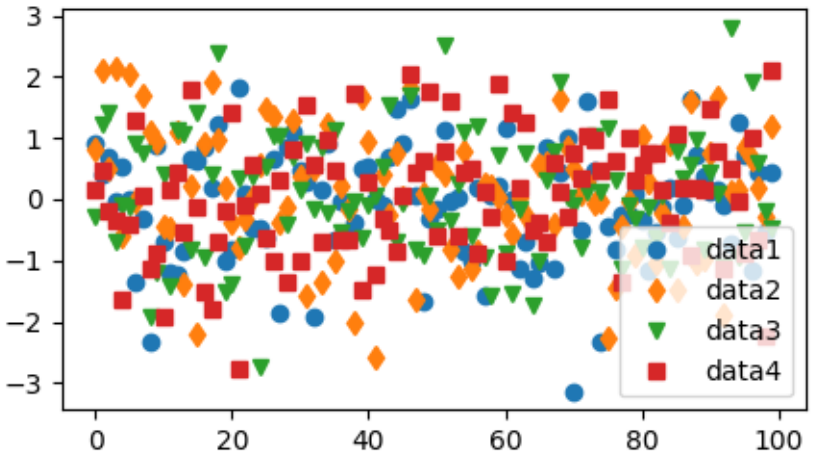



### 2.5.2 Linewidths, linestyles, and markersizes

Line widths are typically in typographic points (1 pt = 1/72 inch) and available for Artists that have stroked lines. Similarly, stroked lines can have a linestyle. See the *linestyles example*.

Marker size depends on the method being used. `plot` specifies markersize in points, and is generally the "diameter" or width of the marker. `scatter` specifies markersize as approximately proportional to the visual area of the marker. There is an array of markerstyles available as string codes (see `markers`), or users can define their own `MarkerStyle` (see *Marker reference*):

```
fig, ax = plt.subplots(figsize=(5, 2.7))
ax.plot(data1, 'o', label='data1')
ax.plot(data2, 'd', label='data2')
ax.plot(data3, 'v', label='data3')
ax.plot(data4, 's', label='data4')
ax.legend()
```

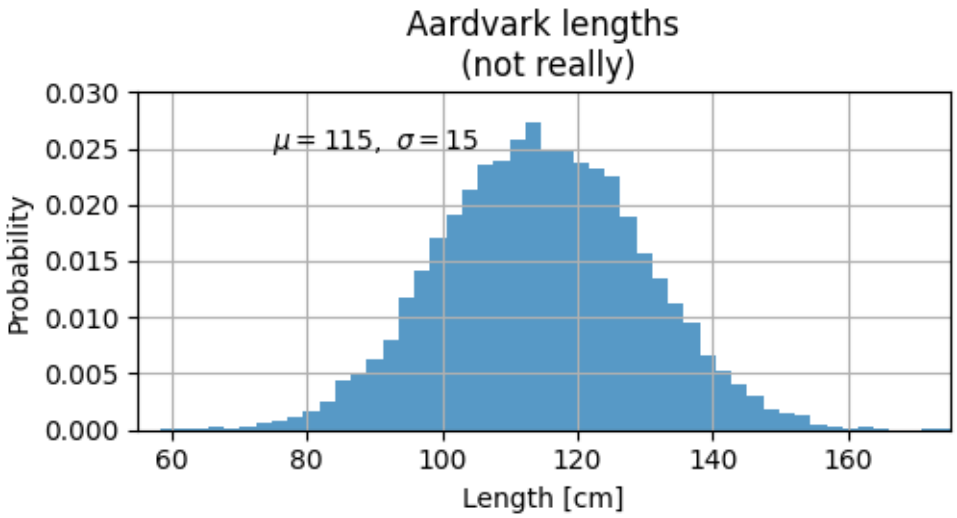



## 2.6 Labelling plots

### 2.6.1 Axes labels and text

`set_xlabel`, `set_ylabel`, and `set_title` are used to add text in the indicated locations (see *Text in Matplotlib* for more discussion). Text can also be directly added to plots using `text`:

```
mu, sigma = 115, 15
x = mu + sigma * np.random.randn(10000)
fig, ax = plt.subplots(figsize=(5, 2.7), layout='constrained')
# the histogram of the data
n, bins, patches = ax.hist(x, 50, density=True, facecolor='C0', alpha=0.75)

ax.set_xlabel('Length [cm]')
ax.set_ylabel('Probability')
ax.set_title('Aardvark lengths\n (not really)')
ax.text(75, .025, r'$\mu=115,\ \sigma=15$')
ax.axis([55, 175, 0, 0.03])
ax.grid(True)
```




All of the *text* functions return a *matplotlib.text.Text* instance. Just as with lines above, you can customize the properties by passing keyword arguments into the text functions:

```
t = ax.set_xlabel('my data', fontsize=14, color='red')
```

These properties are covered in more detail in *Text properties and layout*.

### 2.6.2 Using mathematical expressions in text

Matplotlib accepts TeX equation expressions in any text expression. For example to write the expression $\sigma_i = 15$ in the title, you can write a TeX expression surrounded by dollar signs:

```
ax.set_title(r'$\sigma_i=15$')
```

where the `r` preceding the title string signifies that the string is a *raw* string and not to treat backslashes as python escapes. Matplotlib has a built-in TeX expression parser and layout engine, and ships its own math fonts – for details see *Writing mathematical expressions*. You can also use LaTeX directly to format your text and incorporate the output directly into your display figures or saved postscript – see *Text rendering with LaTeX*.

### 2.6.3 Annotations

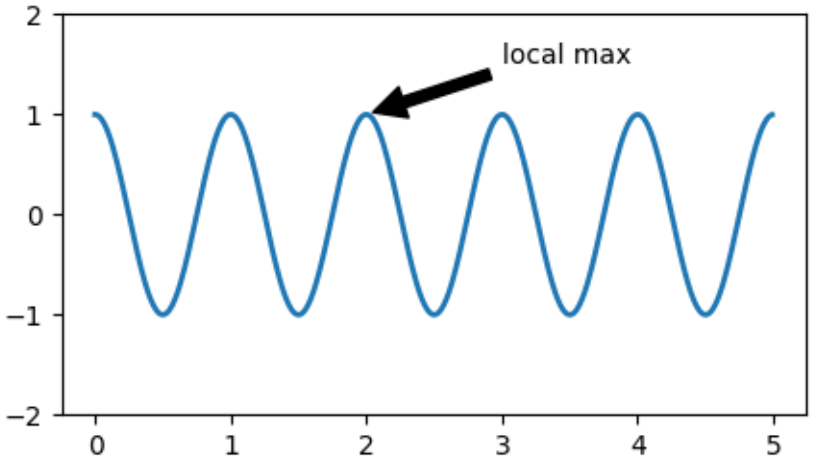
We can also annotate points on a plot, often by connecting an arrow pointing to *xy*, to a piece of text at *xytext*:

```
fig, ax = plt.subplots(figsize=(5, 2.7))

t = np.arange(0.0, 5.0, 0.01)
s = np.cos(2 * np.pi * t)
line, = ax.plot(t, s, lw=2)

ax.annotate('local max', xy=(2, 1), xytext=(3, 1.5),
            arrowprops=dict(facecolor='black', shrink=0.05))
```

(continues on next page)

(continued from previous page)

```
ax.set_ylim(-2, 2)
```

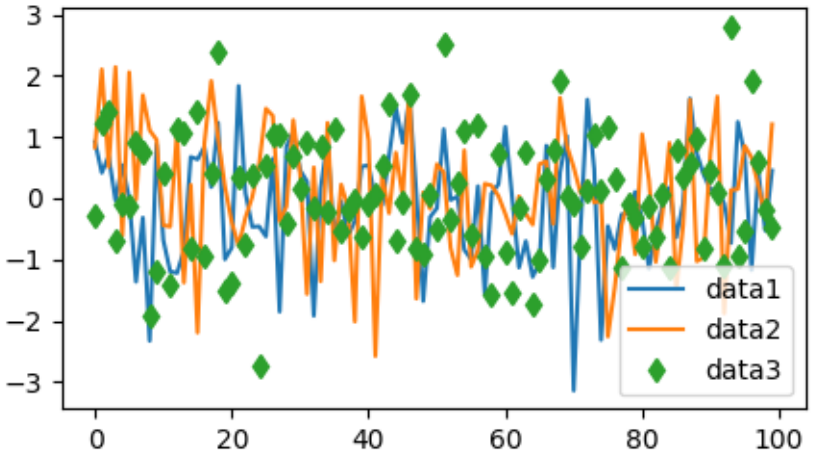



In this basic example, both *xy* and *xytext* are in data coordinates. There are a variety of other coordinate systems one can choose -- see *Basic annotation* and *Advanced annotation* for details. More examples also can be found in *Annotating Plots*.

### 2.6.4 Legends

Often we want to identify lines or markers with a `Axes.legend`:

```
fig, ax = plt.subplots(figsize=(5, 2.7))
ax.plot(np.arange(len(data1)), data1, label='data1')
ax.plot(np.arange(len(data2)), data2, label='data2')
ax.plot(np.arange(len(data3)), data3, 'd', label='data3')
ax.legend()
```



Legends in Matplotlib are quite flexible in layout, placement, and what Artists they can represent. They are discussed in detail in *Legend guide*.

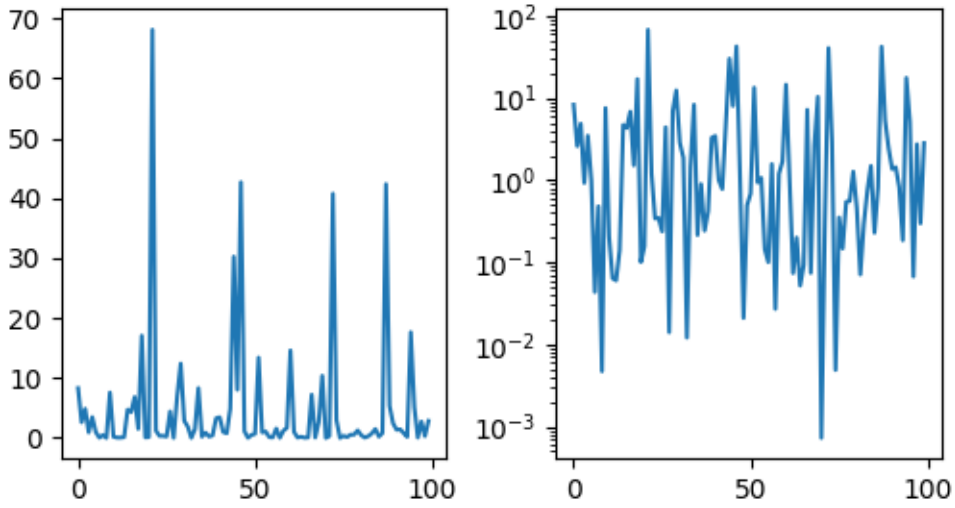
## 2.7 Axis scales and ticks

Each Axes has two (or three) `Axis` objects representing the x- and y-axis. These control the *scale* of the Axis, the tick *locators* and the tick *formatters*. Additional Axes can be attached to display further Axis objects.

### 2.7.1 Scales

In addition to the linear scale, Matplotlib supplies non-linear scales, such as a log-scale. Since log-scales are used so much there are also direct methods like `loglog`, `semilogx`, and `semilogy`. There are a number of scales (see *Scales* for other examples). Here we set the scale manually:

```
fig, axs = plt.subplots(1, 2, figsize=(5, 2.7), layout='constrained')
xdata = np.arange(len(data1))  # make an ordinal for this
data = 10**data1
axs[0].plot(xdata, data)

axs[1].set_yscale('log')
axs[1].plot(xdata, data)
```

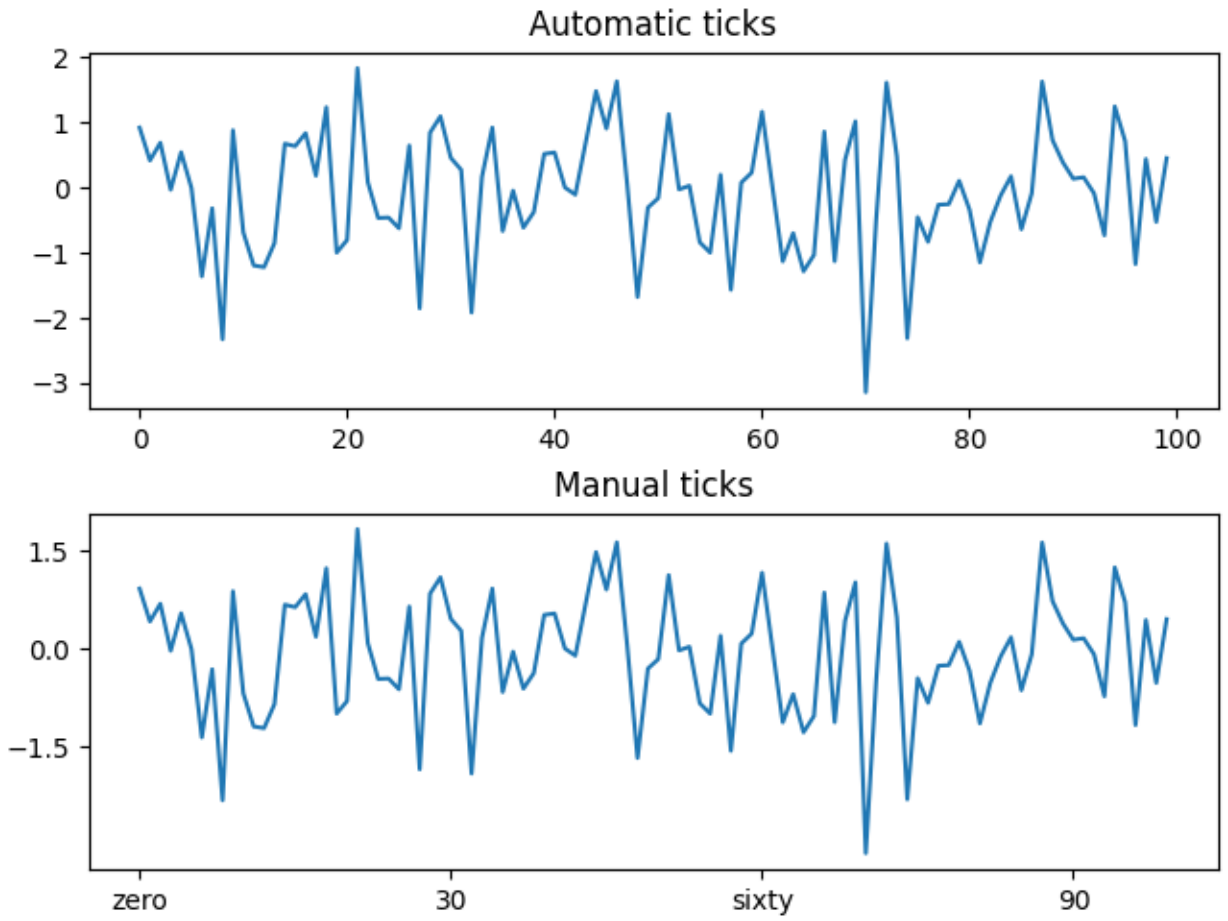



The scale sets the mapping from data values to spacing along the Axis. This happens in both directions, and gets combined into a *transform*, which is the way that Matplotlib maps from data coordinates to Axes, Figure, or screen coordinates. See *Transformations Tutorial*.

### 2.7.2 Tick locators and formatters

Each Axis has a tick *locator* and *formatter* that choose where along the Axis objects to put tick marks. A simple interface to this is `set_xticks`:

```
fig, axs = plt.subplots(2, 1, layout='constrained')
axs[0].plot(xdata, data1)
axs[0].set_title('Automatic ticks')

axs[1].plot(xdata, data1)
axs[1].set_xticks(np.arange(0, 100, 30), ['zero', '30', 'sixty', '90'])
axs[1].set_yticks([-1.5, 0, 1.5])  # note that we don't need to specify labels
axs[1].set_title('Manual ticks')
```

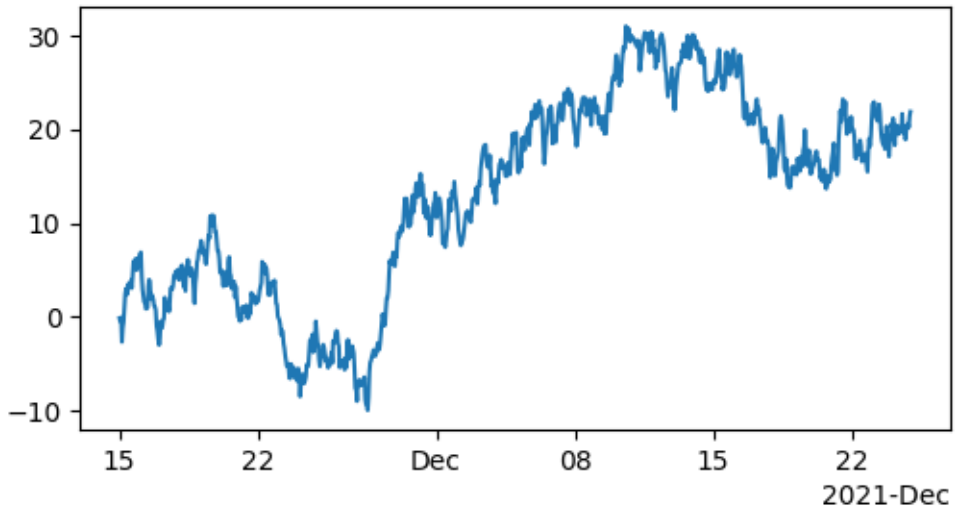



Different scales can have different locators and formatters; for instance the log-scale above uses `LogLocator` and `LogFormatter`. See *Tick locators* and *Tick formatters* for other formatters and locators and information for writing your own.

### 2.7.3 Plotting dates and strings

Matplotlib can handle plotting arrays of dates and arrays of strings, as well as floating point numbers. These get special locators and formatters as appropriate. For dates:

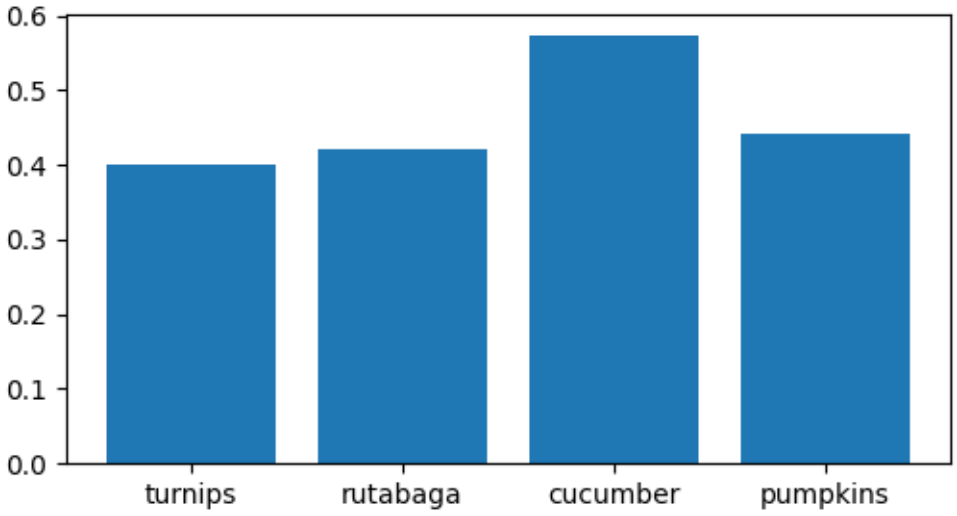
```
from matplotlib.dates import ConciseDateFormatter

fig, ax = plt.subplots(figsize=(5, 2.7), layout='constrained')
dates = np.arange(np.datetime64('2021-11-15'), np.datetime64('2021-12-25'),
                  np.timedelta64(1, 'h'))
data = np.cumsum(np.random.randn(len(dates)))
ax.plot(dates, data)
ax.xaxis.set_major_formatter(ConciseDateFormatter(ax.xaxis.get_major_
 ↪locator()))
```




For more information see the date examples (e.g. *Date tick labels*)

For strings, we get categorical plotting (see: *Plotting categorical variables*).

```
fig, ax = plt.subplots(figsize=(5, 2.7), layout='constrained')
categories = ['turnips', 'rutabaga', 'cucumber', 'pumpkins']

ax.bar(categories, np.random.rand(len(categories)))
```



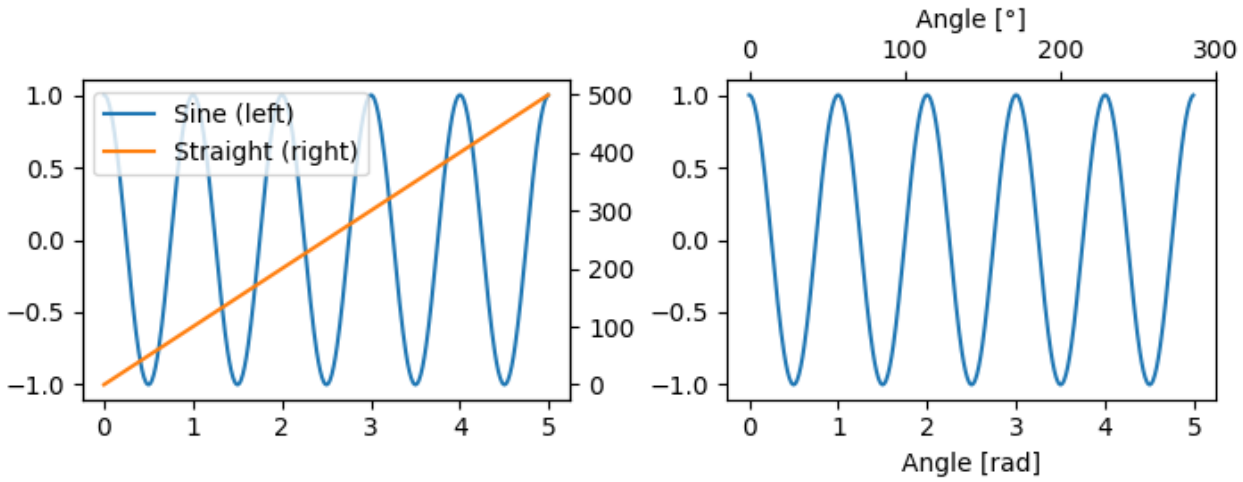

One caveat about categorical plotting is that some methods of parsing text files return a list of strings, even if the strings all represent numbers or dates. If you pass 1000 strings, Matplotlib will think you meant 1000 categories and will add 1000 ticks to your plot!

### 2.7.4 Additional Axis objects

Plotting data of different magnitude in one chart may require an additional y-axis. Such an Axis can be created by using *twinx* to add a new Axes with an invisible x-axis and a y-axis positioned at the right (analogously for *twiny*). See *Plots with different scales* for another example.

Similarly, you can add a *secondary_xaxis* or *secondary_yaxis* having a different scale than the main Axis to represent the data in different scales or units. See *Secondary Axis* for further examples.

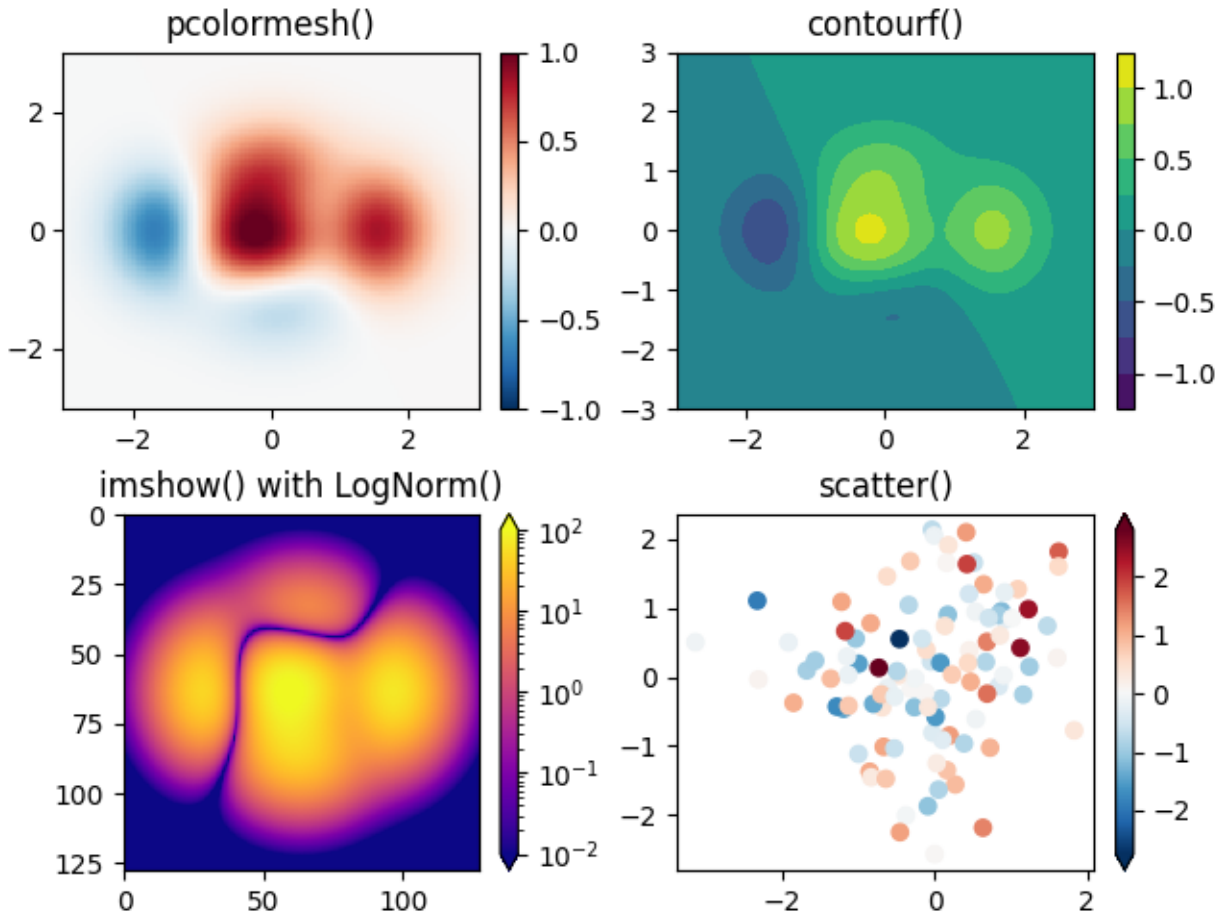
```
fig, (ax1, ax3) = plt.subplots(1, 2, figsize=(7, 2.7), layout='constrained')
l1, = ax1.plot(t, s)
ax2 = ax1.twinx()
l2, = ax2.plot(t, range(len(t)), 'C1')
ax2.legend([l1, l2], ['Sine (left)', 'Straight (right)'])

ax3.plot(t, s)
ax3.set_xlabel('Angle [rad]')
ax4 = ax3.secondary_xaxis('top', functions=(np.rad2deg, np.deg2rad))
ax4.set_xlabel('Angle [°]')
```




## 2.8 Color mapped data

Often we want to have a third dimension in a plot represented by colors in a colormap. Matplotlib has a number of plot types that do this:

```
from matplotlib.colors import LogNorm

X, Y = np.meshgrid(np.linspace(-3, 3, 128), np.linspace(-3, 3, 128))
Z = (1 - X/2 + X**5 + Y**3) * np.exp(-X**2 - Y**2)

fig, axs = plt.subplots(2, 2, layout='constrained')
pc = axs[0, 0].pcolormesh(X, Y, Z, vmin=-1, vmax=1, cmap='RdBu_r')
fig.colorbar(pc, ax=axs[0, 0])
axs[0, 0].set_title('pcolormesh()')

co = axs[0, 1].contourf(X, Y, Z, levels=np.linspace(-1.25, 1.25, 11))
fig.colorbar(co, ax=axs[0, 1])
axs[0, 1].set_title('contourf()')

pc = axs[1, 0].imshow(Z**2 * 100, cmap='plasma', norm=LogNorm(vmin=0.01,␣
 ↪vmax=100))
fig.colorbar(pc, ax=axs[1, 0], extend='both')
axs[1, 0].set_title('imshow() with LogNorm()')

pc = axs[1, 1].scatter(data1, data2, c=data3, cmap='RdBu_r')
fig.colorbar(pc, ax=axs[1, 1], extend='both')
axs[1, 1].set_title('scatter()')
```




### 2.8.1 Colormaps

These are all examples of Artists that derive from `ScalarMappable` objects. They all can set a linear mapping between *vmin* and *vmax* into the colormap specified by *cmap*. Matplotlib has many colormaps to choose from (*Choosing Colormaps in Matplotlib*) you can make your own (*Creating Colormaps in Matplotlib*) or download as third-party packages.

### 2.8.2 Normalizations

Sometimes we want a non-linear mapping of the data to the colormap, as in the `LogNorm` example above. We do this by supplying the ScalarMappable with the *norm* argument instead of *vmin* and *vmax*. More normalizations are shown at *Colormap normalization*.

### 2.8.3 Colorbars

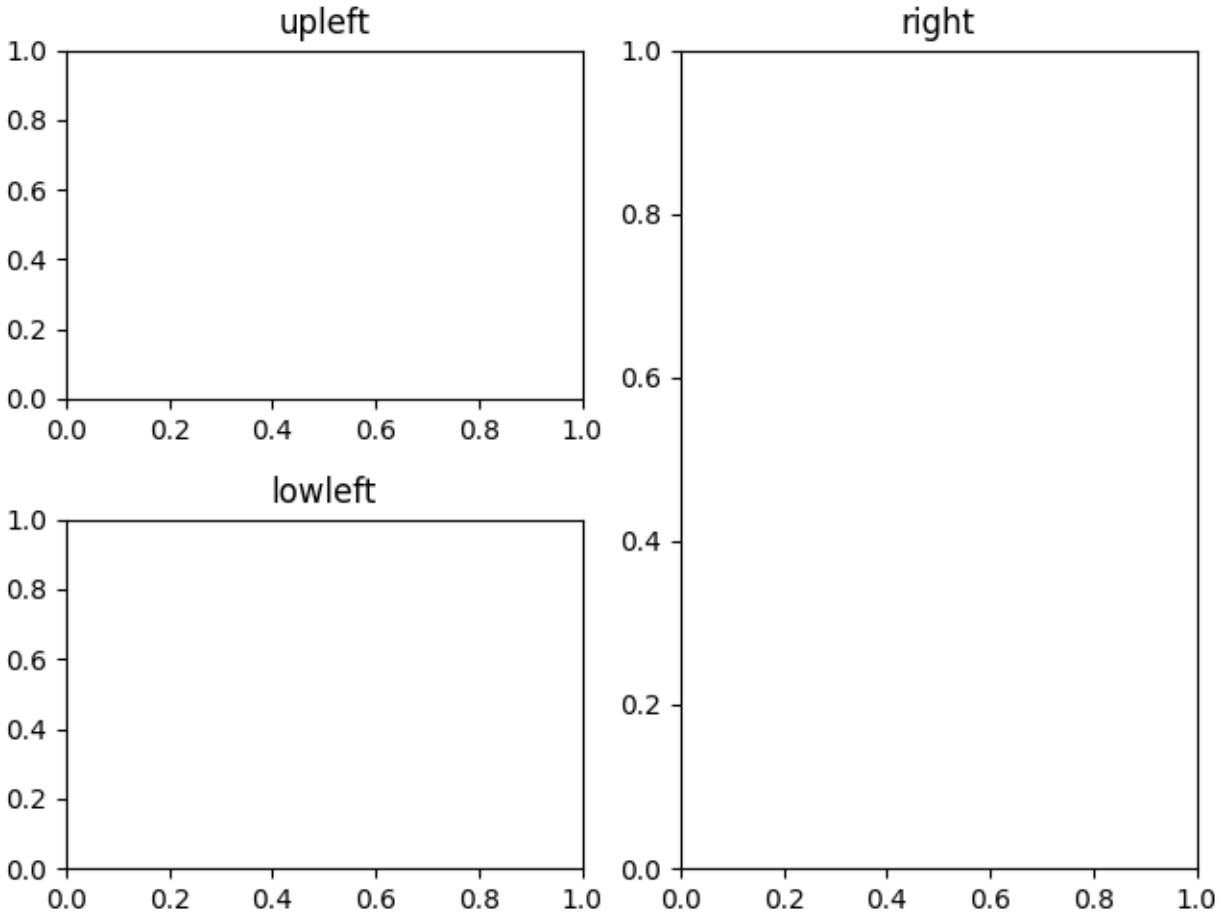
Adding a *colorbar* gives a key to relate the color back to the underlying data. Colorbars are figure-level Artists, and are attached to a ScalarMappable (where they get their information about the norm and colormap) and usually steal space from a parent Axes. Placement of colorbars can be complex: see *Placing colorbars* for details. You can also change the appearance of colorbars with the *extend* keyword to add arrows to the ends, and *shrink* and *aspect* to control the size. Finally, the colorbar will have default locators and formatters appropriate to the norm. These can be changed as for other Axis objects.

## 2.9 Working with multiple Figures and Axes

You can open multiple Figures with multiple calls to `fig = plt.figure()` or `fig2, ax = plt.subplots()`. By keeping the object references you can add Artists to either Figure.

Multiple Axes can be added a number of ways, but the most basic is `plt.subplots()` as used above. One can achieve more complex layouts, with Axes objects spanning columns or rows, using *subplot_mosaic*.

```
fig, axd = plt.subplot_mosaic([['upleft', 'right'],
                               ['lowleft', 'right']], layout='constrained')
axd['upleft'].set_title('upleft')
axd['lowleft'].set_title('lowleft')
axd['right'].set_title('right')
```




Matplotlib has quite sophisticated tools for arranging Axes: See *Arranging multiple Axes in a Figure* and *Complex and semantic figure composition (subplot_mosaic)*.

## 2.10 More reading

For more plot types see *Plot types* and the *API reference*, in particular the *Axes API*.

**Total running time of the script:** (0 minutes 6.236 seconds)

CHAPTER

THREE

# USING MATPLOTLIB

## 3.1 Frequently Asked Questions

### 3.1.1 I don't see a figure window

Please see *Debugging the figure windows not showing*.

### 3.1.2 Why do I have so many ticks, and/or why are they out of order?

One common cause for unexpected tick behavior is passing a *list of strings instead of numbers or datetime objects*. This can easily happen without notice when reading in a comma-delimited text file. Matplotlib treats lists of strings as *categorical* variables (*Plotting categorical variables*), and by default puts one tick per category, and plots them in the order in which they are supplied.

```
import matplotlib.pyplot as plt
import numpy as np

fig, ax = plt.subplots(1, 2, layout='constrained', figsize=(6, 2))

ax[0].set_title('Ticks seem out of order / misplaced')
x = ['5', '20', '1', '9']  # strings
y = [5, 20, 1, 9]
ax[0].plot(x, y, 'd')
ax[0].tick_params(axis='x', labelcolor='red', labelsize=14)

ax[1].set_title('Many ticks')
x = [str(xx) for xx in np.arange(100)]  # strings
y = np.arange(100)
ax[1].plot(x, y)
ax[1].tick_params(axis='x', labelcolor='red', labelsize=14)
```

The solution is to convert the list of strings to numbers or datetime objects (often `np.asarray(numeric_strings, dtype='float')` or `np.asarray(datetime_strings, dtype='datetime64[s]')`).

For more information see *Fixing too many ticks*.

### 3.1.3 Determine the extent of Artists in the Figure

Sometimes we want to know the extent of an Artist. Matplotlib *Artist* objects have a method *Artist.get_window_extent* that will usually return the extent of the artist in pixels. However, some artists, in particular text, must be rendered at least once before their extent is known. Matplotlib supplies *Figure.draw_without_rendering*, which should be called before calling `get_window_extent`.

### 3.1.4 Check whether a figure is empty

Empty can actually mean different things. Does the figure contain any artists? Does a figure with an empty *Axes* still count as empty? Is the figure empty if it was rendered pure white (there may be artists present, but they could be outside the drawing area or transparent)?

For the purpose here, we define empty as: "The figure does not contain any artists except it's background patch." The exception for the background is necessary, because by default every figure contains a *Rectangle* as it's background patch. This definition could be checked via:

```
def is_empty(figure):
    """
    Return whether the figure contains no Artists (other than the default
    background patch).
    """
    contained_artists = figure.get_children()
    return len(contained_artists) <= 1
```

We've decided not to include this as a figure method because this is only one way of defining empty, and checking the above is only rarely necessary. Usually the user or program handling the figure know if they have added something to the figure.

The only reliable way to check whether a figure would render empty is to actually perform such a rendering and inspect the result.

### 3.1.5 Find all objects in a figure of a certain type

Every Matplotlib artist (see *Artist tutorial*) has a method called *findobj()* that can be used to recursively search the artist for any artists it may contain that meet some criteria (e.g., match all *Line2D* instances or match some arbitrary filter function). For example, the following snippet finds every object in the figure which has a `set_color` property and makes the object blue:

```
def myfunc(x):
    return hasattr(x, 'set_color')

for o in fig.findobj(myfunc):
    o.set_color('blue')
```

You can also filter on class instances:

```
import matplotlib.text as text
for o in fig.findobj(text.Text):
    o.set_fontstyle('italic')
```

### 3.1.6 Prevent ticklabels from having an offset

The default formatter will use an offset to reduce the length of the ticklabels. To turn this feature off on a per-axis basis:

```
ax.xaxis.get_major_formatter().set_useOffset(False)
```

set *rcParams["axes.formatter.useoffset"]* (default: `True`), or use a different formatter. See *ticker* for details.

### 3.1.7 Save transparent figures

The *savefig()* command has a keyword argument *transparent* which, if 'True', will make the figure and axes backgrounds transparent when saving, but will not affect the displayed image on the screen.

If you need finer grained control, e.g., you do not want full transparency or you want to affect the screen displayed version as well, you can set the alpha properties directly. The figure has a *Rectangle* instance called *patch* and the axes has a Rectangle instance called *patch*. You can set any property on them directly (*facecolor*, *edgecolor*, *linewidth*, *linestyle*, *alpha*). e.g.:

```
fig = plt.figure()
fig.patch.set_alpha(0.5)
ax = fig.add_subplot(111)
ax.patch.set_alpha(0.5)
```

If you need *all* the figure elements to be transparent, there is currently no global alpha setting, but you can set the alpha channel on individual elements, e.g.:

```
ax.plot(x, y, alpha=0.5)
ax.set_xlabel('volts', alpha=0.5)
```

### 3.1.8 Save multiple plots to one pdf file

Many image file formats can only have one image per file, but some formats support multi-page files. Currently, Matplotlib only provides multi-page output to pdf files, using either the pdf or pgf backends, via the *backend_pdf.PdfPages* and *backend_pgf.PdfPages* classes.

### 3.1.9 Make room for tick labels

By default, Matplotlib uses fixed percentage margins around subplots. This can lead to labels overlapping or being cut off at the figure boundary. There are multiple ways to fix this:

- Manually adapt the subplot parameters using *Figure.subplots_adjust* / *pyplot.subplots_adjust*.
- Use one of the automatic layout mechanisms:
  - constrained layout (*Constrained layout guide*)
  - tight layout (*Tight layout guide*)
- Calculate good values from the size of the plot elements yourself (*Programmatically controlling subplot adjustment*)

### 3.1.10 Align my ylabels across multiple subplots

If you have multiple subplots over one another, and the y data have different scales, you can often get ylabels that do not align vertically across the multiple subplots, which can be unattractive. By default, Matplotlib positions the x location of the ylabel so that it does not overlap any of the y ticks. You can override this default behavior by specifying the coordinates of the label. To learn how, see *Align y-labels*

### 3.1.11 Control the draw order of plot elements

The draw order of plot elements, and thus which elements will be on top, is determined by the *set_zorder* property. See *Zorder Demo* for a detailed description.

### 3.1.12 Make the aspect ratio for plots equal

The Axes property *set_aspect()* controls the aspect ratio of the axes. You can set it to be 'auto', 'equal', or some ratio which controls the ratio:

```
ax = fig.add_subplot(111, aspect='equal')
```

### 3.1.13 Draw multiple y-axis scales

A frequent request is to have two scales for the left and right y-axis, which is possible using *twinx()* (more than two scales are not currently supported, though it is on the wish list). This works pretty well, though there are some quirks when you are trying to interactively pan and zoom, because both scales do not get the signals.

The approach uses *twinx()* (and its sister *twiny()*) to use *2 different axes*, turning the axes rectangular frame off on the 2nd axes to keep it from obscuring the first, and manually setting the tick locs and labels as desired. You can use separate `matplotlib.ticker` formatters and locators as desired because the two axes are independent.

### 3.1.14 Generate images without having a window appear

Simply do not call *show*, and directly save the figure to the desired format:

```
import matplotlib.pyplot as plt
plt.plot([1, 2, 3])
plt.savefig('myfig.png')
```

**See also:**

*Embedding in a web application server (Flask)* for information about running matplotlib inside of a web application.

### 3.1.15 Work with threads

Matplotlib is not thread-safe: in fact, there are known race conditions that affect certain artists. Hence, if you work with threads, it is your responsibility to set up the proper locks to serialize access to Matplotlib artists.

You may be able to work on separate figures from separate threads. However, you must in that case use a *non-interactive backend* (typically Agg), because most GUI backends *require* being run from the main thread as well.

### 3.1.16 Get help

There are a number of good resources for getting help with Matplotlib. There is a good chance your question has already been asked:

- The mailing list archive.
- GitHub issues.
- Stackoverflow questions tagged matplotlib.

If you are unable to find an answer to your question through search, please provide the following information in your e-mail to the mailing list:

- Your operating system (Linux/Unix users: post the output of `uname -a`).
- Matplotlib version:

  ```
  python -c "import matplotlib; print(matplotlib.__version__)"
  ```

- Where you obtained Matplotlib (e.g., your Linux distribution's packages, GitHub, PyPI, or Anaconda).
- Any customizations to your `matplotlibrc` file (see *Customizing Matplotlib with style sheets and rcParams*).
- If the problem is reproducible, please try to provide a *minimal*, standalone Python script that demonstrates the problem. This is *the* critical step. If you can't post a piece of code that we can run and reproduce your error, the chances of getting help are significantly diminished. Very often, the mere

act of trying to minimize your code to the smallest bit that produces the error will help you find a bug in *your* code that is causing the problem.

- Matplotlib provides debugging information through the `logging` library, and a helper function to set the logging level: one can call

```
plt.set_loglevel("info")  # or "debug" for more info
```

to obtain this debugging information.

Standard functions from the `logging` module are also applicable; e.g. one could call `logging.basicConfig(level="DEBUG")` even before importing Matplotlib (this is in particular necessary to get the logging info emitted during Matplotlib's import), or attach a custom handler to the "matplotlib" logger. This may be useful if you use a custom logging configuration.

If you compiled Matplotlib yourself, please also provide:

- your compiler version -- e.g., `gcc --version`.
- the output of:

```
pip install --verbose
```

The beginning of the build output contains lots of details about your platform that are useful for the Matplotlib developers to diagnose your problem.

If you compiled an older version of Matplotlib using the pre-Meson build system, instead provide:

- any changes you have made to `setup.py`/`setupext.py`,
- the output of:

```
rm -rf build
python setup.py build
```

Including this information in your first e-mail to the mailing list will save a lot of time.

You will likely get a faster response writing to the mailing list than filing a bug in the bug tracker. Most developers check the bug tracker only periodically. If your problem has been determined to be a bug and cannot be quickly solved, you may be asked to file a bug in the tracker so the issue doesn't get lost.

## 3.2 Figures and backends

When looking at Matplotlib visualization, you are almost always looking at Artists placed on a *Figure*. In the example below, the figure is the blue region and *add_subplot* has added an *Axes* artist to the *Figure* (see *Parts of a Figure*). A more complicated visualization can add multiple Axes to the Figure, colorbars, legends, annotations, and the Axes themselves can have multiple Artists added to them (e.g. `ax.plot` or `ax.imshow`).

```
fig = plt.figure(figsize=(4, 2), facecolor='lightskyblue',
                 layout='constrained')
fig.suptitle('A nice Matplotlib Figure')
```

(continues on next page)

(continued from previous page)

```
ax = fig.add_subplot()
ax.set_title('Axes', loc='left', fontstyle='oblique', fontsize='medium')
```

### 3.2.1 Introduction to Figures

```
fig = plt.figure(figsize=(2, 2), facecolor='lightskyblue',
                 layout='constrained')
fig.suptitle('Figure')
ax = fig.add_subplot()
ax.set_title('Axes', loc='left', fontstyle='oblique', fontsize='medium')
```

When looking at Matplotlib visualization, you are almost always looking at Artists placed on a *Figure*. In the example above, the figure is the blue region and *add_subplot* has added an *Axes* artist to the *Figure* (see *Parts of a Figure*). A more complicated visualization can add multiple Axes to the Figure, colorbars, legends, annotations, and the Axes themselves can have multiple Artists added to them (e.g. `ax.plot` or `ax.imshow`).


- *Viewing Figures*
  - *Notebooks and IDEs*
  - *Standalone scripts and interactive use*
- *Creating Figures*
  - *Figure options*
  - *Adding Artists*
- *Saving Figures*

## Viewing Figures

We will discuss how to create Figures in more detail below, but first it is helpful to understand how to view a Figure. This varies based on how you are using Matplotlib, and what *Backend* you are using.

## Notebooks and IDEs

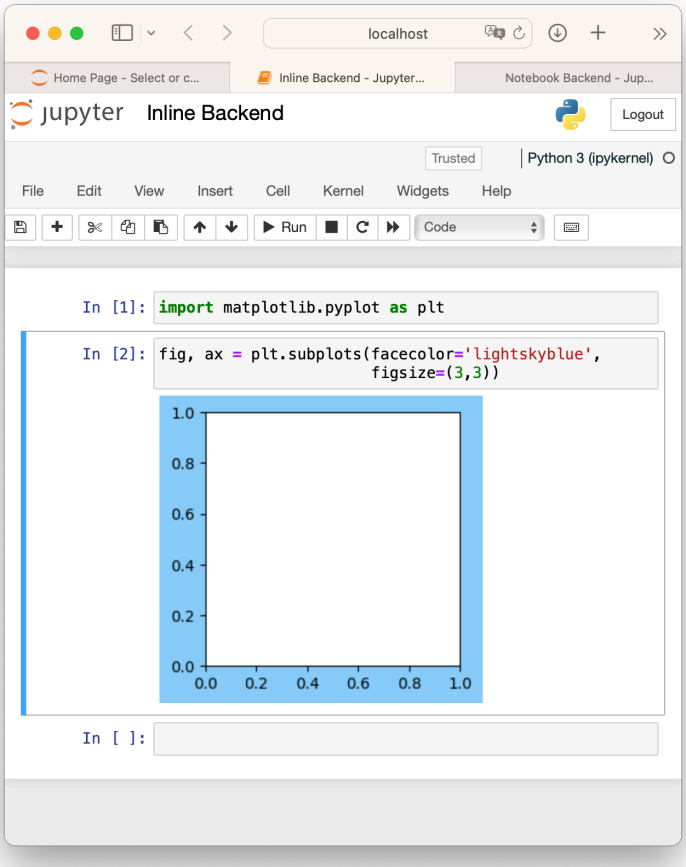



Fig. 1: Screenshot of a Jupyter Notebook, with a figure generated via the default inline backend.

If you are using a Notebook (e.g. Jupyter) or an IDE that renders Notebooks (PyCharm, VSCode, etc), then they have a backend that will render the Matplotlib Figure when a code cell is executed. The default Jupyter backend (`%matplotlib inline`) creates static plots that by default trim or expand the figure size to have a tight box around Artists added to the Figure (see *Saving Figures*, below). For interactive plots in Jupyter you will need to use an ipython "magic" like `%matplotlib widget` for the ipympl backend in `jupyter lab` or `notebook>=7`, or `%matplotlib notebook` for the Matplotlib *notebook* in `notebook<7` or `nbclassic`.

**Note:** The ipympl backend is in a separate package, see *Installing ipympl*.

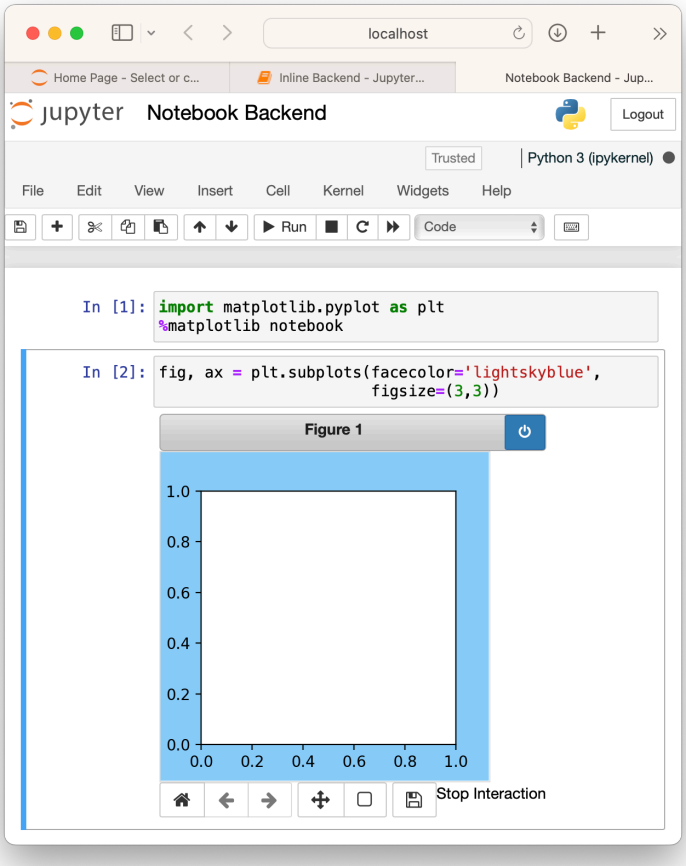



Fig. 2: Screenshot of a Jupyter Notebook with an interactive figure generated via the `%matplotlib notebook` magic. Users should also try the similar widget backend if using JupyterLab.

**See also:**

*Interactive figures*.

### Standalone scripts and interactive use

If the user is on a client with a windowing system, there are a number of *Backends* that can be used to render the Figure to the screen, usually using a Python Qt, Tk, or Wx toolkit, or the native MacOS backend. These are typically chosen either in the user's *matplotlibrc*, or by calling, for example, `matplotlib.use('QtAgg')` at the beginning of a session or script.

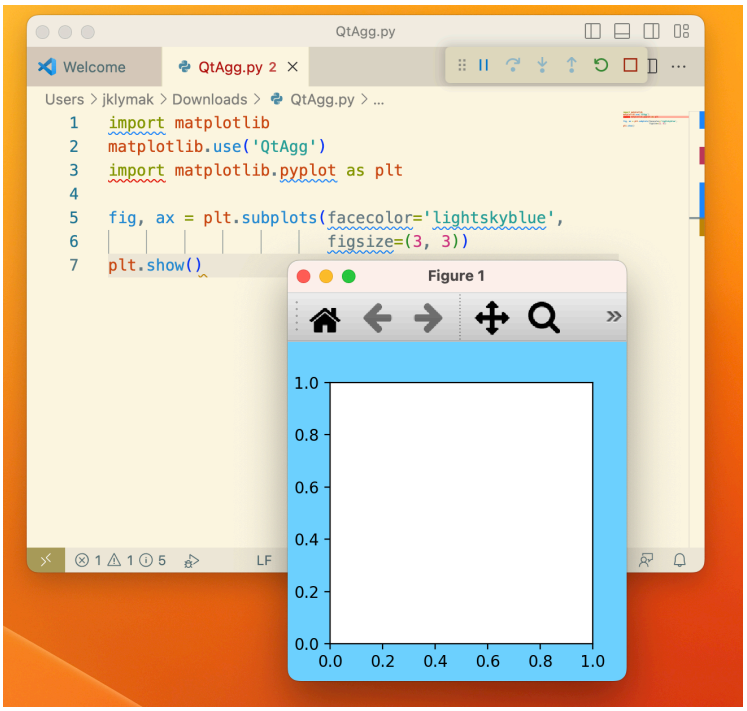



Fig. 3: Screenshot of a Figure generated via a python script and shown using the QtAgg backend.

When run from a script, or interactively (e.g. from an IPython shell) the Figure will not be shown until we call `plt.show()`. The Figure will appear in a new GUI window, and usually will have a toolbar with Zoom, Pan, and other tools for interacting with the Figure. By default, `plt.show()` blocks further interaction from the script or shell until the Figure window is closed, though that can be toggled off for some purposes. For more details, please see *Interactive mode*.

Note that if you are on a client that does not have access to a windowing system, the Figure will fallback to being drawn using the "Agg" backend, and cannot be viewed, though it can be *saved*.

**See also:**

*Interactive figures*.

### Creating Figures

By far the most common way to create a figure is using the *pyplot* interface. As noted in *Matplotlib Application Interfaces (APIs)*, the pyplot interface serves two purposes. One is to spin up the Backend and keep track of GUI windows. The other is a global state for Axes and Artists that allow a short-form API to plotting methods. In the example above, we use pyplot for the first purpose, and create the Figure object, `fig`. As a side effect `fig` is also added to pyplot's global state, and can be accessed via *gcf*.

Users typically want an Axes or a grid of Axes when they create a Figure, so in addition to *figure*, there are convenience methods that return both a Figure and some Axes. A simple grid of Axes can be achieved with *pyplot.subplots* (which simply wraps *Figure.subplots*):

```
fig, axs = plt.subplots(2, 2, figsize=(4, 3), layout='constrained')
```

More complex grids can be achieved with *pyplot.subplot_mosaic* (which wraps *Figure.subplot_mosaic*):

```
fig, axs = plt.subplot_mosaic([['A', 'right'], ['B', 'right']],
                              figsize=(4, 3), layout='constrained')
for ax_name, ax in axs.items():
    ax.text(0.5, 0.5, ax_name, ha='center', va='center')
```

Sometimes we want to have a nested layout in a Figure, with two or more sets of Axes that do not share the same subplot grid. We can use *add_subfigure* or *subfigures* to create virtual figures inside a parent Figure; see *Figure subfigures* for more details.

```
fig = plt.figure(layout='constrained', facecolor='lightskyblue')
fig.suptitle('Figure')
figL, figR = fig.subfigures(1, 2)
figL.set_facecolor('thistle')
axL = figL.subplots(2, 1, sharex=True)
axL[1].set_xlabel('x [m]')
figL.suptitle('Left subfigure')
figR.set_facecolor('paleturquoise')
axR = figR.subplots(1, 2, sharey=True)
axR[0].set_title('Axes 1')
figR.suptitle('Right subfigure')
```

It is possible to directly instantiate a *Figure* instance without using the pyplot interface. This is usually only necessary if you want to create your own GUI application or service that you do not want carrying the pyplot global state. See the embedding examples in *Embedding Matplotlib in graphical user interfaces* for examples of how to do this.

### Figure options

There are a few options available when creating figures. The Figure size on the screen is set by *figsize* and *dpi*. *figsize* is the `(width, height)` of the Figure in inches (or, if preferred, units of 72 typographic points). *dpi* are how many pixels per inch the figure will be rendered at. To make your Figures appear on the screen at the physical size you requested, you should set *dpi* to the same *dpi* as your graphics system. Note that many graphics systems now use a "dpi ratio" to specify how many screen pixels are used to represent a graphics pixel. Matplotlib applies the dpi ratio to the *dpi* passed to the figure to make it have higher resolution, so you should pass the lower number to the figure.

The *facecolor*, *edgecolor*, *linewidth*, and *frameon* options all change the appearance of the figure in expected ways, with *frameon* making the figure transparent if set to *False*.

Finally, the user can specify a layout engine for the figure with the *layout* parameter. Currently Matplotlib supplies *"constrained"*, *"compressed"* and *"tight"* layout engines. These rescale axes inside the Figure to prevent overlap of ticklabels, and try and align axes, and can save significant manual adjustment of artists on a Figure for many common cases.

### Adding Artists

The *Figure* class has a number of methods for adding artists to a *Figure* or a *SubFigure*. By far the most common are to add Axes of various configurations (*add_axes*, *add_subplot*, *subplots*, *subplot_mosaic*) and subfigures (*subfigures*). Colorbars are added to Axes or group of Axes at the Figure level (*colorbar*). It is also possible to have a Figure-level legend (*legend*). Other Artists include figure-wide labels (*suptitle*, *supxlabel*, *supylabel*) and text (*text*). Finally, low-level Artists can be added directly using *add_artist* usually with care being taken to use the appropriate transform. Usually these include `Figure.transFigure` which ranges from 0 to 1 in each direction, and represents the fraction of the current Figure size, or `Figure.dpi_scale_trans` which will be in physical units of inches from the bottom left corner of the Figure (see *Transformations Tutorial* for more details).

### Saving Figures

Finally, Figures can be saved to disk using the *savefig* method. `fig.savefig('MyFigure.png', dpi=200)` will save a PNG formatted figure to the file `MyFigure.png` in the current directory on disk with 200 dots-per-inch resolution. Note that the filename can include a relative or absolute path to any place on the file system.

Many types of output are supported, including raster formats like PNG, GIF, JPEG, TIFF and vector formats like PDF, EPS, and SVG.

By default, the size of the saved Figure is set by the Figure size (in inches) and, for the raster formats, the *dpi*. If *dpi* is not set, then the *dpi* of the Figure is used. Note that *dpi* still has meaning for vector formats like PDF if the Figure includes Artists that have been *rasterized*; the *dpi* specified will be the resolution of the rasterized objects.

It is possible to change the size of the Figure using the *bbox_inches* argument to savefig. This can be specified manually, again in inches. However, by far the most common use is `bbox_inches='tight'`. This option "shrink-wraps", trimming or expanding as needed, the size of the figure so that it is tight around all the artists

in a figure, with a small pad that can be specified by *pad_inches*, which defaults to 0.1 inches. The dashed box in the plot below shows the portion of the figure that would be saved if `bbox_inches='tight'` were used in savefig.

### 3.2.2 Backends

#### What is a backend?

Backends are used for displaying Matplotlib figures (see *Introduction to Figures*), on the screen, or for writing to files. A lot of documentation on the website and in the mailing lists refers to the "backend" and many new users are confused by this term. Matplotlib targets many different use cases and output formats. Some people use Matplotlib interactively from the Python shell and have plotting windows pop up when they type commands. Some people run Jupyter notebooks and draw inline plots for quick data analysis. Others embed Matplotlib into graphical user interfaces like PyQt or PyGObject to build rich applications. Some people use Matplotlib in batch scripts to generate postscript images from numerical simulations, and still others run web application servers to dynamically serve up graphs.

To support all of these use cases, Matplotlib can target different outputs, and each of these capabilities is called a backend; the "frontend" is the user facing code, i.e., the plotting code, whereas the "backend" does all the hard work behind-the-scenes to make the figure. There are two types of backends: user interface backends (for use in PyQt/PySide, PyGObject, Tkinter, wxPython, or macOS/Cocoa); also referred to as "interactive backends") and hardcopy backends to make image files (PNG, SVG, PDF, PS; also referred to as "non-interactive backends").

#### Selecting a backend

There are three ways to configure your backend:

- The *`rcParams["backend"]`* parameter in your `matplotlibrc` file
- The *`MPLBACKEND`* environment variable
- The function *`matplotlib.use()`*

Below is a more detailed description.

If there is more than one configuration present, the last one from the list takes precedence; e.g. calling *`matplotlib.use()`* will override the setting in your `matplotlibrc`.

Without a backend explicitly set, Matplotlib automatically detects a usable backend based on what is available on your system and on whether a GUI event loop is already running. The first usable backend in the following list is selected: MacOSX, QtAgg, GTK4Agg, Gtk3Agg, TkAgg, WxAgg, Agg. The last, Agg, is a non-interactive backend that can only write to files. It is used on Linux, if Matplotlib cannot connect to either an X display or a Wayland display.

Here is a detailed description of the configuration methods:

1. Setting *`rcParams["backend"]`* in your `matplotlibrc` file:

```
backend : qtagg   # use pyqt with antigrain (agg) rendering
```

See also *Customizing Matplotlib with style sheets and rcParams*.

2. Setting the *MPLBACKEND* environment variable:

   You can set the environment variable either for your current shell or for a single script.

   On Unix:

   ```
   > export MPLBACKEND=qtagg
   > python simple_plot.py

   > MPLBACKEND=qtagg python simple_plot.py
   ```

   On Windows, only the former is possible:

   ```
   > set MPLBACKEND=qtagg
   > python simple_plot.py
   ```

   Setting this environment variable will override the `backend` parameter in *any* `matplotlibrc`, even if there is a `matplotlibrc` in your current working directory. Therefore, setting *MPLBACKEND* globally, e.g. in your `.bashrc` or `.profile`, is discouraged as it might lead to counter-intuitive behavior.

3. If your script depends on a specific backend you can use the function *matplotlib.use()*:

   ```
   import matplotlib
   matplotlib.use('qtagg')
   ```

   This should be done before any figure is created, otherwise Matplotlib may fail to switch the backend and raise an ImportError.

   Using *use* will require changes in your code if users want to use a different backend. Therefore, you should avoid explicitly calling *use* unless absolutely necessary.

### The builtin backends

By default, Matplotlib should automatically select a default backend which allows both interactive work and plotting from scripts, with output to the screen and/or to a file, so at least initially, you will not need to worry about the backend. The most common exception is if your Python distribution comes without `tkinter` and you have no other GUI toolkit installed. This happens with certain Linux distributions, where you need to install a Linux package named `python-tk` (or similar).

If, however, you want to write graphical user interfaces, or a web application server (*Embedding in a web application server (Flask)*), or need a better understanding of what is going on, read on. To make things easily more customizable for graphical user interfaces, Matplotlib separates the concept of the renderer (the thing that actually does the drawing) from the canvas (the place where the drawing goes). The canonical renderer for user interfaces is `Agg` which uses the Anti-Grain Geometry C++ library to make a raster (pixel) image of the figure; it is used by the `QtAgg`, `GTK4Agg`, `GTK3Agg`, `wxAgg`, `TkAgg`, and `macosx` backends. An alternative renderer is based on the Cairo library, used by `QtCairo`, etc.

For the rendering engines, users can also distinguish between vector or raster renderers. Vector graphics languages issue drawing commands like "draw a line from this point to this point" and hence are scale free. Raster backends generate a pixel representation of the line whose accuracy depends on a DPI setting.

### Static backends

Here is a summary of the Matplotlib renderers (there is an eponymous backend for each; these are *non-interactive backends*, capable of writing to a file):

| Renderer | Filetypes | Description |
|---|---|---|
| AGG | png | raster graphics -- high quality images using the Anti-Grain Geometry engine. |
| PDF | pdf | vector graphics -- Portable Document Format output. |
| PS | ps, eps | vector graphics -- PostScript output. |
| SVG | svg | vector graphics -- Scalable Vector Graphics output. |
| PGF | pgf, pdf | vector graphics -- using the pgf package. |
| Cairo | png, ps, pdf, svg | raster or vector graphics -- using the Cairo library (requires pycairo or cairocffi). |

To save plots using the non-interactive backends, use the `matplotlib.pyplot.savefig('filename')` method.

### Interactive backends

These are the user interfaces and renderer combinations supported; these are *interactive backends*, capable of displaying to the screen and using appropriate renderers from the table above to write to a file:

| Back-end | Description |
|---|---|
| QtAgg | Agg rendering in a Qt canvas (requires PyQt or Qt for Python, a.k.a. PySide). This backend can be activated in IPython with `%matplotlib qt`. The Qt binding can be selected via the *QT_API* environment variable; see *Qt Bindings* for more details. |
| ipympl | Agg rendering embedded in a Jupyter widget (requires ipympl). This backend can be enabled in a Jupyter notebook with `%matplotlib ipympl` or `%matplotlib widget`. Works with Jupyter `lab` and `notebook>=7`. |
| GTK3 | Agg rendering to a GTK 3.x canvas (requires PyGObject and pycairo). This backend can be activated in IPython with `%matplotlib gtk3`. |
| GTK4 | Agg rendering to a GTK 4.x canvas (requires PyGObject and pycairo). This backend can be activated in IPython with `%matplotlib gtk4`. |
| macosx | Agg rendering into a Cocoa canvas in macOS. This backend can be activated in IPython with `%matplotlib osx`. |
| TkAgg | Agg rendering to a Tk canvas (requires TkInter). This backend can be activated in IPython with `%matplotlib tk`. |
| nbAgg | Embed an interactive figure in a Jupyter classic notebook. This backend can be enabled in Jupyter notebooks via `%matplotlib notebook` or `%matplotlib nbagg`. Works with Jupyter `notebook<7` and `nbclassic`. |
| WebAgg | On `show()` will start a tornado server with an interactive figure. |
| GTK3 | Cairo rendering to a GTK 3.x canvas (requires PyGObject and pycairo). |
| GTK4 | Cairo rendering to a GTK 4.x canvas (requires PyGObject and pycairo). |
| wxAgg | Agg rendering to a wxWidgets canvas (requires wxPython 4). This backend can be activated in IPython with `%matplotlib wx`. |

**Note:** The names of builtin backends are case-insensitive; e.g., 'QtAgg' and 'qtagg' are equivalent.

### ipympl

The ipympl backend is in a separate package that must be explicitly installed if you wish to use it, for example:

```
pip install ipympl
```

or

```
conda install ipympl -c conda-forge
```

See installing ipympl for more details.

### Using non-builtin backends

More generally, any importable backend can be selected by using any of the methods above. If `name.of.the.backend` is the module containing the backend, use `module://name.of.the.backend` as the backend name, e.g. `matplotlib.use('module://name.of.the.backend')`.

Information for backend implementers is available at *Writing a backend -- the pyplot interface*.

### Debugging the figure windows not showing

Sometimes things do not work as expected, usually during an install.

If you are using a Notebook or integrated development environment (see *Notebooks and IDEs*), please consult their documentation for debugging figures not working in their environments.

If you are using one of Matplotlib's graphics backends (see *Standalone scripts and interactive use*), make sure you know which one is being used:

```
import matplotlib

print(matplotlib.get_backend())
```

Try a simple plot to see if the GUI opens:

```
import matplotlib
import matplotlib.pyplot as plt

print(matplotlib.get_backend())
plt.plot((1, 4, 6))
plt.show()
```

If it does not, you perhaps have an installation problem. A good step at this point is to ensure that your GUI toolkit is installed properly, taking Matplotlib out of the testing. Almost all GUI toolkits have a small test program that can be run to test basic functionality. If this test fails, try re-installing.

### QtAgg, QtCairo, Qt5Agg, and Qt5Cairo

Test `PyQt5`.

If you have `PySide` or `PyQt6` installed rather than `PyQt5`, just change the import accordingly:

```
python -c "from PyQt5.QtWidgets import *; app = QApplication([]); win =␣
↪QMainWindow(); win.show(); app.exec()"
```

#### TkAgg and TkCairo

Test `tkinter`:

```
python3 -c "from tkinter import Tk; Tk().mainloop()"
```

#### GTK3Agg, GTK4Agg, GTK3Cairo, GTK4Cairo

Test `Gtk`:

```
python3 -c "from gi.repository import Gtk; win = Gtk.Window(); win.connect(
 ↪'destroy', Gtk.main_quit); win.show(); Gtk.main()"
```

#### wxAgg and wxCairo

Test `wx`:

```
import wx

app = wx.App(False)  # Create a new app, don't redirect stdout/stderr to a␣
 ↪window.
frame = wx.Frame(None, wx.ID_ANY, "Hello World") # A Frame is a top-level␣
 ↪window.
frame.Show(True)     # Show the frame.
app.MainLoop()
```

If the test works for your desired backend but you still cannot get Matplotlib to display a figure, then contact us (see *Get help*).

### 3.2.3 Matplotlib Application Interfaces (APIs)

Matplotlib has two major application interfaces, or styles of using the library:

- An explicit "Axes" interface that uses methods on a Figure or Axes object to create other Artists, and build a visualization step by step. This has also been called an "object-oriented" interface.
- An implicit "pyplot" interface that keeps track of the last Figure and Axes created, and adds Artists to the object it thinks the user wants.

In addition, a number of downstream libraries (like `pandas` and xarray) offer a `plot` method implemented directly on their data classes so that users can call `data.plot()`.

The difference between these interfaces can be a bit confusing, particularly given snippets on the web that use one or the other, or sometimes multiple interfaces in the same example. Here we attempt to point out how the "pyplot" and downstream interfaces relate to the explicit "Axes" interface to help users better navigate the library.

### Native Matplotlib interfaces

#### The explicit "Axes" interface

The "Axes" interface is how Matplotlib is implemented, and many customizations and fine-tuning end up being done at this level.

This interface works by instantiating an instance of a *Figure* class (`fig` below), using a *subplots* method (or similar) on that object to create one or more *Axes* objects (`ax` below), and then calling drawing methods on the Axes (`plot` in this example):

```
import matplotlib.pyplot as plt

fig = plt.figure()
ax = fig.subplots()
ax.plot([1, 2, 3, 4], [0, 0.5, 1, 0.2])
```

We call this an "explicit" interface because each object is explicitly referenced, and used to make the next object. Keeping references to the objects is very flexible, and allows us to customize the objects after they are created, but before they are displayed.

#### The implicit "pyplot" interface

The *pyplot* module shadows most of the *Axes* plotting methods to give the equivalent of the above, where the creation of the Figure and Axes is done for the user:

```
import matplotlib.pyplot as plt

plt.plot([1, 2, 3, 4], [0, 0.5, 1, 0.2])
```

This can be convenient, particularly when doing interactive work or simple scripts. A reference to the current Figure can be retrieved using *gcf* and to the current Axes by *gca*. The *pyplot* module retains a list of Figures, and each Figure retains a list of Axes on the figure for the user so that the following:

```
import matplotlib.pyplot as plt

plt.subplot(1, 2, 1)
plt.plot([1, 2, 3], [0, 0.5, 0.2])

plt.subplot(1, 2, 2)
plt.plot([3, 2, 1], [0, 0.5, 0.2])
```

is equivalent to:

```
import matplotlib.pyplot as plt

plt.subplot(1, 2, 1)
ax = plt.gca()
ax.plot([1, 2, 3], [0, 0.5, 0.2])

plt.subplot(1, 2, 2)
ax = plt.gca()
ax.plot([3, 2, 1], [0, 0.5, 0.2])
```

In the explicit interface, this would be:

```
import matplotlib.pyplot as plt

fig, axs = plt.subplots(1, 2)
axs[0].plot([1, 2, 3], [0, 0.5, 0.2])
axs[1].plot([3, 2, 1], [0, 0.5, 0.2])
```

### Why be explicit?

What happens if you have to backtrack, and operate on an old axes that is not referenced by `plt.gca()`? One simple way is to call `subplot` again with the same arguments. However, that quickly becomes inelegant. You can also inspect the Figure object and get its list of Axes objects, however, that can be misleading (colorbars are Axes too!). The best solution is probably to save a handle to every Axes you create, but if you do that, why not simply create the all the Axes objects at the start?

The first approach is to call `plt.subplot` again:

```
import matplotlib.pyplot as plt

plt.subplot(1, 2, 1)
plt.plot([1, 2, 3], [0, 0.5, 0.2])

plt.subplot(1, 2, 2)
plt.plot([3, 2, 1], [0, 0.5, 0.2])

plt.suptitle('Implicit Interface: re-call subplot')

for i in range(1, 3):
    plt.subplot(1, 2, i)
    plt.xlabel('Boo')
```

The second is to save a handle:

```
import matplotlib.pyplot as plt
```

(continues on next page)

(continued from previous page)

```
axs = []
ax = plt.subplot(1, 2, 1)
axs += [ax]
plt.plot([1, 2, 3], [0, 0.5, 0.2])

ax = plt.subplot(1, 2, 2)
axs += [ax]
plt.plot([3, 2, 1], [0, 0.5, 0.2])

plt.suptitle('Implicit Interface: save handles')

for i in range(2):
    plt.sca(axs[i])
    plt.xlabel('Boo')
```

However, the recommended way would be to be explicit from the outset:

```
import matplotlib.pyplot as plt

fig, axs = plt.subplots(1, 2)
axs[0].plot([1, 2, 3], [0, 0.5, 0.2])
axs[1].plot([3, 2, 1], [0, 0.5, 0.2])
fig.suptitle('Explicit Interface')
for i in range(2):
    axs[i].set_xlabel('Boo')
```

### Third-party library "Data-object" interfaces

Some third party libraries have chosen to implement plotting for their data objects, e.g. `data.plot()`, is seen in `pandas`, xarray, and other third-party libraries. For illustrative purposes, a downstream library may implement a simple data container that has `x` and `y` data stored together, and then implements a `plot` method:

```
import matplotlib.pyplot as plt

# supplied by downstream library:
class DataContainer:

    def __init__(self, x, y):
        """
        Proper docstring here!
        """
        self._x = x
        self._y = y

    def plot(self, ax=None, **kwargs):
        if ax is None:
```

(continues on next page)

(continued from previous page)

```
            ax = plt.gca()
        ax.plot(self._x, self._y, **kwargs)
        ax.set_title('Plotted from DataClass!')
        return ax


# what the user usually calls:
data = DataContainer([0, 1, 2, 3], [0, 0.2, 0.5, 0.3])
data.plot()
```

So the library can hide all the nitty-gritty from the user, and can make a visualization appropriate to the data type, often with good labels, choices of colormaps, and other convenient features.

In the above, however, we may not have liked the title the library provided. Thankfully, they pass us back the Axes from the `plot()` method, and understanding the explicit Axes interface, we could call: `ax.set_title('My preferred title')` to customize the title.

Many libraries also allow their `plot` methods to accept an optional *ax* argument. This allows us to place the visualization in an Axes that we have placed and perhaps customized.

### Summary

Overall, it is useful to understand the explicit "Axes" interface since it is the most flexible and underlies the other interfaces. A user can usually figure out how to drop down to the explicit interface and operate on the underlying objects. While the explicit interface can be a bit more verbose to setup, complicated plots will often end up simpler than trying to use the implicit "pyplot" interface.

---

**Note:** It is sometimes confusing to people that we import `pyplot` for both interfaces. Currently, the `pyplot` module implements the "pyplot" interface, but it also provides top-level Figure and Axes creation methods, and ultimately spins up the graphical user interface, if one is being used. So `pyplot` is still needed regardless of the interface chosen.

---

Similarly, the declarative interfaces provided by partner libraries use the objects accessible by the "Axes" interface, and often accept these as arguments or pass them back from methods. It is usually essential to use the explicit "Axes" interface to perform any customization of the default visualization, or to unpack the data into NumPy arrays and pass directly to Matplotlib.

### Appendix: "Axes" interface with data structures

Most *Axes* methods allow yet another API addressing by passing a *data* object to the method and specifying the arguments as strings:

```
import matplotlib.pyplot as plt

data = {'xdat': [0, 1, 2, 3], 'ydat': [0, 0.2, 0.4, 0.1]}
fig, ax = plt.subplots(figsize=(2, 2))
ax.plot('xdat', 'ydat', data=data)
```

### Appendix: "pylab" interface

There is one further interface that is highly discouraged, and that is to basically do `from matplotlib.pylab import *`. This imports all the functions from `matplotlib.pyplot`, `numpy`, `numpy.fft`, `numpy.linalg`, and `numpy.random`, and some additional functions into the global namespace.

Such a pattern is considered bad practice in modern python, as it clutters the global namespace. Even more severely, in the case of `pylab`, this will overwrite some builtin functions (e.g. the builtin `sum` will be replaced by `numpy.sum`), which can lead to unexpected behavior.

## 3.2.4 Interactive figures

When working with data, interactivity can be invaluable. The pan/zoom and mouse-location tools built into the Matplotlib GUI windows are often sufficient, but you can also use the event system to build customized data exploration tools.

**See also:**

*Introduction to Figures*.

Matplotlib ships with *backends* binding to several GUI toolkits (Qt, Tk, Wx, GTK, macOS, JavaScript) and third party packages provide bindings to kivy and Jupyter Lab. For the figures to be responsive to mouse, keyboard, and paint events, the GUI event loop needs to be integrated with an interactive prompt. We recommend using IPython (see *below*).

The *pyplot* module provides functions for explicitly creating figures that include interactive tools, a toolbar, a tool-tip, and *key bindings*:

***pyplot.figure***

Creates a new empty *Figure* or selects an existing figure

***pyplot.subplots***

Creates a new *Figure* and fills it with a grid of *Axes*

***pyplot.gcf***

Get the current *Figure*. If there is current no figure on the pyplot figure stack, a new figure is created

*pyplot.gca*

Get the current *Axes*. If there is current no Axes on the Figure, a new one is created

Almost all of the functions in *pyplot* pass through the current *Figure* / *Axes* (or create one) as appropriate.

Matplotlib keeps a reference to all of the open figures created via *pyplot.figure* or *pyplot.subplots* so that the figures will not be garbage collected. *Figure*s can be closed and deregistered from *pyplot* individually via *pyplot.close*; all open *Figure*s can be closed via `plt.close('all')`.

**See also:**

For more discussion of Matplotlib's event system and integrated event loops: - *Interactive figures and asynchronous programming* - *Event handling and picking*

## IPython integration

We recommend using IPython for an interactive shell. In addition to all of its features (improved tab-completion, magics, multiline editing, etc), it also ensures that the GUI toolkit event loop is properly integrated with the command line (see *Command prompt integration*).

In this example, we create and modify a figure via an IPython prompt. The figure displays in a QtAgg GUI window. To configure the integration and enable *interactive mode* use the `%matplotlib` magic:

```
In [1]: %matplotlib
Using matplotlib backend: QtAgg

In [2]: import matplotlib.pyplot as plt
```

Create a new figure window:

```
In [3]: fig, ax = plt.subplots()
```

Add a line plot of the data to the window:

```
In [4]: ln, = ax.plot(range(5))
```

Change the color of the line from blue to orange:

```
In [5]: ln.set_color('orange')
```

If you wish to disable automatic redrawing of the plot:

```
In [6]: plt.ioff()
```

If you wish to re-enable automatic redrawing of the plot:

```
In [7]: plt.ion()
```

In recent versions of `Matplotlib` and `IPython`, it is sufficient to import *matplotlib.pyplot* and call *pyplot.ion*. Using the `%` magic is guaranteed to work in all versions of Matplotlib and IPython.

### Interactive mode

| | |
|---|---|
| *pyplot.ion* | Enable interactive mode. |
| *pyplot.ioff* | Disable interactive mode. |
| *pyplot.isinteractive* | Return whether plots are updated after every plotting command. |

| | |
|---|---|
| *pyplot.show* | Display all open figures. |
| *pyplot.pause* | Run the GUI event loop for *interval* seconds. |

Interactive mode controls:

- whether created figures are automatically shown
- whether changes to artists automatically trigger re-drawing existing figures
- when *pyplot.show()* returns if given no arguments: immediately, or after all of the figures have been closed

If in interactive mode:

- newly created figures will be displayed immediately
- figures will automatically redraw when elements are changed
- *pyplot.show()* displays the figures and immediately returns

If not in interactive mode:

- newly created figures and changes to figures are not displayed until
  - *pyplot.show()* is called
  - *pyplot.pause()* is called
  - *FigureCanvasBase.flush_events()* is called
- *pyplot.show()* runs the GUI event loop and does not return until all the plot windows are closed

If you are in non-interactive mode (or created figures while in non-interactive mode) you may need to explicitly call *pyplot.show* to display the windows on your screen. If you only want to run the GUI event loop for a fixed amount of time, you can use *pyplot.pause*. This will block the progress of your code as if you had called `time.sleep`, ensure the current window is shown and re-drawn if needed, and run the GUI event loop for the specified period of time.

The GUI event loop being integrated with your command prompt and the figures being in interactive mode are independent of each other. If you try to use *pyplot.ion* without arranging for the event-loop integration, your figures will appear but will not be interactive while the prompt is waiting for input. You will not be able to pan/zoom and the figure may not even render (the window might appear black, transparent, or as a snapshot of the desktop under it). Conversely, if you configure the event loop integration, displayed figures will be responsive while waiting for input at the prompt, regardless of pyplot's "interactive mode".

No matter what combination of interactive mode setting and event loop integration, figures will be responsive if you use `pyplot.show(block=True)`, *`pyplot.pause`*, or run the GUI main loop in some other way.

**Warning:** Using *`Figure.show`*, it is possible to display a figure on the screen without starting the event loop and without being in interactive mode. This may work (depending on the GUI toolkit) but will likely result in a non-responsive figure.

### Default UI

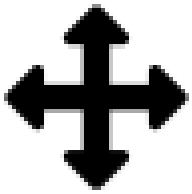
The windows created by *`pyplot`* have an interactive toolbar with navigation buttons and a readout of the data values the cursor is pointing at.

### Interactive navigation

All figure windows come with a navigation toolbar, which can be used to navigate through the data set.

**The `Home`, `Forward` and `Back` buttons**

These are similar to a web browser's home, forward and back controls. `Forward` and `Back` are used to navigate back and forth between previously defined views. They have no meaning unless you have already navigated somewhere else using the pan and zoom buttons. This is analogous to trying to click `Back` on your web browser before visiting a new page or `Forward` before you have gone back to a page -- nothing happens. `Home` takes you to the first, default view of your data.



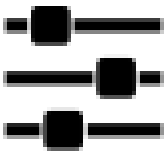
**The `Pan/Zoom` button**

This button has two modes: pan and zoom. Click the `Pan/Zoom` button to activate panning and zooming, then put your mouse somewhere over an axes. Press the left mouse button and hold it to pan the figure, dragging it to a new position. When you release it, the data under the point where you pressed will be moved to the point where you released. If you press 'x' or 'y' while panning the motion will be constrained to the x or y axis, respectively. Press the right mouse button to zoom, dragging it to a new position. The x axis will be zoomed in proportionately to the rightward movement and zoomed out proportionately to the leftward movement. The same is true for the y axis and up/down motions (up zooms in, down zooms out). The point under your mouse when you begin the zoom remains stationary, allowing you to zoom in or out around that point as much as you wish. You can use the modifier keys 'x', 'y' or 'CONTROL' to constrain the zoom to the x axis, the y axis, or aspect ratio preserve, respectively.

With polar plots, the pan and zoom functionality behaves differently. The radius axis labels can be dragged using the left mouse button. The radius scale can be zoomed in and out using the right mouse button.

**The `Zoom-to-Rectangle` button**

Put your mouse somewhere over an axes and press a mouse button. Define a rectangular region by dragging the mouse while holding the button to a new location. When using the left mouse button, the axes view limits will be zoomed to the defined region. When using the right mouse button, the axes view limits will be zoomed out, placing the original axes in the defined region.



**The `Subplot-configuration` button**

Use this button to configure the appearance of the subplot. You can stretch or compress the left, right, top, or bottom side of the subplot, or the space between the rows or space between the columns.

**The `Save` button**

Click this button to launch a file save dialog. You can save files with the following extensions: `png`, `ps`, `eps`, `svg` and `pdf`.

### Navigation keyboard shortcuts

A number of helpful keybindings are registered by default. The following table holds all the default keys, which can be overwritten by use of your *matplotlibrc*.

| Command | Default key binding and rcParam |
|---|---|
| Home/Reset | *rcParams["keymap.home"]* (default: `['h', 'r', 'home']`) |
| Back | *rcParams["keymap.back"]* (default: `['left', 'c', 'backspace', 'MouseButton.BACK']`) |
| Forward | *rcParams["keymap.forward"]* (default: `['right', 'v', 'MouseButton.FORWARD']`) |
| Pan/Zoom | *rcParams["keymap.pan"]* (default: `['p']`) |
| Zoom-to-rect | *rcParams["keymap.zoom"]* (default: `['o']`) |
| Save | *rcParams["keymap.save"]* (default: `['s', 'ctrl+s']`) |
| Toggle fullscreen | *rcParams["keymap.fullscreen"]* (default: `['f', 'ctrl+f']`) |
| Toggle major grids | *rcParams["keymap.grid"]* (default: `['g']`) |
| Toggle minor grids | *rcParams["keymap.grid_minor"]* (default: `['G']`) |
| Toggle x axis scale (log/linear) | *rcParams["keymap.xscale"]* (default: `['k', 'L']`) |
| Toggle y axis scale (log/linear) | *rcParams["keymap.yscale"]* (default: `['l']`) |
| Close Figure | *rcParams["keymap.quit"]* (default: `['ctrl+w', 'cmd+w', 'q']`) |
| Constrain pan/zoom to x axis | hold **x** when panning/zooming with mouse |
| Constrain pan/zoom to y axis | hold **y** when panning/zooming with mouse |
| Preserve aspect ratio | hold **CONTROL** when panning/zooming with mouse |

### Other Python prompts

Interactive mode works in the default Python prompt:

```
>>> import matplotlib.pyplot as plt
>>> plt.ion()
>>>
```

However, this does not ensure that the event hook is properly installed and your figures may not be responsive. Please consult the documentation of your GUI toolkit for details.

### Jupyter Notebooks / JupyterLab

To get interactive figures in the 'classic' notebook or Jupyter lab, use the ipympl backend (must be installed separately) which uses the **ipywidget** framework. If `ipympl` is installed use the magic:

```
%matplotlib widget
```

to select and enable it.

If you only need to use the classic notebook (i.e. `notebook<7`), you can use

```
%matplotlib notebook
```

which uses the *backend_nbagg* backend provided by Matplotlib; however, nbagg does not work in Jupyter Lab.

---

**Note:** To get the interactive functionality described here, you must be using an interactive backend. The default backend in notebooks, the inline backend, is not. `backend_inline` renders the figure once and inserts a static image into the notebook when the cell is executed. Because the images are static, they cannot be panned / zoomed, take user input, or be updated from other cells.

---

### GUIs + Jupyter

You can also use one of the non-`ipympl` GUI backends in a Jupyter Notebook. If you are running your Jupyter kernel locally, the GUI window will spawn on your desktop adjacent to your web browser. If you run your notebook on a remote server, the kernel will try to open the GUI window on the remote computer. Unless you have arranged to forward the xserver back to your desktop, you will not be able to see or interact with the window. It may also raise an exception.

#### PyCharm, Spyder, and VSCode

Many IDEs have built-in integration with Matplotlib, please consult their documentation for configuration details.

### 3.2.5 Interactive figures and asynchronous programming

Matplotlib supports rich interactive figures by embedding figures into a GUI window. The basic interactions of panning and zooming in an Axes to inspect your data is 'baked in' to Matplotlib. This is supported by a full mouse and keyboard event handling system that you can use to build sophisticated interactive graphs.

This guide is meant to be an introduction to the low-level details of how Matplotlib integration with a GUI event loop works. For a more practical introduction to the Matplotlib event API see *event handling system*, Interactive Tutorial, and Interactive Applications using Matplotlib.

#### Event loops

Fundamentally, all user interaction (and networking) is implemented as an infinite loop waiting for events from the user (via the OS) and then doing something about it. For example, a minimal Read Evaluate Print Loop (REPL) is

```
exec_count = 0
while True:
    inp = input(f"[{exec_count}] > ")          # Read
    ret = eval(inp)                            # Evaluate
    print(ret)                                 # Print
    exec_count += 1                            # Loop
```

This is missing many niceties (for example, it exits on the first exception!), but is representative of the event loops that underlie all terminals, GUIs, and servers[1]. In general the *Read* step is waiting on some sort of I/O -- be it user input or the network -- while the *Evaluate* and *Print* are responsible for interpreting the input and then **doing** something about it.

In practice we interact with a framework that provides a mechanism to register callbacks to be run in response to specific events rather than directly implement the I/O loop[2]. For example "when the user clicks on this button, please run this function" or "when the user hits the 'z' key, please run this other function". This allows users to write reactive, event-driven, programs without having to delve into the nitty-gritty[3] details of I/O. The core event loop is sometimes referred to as "the main loop" and is typically started, depending on the library, by methods with names like `_exec`, `run`, or `start`.

[1] A limitation of this design is that you can only wait for one input, if there is a need to multiplex between multiple sources then the loop would look something like

```
fds = [...]
    while True:                        # Loop
        inp = select(fds).read()       # Read
        eval(inp)                      # Evaluate / Print
```

[2] Or you can write your own if you must.

[3] These examples are aggressively dropping many of the complexities that must be dealt with in the real world such as keyboard interrupts, timeouts, bad input, resource allocation and cleanup, etc.

All GUI frameworks (Qt, Wx, Gtk, tk, macOS, or web) have some method of capturing user interactions and passing them back to the application (for example `Signal` / `Slot` framework in Qt) but the exact details depend on the toolkit. Matplotlib has a *backend* for each GUI toolkit we support which uses the toolkit API to bridge the toolkit UI events into Matplotlib's *event handling system*. You can then use *`FigureCanvasBase.mpl_connect`* to connect your function to Matplotlib's event handling system. This allows you to directly interact with your data and write GUI toolkit agnostic user interfaces.

### Command prompt integration

So far, so good. We have the REPL (like the IPython terminal) that lets us interactively send code to the interpreter and get results back. We also have the GUI toolkit that runs an event loop waiting for user input and lets us register functions to be run when that happens. However, if we want to do both we have a problem: the prompt and the GUI event loop are both infinite loops that each think *they* are in charge! In order for both the prompt and the GUI windows to be responsive we need a method to allow the loops to 'timeshare' :

1. let the GUI main loop block the python process when you want interactive windows
2. let the CLI main loop block the python process and intermittently run the GUI loop
3. fully embed python in the GUI (but this is basically writing a full application)

### Blocking the prompt

| | |
|---|---|
| *`pyplot.show`* | Display all open figures. |
| *`pyplot.pause`* | Run the GUI event loop for *interval* seconds. |
| *`backend_bases.FigureCanvasBase.start_event_loop`* | Start a blocking event loop. |
| *`backend_bases.FigureCanvasBase.stop_event_loop`* | Stop the current blocking event loop. |

The simplest "integration" is to start the GUI event loop in 'blocking' mode and take over the CLI. While the GUI event loop is running you cannot enter new commands into the prompt (your terminal may echo the characters typed into the terminal, but they will not be sent to the Python interpreter because it is busy running the GUI event loop), but the figure windows will be responsive. Once the event loop is stopped (leaving any still open figure windows non-responsive) you will be able to use the prompt again. Re-starting the event loop will make any open figure responsive again (and will process any queued up user interaction).

To start the event loop until all open figures are closed, use *`pyplot.show`* as

```
pyplot.show(block=True)
```

To start the event loop for a fixed amount of time (in seconds) use *`pyplot.pause`*.

If you are not using *`pyplot`* you can start and stop the event loops via *`FigureCanvasBase.start_event_loop`* and *`FigureCanvasBase.stop_event_loop`*. However, in most contexts where you would not be using *`pyplot`* you are embedding Matplotlib in a large GUI application and the GUI event loop should already be running for the application.

Away from the prompt, this technique can be very useful if you want to write a script that pauses for user interaction, or displays a figure between polling for additional data. See *Scripts and functions* for more details.

### Input hook integration

While running the GUI event loop in a blocking mode or explicitly handling UI events is useful, we can do better! We really want to be able to have a usable prompt **and** interactive figure windows.

We can do this using the 'input hook' feature of the interactive prompt. This hook is called by the prompt as it waits for the user to type (even for a fast typist the prompt is mostly waiting for the human to think and move their fingers). Although the details vary between prompts the logic is roughly

1. start to wait for keyboard input
2. start the GUI event loop
3. as soon as the user hits a key, exit the GUI event loop and handle the key
4. repeat

This gives us the illusion of simultaneously having interactive GUI windows and an interactive prompt. Most of the time the GUI event loop is running, but as soon as the user starts typing the prompt takes over again.

This time-share technique only allows the event loop to run while python is otherwise idle and waiting for user input. If you want the GUI to be responsive during long running code it is necessary to periodically flush the GUI event queue as described in *Explicitly spinning the event Loop*. In this case it is your code, not the REPL, which is blocking the process so you need to handle the "time-share" manually. Conversely, a very slow figure draw will block the prompt until it finishes drawing.

### Full embedding

It is also possible to go the other direction and fully embed figures (and a Python interpreter) in a rich native application. Matplotlib provides classes for each toolkit which can be directly embedded in GUI applications (this is how the built-in windows are implemented!). See *Embedding Matplotlib in graphical user interfaces* for more details.

### Scripts and functions

| | |
|---|---|
| `backend_bases.FigureCanvasBase.flush_events` | Flush the GUI events for the figure. |
| `backend_bases.FigureCanvasBase.draw_idle` | Request a widget redraw once control returns to the GUI event loop. |
| `figure.Figure.ginput` | Blocking call to interact with a figure. |
| `pyplot.ginput` | Blocking call to interact with a figure. |
| `pyplot.show` | Display all open figures. |
| `pyplot.pause` | Run the GUI event loop for *interval* seconds. |

There are several use-cases for using interactive figures in scripts:

- capture user input to steer the script
- progress updates as a long running script progresses
- streaming updates from a data source

### Blocking functions

If you only need to collect points in an Axes you can use *Figure.ginput*. However if you have written some custom event handling or are using *widgets* you will need to manually run the GUI event loop using the methods described *above*.

You can also use the methods described in *Blocking the prompt* to suspend run the GUI event loop. Once the loop exits your code will resume. In general, any place you would use `time.sleep` you can use *pyplot.pause* instead with the added benefit of interactive figures.

For example, if you want to poll for data you could use something like

```
fig, ax = plt.subplots()
ln, = ax.plot([], [])

while True:
    x, y = get_new_data()
    ln.set_data(x, y)
    plt.pause(1)
```

which would poll for new data and update the figure at 1Hz.

### Explicitly spinning the event Loop

| | |
|---|---|
| *backend_bases.FigureCanvasBase.flush_events* | Flush the GUI events for the figure. |
| *backend_bases.FigureCanvasBase.draw_idle* | Request a widget redraw once control returns to the GUI event loop. |

If you have open windows that have pending UI events (mouse clicks, button presses, or draws) you can explicitly process those events by calling *FigureCanvasBase.flush_events*. This will run the GUI event loop until all UI events currently waiting have been processed. The exact behavior is backend-dependent but typically events on all figure are processed and only events waiting to be processed (not those added during processing) will be handled.

For example

```
import time
import matplotlib.pyplot as plt
import numpy as np
```

(continues on next page)

(continued from previous page)

```
plt.ion()

fig, ax = plt.subplots()
th = np.linspace(0, 2*np.pi, 512)
ax.set_ylim(-1.5, 1.5)

ln, = ax.plot(th, np.sin(th))

def slow_loop(N, ln):
    for j in range(N):
        time.sleep(.1)  # to simulate some work
        ln.figure.canvas.flush_events()

slow_loop(100, ln)
```

While this will feel a bit laggy (as we are only processing user input every 100ms whereas 20-30ms is what feels "responsive") it will respond.

If you make changes to the plot and want it re-rendered you will need to call *draw_idle* to request that the canvas be re-drawn. This method can be thought of *draw_soon* in analogy to `asyncio.loop.call_soon`.

We can add this to our example above as

```
def slow_loop(N, ln):
    for j in range(N):
        time.sleep(.1)  # to simulate some work
        if j % 10:
            ln.set_ydata(np.sin(((j // 10) % 5 * th)))
            ln.figure.canvas.draw_idle()

        ln.figure.canvas.flush_events()

slow_loop(100, ln)
```

The more frequently you call *FigureCanvasBase.flush_events* the more responsive your figure will feel but at the cost of spending more resources on the visualization and less on your computation.

### Stale artists

Artists (as of Matplotlib 1.5) have a **stale** attribute which is `True` if the internal state of the artist has changed since the last time it was rendered. By default the stale state is propagated up to the Artists parents in the draw tree, e.g., if the color of a *Line2D* instance is changed, the *Axes* and *Figure* that contain it will also be marked as "stale". Thus, `fig.stale` will report if any artist in the figure has been modified and is out of sync with what is displayed on the screen. This is intended to be used to determine if `draw_idle` should be called to schedule a re-rendering of the figure.

Each artist has a `Artist.stale_callback` attribute which holds a callback with the signature

```
def callback(self: Artist, val: bool) -> None:
    ...
```

which by default is set to a function that forwards the stale state to the artist's parent. If you wish to suppress a given artist from propagating set this attribute to None.

*Figure* instances do not have a containing artist and their default callback is None. If you call *pyplot.ion* and are not in IPython we will install a callback to invoke *draw_idle* whenever the *Figure* becomes stale. In IPython we use the 'post_execute' hook to invoke *draw_idle* on any stale figures after having executed the user's input, but before returning the prompt to the user. If you are not using *pyplot* you can use the callback Figure.stale_callback attribute to be notified when a figure has become stale.

### Idle draw

| | |
|---|---|
| *backend_bases.FigureCanvasBase.draw* | Render the *Figure*. |
| *backend_bases.FigureCanvasBase.draw_idle* | Request a widget redraw once control returns to the GUI event loop. |
| *backend_bases.FigureCanvasBase.flush_events* | Flush the GUI events for the figure. |

In almost all cases, we recommend using *backend_bases.FigureCanvasBase.draw_idle* over *backend_bases.FigureCanvasBase.draw*. draw forces a rendering of the figure whereas draw_idle schedules a rendering the next time the GUI window is going to re-paint the screen. This improves performance by only rendering pixels that will be shown on the screen. If you want to be sure that the screen is updated as soon as possible do

```
fig.canvas.draw_idle()
fig.canvas.flush_events()
```

### Threading

Most GUI frameworks require that all updates to the screen, and hence their main event loop, run on the main thread. This makes pushing periodic updates of a plot to a background thread impossible. Although it seems backwards, it is typically easier to push your computations to a background thread and periodically update the figure on the main thread.

In general Matplotlib is not thread safe. If you are going to update *Artist* objects in one thread and draw from another you should make sure that you are locking in the critical sections.

#### Eventloop integration mechanism

#### CPython / readline

The Python C API provides a hook, `PyOS_InputHook`, to register a function to be run ("The function will be called when Python's interpreter prompt is about to become idle and wait for user input from the terminal."). This hook can be used to integrate a second event loop (the GUI event loop) with the python input prompt loop. The hook functions typically exhaust all pending events on the GUI event queue, run the main loop for a short fixed amount of time, or run the event loop until a key is pressed on stdin.

Matplotlib does not currently do any management of `PyOS_InputHook` due to the wide range of ways that Matplotlib is used. This management is left to downstream libraries -- either user code or the shell. Interactive figures, even with Matplotlib in 'interactive mode', may not work in the vanilla python repl if an appropriate `PyOS_InputHook` is not registered.

Input hooks, and helpers to install them, are usually included with the python bindings for GUI toolkits and may be registered on import. IPython also ships input hook functions for all of the GUI frameworks Matplotlib supports which can be installed via `%matplotlib`. This is the recommended method of integrating Matplotlib and a prompt.

#### IPython / prompt_toolkit

With IPython >= 5.0 IPython has changed from using CPython's readline based prompt to a `prompt_toolkit` based prompt. `prompt_toolkit` has the same conceptual input hook, which is fed into `prompt_toolkit` via the `IPython.terminal.interactiveshell.TerminalInteractiveShell.inputhook()` method. The source for the `prompt_toolkit` input hooks lives at `IPython.terminal.pt_inputhooks`.

### 3.2.6 Event handling and picking

Matplotlib works with a number of user interface toolkits (wxpython, tkinter, qt, gtk, and macOS) and in order to support features like interactive panning and zooming of figures, it is helpful to the developers to have an API for interacting with the figure via key presses and mouse movements that is "GUI neutral" so we don't have to repeat a lot of code across the different user interfaces. Although the event handling API is GUI neutral, it is based on the GTK model, which was the first user interface Matplotlib supported. The events that are triggered are also a bit richer vis-a-vis Matplotlib than standard GUI events, including information like which *Axes* the event occurred in. The events also understand the Matplotlib coordinate system, and report event locations in both pixel and data coordinates.

### Event connections

To receive events, you need to write a callback function and then connect your function to the event manager, which is part of the *FigureCanvasBase*. Here is a simple example that prints the location of the mouse click and which button was pressed:

```
fig, ax = plt.subplots()
ax.plot(np.random.rand(10))

def onclick(event):
    print('%s click: button=%d, x=%d, y=%d, xdata=%f, ydata=%f' %
          ('double' if event.dblclick else 'single', event.button,
           event.x, event.y, event.xdata, event.ydata))

cid = fig.canvas.mpl_connect('button_press_event', onclick)
```

The *FigureCanvasBase.mpl_connect* method returns a connection id (an integer), which can be used to disconnect the callback via

```
fig.canvas.mpl_disconnect(cid)
```

---

**Note:** The canvas retains only weak references to instance methods used as callbacks. Therefore, you need to retain a reference to instances owning such methods. Otherwise the instance will be garbage-collected and the callback will vanish.

This does not affect free functions used as callbacks.

---

Here are the events that you can connect to, the class instances that are sent back to you when the event occurs, and the event descriptions:

| Event name | Class | Description |
|---|---|---|
| 'button_press_event' | *MouseEvent* | mouse button is pressed |
| 'button_release_event' | *MouseEvent* | mouse button is released |
| 'close_event' | *CloseEvent* | figure is closed |
| 'draw_event' | *DrawEvent* | canvas has been drawn (but screen widget not updated yet) |
| 'key_press_event' | *KeyEvent* | key is pressed |
| 'key_release_event' | *KeyEvent* | key is released |
| 'motion_notify_event' | *MouseEvent* | mouse moves |
| 'pick_event' | *PickEvent* | artist in the canvas is selected |
| 'resize_event' | *ResizeEvent* | figure canvas is resized |
| 'scroll_event' | *MouseEvent* | mouse scroll wheel is rolled |
| 'figure_enter_event' | *LocationEvent* | mouse enters a new figure |
| 'figure_leave_event' | *LocationEvent* | mouse leaves a figure |
| 'axes_enter_event' | *LocationEvent* | mouse enters a new axes |
| 'axes_leave_event' | *LocationEvent* | mouse leaves an axes |

**Note:** When connecting to 'key_press_event' and 'key_release_event' events, you may encounter inconsistencies between the different user interface toolkits that Matplotlib works with. This is due to inconsistencies/limitations of the user interface toolkit. The following table shows some basic examples of what you may expect to receive as key(s) (using a QWERTY keyboard layout) from the different user interface toolkits, where a comma separates different keys:

| Key(s) Pressed | Tkinter | Qt | macosx | WebAgg | GTK | WxPython |
|---|---|---|---|---|---|---|
| `Shift+2` | shift, @ | shift, @ | shift, @ | shift, @ | shift, @ | shift, shift+2 |
| `Shift+F1` | shift, shift+f1 | shift, shift+f1 | shift, shift+f1 | shift, shift+f1 | shift, shift+f1 | shift, shift+f1 |
| `Shift` | shift | shift | shift | shift | shift | shift |
| `Control` | control | control | control | control | control | control |
| `Alt` | alt | alt | alt | alt | alt | alt |
| `AltGr` | iso_level3_shi | *nothing* | | alt | iso_level3_shi | *nothing* |
| `CapsLock` | caps_lock | caps_lock | caps_lock | caps_lock | caps_lock | caps_lock |
| `CapsLock+a` | caps_lock, A | caps_lock, a | caps_lock, a | caps_lock, A | caps_lock, A | caps_lock, a |
| `a` | a | a | a | a | a | a |
| `Shift+a` | shift, A | shift, A | shift, A | shift, A | shift, A | shift, A |
| `CapsLock+Sh` | caps_lock, shift, a | caps_lock, shift, A | caps_lock, shift, A | caps_lock, shift, a | caps_lock, shift, a | caps_lock, shift, A |
| `Ctrl+Shi` | control, ctrl+shift, ctrl+meta | control, ctrl+shift, ctrl+meta | control, ctrl+shift, ctrl+alt+shift | control, ctrl+shift, ctrl+meta | control, ctrl+shift, ctrl+meta | control, ctrl+shift, ctrl+alt |
| `Ctrl+Shi` | control, ctrl+shift, ctrl+a | control, ctrl+shift, ctrl+A | control, ctrl+shift, ctrl+A | control, ctrl+shift, ctrl+A | control, ctrl+shift, ctrl+A | control, ctrl+shift, ctrl+A |
| `F1` | f1 | f1 | f1 | f1 | f1 | f1 |
| `Ctrl+F1` | control, ctrl+f1 | control, ctrl+f1 | control, *nothing* | control, ctrl+f1 | control, ctrl+f1 | control, ctrl+f1 |

Matplotlib attaches some keypress callbacks by default for interactivity; they are documented in the *Navigation keyboard shortcuts* section.

### Event attributes

All Matplotlib events inherit from the base class `matplotlib.backend_bases.Event`, which stores the attributes:

**`name`**

the event name

**`canvas`**

the FigureCanvas instance generating the event

**`guiEvent`**

the GUI event that triggered the Matplotlib event

The most common events that are the bread and butter of event handling are key press/release events and mouse press/release and movement events. The *KeyEvent* and *MouseEvent* classes that handle these events are both derived from the LocationEvent, which has the following attributes

**x, y**

mouse x and y position in pixels from left and bottom of canvas

**inaxes**

the *Axes* instance over which the mouse is, if any; else None

**xdata, ydata**

mouse x and y position in data coordinates, if the mouse is over an axes

Let's look a simple example of a canvas, where a simple line segment is created every time a mouse is pressed:

```
from matplotlib import pyplot as plt

class LineBuilder:
    def __init__(self, line):
        self.line = line
        self.xs = list(line.get_xdata())
        self.ys = list(line.get_ydata())
        self.cid = line.figure.canvas.mpl_connect('button_press_event', self)

    def __call__(self, event):
        print('click', event)
        if event.inaxes!=self.line.axes: return
        self.xs.append(event.xdata)
        self.ys.append(event.ydata)
        self.line.set_data(self.xs, self.ys)
        self.line.figure.canvas.draw()

fig, ax = plt.subplots()
ax.set_title('click to build line segments')
line, = ax.plot([0], [0])  # empty line
linebuilder = LineBuilder(line)

plt.show()
```

The *MouseEvent* that we just used is a *LocationEvent*, so we have access to the data and pixel coordinates via `(event.x, event.y)` and `(event.xdata, event.ydata)`. In addition to the `LocationEvent` attributes, it also has:

**button**

the button pressed: None, *MouseButton*, 'up', or 'down' (up and down are used for scroll events)

**key**

the key pressed: None, any character, 'shift', 'win', or 'control'

### Draggable rectangle exercise

Write draggable rectangle class that is initialized with a *Rectangle* instance but will move its xy location when dragged. Hint: you will need to store the original xy location of the rectangle which is stored as rect.xy and connect to the press, motion and release mouse events. When the mouse is pressed, check to see if the click occurs over your rectangle (see Rectangle.contains) and if it does, store the rectangle xy and the location of the mouse click in data coords. In the motion event callback, compute the deltax and deltay of the mouse movement, and add those deltas to the origin of the rectangle you stored. The redraw the figure. On the button release event, just reset all the button press data you stored as None.

Here is the solution:

```
import numpy as np
import matplotlib.pyplot as plt

class DraggableRectangle:
    def __init__(self, rect):
        self.rect = rect
        self.press = None

    def connect(self):
        """Connect to all the events we need."""
        self.cidpress = self.rect.figure.canvas.mpl_connect(
            'button_press_event', self.on_press)
        self.cidrelease = self.rect.figure.canvas.mpl_connect(
            'button_release_event', self.on_release)
        self.cidmotion = self.rect.figure.canvas.mpl_connect(
            'motion_notify_event', self.on_motion)

    def on_press(self, event):
        """Check whether mouse is over us; if so, store some data."""
        if event.inaxes != self.rect.axes:
            return
        contains, attrd = self.rect.contains(event)
        if not contains:
            return
        print('event contains', self.rect.xy)
        self.press = self.rect.xy, (event.xdata, event.ydata)

    def on_motion(self, event):
        """Move the rectangle if the mouse is over us."""
        if self.press is None or event.inaxes != self.rect.axes:
            return
        (x0, y0), (xpress, ypress) = self.press
        dx = event.xdata - xpress
        dy = event.ydata - ypress
        # print(f'x0={x0}, xpress={xpress}, event.xdata={event.xdata}, '
        #       f'dx={dx}, x0+dx={x0+dx}')
        self.rect.set_x(x0+dx)
        self.rect.set_y(y0+dy)

        self.rect.figure.canvas.draw()
```

(continues on next page)

(continued from previous page)

```
    def on_release(self, event):
        """Clear button press information."""
        self.press = None
        self.rect.figure.canvas.draw()

    def disconnect(self):
        """Disconnect all callbacks."""
        self.rect.figure.canvas.mpl_disconnect(self.cidpress)
        self.rect.figure.canvas.mpl_disconnect(self.cidrelease)
        self.rect.figure.canvas.mpl_disconnect(self.cidmotion)

fig, ax = plt.subplots()
rects = ax.bar(range(10), 20*np.random.rand(10))
drs = []
for rect in rects:
    dr = DraggableRectangle(rect)
    dr.connect()
    drs.append(dr)

plt.show()
```

**Extra credit**: Use blitting to make the animated drawing faster and smoother.

Extra credit solution:

```
# Draggable rectangle with blitting.
import numpy as np
import matplotlib.pyplot as plt

class DraggableRectangle:
    lock = None  # only one can be animated at a time

    def __init__(self, rect):
        self.rect = rect
        self.press = None
        self.background = None

    def connect(self):
        """Connect to all the events we need."""
        self.cidpress = self.rect.figure.canvas.mpl_connect(
            'button_press_event', self.on_press)
        self.cidrelease = self.rect.figure.canvas.mpl_connect(
            'button_release_event', self.on_release)
        self.cidmotion = self.rect.figure.canvas.mpl_connect(
            'motion_notify_event', self.on_motion)

    def on_press(self, event):
        """Check whether mouse is over us; if so, store some data."""
        if (event.inaxes != self.rect.axes
                or DraggableRectangle.lock is not None):
            return
```

(continues on next page)

(continued from previous page)

```
        contains, attrd = self.rect.contains(event)
        if not contains:
            return
        print('event contains', self.rect.xy)
        self.press = self.rect.xy, (event.xdata, event.ydata)
        DraggableRectangle.lock = self

        # draw everything but the selected rectangle and store the pixel␣
↪buffer
        canvas = self.rect.figure.canvas
        axes = self.rect.axes
        self.rect.set_animated(True)
        canvas.draw()
        self.background = canvas.copy_from_bbox(self.rect.axes.bbox)

        # now redraw just the rectangle
        axes.draw_artist(self.rect)

        # and blit just the redrawn area
        canvas.blit(axes.bbox)

    def on_motion(self, event):
        """Move the rectangle if the mouse is over us."""
        if (event.inaxes != self.rect.axes
                or DraggableRectangle.lock is not self):
            return
        (x0, y0), (xpress, ypress) = self.press
        dx = event.xdata - xpress
        dy = event.ydata - ypress
        self.rect.set_x(x0+dx)
        self.rect.set_y(y0+dy)

        canvas = self.rect.figure.canvas
        axes = self.rect.axes
        # restore the background region
        canvas.restore_region(self.background)

        # redraw just the current rectangle
        axes.draw_artist(self.rect)

        # blit just the redrawn area
        canvas.blit(axes.bbox)

    def on_release(self, event):
        """Clear button press information."""
        if DraggableRectangle.lock is not self:
            return

        self.press = None
        DraggableRectangle.lock = None

        # turn off the rect animation property and reset the background
```

(continues on next page)

(continued from previous page)

```
        self.rect.set_animated(False)
        self.background = None

        # redraw the full figure
        self.rect.figure.canvas.draw()

    def disconnect(self):
        """Disconnect all callbacks."""
        self.rect.figure.canvas.mpl_disconnect(self.cidpress)
        self.rect.figure.canvas.mpl_disconnect(self.cidrelease)
        self.rect.figure.canvas.mpl_disconnect(self.cidmotion)

fig, ax = plt.subplots()
rects = ax.bar(range(10), 20*np.random.rand(10))
drs = []
for rect in rects:
    dr = DraggableRectangle(rect)
    dr.connect()
    drs.append(dr)

plt.show()
```

### Mouse enter and leave

If you want to be notified when the mouse enters or leaves a figure or axes, you can connect to the figure/axes enter/leave events. Here is a simple example that changes the colors of the axes and figure background that the mouse is over:

```
"""
Illustrate the figure and axes enter and leave events by changing the
frame colors on enter and leave
"""
import matplotlib.pyplot as plt

def enter_axes(event):
    print('enter_axes', event.inaxes)
    event.inaxes.patch.set_facecolor('yellow')
    event.canvas.draw()

def leave_axes(event):
    print('leave_axes', event.inaxes)
    event.inaxes.patch.set_facecolor('white')
    event.canvas.draw()

def enter_figure(event):
    print('enter_figure', event.canvas.figure)
    event.canvas.figure.patch.set_facecolor('red')
    event.canvas.draw()

def leave_figure(event):
```

(continues on next page)

(continued from previous page)

```
    print('leave_figure', event.canvas.figure)
    event.canvas.figure.patch.set_facecolor('grey')
    event.canvas.draw()

fig1, axs = plt.subplots(2)
fig1.suptitle('mouse hover over figure or axes to trigger events')

fig1.canvas.mpl_connect('figure_enter_event', enter_figure)
fig1.canvas.mpl_connect('figure_leave_event', leave_figure)
fig1.canvas.mpl_connect('axes_enter_event', enter_axes)
fig1.canvas.mpl_connect('axes_leave_event', leave_axes)

fig2, axs = plt.subplots(2)
fig2.suptitle('mouse hover over figure or axes to trigger events')

fig2.canvas.mpl_connect('figure_enter_event', enter_figure)
fig2.canvas.mpl_connect('figure_leave_event', leave_figure)
fig2.canvas.mpl_connect('axes_enter_event', enter_axes)
fig2.canvas.mpl_connect('axes_leave_event', leave_axes)

plt.show()
```

### Object picking

You can enable picking by setting the `picker` property of an *`Artist`* (such as *`Line2D`*, *`Text`*, *`Patch`*, *`Polygon`*, *`AxesImage`*, etc.)

The `picker` property can be set using various types:

**`None`**

Picking is disabled for this artist (default).

**`boolean`**

If True, then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist.

**`callable`**

If picker is a callable, it is a user supplied function which determines whether the artist is hit by the mouse event. The signature is `hit, props = picker(artist, mouseevent)` to determine the hit test. If the mouse event is over the artist, return `hit = True`; `props` is a dictionary of properties that become additional attributes on the *`PickEvent`*.

The artist's `pickradius` property can additionally be set to a tolerance value in points (there are 72 points per inch) that determines how far the mouse can be and still trigger a mouse event.

After you have enabled an artist for picking by setting the `picker` property, you need to connect a handler to the figure canvas pick_event to get pick callbacks on mouse press events. The handler typically looks like

```
def pick_handler(event):
    mouseevent = event.mouseevent
    artist = event.artist
    # now do something with this...
```

The *PickEvent* passed to your callback always has the following attributes:

**mouseevent**

The *MouseEvent* that generate the pick event. See *event-attributes* for a list of useful attributes on the mouse event.

**artist**

The *Artist* that generated the pick event.

Additionally, certain artists like *Line2D* and *PatchCollection* may attach additional metadata, like the indices of the data that meet the picker criteria (e.g., all the points in the line that are within the specified `pickradius` tolerance).

### Simple picking example

In the example below, we enable picking on the line and set a pick radius tolerance in points. The `onpick` callback function will be called when the pick event it within the tolerance distance from the line, and has the indices of the data vertices that are within the pick distance tolerance. Our `onpick` callback function simply prints the data that are under the pick location. Different Matplotlib Artists can attach different data to the PickEvent. For example, `Line2D` attaches the ind property, which are the indices into the line data under the pick point. See `Line2D.pick` for details on the `PickEvent` properties of the line.

```
import numpy as np
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
ax.set_title('click on points')

line, = ax.plot(np.random.rand(100), 'o',
                picker=True, pickradius=5)  # 5 points tolerance

def onpick(event):
    thisline = event.artist
    xdata = thisline.get_xdata()
    ydata = thisline.get_ydata()
    ind = event.ind
    points = tuple(zip(xdata[ind], ydata[ind]))
    print('onpick points:', points)

fig.canvas.mpl_connect('pick_event', onpick)

plt.show()
```

### Picking exercise

Create a data set of 100 arrays of 1000 Gaussian random numbers and compute the sample mean and standard deviation of each of them (hint: NumPy arrays have a mean and std method) and make a xy marker plot of the 100 means vs. the 100 standard deviations. Connect the line created by the plot command to the pick event, and plot the original time series of the data that generated the clicked on points. If more than one point is within the tolerance of the clicked on point, you can use multiple subplots to plot the multiple time series.

Exercise solution:

```
"""
Compute the mean and stddev of 100 data sets and plot mean vs. stddev.
When you click on one of the (mean, stddev) points, plot the raw dataset
that generated that point.
"""

import numpy as np
import matplotlib.pyplot as plt

X = np.random.rand(100, 1000)
xs = np.mean(X, axis=1)
ys = np.std(X, axis=1)

fig, ax = plt.subplots()
ax.set_title('click on point to plot time series')
line, = ax.plot(xs, ys, 'o', picker=True, pickradius=5)  # 5 points tolerance


def onpick(event):
    if event.artist != line:
        return
    n = len(event.ind)
    if not n:
        return
    fig, axs = plt.subplots(n, squeeze=False)
    for dataind, ax in zip(event.ind, axs.flat):
        ax.plot(X[dataind])
        ax.text(0.05, 0.9,
                f"$\\mu$={xs[dataind]:1.3f}\n$\\sigma$={ys[dataind]:1.3f}",
                transform=ax.transAxes, verticalalignment='top')
        ax.set_ylim(-0.5, 1.5)
    fig.show()
    return True


fig.canvas.mpl_connect('pick_event', onpick)
plt.show()
```

### 3.2.7 Writing a backend -- the pyplot interface

This page assumes general understanding of the information in the *Backends* page, and is instead intended as reference for third-party backend implementers. It also only deals with the interaction between backends and *pyplot*, not with the rendering side, which is described in *backend_template*.

There are two APIs for defining backends: a new canvas-based API (introduced in Matplotlib 3.6), and an older function-based API. The new API is simpler to implement because many methods can be inherited from "parent backends". It is recommended if back-compatibility for Matplotlib < 3.6 is not a concern. However, the old API remains supported.

Fundamentally, a backend module needs to provide information to *pyplot*, so that

1. *pyplot.figure()* can create a new *Figure* instance and associate it with an instance of a backend-provided canvas class, itself hosted in an instance of a backend-provided manager class.
2. *pyplot.show()* can show all figures and start the GUI event loop (if any).

To do so, the backend module must define a `backend_module.FigureCanvas` subclass of *FigureCanvasBase*. In the canvas-based API, this is the only strict requirement for backend modules. The function-based API additionally requires many module-level functions to be defined.

#### Canvas-based API (Matplotlib >= 3.6)

1. **Creating a figure**: *pyplot.figure()* calls `figure = Figure(); FigureCanvas.new_manager(figure, num)` (`new_manager` is a classmethod) to instantiate a canvas and a manager and set up the `figure.canvas` and `figure.canvas.manager` attributes. Figure unpickling uses the same approach, but replaces the newly instantiated `Figure()` by the unpickled figure.

   Interactive backends should customize the effect of `new_manager` by setting the `FigureCanvas.manager_class` attribute to the desired manager class, and additionally (if the canvas cannot be created before the manager, as in the case of the wx backends) by overriding the `FigureManager.create_with_canvas` classmethod. (Non-interactive backends can normally use a trivial `FigureManagerBase` and can therefore skip this step.)

   After a new figure is registered with *pyplot* (either via *pyplot.figure()* or via unpickling), if in interactive mode, *pyplot* will call its canvas' `draw_idle()` method, which can be overridden as desired.

2. **Showing figures**: *pyplot.show()* calls `FigureCanvas.manager_class.pyplot_show()` (a classmethod), forwarding any arguments, to start the main event loop.

   By default, `pyplot_show()` checks whether there are any `managers` registered with *pyplot* (exiting early if not), calls `manager.show()` on all such managers, and then, if called with `block=True` (or with the default `block=None` and out of IPython's pylab mode and not in interactive mode), calls `FigureCanvas.manager_class.start_main_loop()` (a classmethod) to start the main event loop. Interactive backends should therefore override the `FigureCanvas.manager_class.start_main_loop` classmethod accordingly (or alternatively, they may also directly override `FigureCanvas.manager_class.pyplot_show` directly).

### Function-based API

1. **Creating a figure**: *pyplot.figure()* calls `new_figure_manager(num, *args, **kwargs)` (which also takes care of creating the new figure as `Figure(*args, **kwargs)`); unpickling calls `new_figure_manager_given_figure(num, figure)`.

   Furthermore, in interactive mode, the first draw of the newly registered figure can be customized by providing a module-level `draw_if_interactive()` function. (In the new canvas-based API, this function is not taken into account anymore.)

2. **Showing figures**: *pyplot.show()* calls a module-level `show()` function, which is typically generated via the `ShowBase` class and its `mainloop` method.

### Registering a backend

For a new backend to be usable via `matplotlib.use()` or IPython `%matplotlib` magic command, it must be compatible with one of the three ways supported by the *BackendRegistry*:

#### Built-in

A backend built into Matplotlib must have its name and `FigureCanvas.required_interactive_framework` hard-coded in the *BackendRegistry*. If the backend module is not `f"matplotlib.backends.backend_{backend_name.lower()}"` then there must also be an entry in the `BackendRegistry._name_to_module`.

#### module:// syntax

Any backend in a separate module (not built into Matplotlib) can be used by specifying the path to the module in the form `module://some.backend.module`. An example is `module://mplcairo.qt` for mplcairo. The backend's interactive framework will be taken from its `FigureCanvas.required_interactive_framework`.

#### Entry point

An external backend module can self-register as a backend using an `entry point` in its `pyproject.toml` such as the one used by `matplotlib-inline`:

```
[project.entry-points."matplotlib.backend"]
inline = "matplotlib_inline.backend_inline"
```

The backend's interactive framework will be taken from its `FigureCanvas.required_interactive_framework`. All entry points are loaded together but only when first needed, such as when a backend name is not recognised as a built-in backend, or when *list_all()* is first called.

## 3.3 Axes and subplots

Matplotlib *Axes* are the gateway to creating your data visualizations. Once an Axes is placed on a figure there are many methods that can be used to add data to the Axes. An Axes typically has a pair of *Axis* Artists that define the data coordinate system, and include methods to add annotations like x- and y-labels, titles, and legends.

### 3.3.1 Introduction to Axes (or Subplots)

Matplotlib *Axes* are the gateway to creating your data visualizations. Once an Axes is placed on a figure there are many methods that can be used to add data to the Axes. An Axes typically has a pair of *Axis* Artists that define the data coordinate system, and include methods to add annotations like x- and y-labels, titles, and legends.

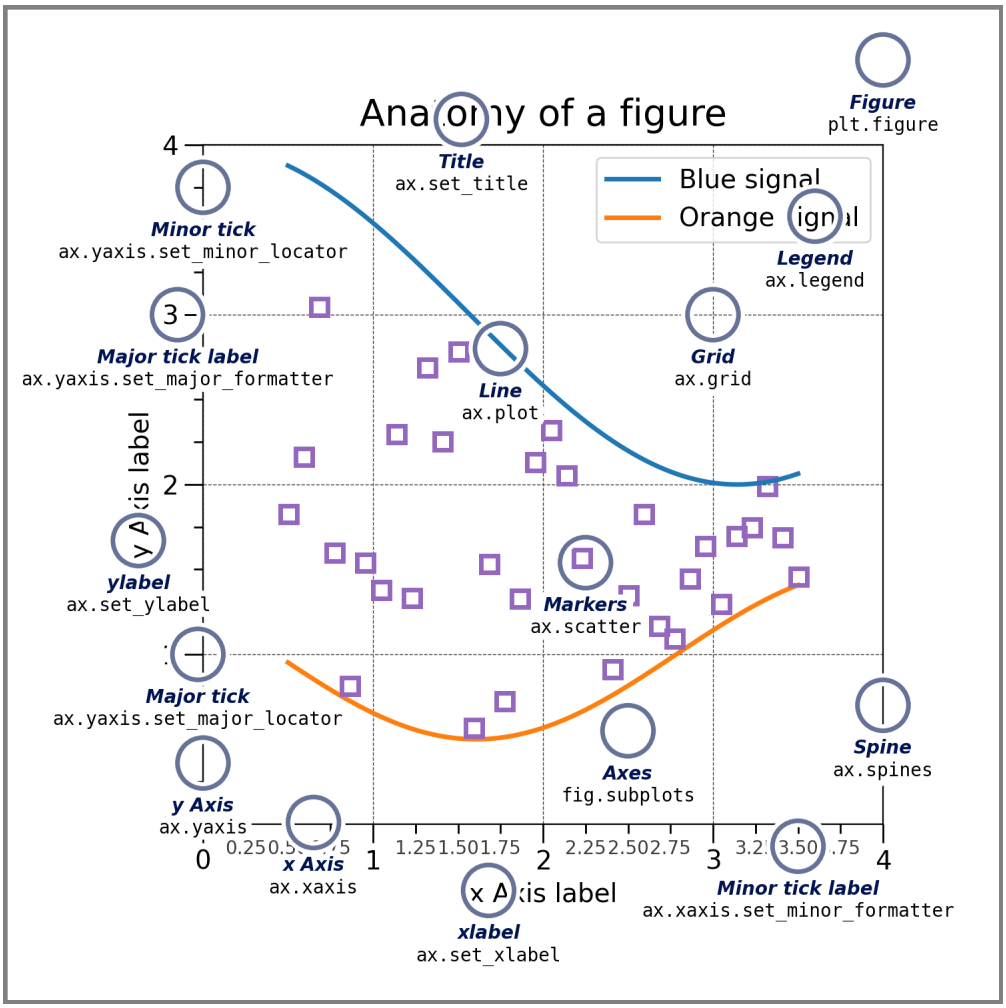



Fig. 4: Anatomy of a Figure

In the picture above, the Axes object was created with `ax = fig.subplots()`. Everything else on

the figure was created with methods on this `ax` object, or can be accessed from it. If we want to change the label on the x-axis, we call `ax.set_xlabel('New Label')`, if we want to plot some data we call `ax.plot(x, y)`. Indeed, in the figure above, the only Artist that is not part of the Axes is the Figure itself, so the *axes.Axes* class is really the gateway to much of Matplotlib's functionality.

Note that Axes are so fundamental to the operation of Matplotlib that a lot of material here is duplicate of that in *Quick start guide*.

### Creating Axes

```
import matplotlib.pyplot as plt
import numpy as np

fig, axs = plt.subplots(ncols=2, nrows=2, figsize=(3.5, 2.5),
                        layout="constrained")
# for each Axes, add an artist, in this case a nice label in the middle...
for row in range(2):
    for col in range(2):
        axs[row, col].annotate(f'axs[{row}, {col}]', (0.5, 0.5),
                               transform=axs[row, col].transAxes,
                               ha='center', va='center', fontsize=18,
                               color='darkgrey')
fig.suptitle('plt.subplots()')
```

Axes are added using methods on *Figure* objects, or via the *pyplot* interface. These methods are discussed in more detail in *Creating Figures* and *Arranging multiple Axes in a Figure*. However, for instance *add_axes* will manually position an Axes on the page. In the example above *subplots* put a grid of subplots on the figure, and `axs` is a (2, 2) array of Axes, each of which can have data added to them.

There are a number of other methods for adding Axes to a Figure:

- *Figure.add_axes*: manually position an Axes. `fig.add_axes([0, 0, 1, 1])` makes an Axes that fills the whole figure.
- *pyplot.subplots* and *Figure.subplots*: add a grid of Axes as in the example above. The pyplot version returns both the Figure object and an array of Axes. Note that `fig, ax = plt.subplots()` adds a single Axes to a Figure.
- *pyplot.subplot_mosaic* and *Figure.subplot_mosaic*: add a grid of named Axes and return a dictionary of axes. For `fig, axs = plt.subplot_mosaic([['left', 'right'], ['bottom', 'bottom']])`, `axs['left']` is an Axes in the top row on the left, and `axs['bottom']` is an Axes that spans both columns on the bottom.

See *Arranging multiple Axes in a Figure* for more detail on how to arrange grids of Axes on a Figure.

### Axes plotting methods

Most of the high-level plotting methods are accessed from the *axes.Axes* class. See the API documentation for a full curated list, and *Plot types* for examples. A basic example is *axes.Axes.plot*:

```
fig, ax = plt.subplots(figsize=(4, 3))
np.random.seed(19680801)
t = np.arange(100)
x = np.cumsum(np.random.randn(100))
lines = ax.plot(t, x)
```

Note that `plot` returns a list of *lines* Artists which can subsequently be manipulated, as discussed in *Introduction to Artists*.

A very incomplete list of plotting methods is below. Again, see *Plot types* for more examples, and *axes.Axes* for the full list of methods.

| | |
|---|---|
| *Pairwise data* | *plot*, *scatter*, *bar*, *step*, |
| Array objects | *pcolormesh*, *contour*, *quiver*, *streamplot*, *imshow* |
| *Statistical distributions* | *hist*, *errorbar*, *hist2d*, *pie*, *boxplot*, *violinplot* |
| *Irregularly gridded data* | *tricontour*, *tripcolor* |

### Axes labelling and annotation

Usually we want to label the Axes with an xlabel, ylabel, and title, and often we want to have a legend to differentiate plot elements. The *Axes* class has a number of methods to create these annotations.

```
fig, ax = plt.subplots(figsize=(5, 3), layout='constrained')
np.random.seed(19680801)
t = np.arange(200)
x = np.cumsum(np.random.randn(200))
y = np.cumsum(np.random.randn(200))
linesx = ax.plot(t, x, label='Random walk x')
linesy = ax.plot(t, y, label='Random walk y')

ax.set_xlabel('Time [s]')
ax.set_ylabel('Distance [km]')
ax.set_title('Random walk example')
ax.legend()
```

These methods are relatively straight-forward, though there are a number of *Text properties and layout* that can be set on the text objects, like *fontsize*, *fontname*, *horizontalalignment*. Legends can be much more complicated; see *Legend guide* for more details.

Note that text can also be added to axes using *text*, and *annotate*. This can be quite sophisticated: see *Text properties and layout* and *Annotations* for more information.

### Axes limits, scales, and ticking

Each Axes has two (or more) *Axis* objects, that can be accessed via `xaxis` and `yaxis` properties. These have substantial number of methods on them, and for highly customizable Axis-es it is useful to read the API at *Axis*. However, the Axes class offers a number of helpers for the most common of these methods. Indeed, the *set_xlabel*, discussed above, is a helper for the *set_label_text*.

Other important methods set the extent on the axes (*set_xlim*, *set_ylim*), or more fundamentally the scale of the axes. So for instance, we can make an Axis have a logarithmic scale, and zoom in on a sub-portion of the data:

```
fig, ax = plt.subplots(figsize=(4, 2.5), layout='constrained')
np.random.seed(19680801)
t = np.arange(200)
x = 2**np.cumsum(np.random.randn(200))
linesx = ax.plot(t, x)
ax.set_yscale('log')
ax.set_xlim([20, 180])
```

The Axes class also has helpers to deal with Axis ticks and their labels. Most straight-forward is *set_xticks* and *set_yticks* which manually set the tick locations and optionally their labels. Minor ticks can be toggled with *minorticks_on* or *minorticks_off*.

Many aspects of Axes ticks and tick labeling can be adjusted using *tick_params*. For instance, to label the top of the axes instead of the bottom,color the ticks red, and color the ticklabels green:

```
fig, ax = plt.subplots(figsize=(4, 2.5))
ax.plot(np.arange(10))
ax.tick_params(top=True, labeltop=True, color='red', axis='x',
               labelcolor='green')
```

More fine-grained control on ticks, setting scales, and controlling the Axis can be highly customized beyond these Axes-level helpers.

### Axes layout

Sometimes it is important to set the aspect ratio of a plot in data space, which we can do with *set_aspect*:

```
fig, axs = plt.subplots(ncols=2, figsize=(7, 2.5), layout='constrained')
np.random.seed(19680801)
t = np.arange(200)
x = np.cumsum(np.random.randn(200))
axs[0].plot(t, x)
axs[0].set_title('aspect="auto"')

axs[1].plot(t, x)
axs[1].set_aspect(3)
axs[1].set_title('aspect=3')
```

### 3.3.2 Arranging multiple Axes in a Figure

Often more than one Axes is wanted on a figure at a time, usually organized into a regular grid. Matplotlib has a variety of tools for working with grids of Axes that have evolved over the history of the library. Here we will discuss the tools we think users should use most often, the tools that underpin how Axes are organized, and mention some of the older tools.

---

**Note:** Matplotlib uses *Axes* to refer to the drawing area that contains data, x- and y-axis, ticks, labels, title, etc. See *Parts of a Figure* for more details. Another term that is often used is "subplot", which refers to an Axes that is in a grid with other Axes objects.

---

#### Overview

#### Create grid-shaped combinations of Axes

***subplots***

The primary function used to create figures and a grid of Axes. It creates and places all Axes on the figure at once, and returns an object array with handles for the Axes in the grid. See `Figure.subplots`.

or

***subplot_mosaic***

A simple way to create figures and a grid of Axes, with the added flexibility that Axes can also span rows or columns. The Axes are returned in a labelled dictionary instead of an array. See also `Figure.subplot_mosaic` and *Complex and semantic figure composition (subplot_mosaic)*.

Sometimes it is natural to have more than one distinct group of Axes grids, in which case Matplotlib has the concept of `SubFigure`:

***SubFigure***

A virtual figure within a figure.

#### Underlying tools

Underlying these are the concept of a `GridSpec` and a `SubplotSpec`:

***GridSpec***

Specifies the geometry of the grid that a subplot will be placed. The number of rows and number of columns of the grid need to be set. Optionally, the subplot layout parameters (e.g., left, right, etc.) can be tuned.

***SubplotSpec***

Specifies the location of the subplot in the given *GridSpec*.

### Adding single Axes at a time

The above functions create all Axes in a single function call. It is also possible to add Axes one at a time, and this was originally how Matplotlib used to work. Doing so is generally less elegant and flexible, though sometimes useful for interactive work or to place an Axes in a custom location:

***add_axes***

Adds a single Axes at a location specified by `[left, bottom, width, height]` in fractions of figure width or height.

***subplot* or *Figure.add_subplot***

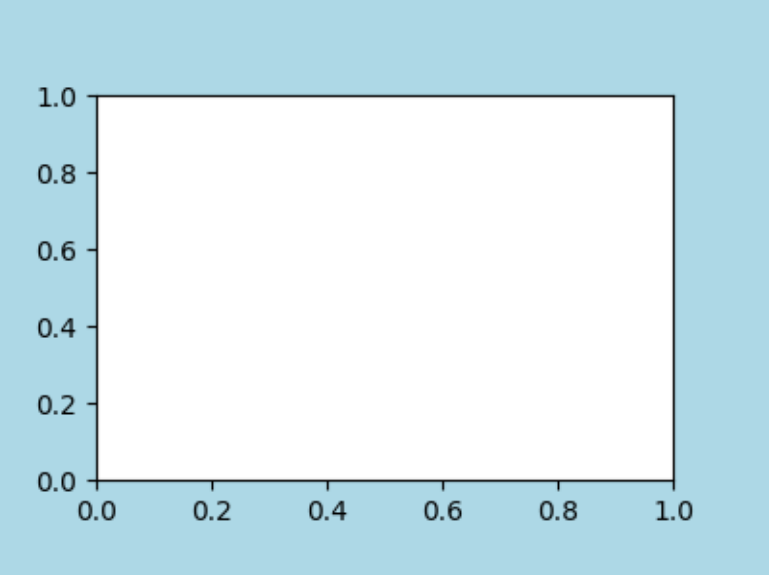
Adds a single subplot on a figure, with 1-based indexing (inherited from Matlab). Columns and rows can be spanned by specifying a range of grid cells.

***subplot2grid***

Similar to *pyplot.subplot*, but uses 0-based indexing and two-d python slicing to choose cells.

As a simple example of manually adding an Axes *ax*, lets add a 3 inch x 2 inch Axes to a 4 inch x 3 inch figure. Note that the location of the subplot is defined as [left, bottom, width, height] in figure-normalized units:

```
import matplotlib.pyplot as plt
import numpy as np

w, h = 4, 3
margin = 0.5
fig = plt.figure(figsize=(w, h), facecolor='lightblue')
ax = fig.add_axes([margin / w, margin / h, (w - 2 * margin) / w,
                   (h - 2 * margin) / h])
```

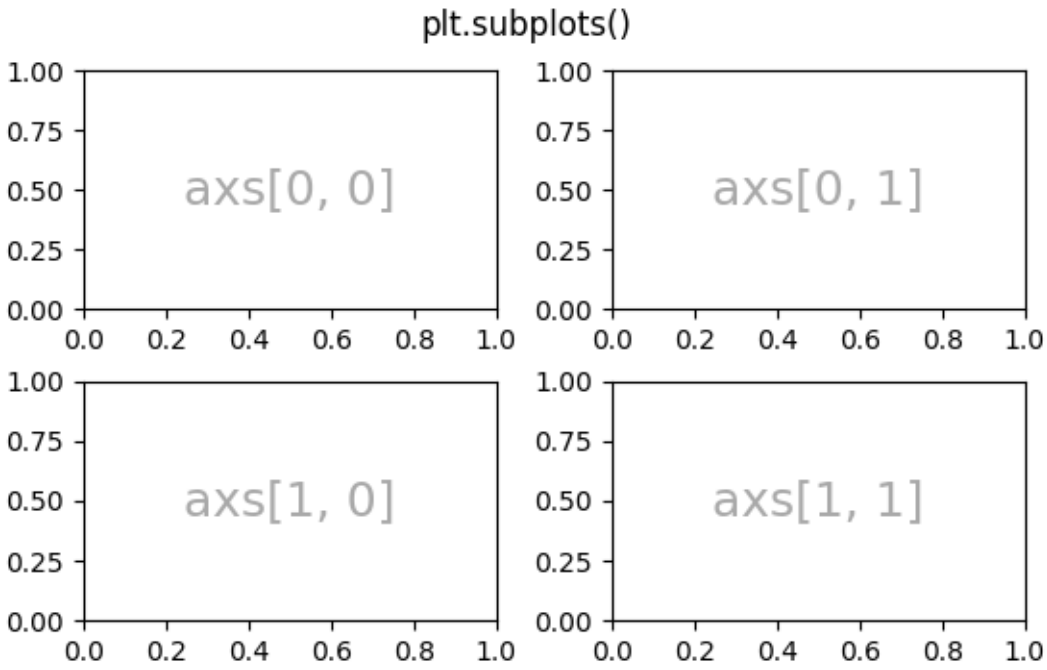



### High-level methods for making grids

#### Basic 2x2 grid

We can create a basic 2-by-2 grid of Axes using *subplots*. It returns a *Figure* instance and an array of *Axes* objects. The Axes objects can be used to access methods to place artists on the Axes; here we use *annotate*, but other examples could be *plot*, *pcolormesh*, etc.

```
fig, axs = plt.subplots(ncols=2, nrows=2, figsize=(5.5, 3.5),
                        layout="constrained")
# add an artist, in this case a nice label in the middle...
for row in range(2):
    for col in range(2):
        axs[row, col].annotate(f'axs[{row}, {col}]', (0.5, 0.5),
                               transform=axs[row, col].transAxes,
                               ha='center', va='center', fontsize=18,
                               color='darkgrey')
fig.suptitle('plt.subplots()')
```

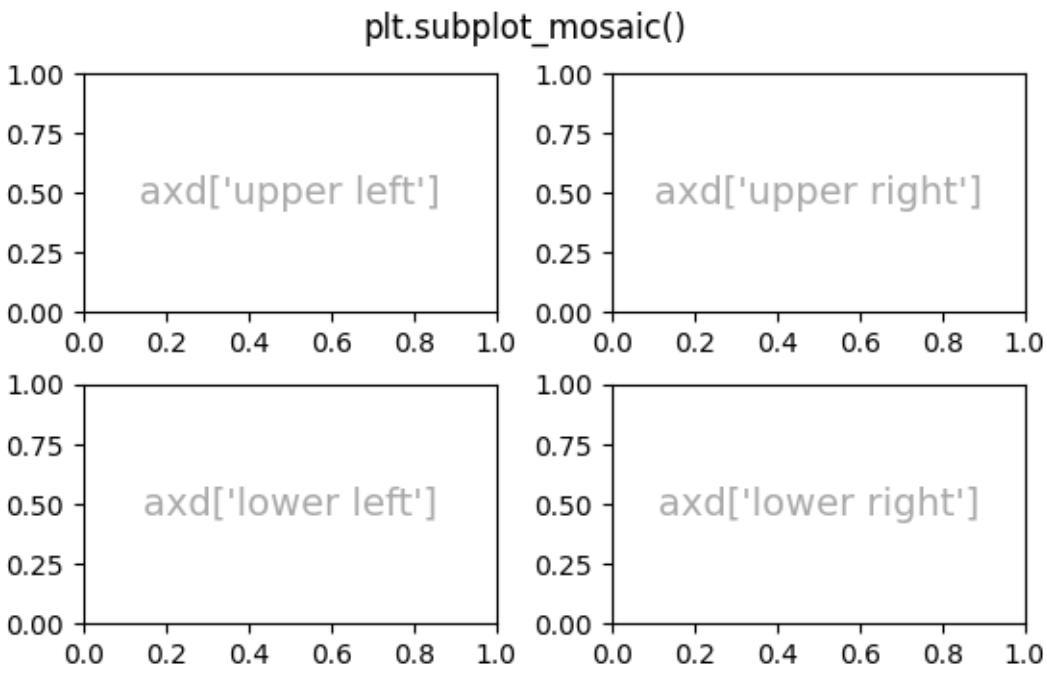



We will annotate a lot of Axes, so let's encapsulate the annotation, rather than having that large piece of annotation code every time we need it:

```
def annotate_axes(ax, text, fontsize=18):
    ax.text(0.5, 0.5, text, transform=ax.transAxes,
            ha="center", va="center", fontsize=fontsize, color="darkgrey")
```

The same effect can be achieved with *subplot_mosaic*, but the return type is a dictionary instead of an array, where the user can give the keys useful meanings. Here we provide two lists, each list representing a row, and each element in the list a key representing the column.

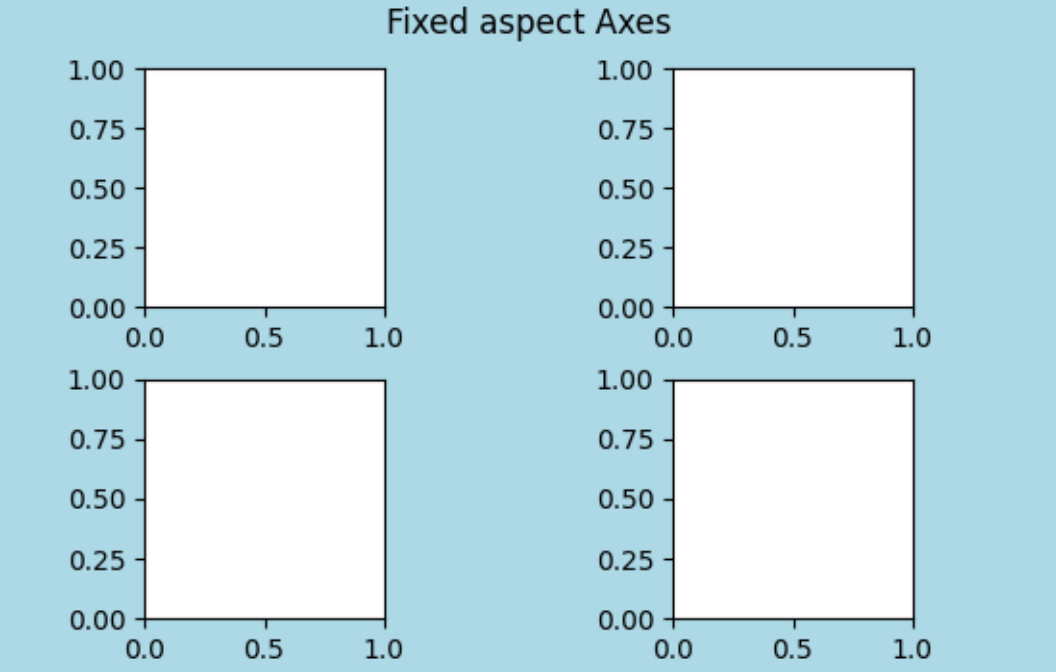
```
fig, axd = plt.subplot_mosaic([['upper left', 'upper right'],
                               ['lower left', 'lower right']],
                              figsize=(5.5, 3.5), layout="constrained")
for k, ax in axd.items():
    annotate_axes(ax, f'axd[{k!r}]', fontsize=14)
fig.suptitle('plt.subplot_mosaic()')
```




### Grids of fixed-aspect ratio Axes

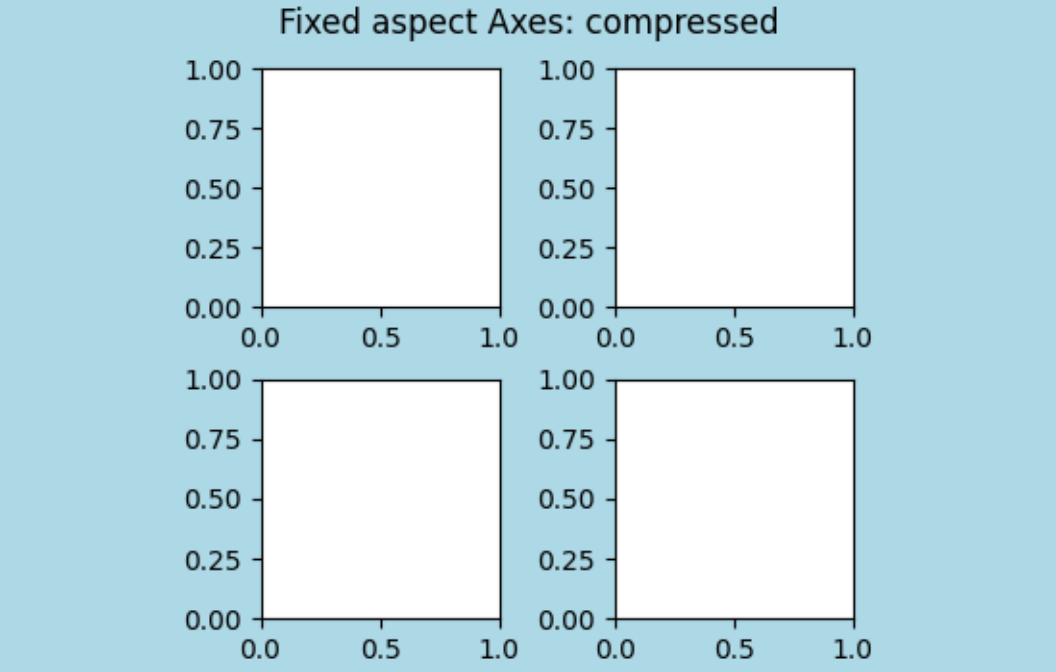
Fixed-aspect ratio Axes are common for images or maps. However, they present a challenge to layout because two sets of constraints are being imposed on the size of the Axes - that they fit in the figure and that they have a set aspect ratio. This leads to large gaps between Axes by default:

```
fig, axs = plt.subplots(2, 2, layout="constrained",
                        figsize=(5.5, 3.5), facecolor='lightblue')
for ax in axs.flat:
    ax.set_aspect(1)
fig.suptitle('Fixed aspect Axes')
```




One way to address this is to change the aspect of the figure to be close to the aspect ratio of the Axes, however that requires trial and error. Matplotlib also supplies `layout="compressed"`, which will work with simple grids to reduce the gaps between Axes. (The `mpl_toolkits` also provides *`ImageGrid`* to accomplish a similar effect, but with a non-standard Axes class).

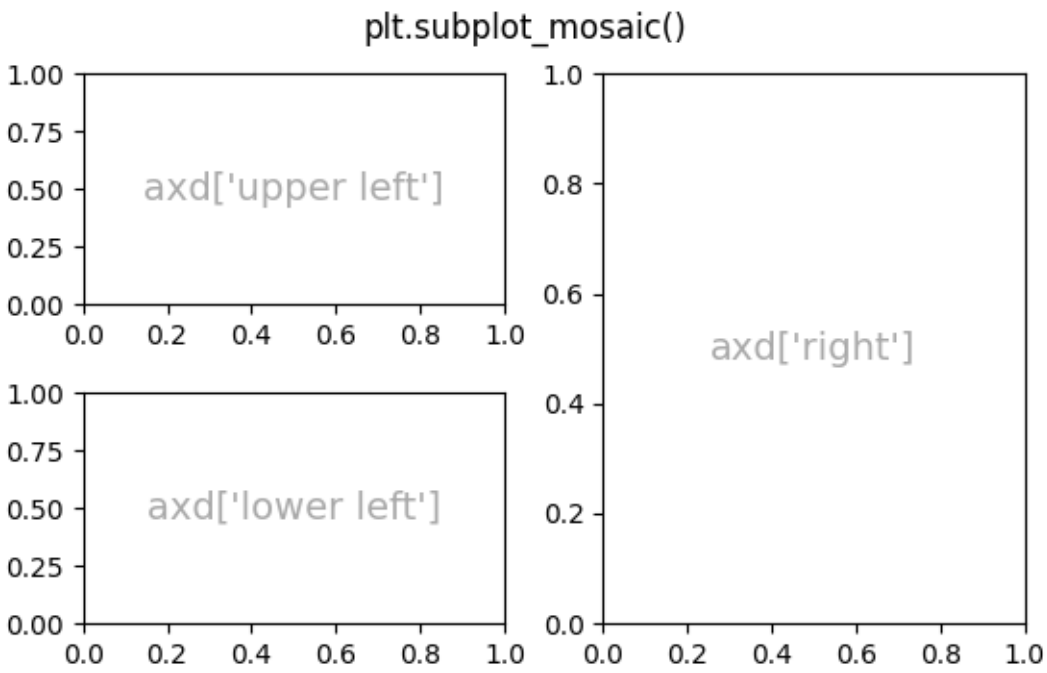
```
fig, axs = plt.subplots(2, 2, layout="compressed", figsize=(5.5, 3.5),
                        facecolor='lightblue')
for ax in axs.flat:
    ax.set_aspect(1)
fig.suptitle('Fixed aspect Axes: compressed')
```




### Axes spanning rows or columns in a grid

Sometimes we want Axes to span rows or columns of the grid. There are actually multiple ways to accomplish this, but the most convenient is probably to use *subplot_mosaic* by repeating one of the keys:

```
fig, axd = plt.subplot_mosaic([['upper left', 'right'],
                               ['lower left', 'right']],
                              figsize=(5.5, 3.5), layout="constrained")
for k, ax in axd.items():
    annotate_axes(ax, f'axd[{k!r}]', fontsize=14)
fig.suptitle('plt.subplot_mosaic()')
```

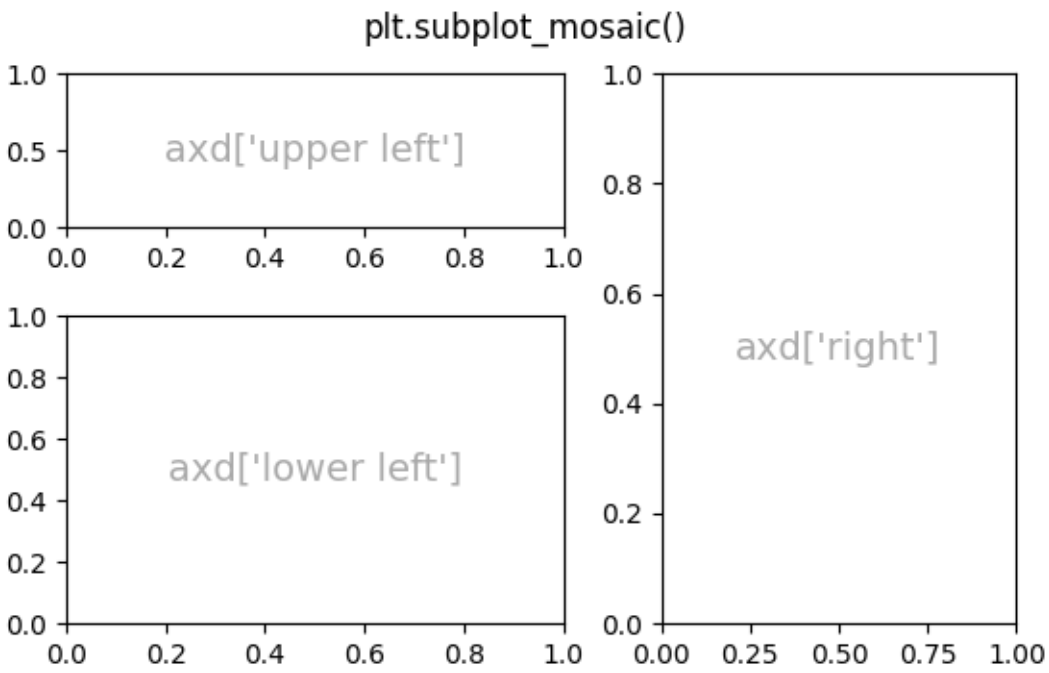



See below for the description of how to do the same thing using *GridSpec* or *subplot2grid*.

### Variable widths or heights in a grid

Both *subplots* and *subplot_mosaic* allow the rows in the grid to be different heights, and the columns to be different widths using the *gridspec_kw* keyword argument. Spacing parameters accepted by *GridSpec* can be passed to *subplots* and *subplot_mosaic*:

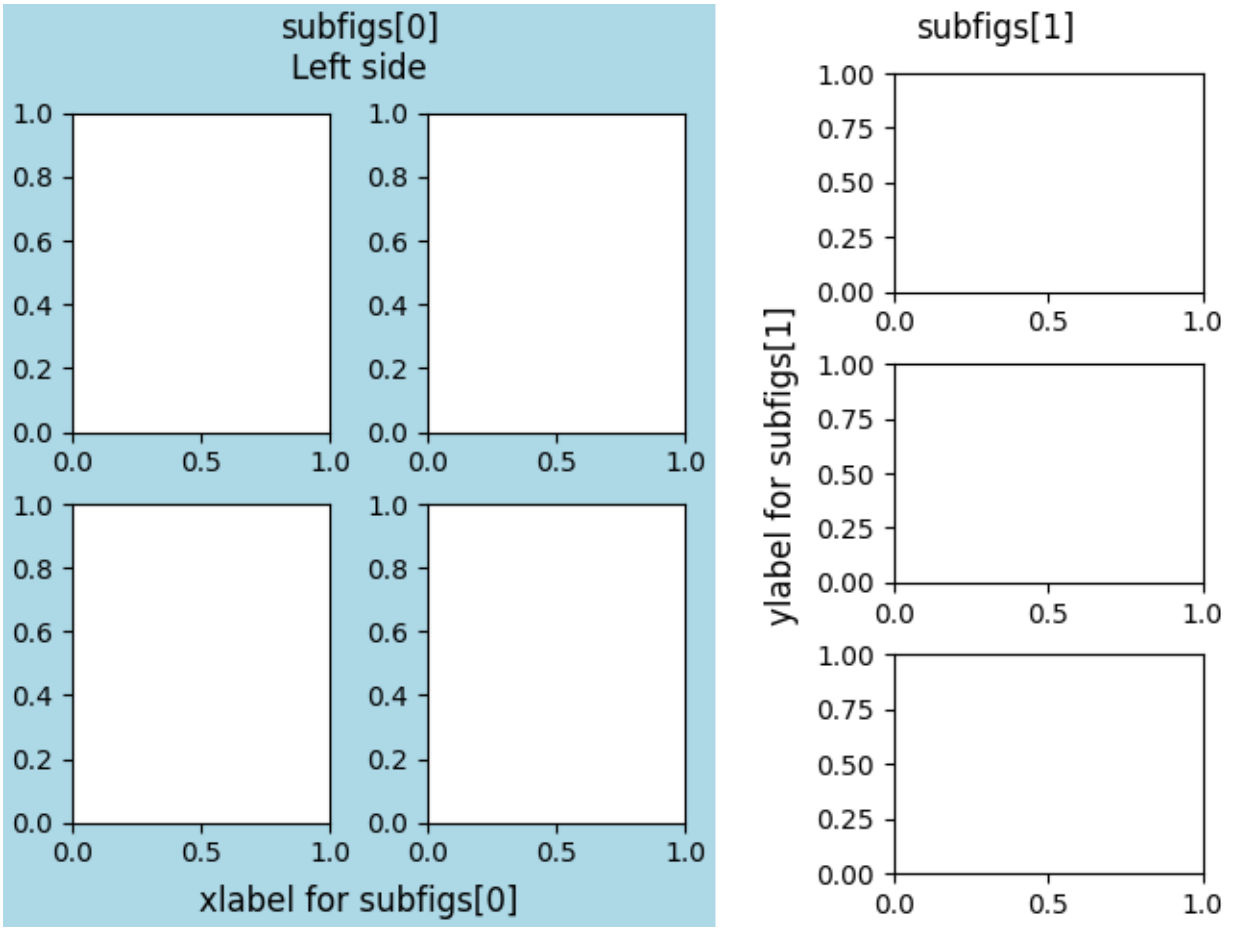
```
gs_kw = dict(width_ratios=[1.4, 1], height_ratios=[1, 2])
fig, axd = plt.subplot_mosaic([['upper left', 'right'],
                               ['lower left', 'right']],
                              gridspec_kw=gs_kw, figsize=(5.5, 3.5),
                              layout="constrained")
for k, ax in axd.items():
    annotate_axes(ax, f'axd[{k!r}]', fontsize=14)
fig.suptitle('plt.subplot_mosaic()')
```




### Nested Axes layouts

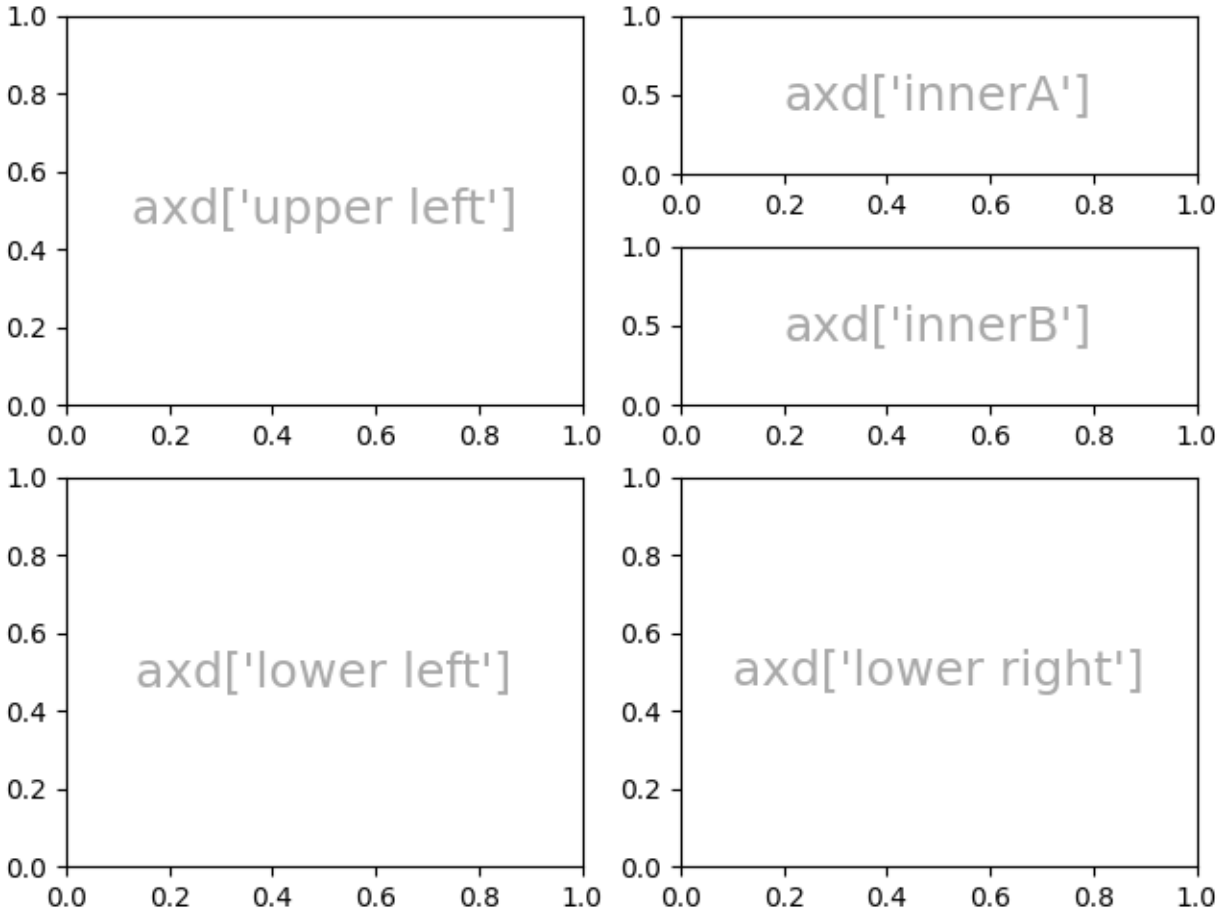
Sometimes it is helpful to have two or more grids of Axes that may not need to be related to one another. The most simple way to accomplish this is to use *Figure.subfigures*. Note that the subfigure layouts are independent, so the Axes spines in each subfigure are not necessarily aligned. See below for a more verbose way to achieve the same effect with *GridSpecFromSubplotSpec*.

```
fig = plt.figure(layout="constrained")
subfigs = fig.subfigures(1, 2, wspace=0.07, width_ratios=[1.5, 1.])
axs0 = subfigs[0].subplots(2, 2)
subfigs[0].set_facecolor('lightblue')
subfigs[0].suptitle('subfigs[0]\nLeft side')
subfigs[0].supxlabel('xlabel for subfigs[0]')

axs1 = subfigs[1].subplots(3, 1)
subfigs[1].suptitle('subfigs[1]')
subfigs[1].supylabel('ylabel for subfigs[1]')
```




It is also possible to nest Axes using *subplot_mosaic* using nested lists. This method does not use subfigures, like above, so lacks the ability to add per-subfigure `suptitle` and `supxlabel`, etc. Rather it is a convenience wrapper around the *subgridspec* method described below.

```
inner = [['innerA'],
         ['innerB']]
outer = [['upper left',  inner],
          ['lower left', 'lower right']]

fig, axd = plt.subplot_mosaic(outer, layout="constrained")
for k, ax in axd.items():
    annotate_axes(ax, f'axd[{k!r}]')
```




### Low-level and advanced grid methods

Internally, the arrangement of a grid of Axes is controlled by creating instances of *GridSpec* and *SubplotSpec*. *GridSpec* defines a (possibly non-uniform) grid of cells. Indexing into the *GridSpec* returns a SubplotSpec that covers one or more grid cells, and can be used to specify the location of an Axes.

The following examples show how to use low-level methods to arrange Axes using *GridSpec* objects.

### Basic 2x2 grid

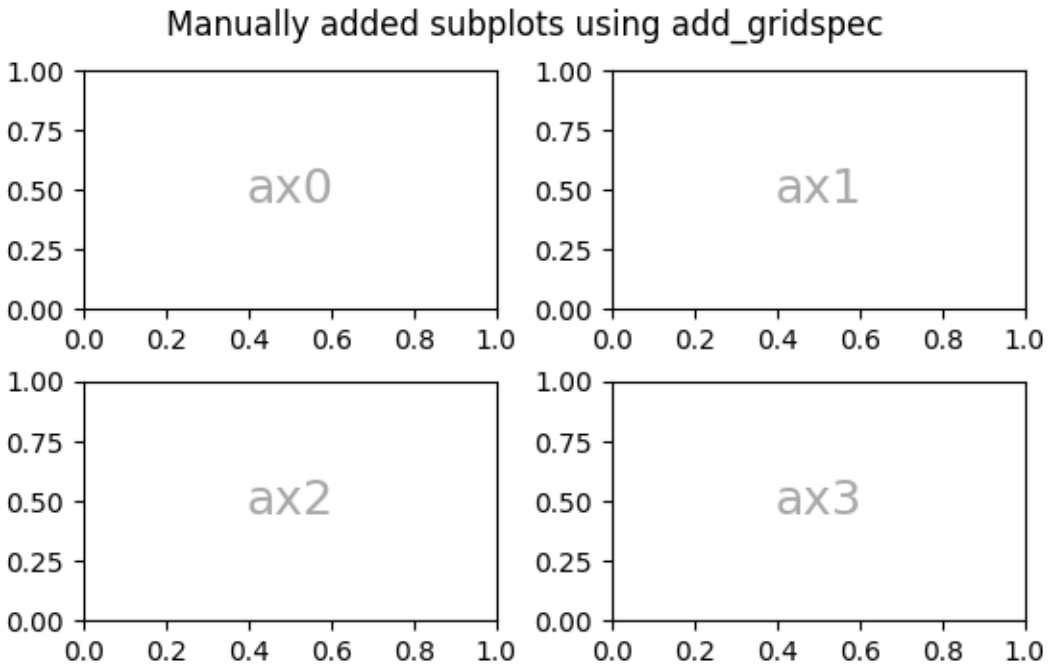
We can accomplish a 2x2 grid in the same manner as `plt.subplots(2, 2)`:

```
fig = plt.figure(figsize=(5.5, 3.5), layout="constrained")
spec = fig.add_gridspec(ncols=2, nrows=2)

ax0 = fig.add_subplot(spec[0, 0])
annotate_axes(ax0, 'ax0')

ax1 = fig.add_subplot(spec[0, 1])
annotate_axes(ax1, 'ax1')
```

(continues on next page)

(continued from previous page)

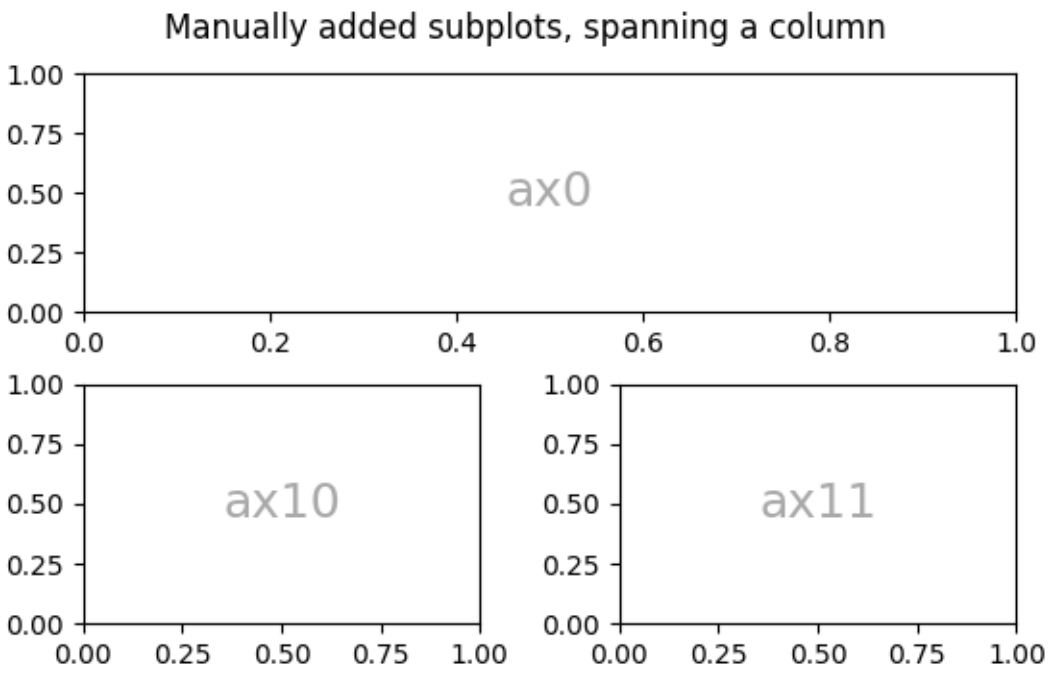
```
ax2 = fig.add_subplot(spec[1, 0])
annotate_axes(ax2, 'ax2')

ax3 = fig.add_subplot(spec[1, 1])
annotate_axes(ax3, 'ax3')

fig.suptitle('Manually added subplots using add_gridspec')
```




### Axes spanning rows or grids in a grid

We can index the *spec* array using NumPy slice syntax and the new Axes will span the slice. This would be the same as `fig, axd = plt.subplot_mosaic([['ax0', 'ax0'], ['ax1', 'ax2']], ...)`:

```
fig = plt.figure(figsize=(5.5, 3.5), layout="constrained")
spec = fig.add_gridspec(2, 2)

ax0 = fig.add_subplot(spec[0, :])
annotate_axes(ax0, 'ax0')

ax10 = fig.add_subplot(spec[1, 0])
annotate_axes(ax10, 'ax10')

ax11 = fig.add_subplot(spec[1, 1])
annotate_axes(ax11, 'ax11')

fig.suptitle('Manually added subplots, spanning a column')
```

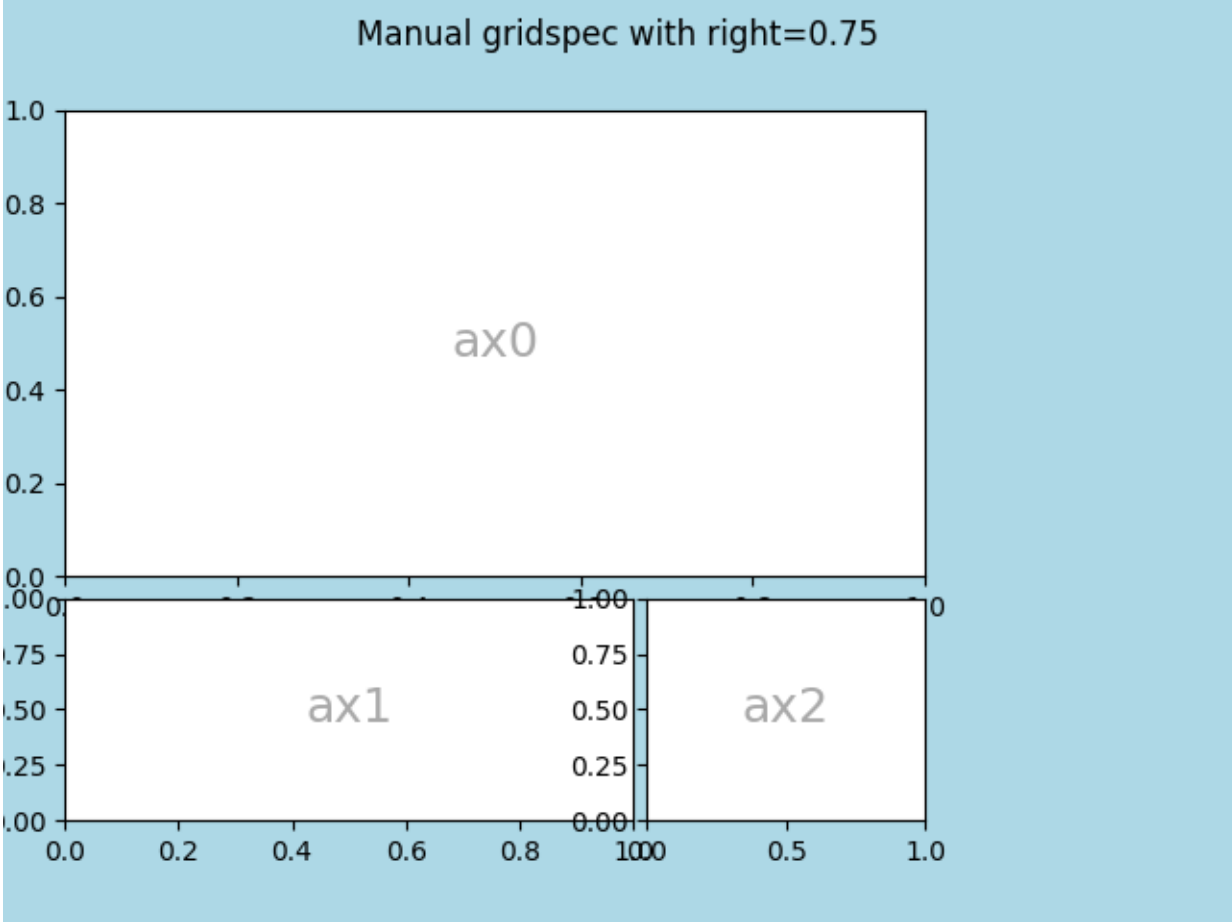



### Manual adjustments to a *GridSpec* layout

When a *GridSpec* is explicitly used, you can adjust the layout parameters of subplots that are created from the *GridSpec*. Note this option is not compatible with *constrained layout* or `Figure.tight_layout` which both ignore *left* and *right* and adjust subplot sizes to fill the figure. Usually such manual placement requires iterations to make the Axes tick labels not overlap the Axes.

These spacing parameters can also be passed to `subplots` and `subplot_mosaic` as the *gridspec_kw* argument.

```
fig = plt.figure(layout=None, facecolor='lightblue')
gs = fig.add_gridspec(nrows=3, ncols=3, left=0.05, right=0.75,
                      hspace=0.1, wspace=0.05)
ax0 = fig.add_subplot(gs[:-1, :])
annotate_axes(ax0, 'ax0')
ax1 = fig.add_subplot(gs[-1, :-1])
annotate_axes(ax1, 'ax1')
ax2 = fig.add_subplot(gs[-1, -1])
annotate_axes(ax2, 'ax2')
fig.suptitle('Manual gridspec with right=0.75')
```




### Nested layouts with SubplotSpec

You can create nested layout similar to *subfigures* using *subgridspec*. Here the Axes spines *are* aligned.

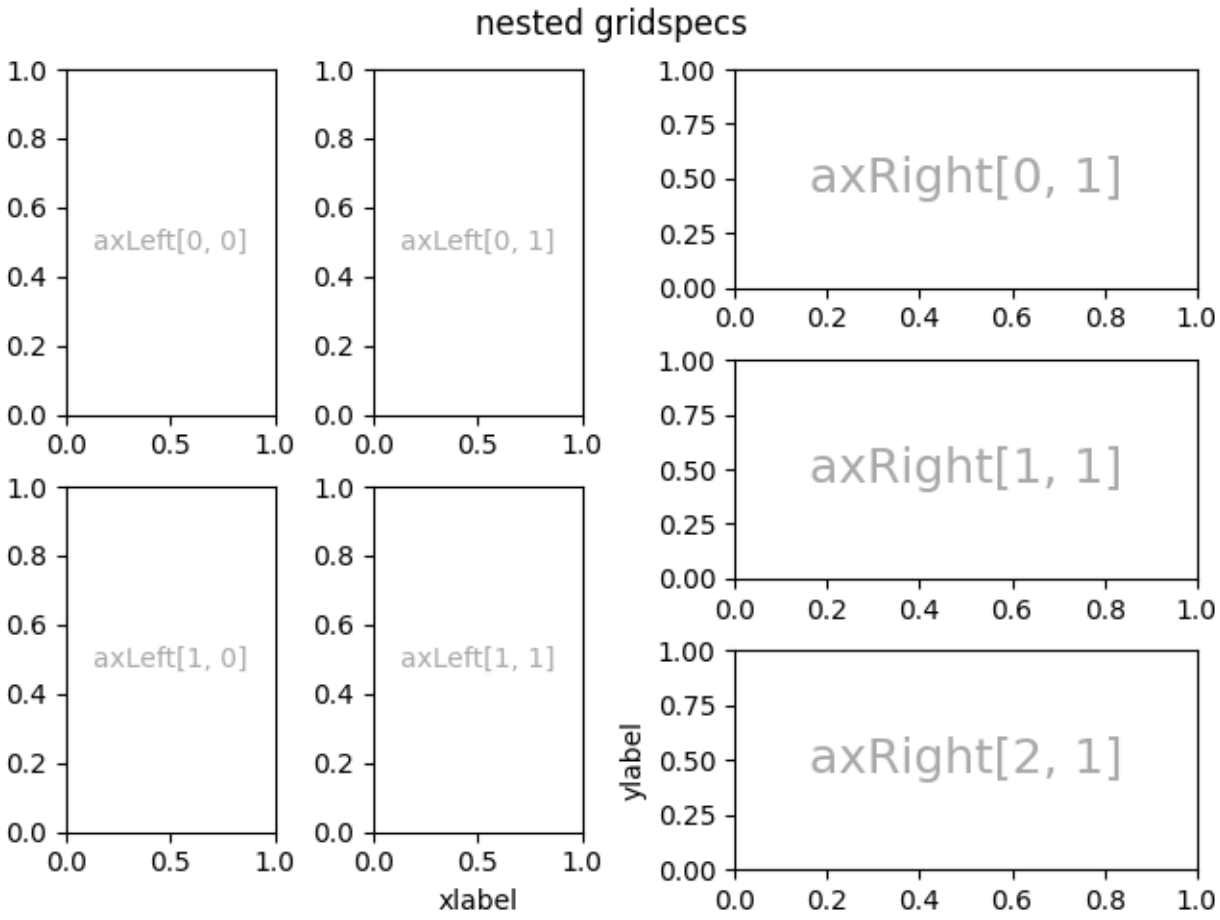
Note this is also available from the more verbose *gridspec.GridSpecFromSubplotSpec*.

```
fig = plt.figure(layout="constrained")
gs0 = fig.add_gridspec(1, 2)

gs00 = gs0[0].subgridspec(2, 2)
gs01 = gs0[1].subgridspec(3, 1)

for a in range(2):
    for b in range(2):
        ax = fig.add_subplot(gs00[a, b])
        annotate_axes(ax, f'axLeft[{a}, {b}]', fontsize=10)
        if a == 1 and b == 1:
            ax.set_xlabel('xlabel')
for a in range(3):
    ax = fig.add_subplot(gs01[a])
    annotate_axes(ax, f'axRight[{a}, {b}]')
```

(continues on next page)

(continued from previous page)

```
    if a == 2:
        ax.set_ylabel('ylabel')

fig.suptitle('nested gridspecs')
```



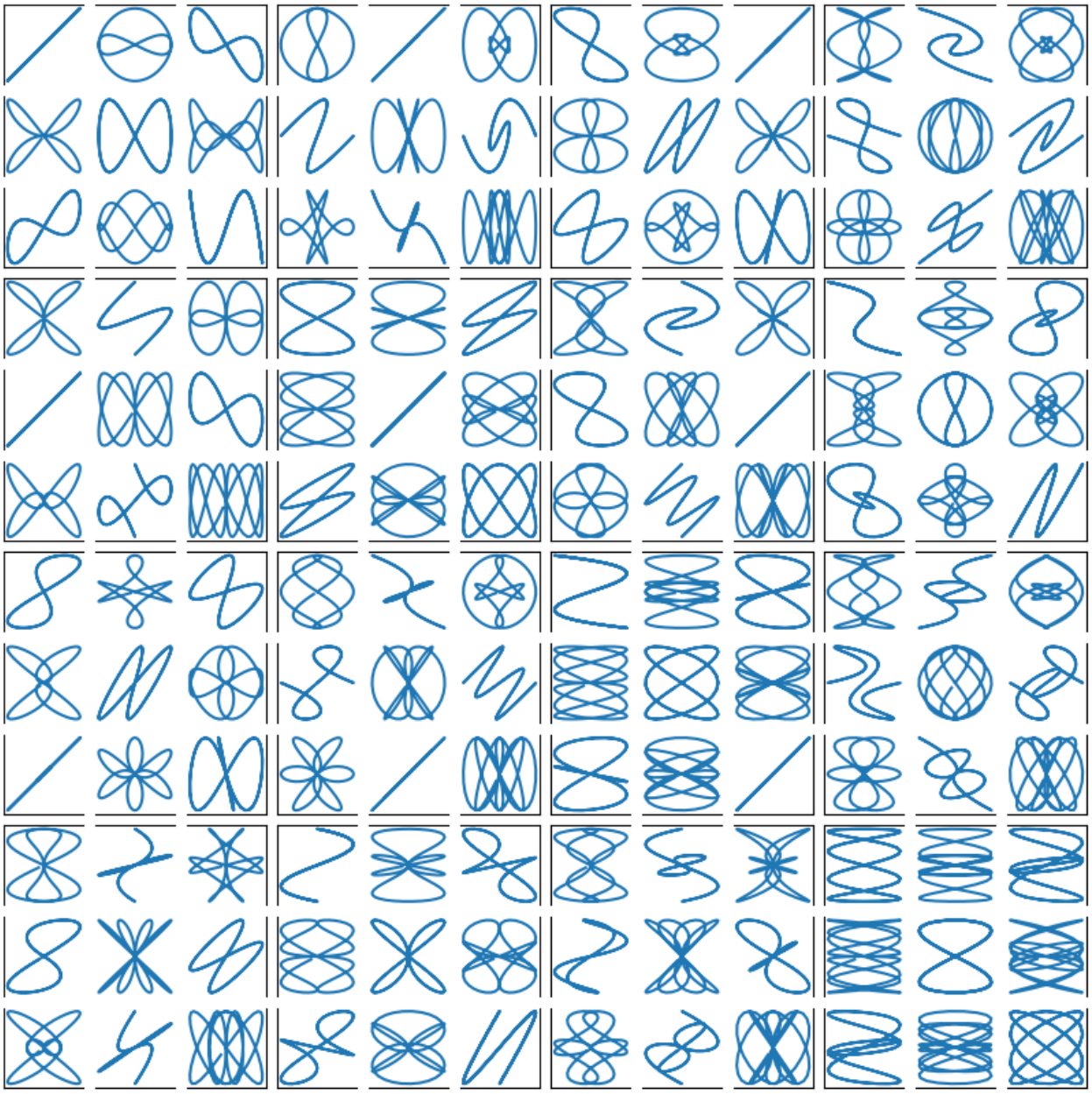

Here's a more sophisticated example of nested *GridSpec*: We create an outer 4x4 grid with each cell containing an inner 3x3 grid of Axes. We outline the outer 4x4 grid by hiding appropriate spines in each of the inner 3x3 grids.

```
def squiggle_xy(a, b, c, d, i=np.arange(0.0, 2*np.pi, 0.05)):
    return np.sin(i*a)*np.cos(i*b), np.sin(i*c)*np.cos(i*d)

fig = plt.figure(figsize=(8, 8), layout='constrained')
outer_grid = fig.add_gridspec(4, 4, wspace=0, hspace=0)

for a in range(4):
    for b in range(4):
        # gridspec inside gridspec
        inner_grid = outer_grid[a, b].subgridspec(3, 3, wspace=0, hspace=0)
        axs = inner_grid.subplots()  # Create all subplots for the inner grid.
        for (c, d), ax in np.ndenumerate(axs):
            ax.plot(*squiggle_xy(a + 1, b + 1, c + 1, d + 1))
```

(continues on next page)

(continued from previous page)

```
            ax.set(xticks=[], yticks=[])

# show only the outside spines
for ax in fig.get_axes():
    ss = ax.get_subplotspec()
    ax.spines.top.set_visible(ss.is_first_row())
    ax.spines.bottom.set_visible(ss.is_last_row())
    ax.spines.left.set_visible(ss.is_first_col())
    ax.spines.right.set_visible(ss.is_last_col())

plt.show()
```



### More reading

- More details about *subplot mosaic*.
- More details about *constrained layout*, used to align spacing in most of these examples.

---

**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.pyplot.subplots*
- *matplotlib.pyplot.subplot_mosaic*
- *matplotlib.figure.Figure.add_gridspec*
- *matplotlib.figure.Figure.add_subplot*
- *matplotlib.gridspec.GridSpec*
- *matplotlib.gridspec.SubplotSpec.subgridspec*
- *matplotlib.gridspec.GridSpecFromSubplotSpec*

---

**Total running time of the script:** (0 minutes 9.089 seconds)

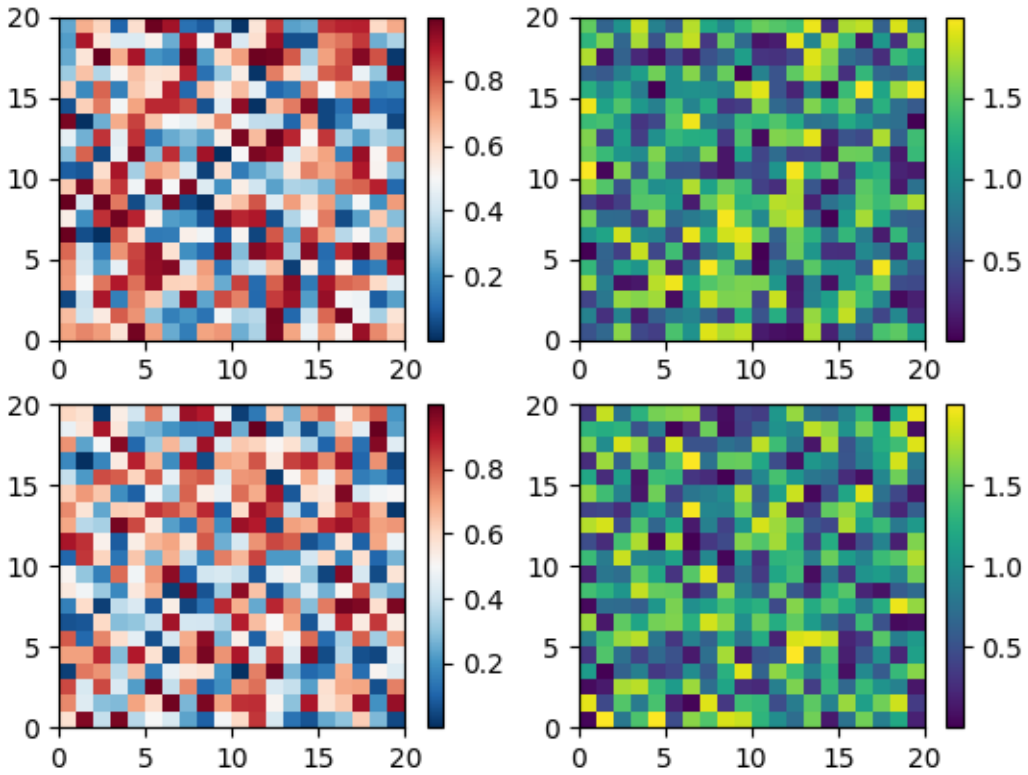
## 3.3.3 Placing colorbars

Colorbars indicate the quantitative extent of image data. Placing in a figure is non-trivial because room needs to be made for them.

### Automatic placement of colorbars

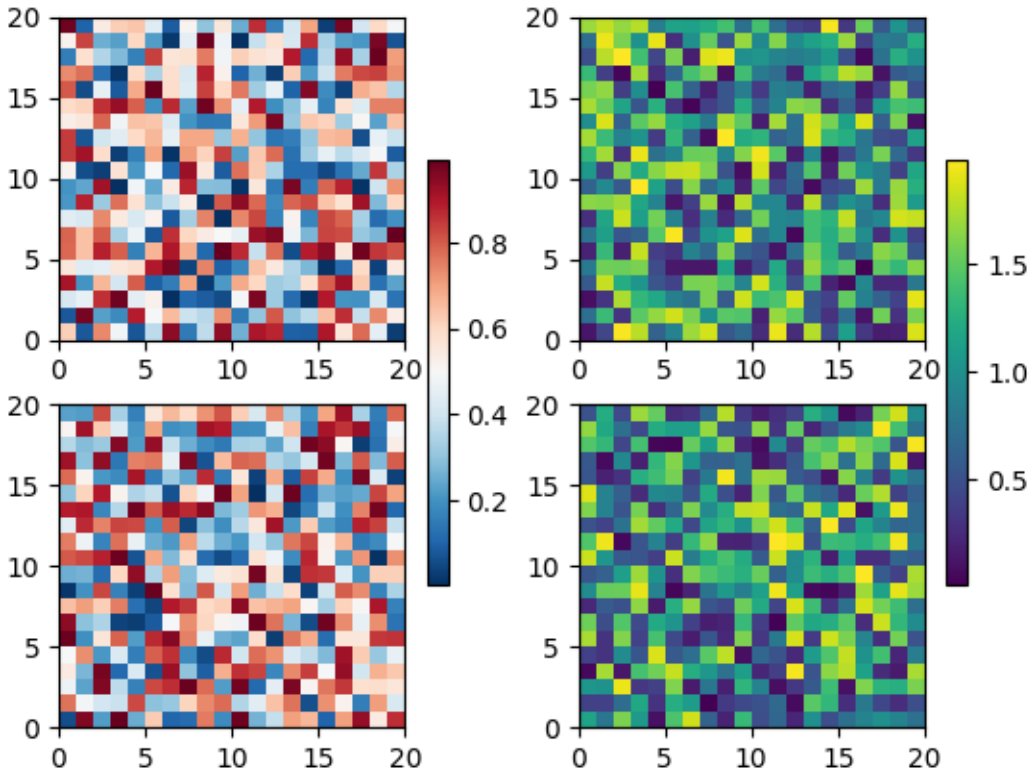
The simplest case is just attaching a colorbar to each Axes. Note in this example that the colorbars steal some space from the parent Axes.

```
import matplotlib.pyplot as plt
import numpy as np

# Fixing random state for reproducibility
np.random.seed(19680801)

fig, axs = plt.subplots(2, 2)
cmaps = ['RdBu_r', 'viridis']
for col in range(2):
    for row in range(2):
        ax = axs[row, col]
        pcm = ax.pcolormesh(np.random.random((20, 20)) * (col + 1),
                            cmap=cmaps[col])
        fig.colorbar(pcm, ax=ax)
```



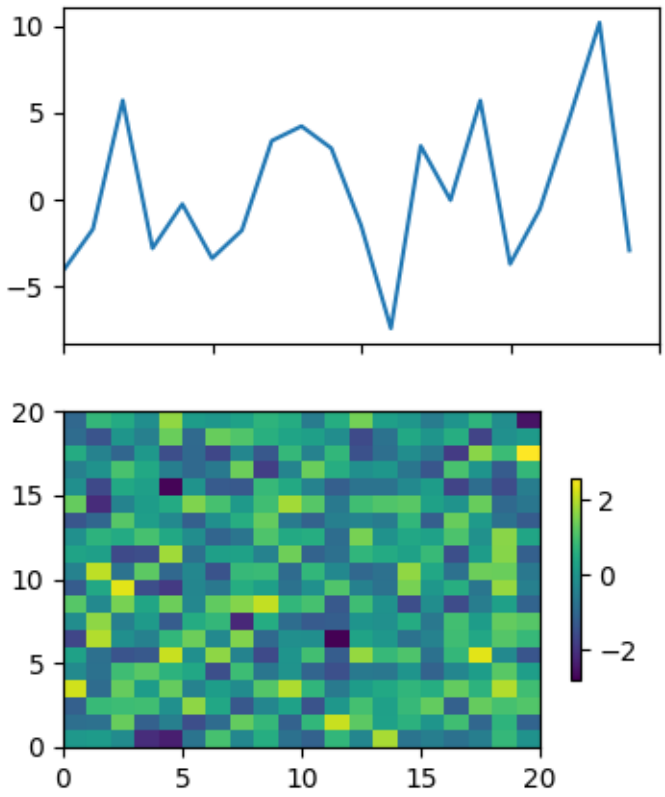

The first column has the same type of data in both rows, so it may be desirable to have just one colorbar. We do this by passing *Figure.colorbar* a list of Axes with the *ax* kwarg.

```
fig, axs = plt.subplots(2, 2)
cmaps = ['RdBu_r', 'viridis']
for col in range(2):
    for row in range(2):
        ax = axs[row, col]
        pcm = ax.pcolormesh(np.random.random((20, 20)) * (col + 1),
                            cmap=cmaps[col])
    fig.colorbar(pcm, ax=axs[:, col], shrink=0.6)
```



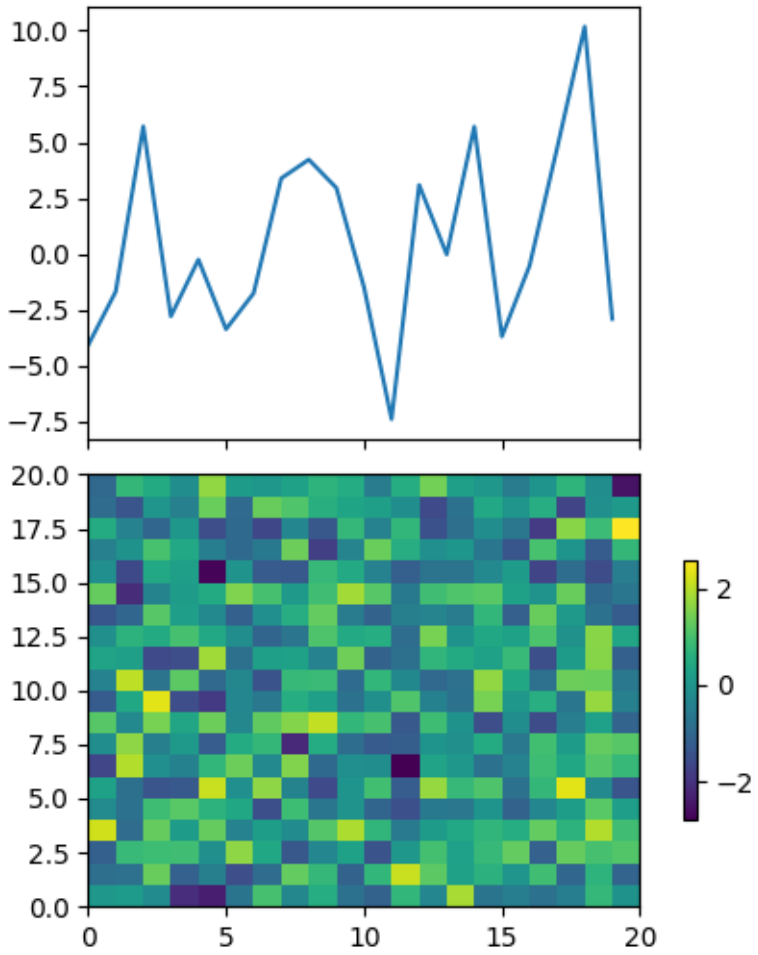

The stolen space can lead to Axes in the same subplot layout being different sizes, which is often undesired if the the x-axis on each plot is meant to be comparable as in the following:

```
fig, axs = plt.subplots(2, 1, figsize=(4, 5), sharex=True)
X = np.random.randn(20, 20)
axs[0].plot(np.sum(X, axis=0))
pcm = axs[1].pcolormesh(X)
fig.colorbar(pcm, ax=axs[1], shrink=0.6)
```



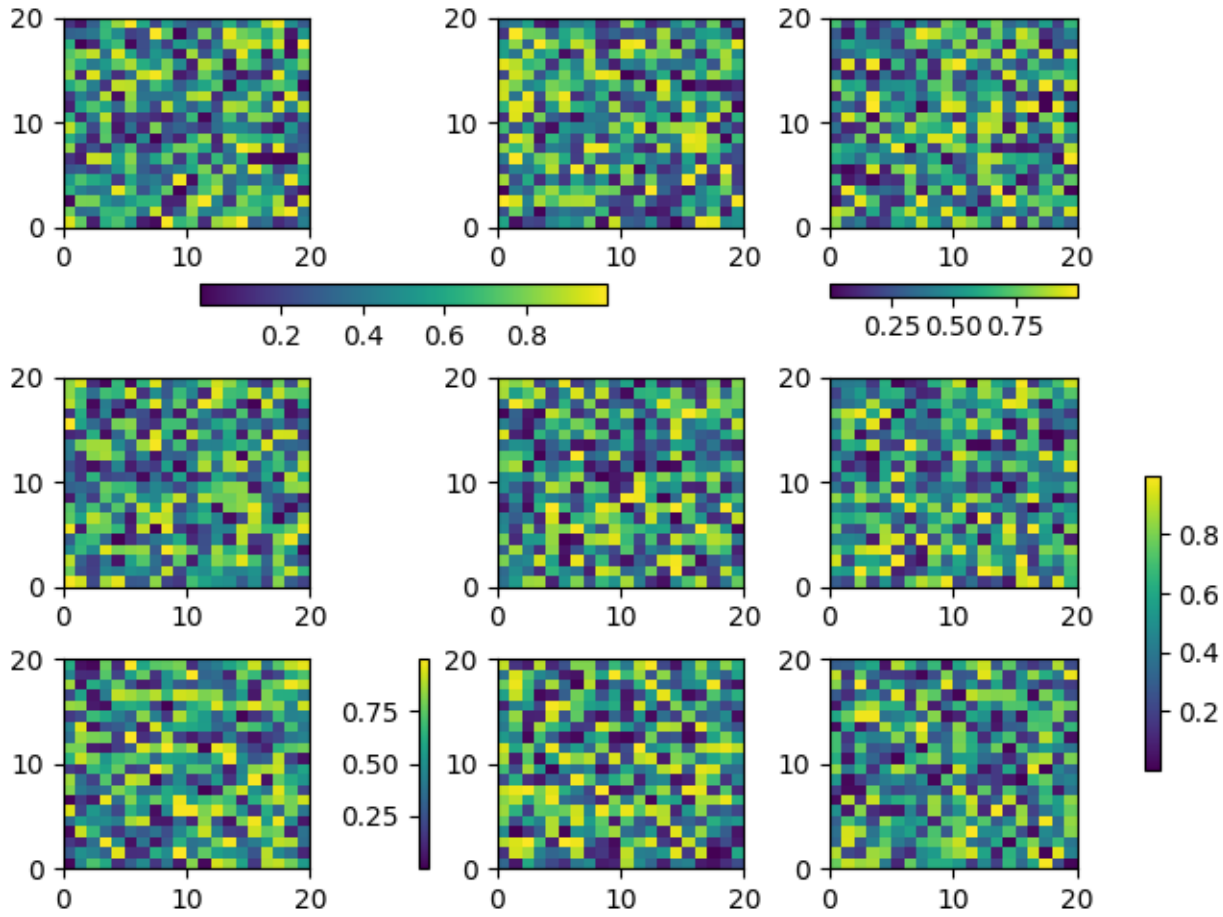

This is usually undesired, and can be worked around in various ways, e.g. adding a colorbar to the other Axes and then removing it. However, the most straightforward is to use *constrained layout*:

```
fig, axs = plt.subplots(2, 1, figsize=(4, 5), sharex=True, layout='constrained
 ↪')
axs[0].plot(np.sum(X, axis=0))
pcm = axs[1].pcolormesh(X)
fig.colorbar(pcm, ax=axs[1], shrink=0.6)
```




Relatively complicated colorbar layouts are possible using this paradigm. Note that this example works far better with `layout='constrained'`

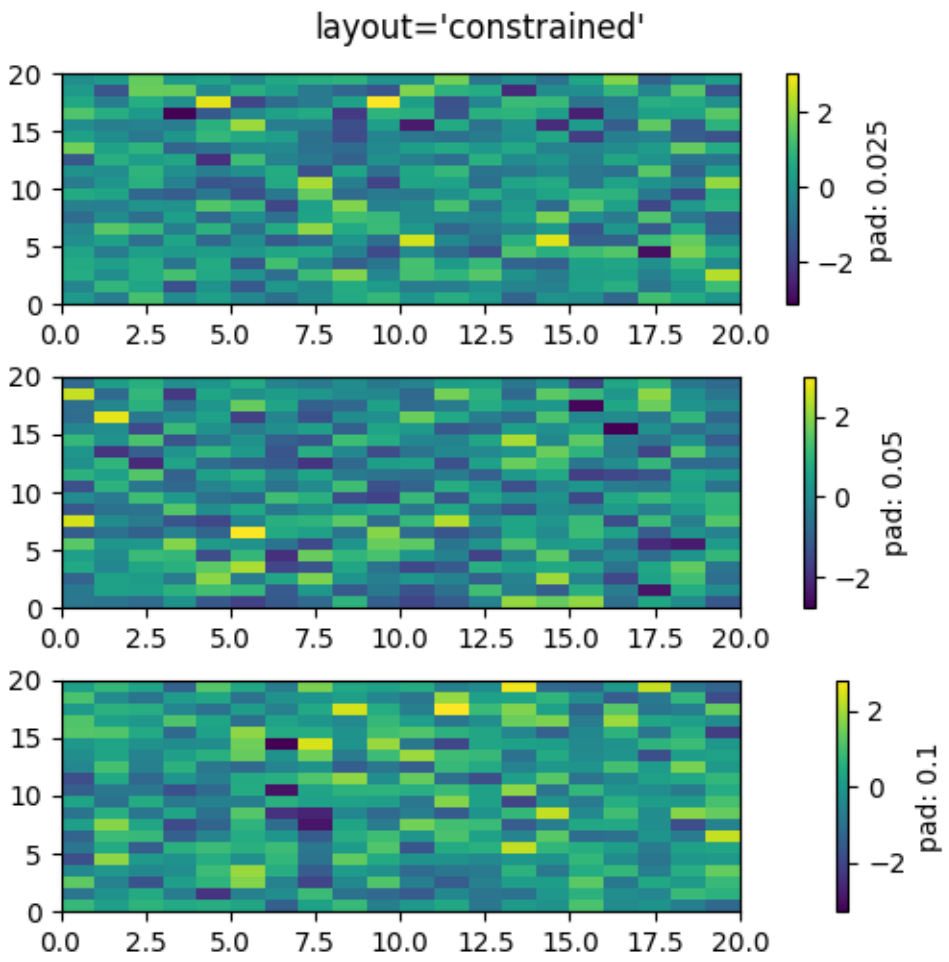
```
fig, axs = plt.subplots(3, 3, layout='constrained')
for ax in axs.flat:
    pcm = ax.pcolormesh(np.random.random((20, 20)))

fig.colorbar(pcm, ax=axs[0, :2], shrink=0.6, location='bottom')
fig.colorbar(pcm, ax=[axs[0, 2]], location='bottom')
fig.colorbar(pcm, ax=axs[1:, :], location='right', shrink=0.6)
fig.colorbar(pcm, ax=[axs[2, 1]], location='left')
```




### Adjusting the spacing between colorbars and parent Axes

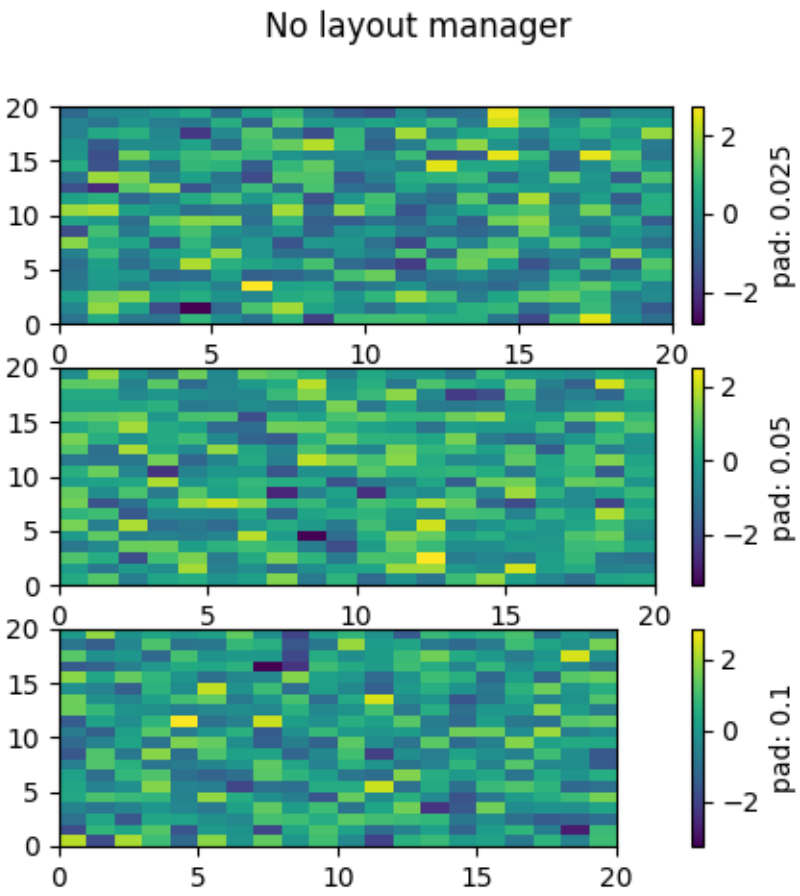
The distance a colorbar is from the parent Axes can be adjusted with the *pad* keyword argument. This is in units of fraction of the parent Axes width, and the default for a vertical Axes is 0.05 (or 0.15 for a horizontal Axes).

```
fig, axs = plt.subplots(3, 1, layout='constrained', figsize=(5, 5))
for ax, pad in zip(axs, [0.025, 0.05, 0.1]):
    pcm = ax.pcolormesh(np.random.randn(20, 20), cmap='viridis')
    fig.colorbar(pcm, ax=ax, pad=pad, label=f'pad: {pad}')
fig.suptitle("layout='constrained'")
```




Note that if you do not use constrained layout, the pad command makes the parent Axes shrink:

```
fig, axs = plt.subplots(3, 1, figsize=(5, 5))
for ax, pad in zip(axs, [0.025, 0.05, 0.1]):
    pcm = ax.pcolormesh(np.random.randn(20, 20), cmap='viridis')
    fig.colorbar(pcm, ax=ax, pad=pad, label=f'pad: {pad}')
fig.suptitle("No layout manager")
```



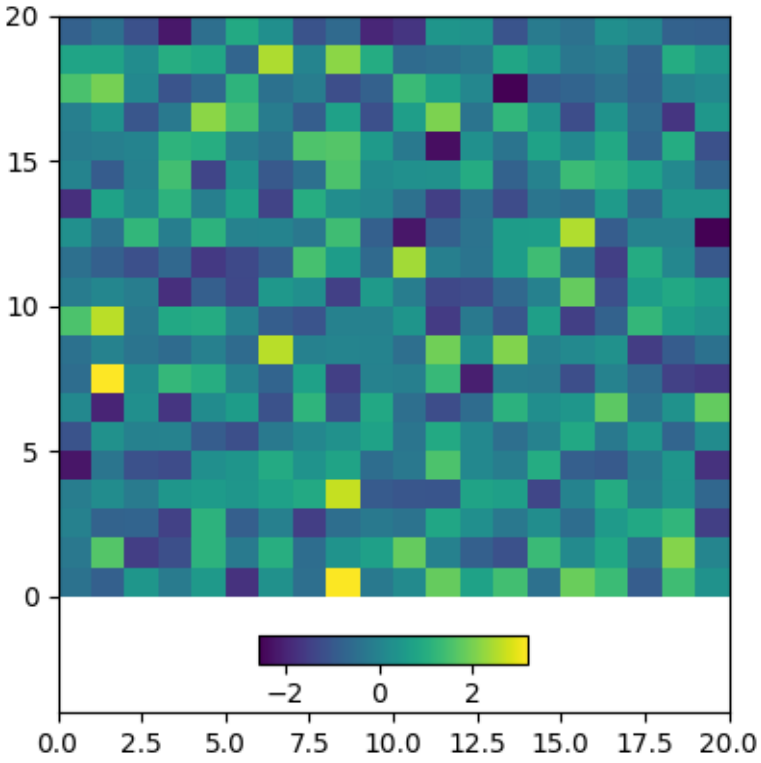

### Manual placement of colorbars

Sometimes the automatic placement provided by `colorbar` does not give the desired effect. We can manually create an Axes and tell `colorbar` to use that Axes by passing the Axes to the *cax* keyword argument.

### Using `inset_axes`

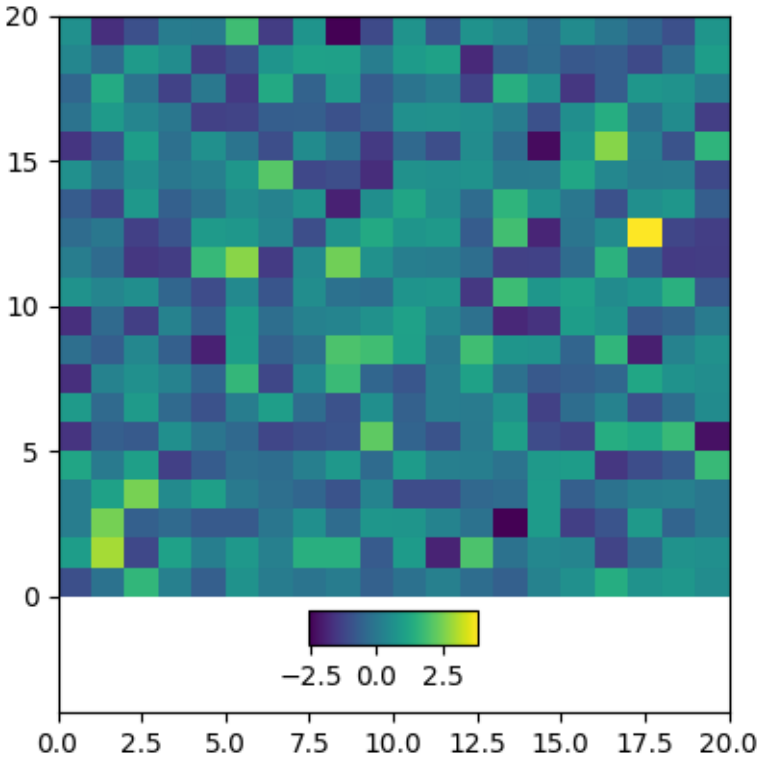
We can manually create any type of Axes for the colorbar to use, but an *`Axes.inset_axes`* is useful because it is a child of the parent Axes and can be positioned relative to the parent. Here we add a colorbar centered near the bottom of the parent Axes.

```
fig, ax = plt.subplots(layout='constrained', figsize=(4, 4))
pcm = ax.pcolormesh(np.random.randn(20, 20), cmap='viridis')
ax.set_ylim([-4, 20])
cax = ax.inset_axes([0.3, 0.07, 0.4, 0.04])
fig.colorbar(pcm, cax=cax, orientation='horizontal')
```




*Axes.inset_axes* can also specify its position in data coordinates using the *transform* keyword argument if you want your Axes at a certain data position on the graph:

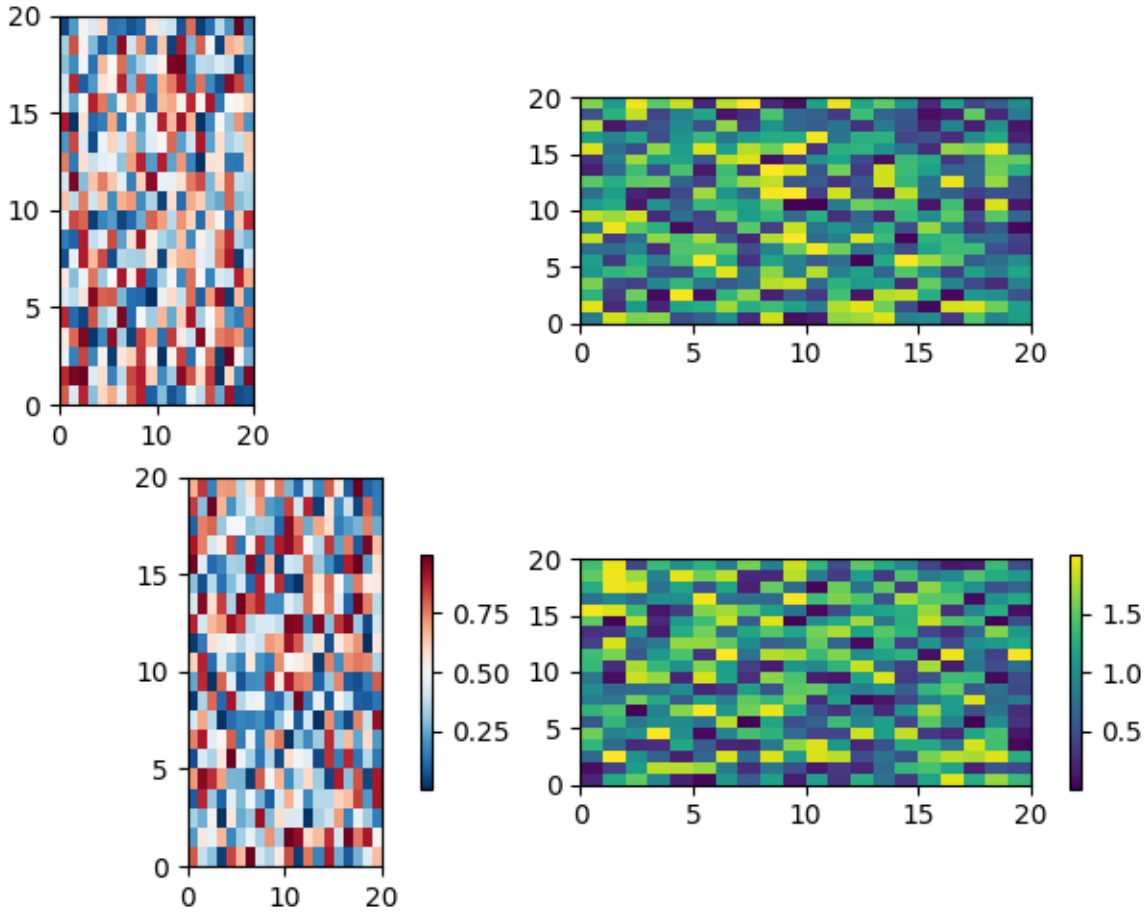
```
fig, ax = plt.subplots(layout='constrained', figsize=(4, 4))
pcm = ax.pcolormesh(np.random.randn(20, 20), cmap='viridis')
ax.set_ylim([-4, 20])
cax = ax.inset_axes([7.5, -1.7, 5, 1.2], transform=ax.transData)
fig.colorbar(pcm, cax=cax, orientation='horizontal')
```




### Colorbars attached to fixed-aspect-ratio Axes

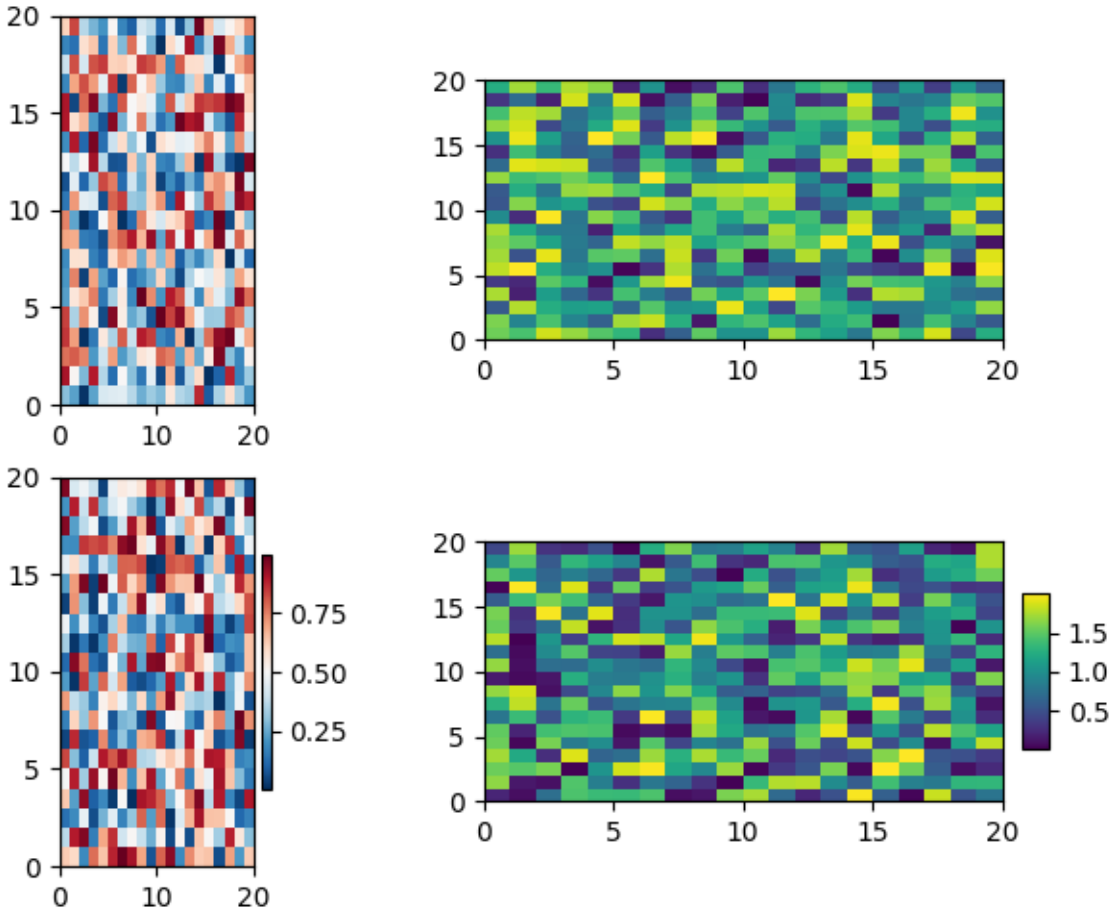
Placing colorbars for Axes with a fixed aspect ratio pose a particular challenge as the parent Axes changes size depending on the data view.

```
fig, axs = plt.subplots(2, 2,  layout='constrained')
cmaps = ['RdBu_r', 'viridis']
for col in range(2):
    for row in range(2):
        ax = axs[row, col]
        pcm = ax.pcolormesh(np.random.random((20, 20)) * (col + 1),
                            cmap=cmaps[col])
        if col == 0:
            ax.set_aspect(2)
        else:
            ax.set_aspect(1/2)
        if row == 1:
            fig.colorbar(pcm, ax=ax, shrink=0.6)
```




We solve this problem using *Axes.inset_axes* to locate the Axes in "axes coordinates" (see *Transformations Tutorial*). Note that if you zoom in on the parent Axes, and thus change the shape of it, the colorbar will also change position.

```
fig, axs = plt.subplots(2, 2, layout='constrained')
cmaps = ['RdBu_r', 'viridis']
for col in range(2):
    for row in range(2):
        ax = axs[row, col]
        pcm = ax.pcolormesh(np.random.random((20, 20)) * (col + 1),
                            cmap=cmaps[col])
        if col == 0:
            ax.set_aspect(2)
        else:
            ax.set_aspect(1/2)
        if row == 1:
            cax = ax.inset_axes([1.04, 0.2, 0.05, 0.6])
            fig.colorbar(pcm, cax=cax)
```




**See also:**

*The axes_grid1 toolkit* has methods for manually creating colorbar Axes as well:

- *Controlling the position and size of colorbars with Inset Axes*
- *Colorbar with AxesDivider*

**Total running time of the script:** (0 minutes 5.440 seconds)

### 3.3.4 Autoscaling Axis

The limits on an axis can be set manually (e.g. `ax.set_xlim(xmin, xmax)`) or Matplotlib can set them automatically based on the data already on the Axes. There are a number of options to this autoscaling behaviour, discussed below.

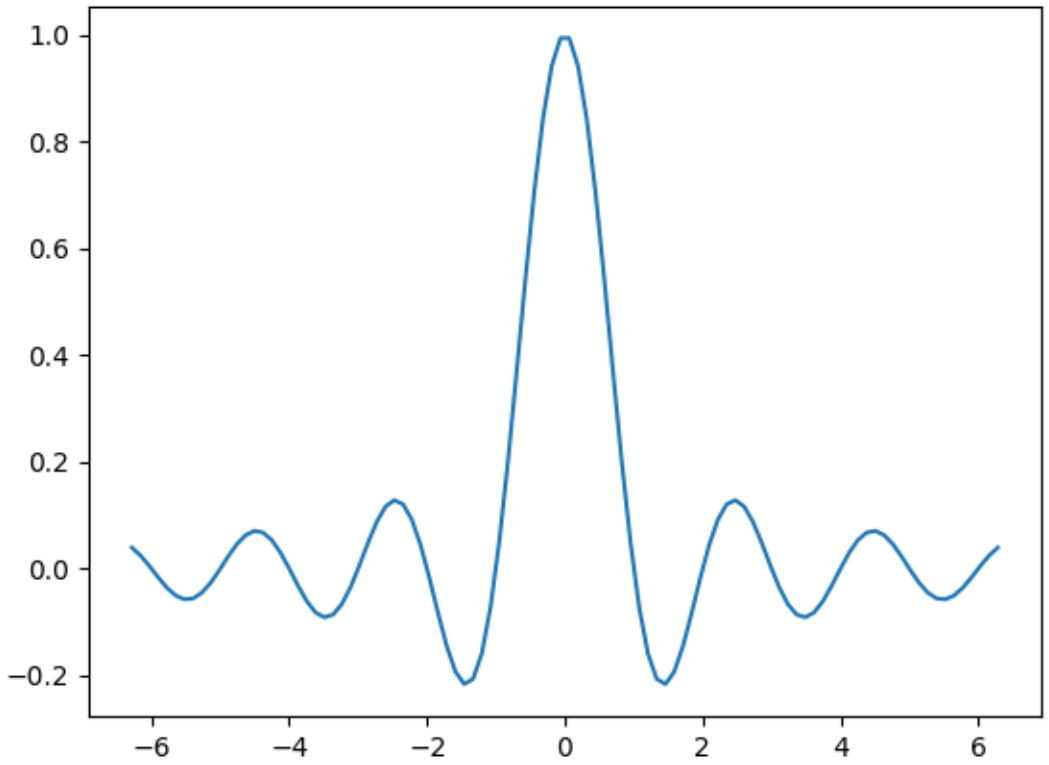
We will start with a simple line plot showing that autoscaling extends the axis limits 5% beyond the data limits (-2π, 2π).

```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib as mpl
```

(continues on next page)

(continued from previous page)

```
x = np.linspace(-2 * np.pi, 2 * np.pi, 100)
y = np.sinc(x)

fig, ax = plt.subplots()
ax.plot(x, y)
```




### Margins

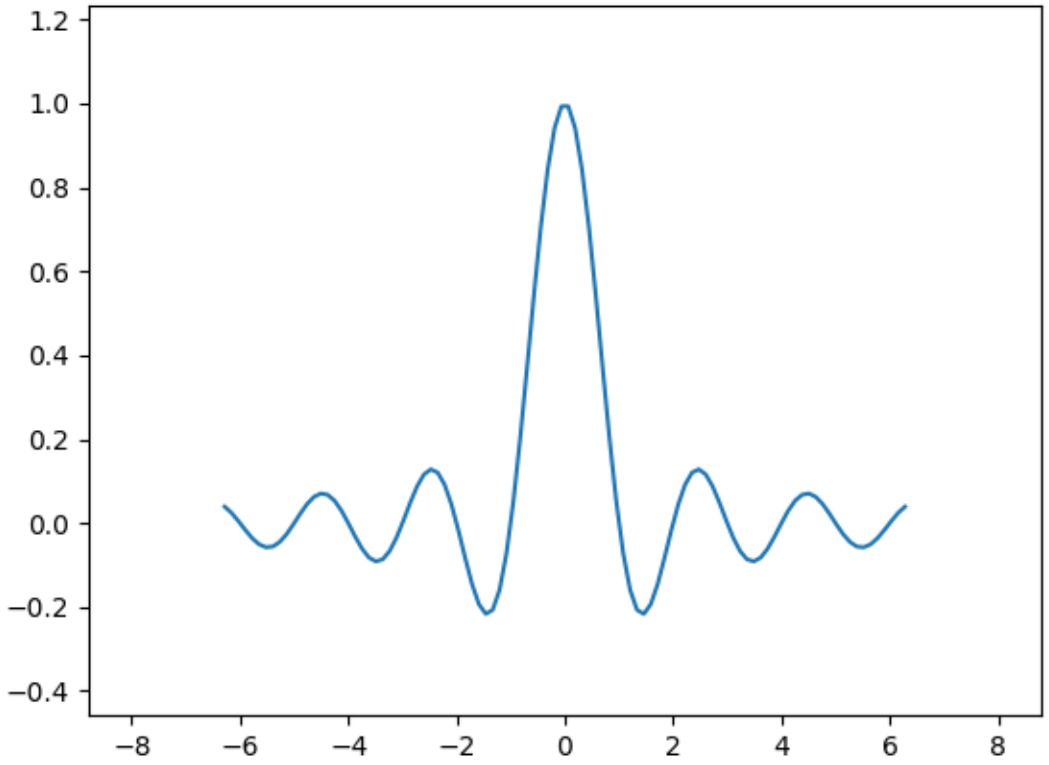
The default margin around the data limits is 5%, which is based on the default configuration setting of *rcParams["axes.xmargin"]* (default: 0.05), *rcParams["axes.ymargin"]* (default: 0.05), and *rcParams["axes.zmargin"]* (default: 0.05):

```
print(ax.margins())
```

```
(0.05, 0.05)
```

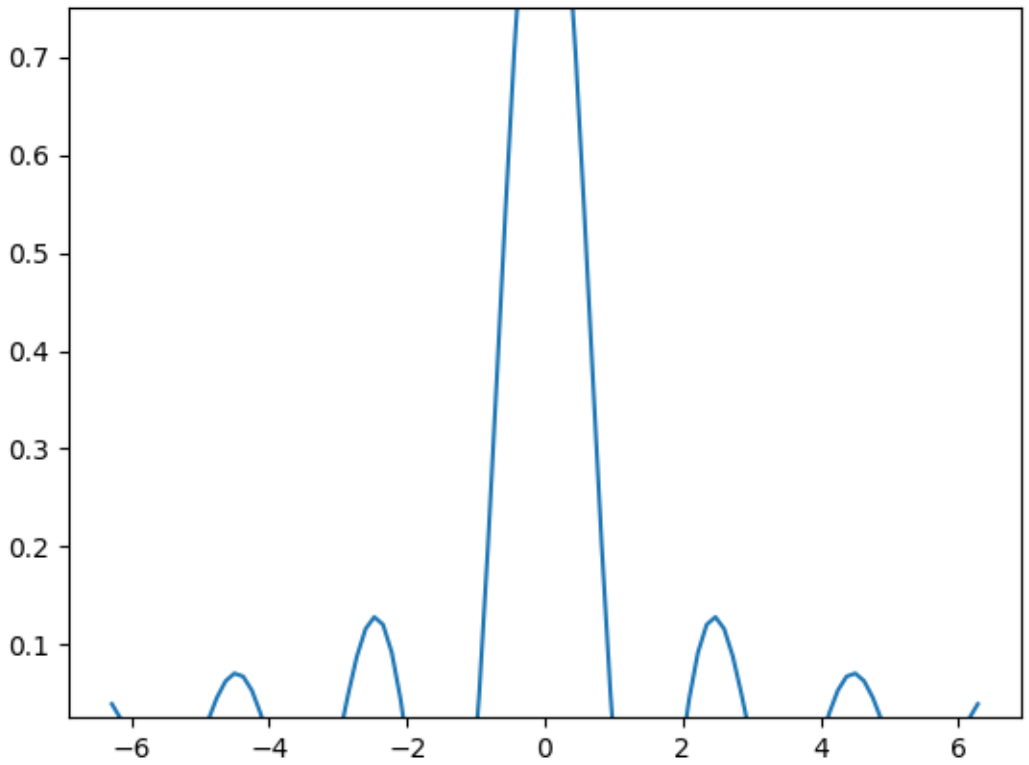
The margin size can be overridden to make them smaller or larger using *margins*:

```
fig, ax = plt.subplots()
ax.plot(x, y)
ax.margins(0.2, 0.2)
```




In general, margins can be in the range (-0.5, ∞), where negative margins set the axes limits to a subrange of the data range, i.e. they clip data. Using a single number for margins affects both axes, a single margin can be customized using keyword arguments `x` or `y`, but positional and keyword interface cannot be combined.

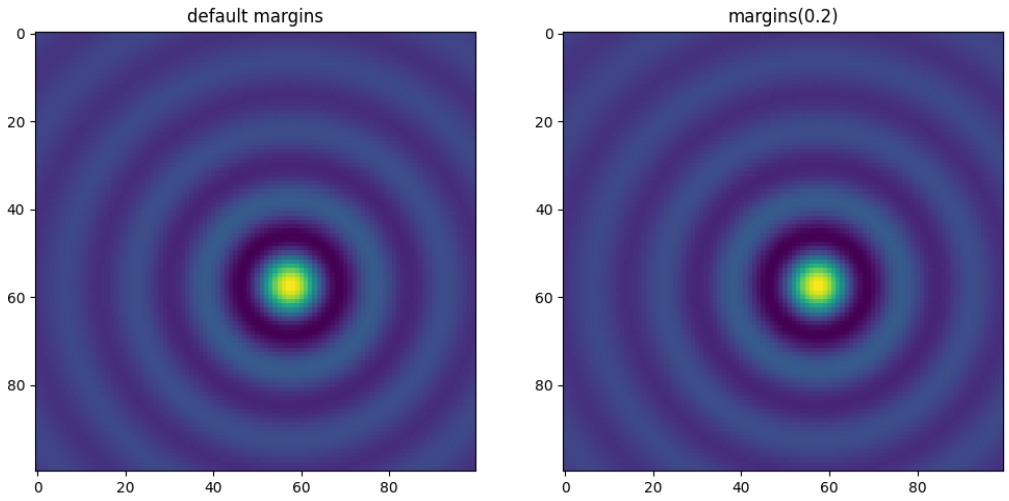
```
fig, ax = plt.subplots()
ax.plot(x, y)
ax.margins(y=-0.2)
```




### Sticky edges

There are plot elements (*Artist*s) that are usually used without margins. For example false-color images (e.g. created with *Axes.imshow*) are not considered in the margins calculation.

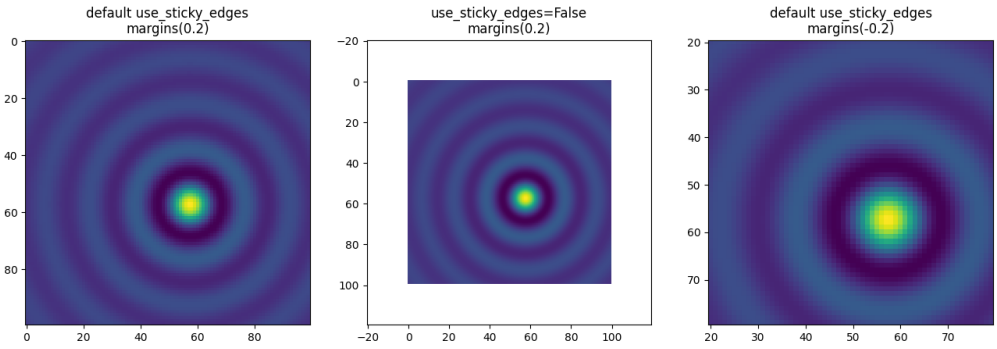
```
xx, yy = np.meshgrid(x, x)
zz = np.sinc(np.sqrt((xx - 1)**2 + (yy - 1)**2))

fig, ax = plt.subplots(ncols=2, figsize=(12, 8))
ax[0].imshow(zz)
ax[0].set_title("default margins")
ax[1].imshow(zz)
ax[1].margins(0.2)
ax[1].set_title("margins(0.2)")
```




This override of margins is determined by "sticky edges", a property of *Artist* class that can suppress adding margins to axis limits. The effect of sticky edges can be disabled on an Axes by changing *use_sticky_edges*. Artists have a property *Artist.sticky_edges*, and the values of sticky edges can be changed by writing to `Artist.sticky_edges.x` or `Artist.sticky_edges.y`.

The following example shows how overriding works and when it is needed.

```
fig, ax = plt.subplots(ncols=3, figsize=(16, 10))
ax[0].imshow(zz)
ax[0].margins(0.2)
ax[0].set_title("default use_sticky_edges\nmargins(0.2)")
ax[1].imshow(zz)
ax[1].margins(0.2)
ax[1].use_sticky_edges = False
ax[1].set_title("use_sticky_edges=False\nmargins(0.2)")
ax[2].imshow(zz)
ax[2].margins(-0.2)
ax[2].set_title("default use_sticky_edges\nmargins(-0.2)")
```



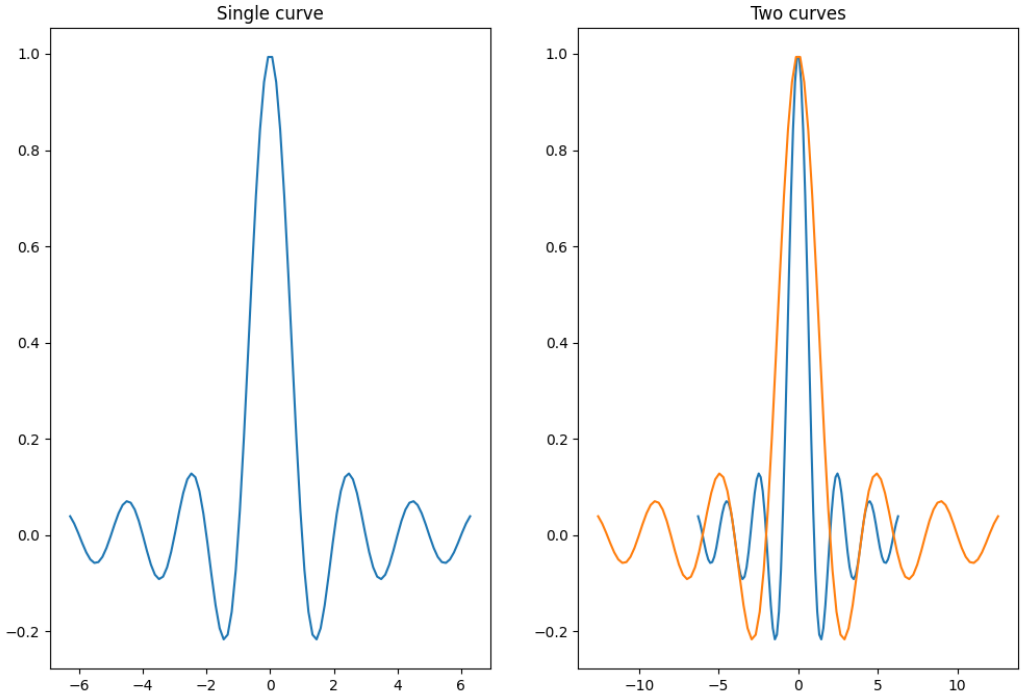

We can see that setting `use_sticky_edges` to *False* renders the image with requested margins.

While sticky edges don't increase the axis limits through extra margins, negative margins are still taken into account. This can be seen in the reduced limits of the third image.

### Controlling autoscale

By default, the limits are recalculated every time you add a new curve to the plot:

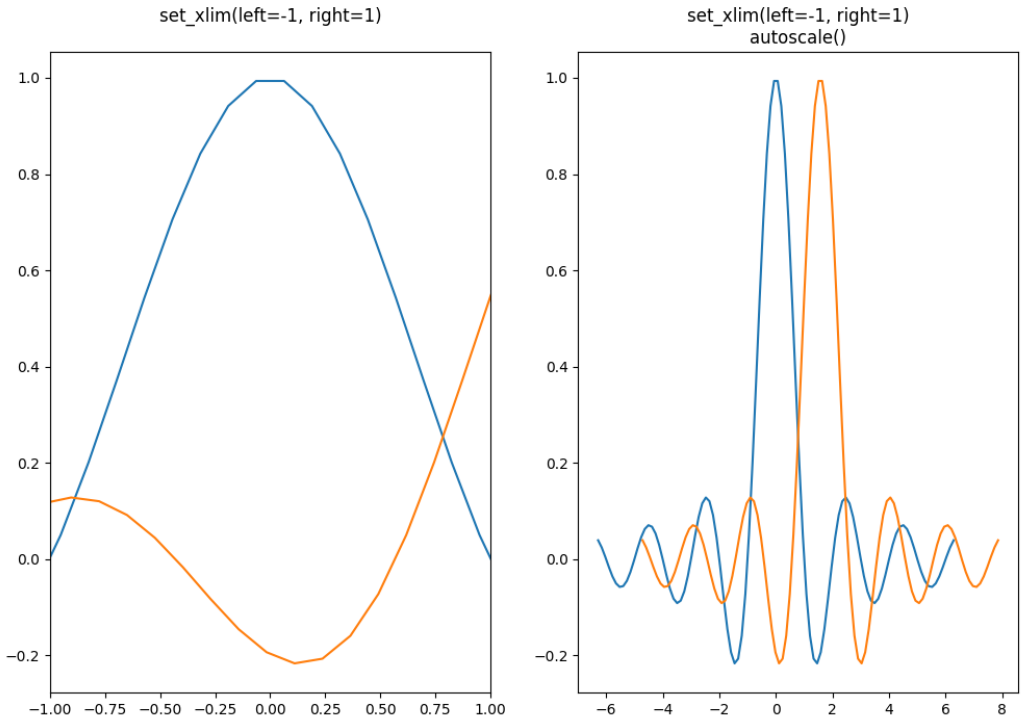
```
fig, ax = plt.subplots(ncols=2, figsize=(12, 8))
ax[0].plot(x, y)
ax[0].set_title("Single curve")
ax[1].plot(x, y)
ax[1].plot(x * 2.0, y)
ax[1].set_title("Two curves")
```




However, there are cases when you don't want to automatically adjust the viewport to new data.

One way to disable autoscaling is to manually set the axis limit. Let's say that we want to see only a part of the data in greater detail. Setting the `xlim` persists even if we add more curves to the data. To recalculate the new limits calling *`Axes.autoscale`* will toggle the functionality manually.

```
fig, ax = plt.subplots(ncols=2, figsize=(12, 8))
ax[0].plot(x, y)
ax[0].set_xlim(left=-1, right=1)
ax[0].plot(x + np.pi * 0.5, y)
ax[0].set_title("set_xlim(left=-1, right=1)\n")
ax[1].plot(x, y)
ax[1].set_xlim(left=-1, right=1)
ax[1].plot(x + np.pi * 0.5, y)
ax[1].autoscale()
ax[1].set_title("set_xlim(left=-1, right=1)\nautoscale()")
```



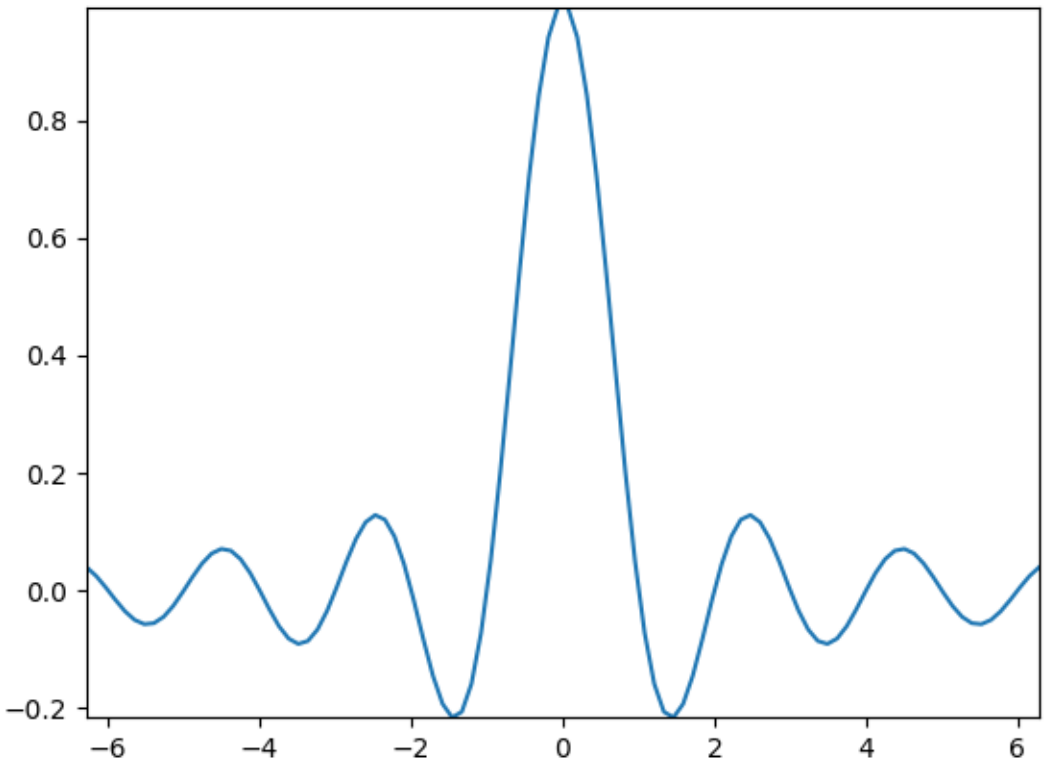

We can check that the first plot has autoscale disabled and that the second plot has it enabled again by using *Axes.get_autoscale_on()*:

```
print(ax[0].get_autoscale_on())  # False means disabled
print(ax[1].get_autoscale_on())  # True means enabled -> recalculated
```

```
False
True
```

Arguments of the autoscale function give us precise control over the process of autoscaling. A combination of arguments `enable`, and `axis` sets the autoscaling feature for the selected axis (or both). The argument `tight` sets the margin of the selected axis to zero. To preserve settings of either `enable` or `tight` you can set the opposite one to *None*, that way it should not be modified. However, setting `enable` to *None* and tight to *True* affects both axes regardless of the `axis` argument.

```
fig, ax = plt.subplots()
ax.plot(x, y)
ax.margins(0.2, 0.2)
ax.autoscale(enable=None, axis="x", tight=True)

print(ax.margins())
```

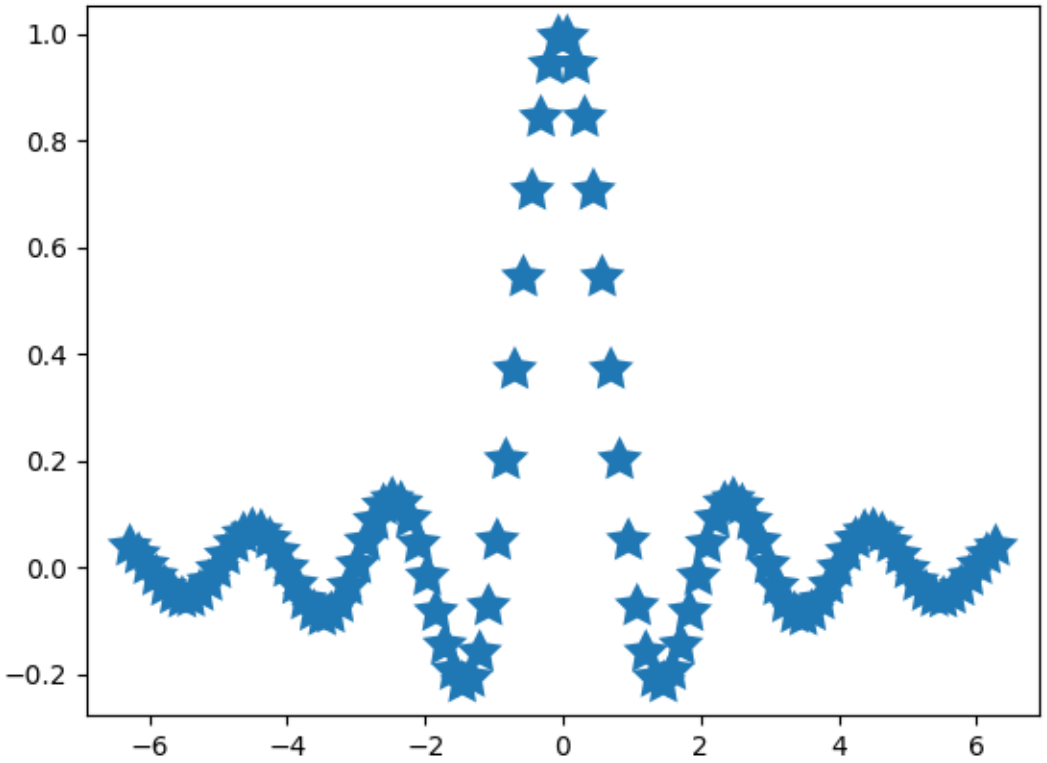



```
(0, 0)
```

### Working with collections

Autoscale works out of the box for all lines, patches, and images added to the Axes. One of the artists that it won't work with is a *Collection*. After adding a collection to the Axes, one has to manually trigger the *autoscale_view()* to recalculate axes limits.

```
fig, ax = plt.subplots()
collection = mpl.collections.StarPolygonCollection(
    5, rotation=0, sizes=(250,),  # five point star, zero angle, size 250px
    offsets=np.column_stack([x, y]),  # Set the positions
    offset_transform=ax.transData,  # Propagate transformations of the Axes
)
ax.add_collection(collection)
ax.autoscale_view()
```




**Total running time of the script:** (0 minutes 5.729 seconds)

### 3.3.5 Axis scales

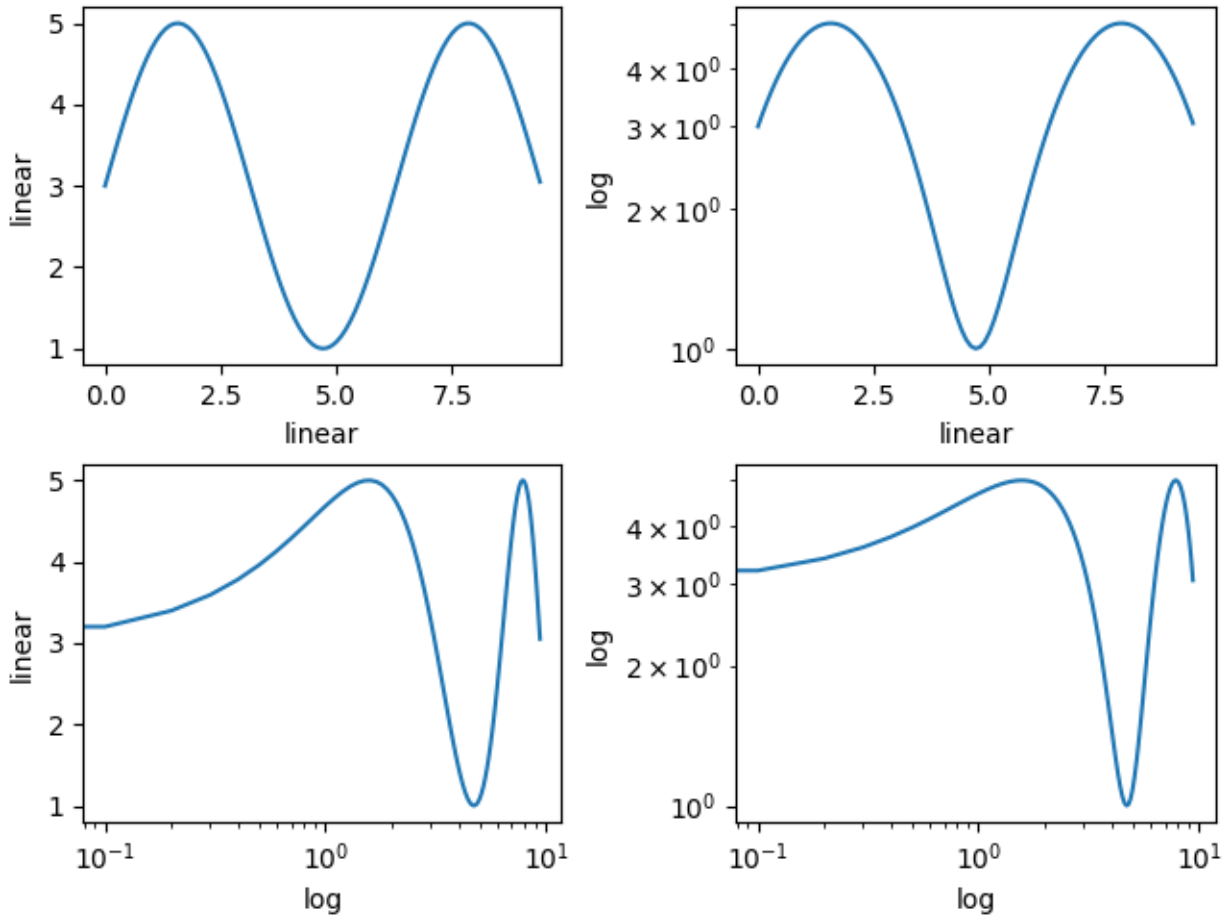
By default Matplotlib displays data on the axis using a linear scale. Matplotlib also supports logarithmic scales, and other less common scales as well. Usually this can be done directly by using the *set_xscale* or *set_yscale* methods.

```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib.scale as mscale
from matplotlib.ticker import FixedLocator, NullFormatter

fig, axs = plt.subplot_mosaic([['linear', 'linear-log'],
                               ['log-linear', 'log-log']], layout='constrained
 ↪')

x = np.arange(0, 3*np.pi, 0.1)
y = 2 * np.sin(x) + 3

ax = axs['linear']
```

(continues on next page)

(continued from previous page)

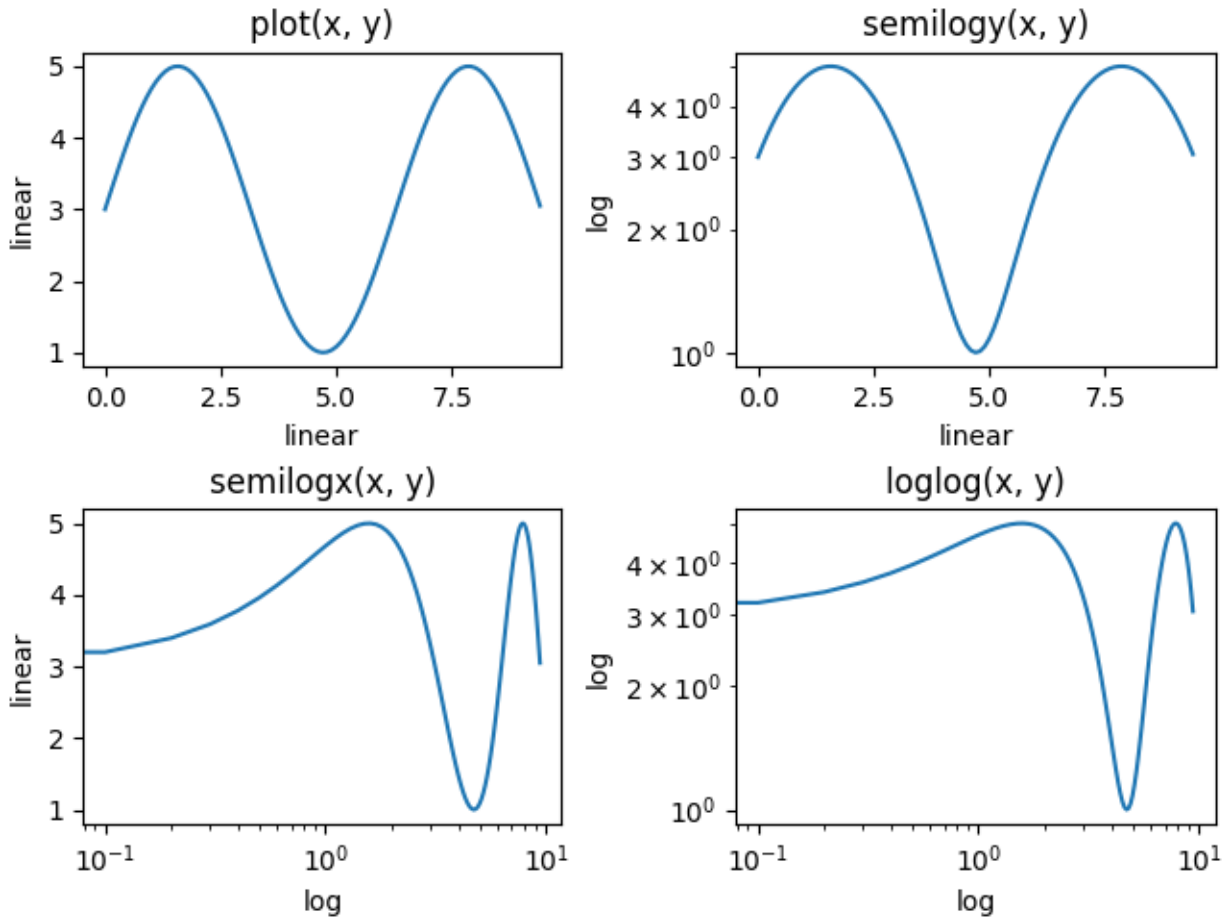
```
ax.plot(x, y)
ax.set_xlabel('linear')
ax.set_ylabel('linear')

ax = axs['linear-log']
ax.plot(x, y)
ax.set_yscale('log')
ax.set_xlabel('linear')
ax.set_ylabel('log')

ax = axs['log-linear']
ax.plot(x, y)
ax.set_xscale('log')
ax.set_xlabel('log')
ax.set_ylabel('linear')

ax = axs['log-log']
ax.plot(x, y)
ax.set_xscale('log')
ax.set_yscale('log')
ax.set_xlabel('log')
ax.set_ylabel('log')
```



### loglog and semilogx/y

The logarithmic axis is used so often that there are a set helper functions, that do the same thing: *semilogy*, *semilogx*, and *loglog*.

```
fig, axs = plt.subplot_mosaic([['linear', 'linear-log'],
                               ['log-linear', 'log-log']], layout='constrained
 ↪')

x = np.arange(0, 3*np.pi, 0.1)
y = 2 * np.sin(x) + 3

ax = axs['linear']
ax.plot(x, y)
ax.set_xlabel('linear')
ax.set_ylabel('linear')
ax.set_title('plot(x, y)')

ax = axs['linear-log']
ax.semilogy(x, y)
ax.set_xlabel('linear')
ax.set_ylabel('log')
ax.set_title('semilogy(x, y)')

ax = axs['log-linear']
ax.semilogx(x, y)
ax.set_xlabel('log')
ax.set_ylabel('linear')
ax.set_title('semilogx(x, y)')

ax = axs['log-log']
ax.loglog(x, y)
ax.set_xlabel('log')
ax.set_ylabel('log')
ax.set_title('loglog(x, y)')
```




### Other built-in scales

There are other scales that can be used. The list of registered scales can be returned from *scale.get_scale_names*:

```
print(mscale.get_scale_names())
```

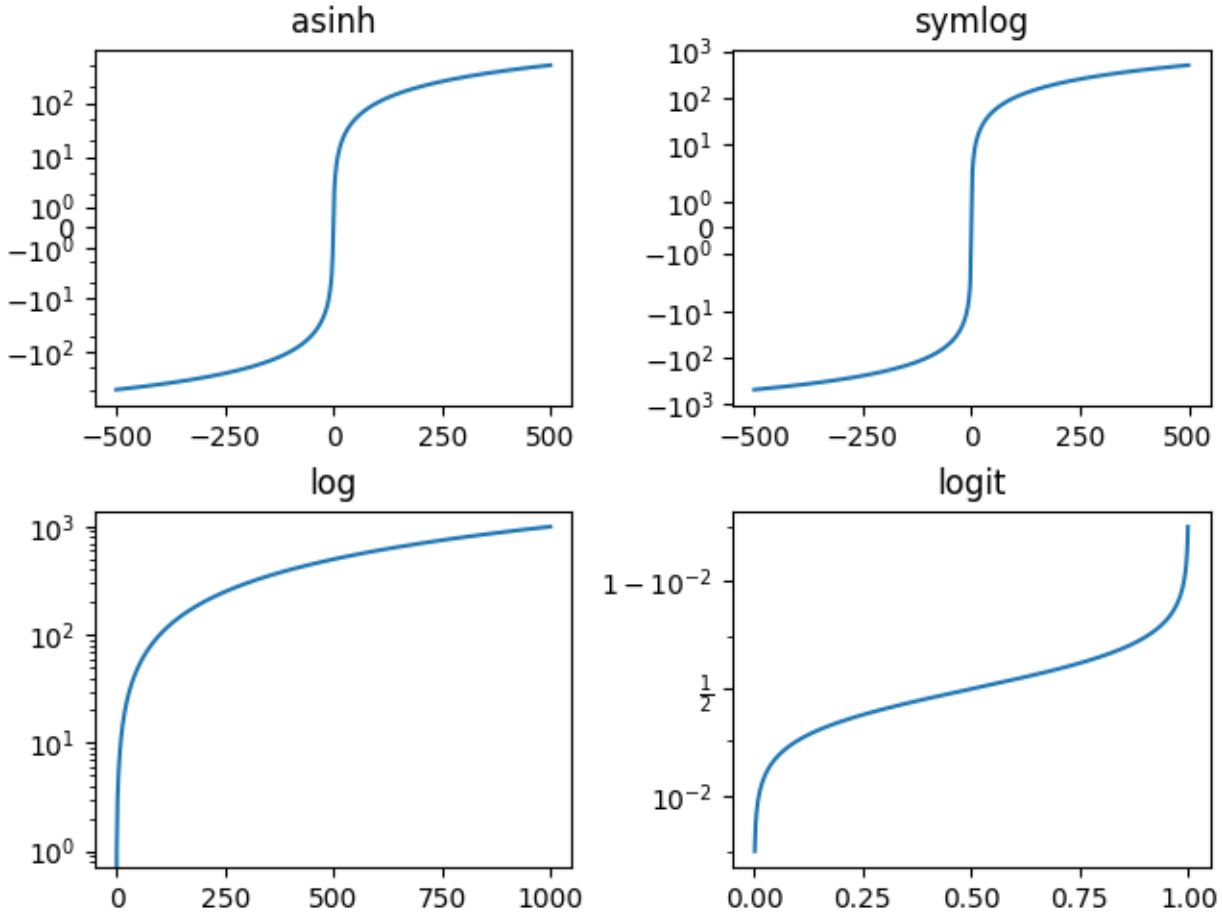
```
['asinh', 'function', 'functionlog', 'linear', 'log', 'logit', 'mercator',
 ↪'symlog']
```

```
fig, axs = plt.subplot_mosaic([['asinh', 'symlog'],
                               ['log', 'logit']], layout='constrained')

x = np.arange(0, 1000)

for name, ax in axs.items():
    if name in ['asinh', 'symlog']:
        yy = x - np.mean(x)
    elif name in ['logit']:
        yy = (x-np.min(x))
        yy = yy / np.max(np.abs(yy))
```

(continues on next page)

(continued from previous page)

```
    else:
        yy = x

    ax.plot(yy, yy)
    ax.set_yscale(name)
    ax.set_title(name)
```




### Optional arguments for scales

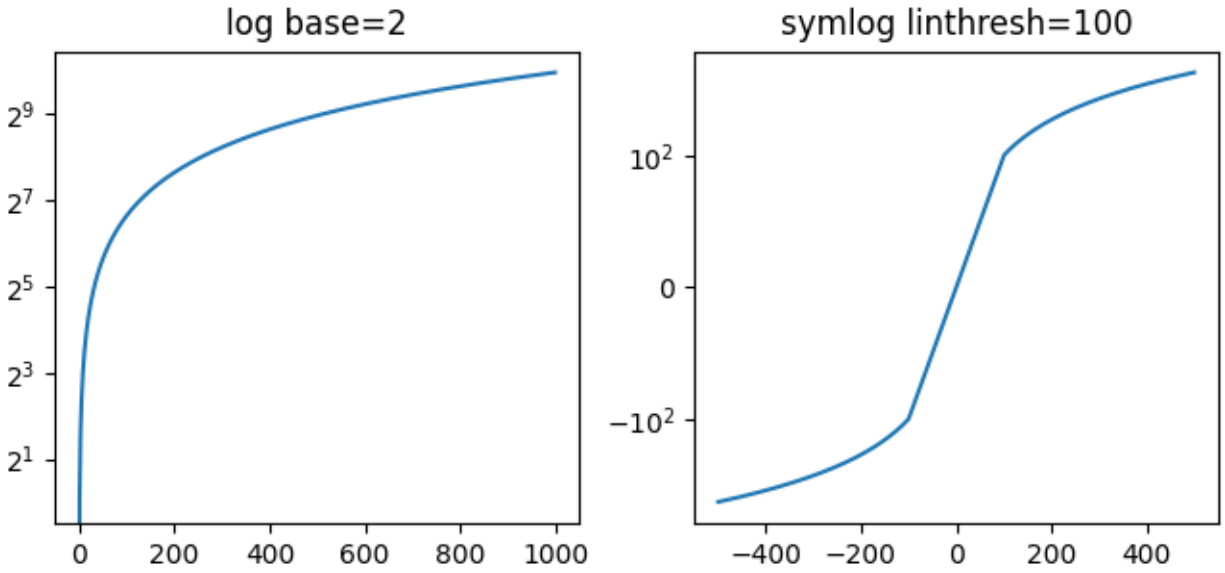
Some of the default scales have optional arguments. These are documented in the API reference for the respective scales at *scale*. One can change the base of the logarithm being plotted (eg 2 below) or the linear threshold range for `'symlog'`.

```
fig, axs = plt.subplot_mosaic([['log', 'symlog']], layout='constrained',
                              figsize=(6.4, 3))

for name, ax in axs.items():
    if name in ['log']:
        ax.plot(x, x)
        ax.set_yscale('log', base=2)
```

(continues on next page)

(continued from previous page)

```
        ax.set_title('log base=2')
    else:
        ax.plot(x - np.mean(x), x - np.mean(x))
        ax.set_yscale('symlog', linthresh=100)
        ax.set_title('symlog linthresh=100')
```




### Arbitrary function scales

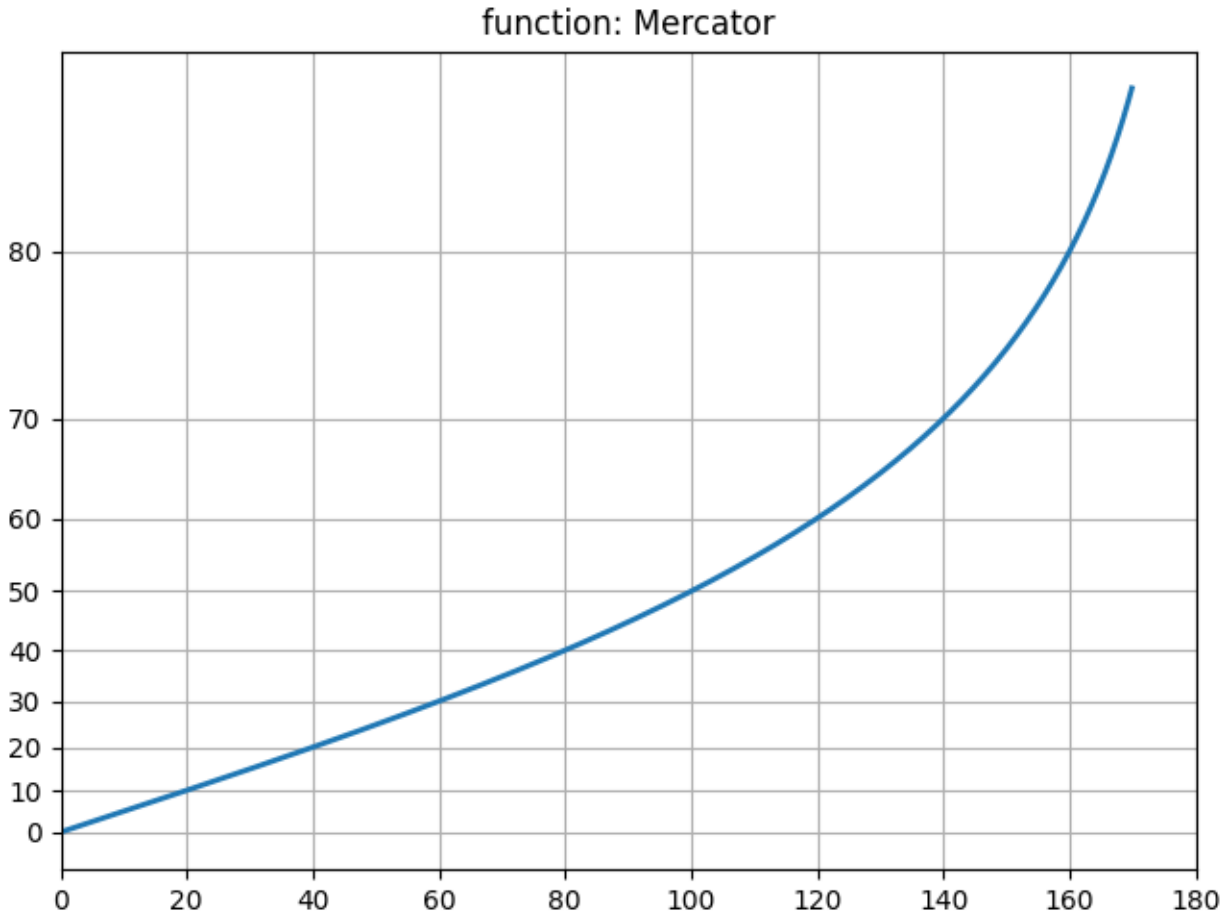
Users can define a full scale class and pass that to *set_xscale* and *set_yscale* (see *Custom scale*). A short cut for this is to use the 'function' scale, and pass as extra arguments a `forward` and an `inverse` function. The following performs a Mercator transform to the y-axis.

```
# Function Mercator transform
def forward(a):
    a = np.deg2rad(a)
    return np.rad2deg(np.log(np.abs(np.tan(a) + 1.0 / np.cos(a))))


def inverse(a):
    a = np.deg2rad(a)
    return np.rad2deg(np.arctan(np.sinh(a)))


t = np.arange(0, 170.0, 0.1)
s = t / 2.

fig, ax = plt.subplots(layout='constrained')
ax.plot(t, s, '-', lw=2)

ax.set_yscale('function', functions=(forward, inverse))
ax.set_title('function: Mercator')
```

(continues on next page)

(continued from previous page)

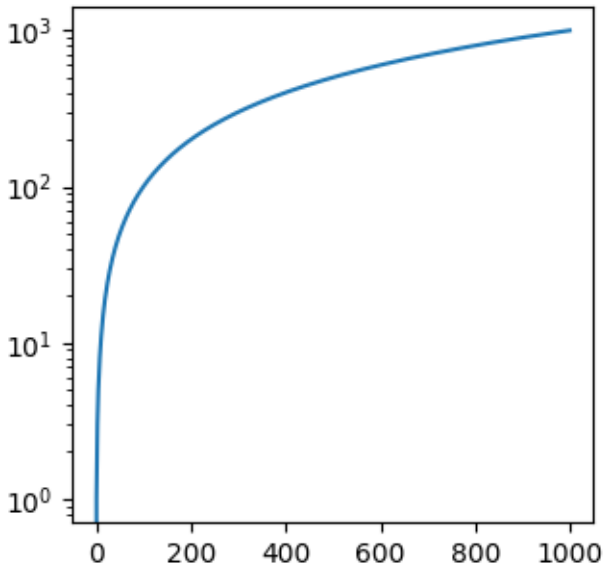
```
ax.grid(True)
ax.set_xlim([0, 180])
ax.yaxis.set_minor_formatter(NullFormatter())
ax.yaxis.set_major_locator(FixedLocator(np.arange(0, 90, 10)))
```




### What is a "scale"?

A scale is an object that gets attached to an axis. The class documentation is at *scale*. *set_xscale* and *set_yscale* set the scale on the respective Axis objects. You can determine the scale on an axis with *get_scale*:

```
fig, ax = plt.subplots(layout='constrained',
                               figsize=(3.2, 3))
ax.semilogy(x, x)

print(ax.xaxis.get_scale())
print(ax.yaxis.get_scale())
```




```
linear
log
```

Setting a scale does three things. First it defines a transform on the axis that maps between data values to position along the axis. This transform can be accessed via `get_transform`:

```
print(ax.yaxis.get_transform())
```

```
LogTransform(base=10, nonpositive='clip')
```

Transforms on the axis are a relatively low-level concept, but is one of the important roles played by `set_scale`.

Setting the scale also sets default tick locators (*`ticker`*) and tick formatters appropriate for the scale. An axis with a 'log' scale has a *`LogLocator`* to pick ticks at decade intervals, and a *`LogFormatter`* to use scientific notation on the decades.

```
print('X axis')
print(ax.xaxis.get_major_locator())
print(ax.xaxis.get_major_formatter())

print('Y axis')
print(ax.yaxis.get_major_locator())
print(ax.yaxis.get_major_formatter())
```

```
X axis
<matplotlib.ticker.AutoLocator object at 0x7f2bc4a32c60>
<matplotlib.ticker.ScalarFormatter object at 0x7f2bc473aff0>
Y axis
<matplotlib.ticker.LogLocator object at 0x7f2bbb5995e0>
<matplotlib.ticker.LogFormatterSciNotation object at 0x7f2bbb47d910>
```

**Total running time of the script:** (0 minutes 4.035 seconds)

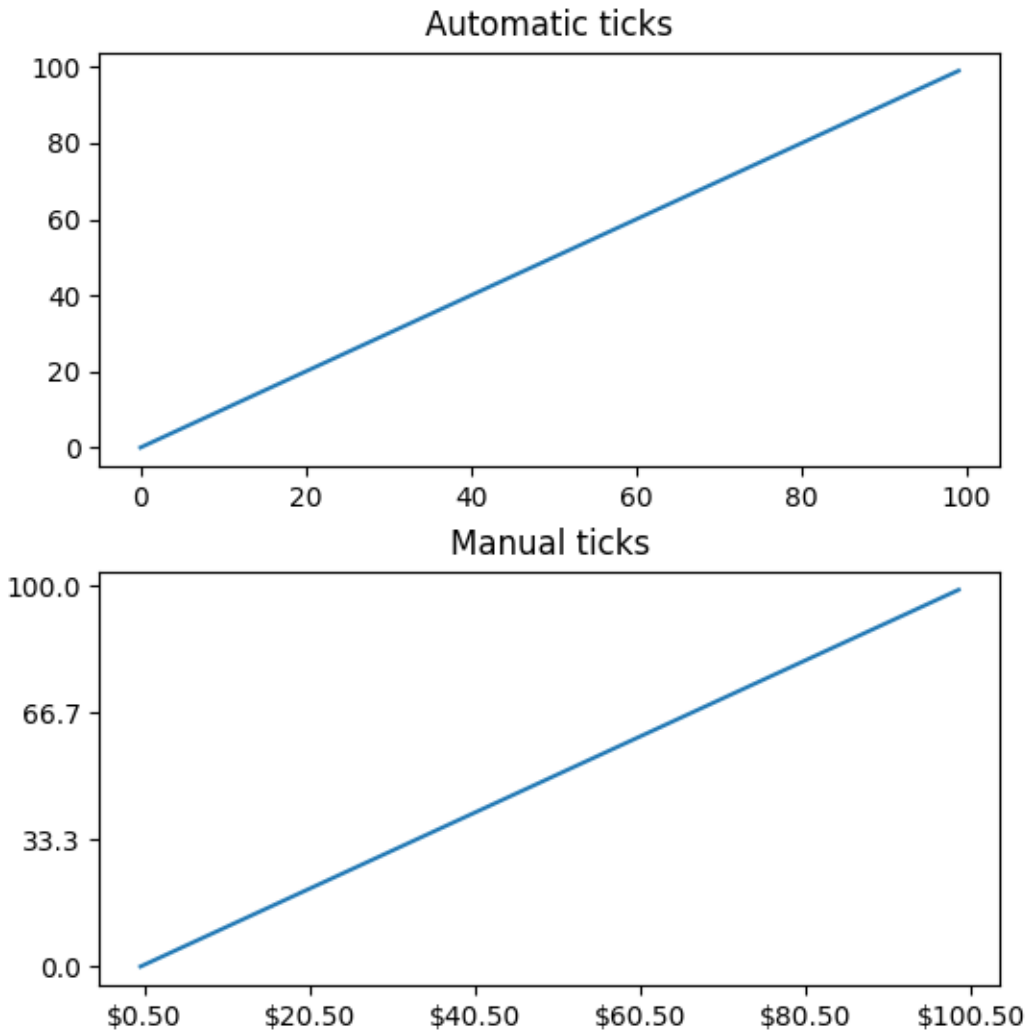
### 3.3.6 Axis ticks

The x and y Axis on each Axes have default tick "locators" and "formatters" that depend on the scale being used (see *Axis scales*). It is possible to customize the ticks and tick labels with either high-level methods like `set_xticks` or set the locators and formatters directly on the axis.

#### Manual location and formats

The simplest method to customize the tick locations and formats is to use `set_xticks` and `set_yticks`. These can be used on either the major or the minor ticks.

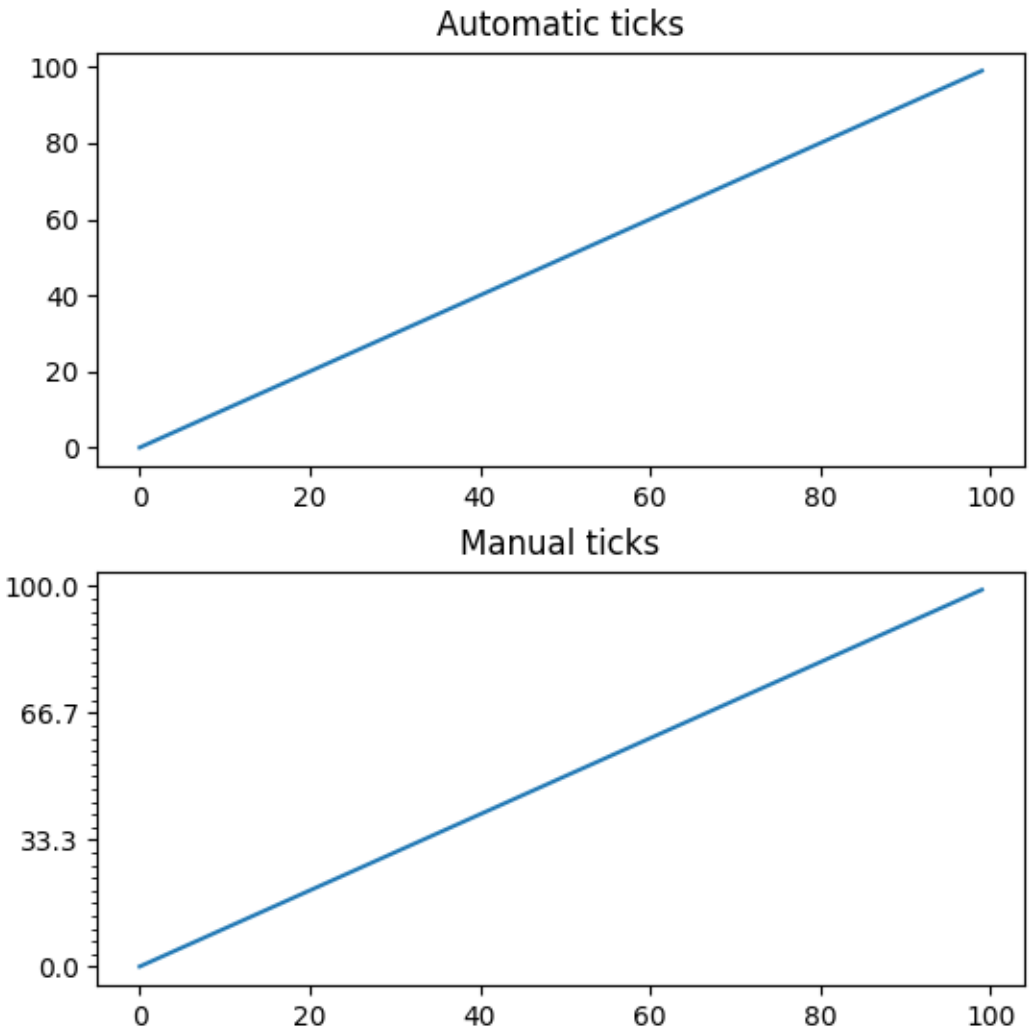
```
import numpy as np
import matplotlib.pyplot as plt

import matplotlib.ticker as ticker


fig, axs = plt.subplots(2, 1, figsize=(5.4, 5.4), layout='constrained')
x = np.arange(100)
for nn, ax in enumerate(axs):
    ax.plot(x, x)
    if nn == 1:
        ax.set_title('Manual ticks')
        ax.set_yticks(np.arange(0, 100.1, 100/3))
        xticks = np.arange(0.50, 101, 20)
        xlabels = [f'\\${x:1.2f}' for x in xticks]
        ax.set_xticks(xticks, labels=xlabels)
    else:
        ax.set_title('Automatic ticks')
```




Note that the length of the `labels` argument must have the same length as the array used to specify the ticks.

By default *`set_xticks`* and *`set_yticks`* act on the major ticks of an Axis, however it is possible to add minor ticks:

```
fig, axs = plt.subplots(2, 1, figsize=(5.4, 5.4), layout='constrained')
x = np.arange(100)
for nn, ax in enumerate(axs):
    ax.plot(x, x)
    if nn == 1:
        ax.set_title('Manual ticks')
        ax.set_yticks(np.arange(0, 100.1, 100/3))
        ax.set_yticks(np.arange(0, 100.1, 100/30), minor=True)
    else:
        ax.set_title('Automatic ticks')
```




### Locators and Formatters

Manually setting the ticks as above works well for specific final plots, but does not adapt as the user interacts with the Axes. At a lower level, Matplotlib has `Locators` that are meant to automatically choose ticks depending on the current view limits of the axis, and `Formatters` that are meant to format the tick labels automatically.

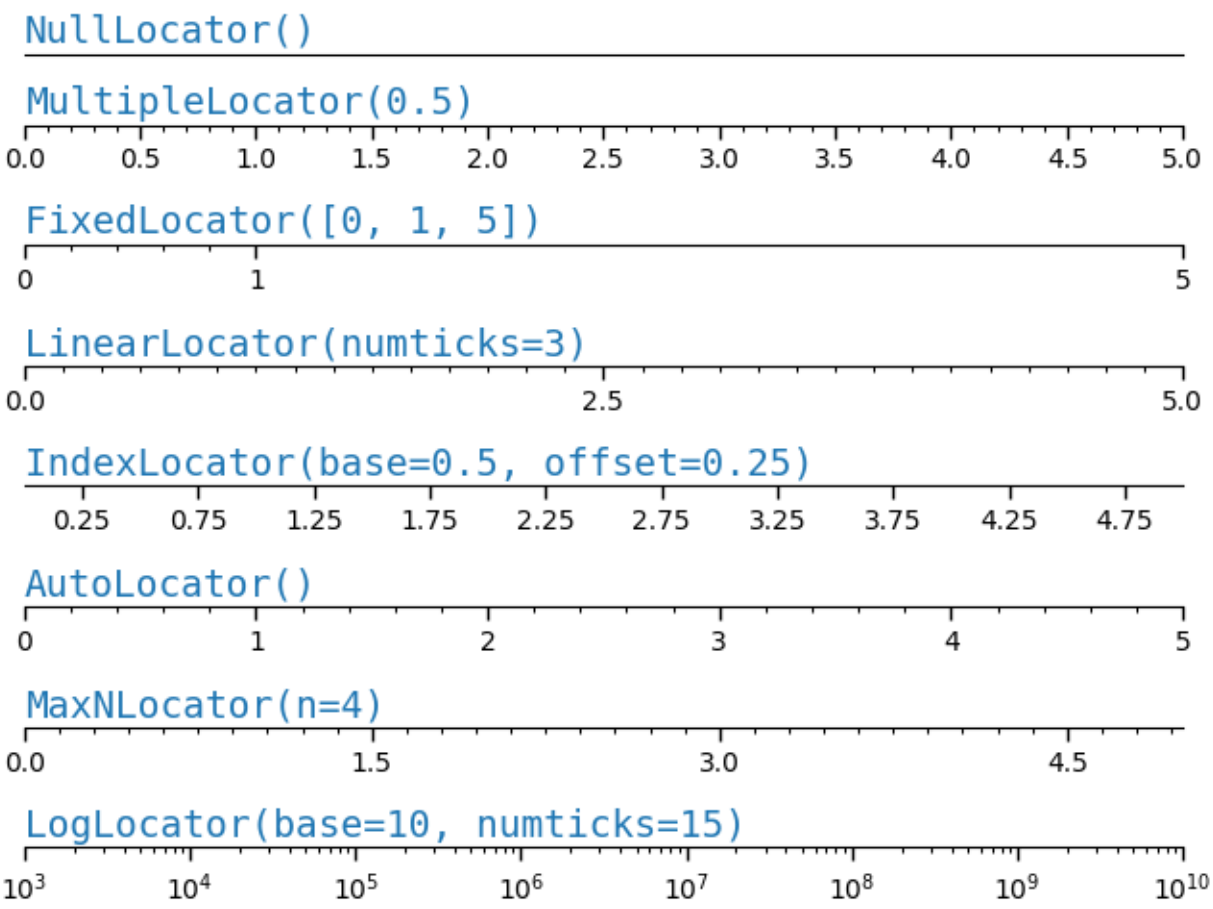
The full list of locators provided by Matplotlib are listed at *Tick locating*, and the formatters at *Tick formatting*.

```
def setup(ax, title):
    """Set up common parameters for the Axes in the example."""
    # only show the bottom spine
    ax.yaxis.set_major_locator(ticker.NullLocator())
    ax.spines[['left', 'right', 'top']].set_visible(False)

    ax.xaxis.set_ticks_position('bottom')
    ax.tick_params(which='major', width=1.00, length=5)
    ax.tick_params(which='minor', width=0.75, length=2.5)
```

(continues on next page)

(continued from previous page)

```
    ax.set_xlim(0, 5)
    ax.set_ylim(0, 1)
    ax.text(0.0, 0.2, title, transform=ax.transAxes,
            fontsize=14, fontname='Monospace', color='tab:blue')


fig, axs = plt.subplots(8, 1, layout='constrained')

# Null Locator
setup(axs[0], title="NullLocator()")
axs[0].xaxis.set_major_locator(ticker.NullLocator())
axs[0].xaxis.set_minor_locator(ticker.NullLocator())

# Multiple Locator
setup(axs[1], title="MultipleLocator(0.5)")
axs[1].xaxis.set_major_locator(ticker.MultipleLocator(0.5))
axs[1].xaxis.set_minor_locator(ticker.MultipleLocator(0.1))

# Fixed Locator
setup(axs[2], title="FixedLocator([0, 1, 5])")
axs[2].xaxis.set_major_locator(ticker.FixedLocator([0, 1, 5]))
axs[2].xaxis.set_minor_locator(ticker.FixedLocator(np.linspace(0.2, 0.8, 4)))

# Linear Locator
setup(axs[3], title="LinearLocator(numticks=3)")
axs[3].xaxis.set_major_locator(ticker.LinearLocator(3))
axs[3].xaxis.set_minor_locator(ticker.LinearLocator(31))

# Index Locator
setup(axs[4], title="IndexLocator(base=0.5, offset=0.25)")
axs[4].plot(range(0, 5), [0]*5, color='white')
axs[4].xaxis.set_major_locator(ticker.IndexLocator(base=0.5, offset=0.25))

# Auto Locator
setup(axs[5], title="AutoLocator()")
axs[5].xaxis.set_major_locator(ticker.AutoLocator())
axs[5].xaxis.set_minor_locator(ticker.AutoMinorLocator())

# MaxN Locator
setup(axs[6], title="MaxNLocator(n=4)")
axs[6].xaxis.set_major_locator(ticker.MaxNLocator(4))
axs[6].xaxis.set_minor_locator(ticker.MaxNLocator(40))

# Log Locator
setup(axs[7], title="LogLocator(base=10, numticks=15)")
axs[7].set_xlim(10**3, 10**10)
axs[7].set_xscale('log')
axs[7].xaxis.set_major_locator(ticker.LogLocator(base=10, numticks=15))
plt.show()
```




Similarly, we can specify "Formatters" for the major and minor ticks on each axis.

The tick format is configured via the function *set_major_formatter* or *set_minor_formatter*. It accepts:

- a format string, which implicitly creates a *StrMethodFormatter*.
- a function, implicitly creates a *FuncFormatter*.
- an instance of a *Formatter* subclass. The most common are
  - *NullFormatter*: No labels on the ticks.
  - *StrMethodFormatter*: Use string `str.format` method.
  - *FormatStrFormatter*: Use %-style formatting.
  - *FuncFormatter*: Define labels through a function.
  - *FixedFormatter*: Set the label strings explicitly.
  - *ScalarFormatter*: Default formatter for scalars: auto-pick the format string.
  - *PercentFormatter*: Format labels as a percentage.

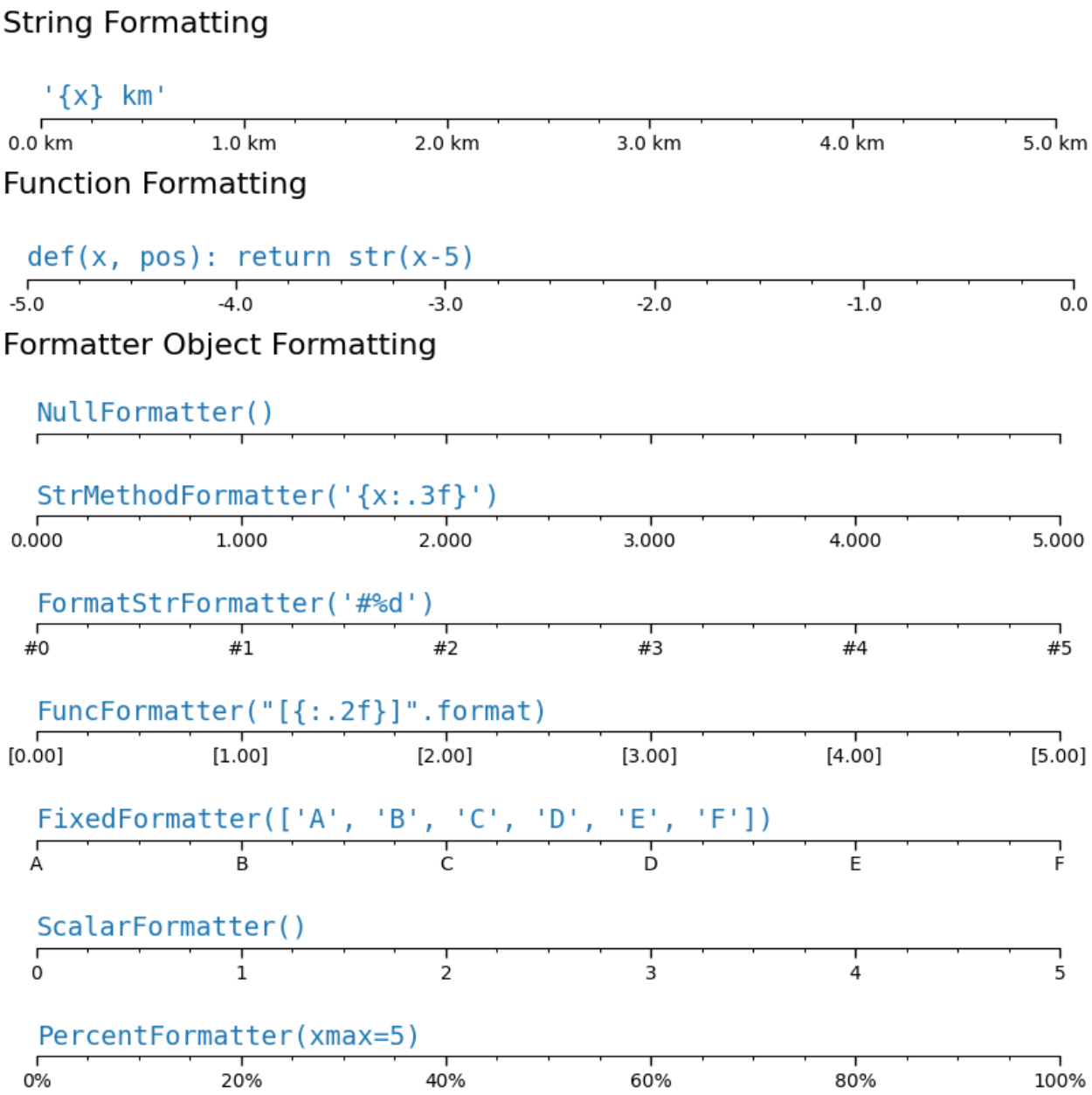
See *Tick formatting* for the complete list.

```
def setup(ax, title):
    """Set up common parameters for the Axes in the example."""
    # only show the bottom spine
    ax.yaxis.set_major_locator(ticker.NullLocator())
    ax.spines[['left', 'right', 'top']].set_visible(False)

    # define tick positions
    ax.xaxis.set_major_locator(ticker.MultipleLocator(1.00))
    ax.xaxis.set_minor_locator(ticker.MultipleLocator(0.25))

    ax.xaxis.set_ticks_position('bottom')
    ax.tick_params(which='major', width=1.00, length=5)
    ax.tick_params(which='minor', width=0.75, length=2.5, labelsize=10)
    ax.set_xlim(0, 5)
    ax.set_ylim(0, 1)
    ax.text(0.0, 0.2, title, transform=ax.transAxes,
            fontsize=14, fontname='Monospace', color='tab:blue')


fig = plt.figure(figsize=(8, 8), layout='constrained')
fig0, fig1, fig2 = fig.subfigures(3, height_ratios=[1.5, 1.5, 7.5])

fig0.suptitle('String Formatting', fontsize=16, x=0, ha='left')
ax0 = fig0.subplots()

setup(ax0, title="'{x} km'")
ax0.xaxis.set_major_formatter('{x} km')

fig1.suptitle('Function Formatting', fontsize=16, x=0, ha='left')
ax1 = fig1.subplots()

setup(ax1, title="def(x, pos): return str(x-5)")
ax1.xaxis.set_major_formatter(lambda x, pos: str(x-5))

fig2.suptitle('Formatter Object Formatting', fontsize=16, x=0, ha='left')
axs2 = fig2.subplots(7, 1)

setup(axs2[0], title="NullFormatter()")
axs2[0].xaxis.set_major_formatter(ticker.NullFormatter())

setup(axs2[1], title="StrMethodFormatter('{x:.3f}')")
axs2[1].xaxis.set_major_formatter(ticker.StrMethodFormatter("{x:.3f}"))

setup(axs2[2], title="FormatStrFormatter('#%d')")
axs2[2].xaxis.set_major_formatter(ticker.FormatStrFormatter("#%d"))


def fmt_two_digits(x, pos):
    return f'[{x:.2f}]'


setup(axs2[3], title='FuncFormatter("[{:.2f}]".format)')
```

(continues on next page)

(continued from previous page)

```
axs2[3].xaxis.set_major_formatter(ticker.FuncFormatter(fmt_two_digits))

setup(axs2[4], title="FixedFormatter(['A', 'B', 'C', 'D', 'E', 'F'])")
# FixedFormatter should only be used together with FixedLocator.
# Otherwise, one cannot be sure where the labels will end up.
positions = [0, 1, 2, 3, 4, 5]
labels = ['A', 'B', 'C', 'D', 'E', 'F']
axs2[4].xaxis.set_major_locator(ticker.FixedLocator(positions))
axs2[4].xaxis.set_major_formatter(ticker.FixedFormatter(labels))

setup(axs2[5], title="ScalarFormatter()")
axs2[5].xaxis.set_major_formatter(ticker.ScalarFormatter(useMathText=True))

setup(axs2[6], title="PercentFormatter(xmax=5)")
axs2[6].xaxis.set_major_formatter(ticker.PercentFormatter(xmax=5))
```




### Styling ticks (tick parameters)

The appearance of ticks can be controlled at a low level by finding the individual *Tick* on the axis. However, usually it is simplest to use *tick_params* to change all the objects at once.

The `tick_params` method can change the properties of ticks:

- length
- direction (in or out of the frame)
- colors
- width and length

- and whether the ticks are drawn at the bottom, top, left, or right of the Axes.

It also can control the tick labels:

- labelsize (fontsize)
- labelcolor (color of the label)
- labelrotation
- labelbottom, labeltop, labelleft, labelright

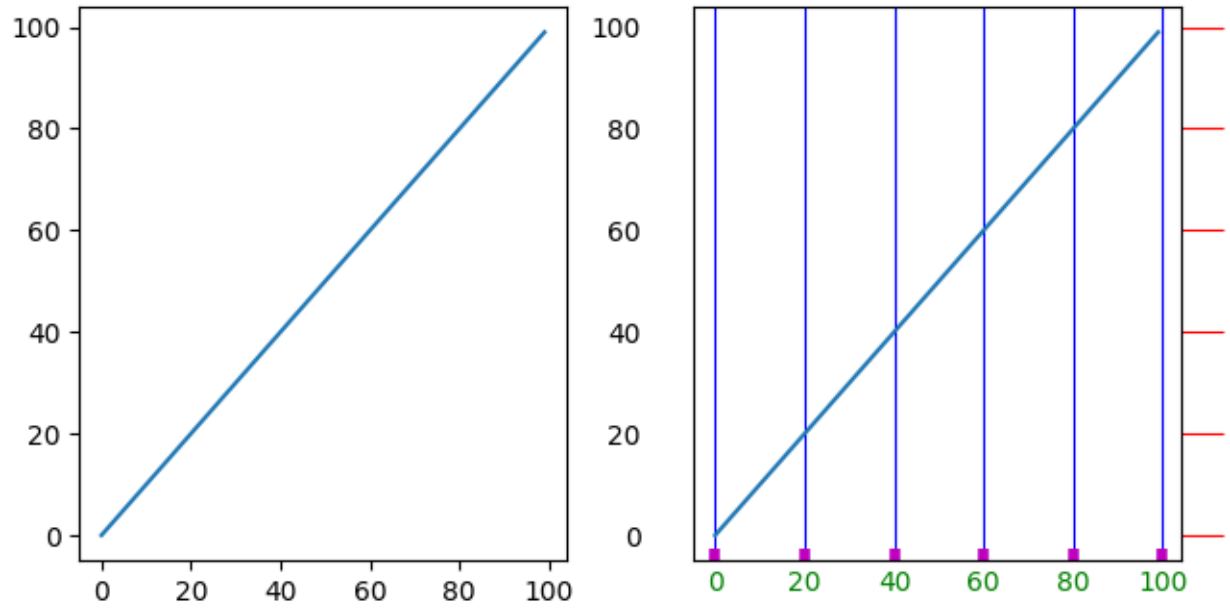
In addition there is a *pad* keyword argument that specifies how far the tick label is from the tick.

Finally, the grid linestyles can be set:

- grid_color
- grid_alpha
- grid_linewidth
- grid_linestyle

All these properties can be restricted to one axis, and can be applied to just the major or minor ticks

```
fig, axs = plt.subplots(1, 2, figsize=(6.4, 3.2), layout='constrained')

for nn, ax in enumerate(axs):
    ax.plot(np.arange(100))
    if nn == 1:
        ax.grid('on')
        ax.tick_params(right=True, left=False, axis='y', color='r', length=16,
                       grid_color='none')
        ax.tick_params(axis='x', color='m', length=4, direction='in', width=4,
                       labelcolor='g', grid_color='b')
```



**Total running time of the script:** (0 minutes 3.429 seconds)

### 3.3.7 Plotting dates and strings

The most basic way to use Matplotlib plotting methods is to pass coordinates in as numerical numpy arrays. For example, `plot(x, y)` will work if `x` and `y` are numpy arrays of floats (or integers). Plotting methods will also work if `numpy.asarray` will convert `x` and `y` to an array of floating point numbers; e.g. `x` could be a python list.

Matplotlib also has the ability to convert other data types if a "unit converter" exists for the data type. Matplotlib has two built-in converters, one for dates and the other for lists of strings. Other downstream libraries have their own converters to handle their data types.

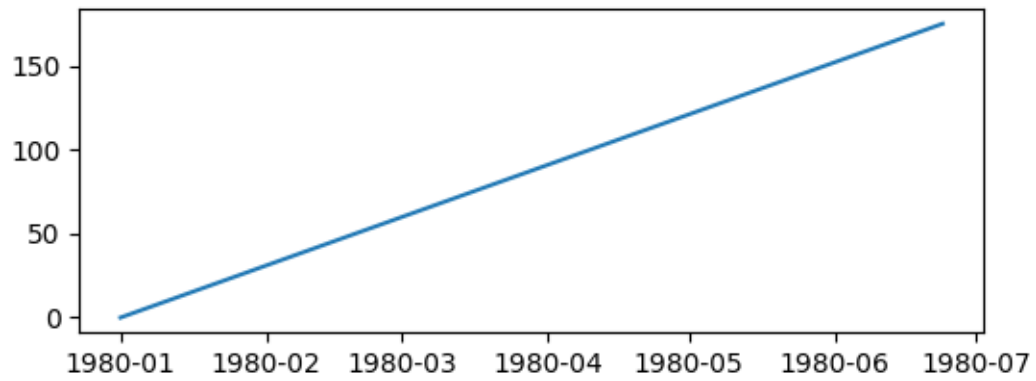
The method to add converters to Matplotlib is described in *matplotlib.units*. Here we briefly overview the built-in date and string converters.

#### Date conversion

If `x` and/or `y` are a list of `datetime` or an array of `numpy.datetime64`, Matplotlib has a built-in converter that will convert the datetime to a float, and add tick locators and formatters to the axis that are appropriate for dates. See *matplotlib.dates*.

In the following example, the x-axis gains a converter that converts from `numpy.datetime64` to float, and a locator that put ticks at the beginning of the month, and a formatter that label the ticks appropriately:

```
import numpy as np

import matplotlib.dates as mdates
import matplotlib.units as munits

import matplotlib.pyplot as plt

fig, ax = plt.subplots(figsize=(5.4, 2), layout='constrained')
time = np.arange('1980-01-01', '1980-06-25', dtype='datetime64[D]')
x = np.arange(len(time))
ax.plot(time, x)
```

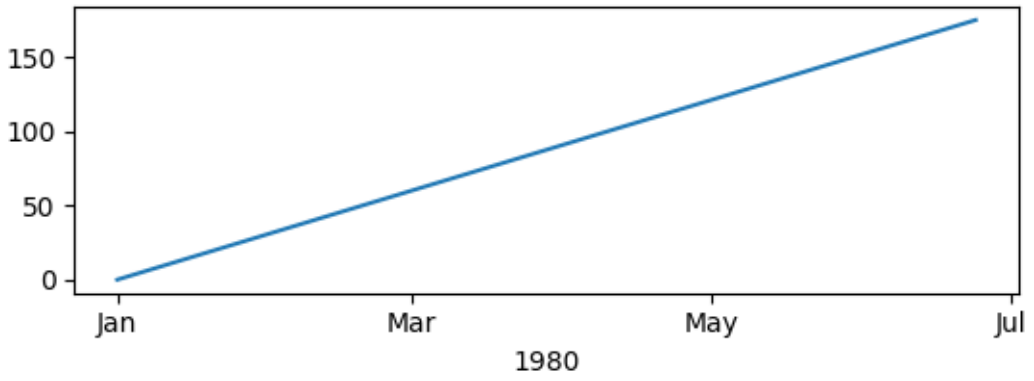


Note that if we try to plot a float on the x-axis, it will be plotted in units of days since the "epoch" for the converter, in this case 1970-01-01 (see *Matplotlib date format*). So when we plot the value 0, the ticks start at 1970-01-01. (The locator also now chooses every two years for a tick instead of every month):

```
fig, ax = plt.subplots(figsize=(5.4, 2), layout='constrained')
time = np.arange('1980-01-01', '1980-06-25', dtype='datetime64[D]')
x = np.arange(len(time))
ax.plot(time, x)
# 0 gets labeled as 1970-01-01
ax.plot(0, 0, 'd')
ax.text(0, 0, ' Float x=0', rotation=45)
```

We can customize the locator and the formatter; see *Date tick locators* and *Date formatters* for a complete list, and *Date tick locators and formatters* for examples of them in use. Here we locate by every second month, and format just with the month's 3-letter name using `"%b"` (see `strftime` for format codes):

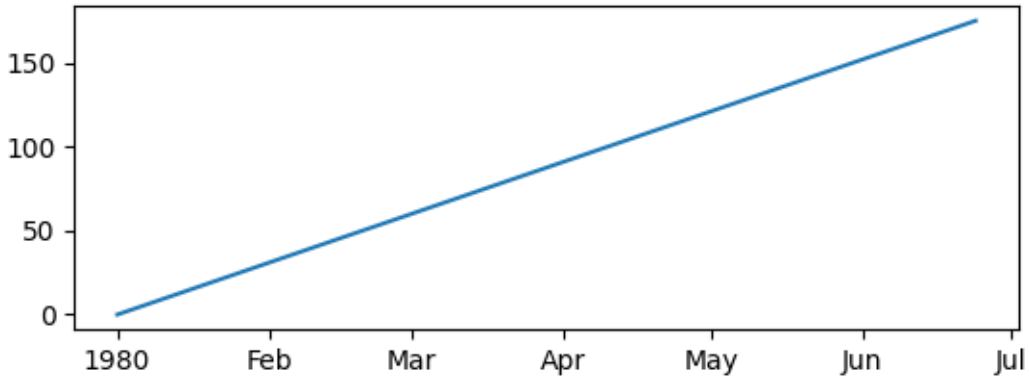
```
fig, ax = plt.subplots(figsize=(5.4, 2), layout='constrained')
time = np.arange('1980-01-01', '1980-06-25', dtype='datetime64[D]')
x = np.arange(len(time))
ax.plot(time, x)
ax.xaxis.set_major_locator(mdates.MonthLocator(bymonth=np.arange(1, 13, 2)))
ax.xaxis.set_major_formatter(mdates.DateFormatter('%b'))
ax.set_xlabel('1980')
```




The default locator is the `AutoDateLocator`, and the default Formatter `AutoDateFormatter`. There are also "concise" formatter and locators that give a more compact labelling, and can be set via rcParams. Note how instead of the redundant "Jan" label at the start of the year, "1980" is used instead. See *Formatting*

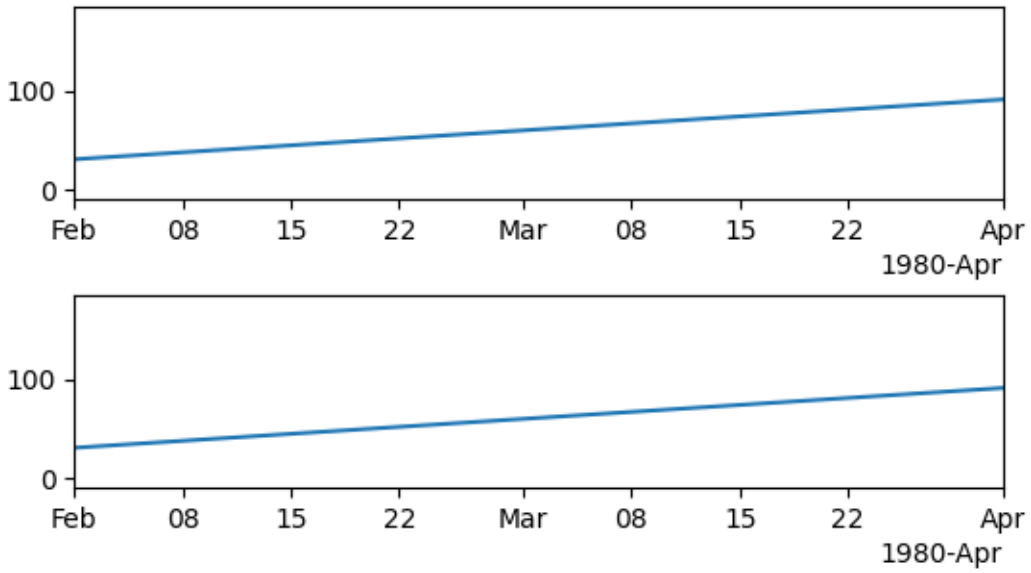
*date ticks using ConciseDateFormatter* for more examples.

```
plt.rcParams['date.converter'] = 'concise'

fig, ax = plt.subplots(figsize=(5.4, 2), layout='constrained')
time = np.arange('1980-01-01', '1980-06-25', dtype='datetime64[D]')
x = np.arange(len(time))
ax.plot(time, x)
```




We can set the limits on the axis either by passing the appropriate dates as limits, or by passing a floating-point value in the proper units of days since the epoch. If we need it, we can get this value from *date2num*.

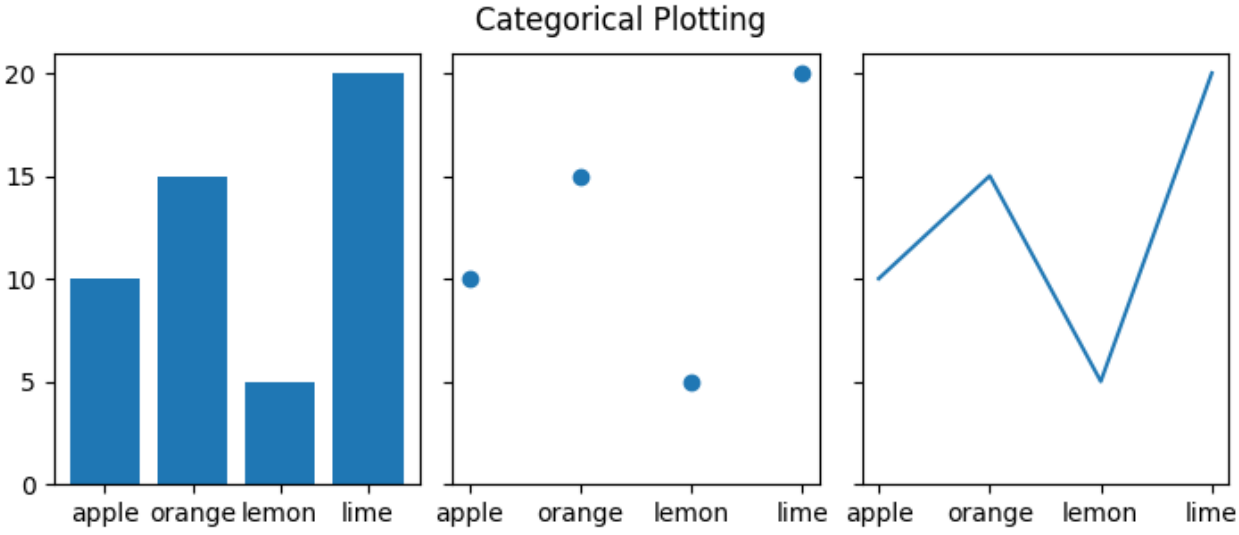
```
fig, axs = plt.subplots(2, 1, figsize=(5.4, 3), layout='constrained')
for ax in axs.flat:
    time = np.arange('1980-01-01', '1980-06-25', dtype='datetime64[D]')
    x = np.arange(len(time))
    ax.plot(time, x)

# set xlim using datetime64:
axs[0].set_xlim(np.datetime64('1980-02-01'), np.datetime64('1980-04-01'))

# set xlim using floats:
# Note can get from mdates.date2num(np.datetime64('1980-02-01'))
axs[1].set_xlim(3683, 3683+60)
```




### String conversion: categorical plots

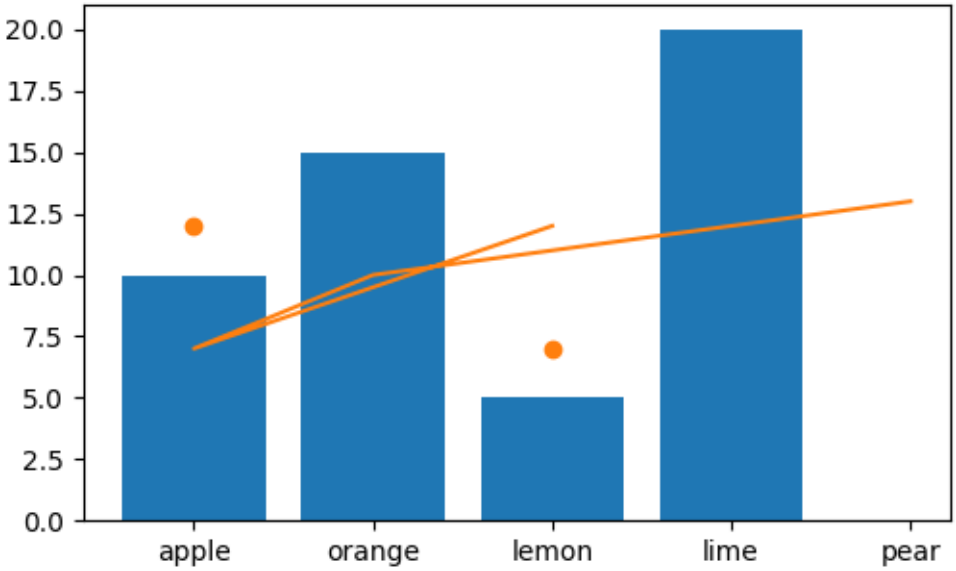
Sometimes we want to label categories on an axis rather than numbers. Matplotlib allows this using a "categorical" converter (see *category*).

```
data = {'apple': 10, 'orange': 15, 'lemon': 5, 'lime': 20}
names = list(data.keys())
values = list(data.values())

fig, axs = plt.subplots(1, 3, figsize=(7, 3), sharey=True, layout='constrained
 ↪')
axs[0].bar(names, values)
axs[1].scatter(names, values)
axs[2].plot(names, values)
fig.suptitle('Categorical Plotting')
```



Note that the "categories" are plotted in the order that they are first specified and that subsequent plotting in a different order will not affect the original order. Further, new additions will be added on the end (see "pear" below):

```
fig, ax = plt.subplots(figsize=(5, 3), layout='constrained')
ax.bar(names, values)

# plot in a different order:
ax.scatter(['lemon', 'apple'], [7, 12])

# add a new category, "pear", and put the other categories in a different␣
↪order:
ax.plot(['pear', 'orange', 'apple', 'lemon'], [13, 10, 7, 12], color='C1')
```




Note that when using `plot` like in the above, the order of the plotting is mapped onto the original order of the data, so the new line goes in the order specified.

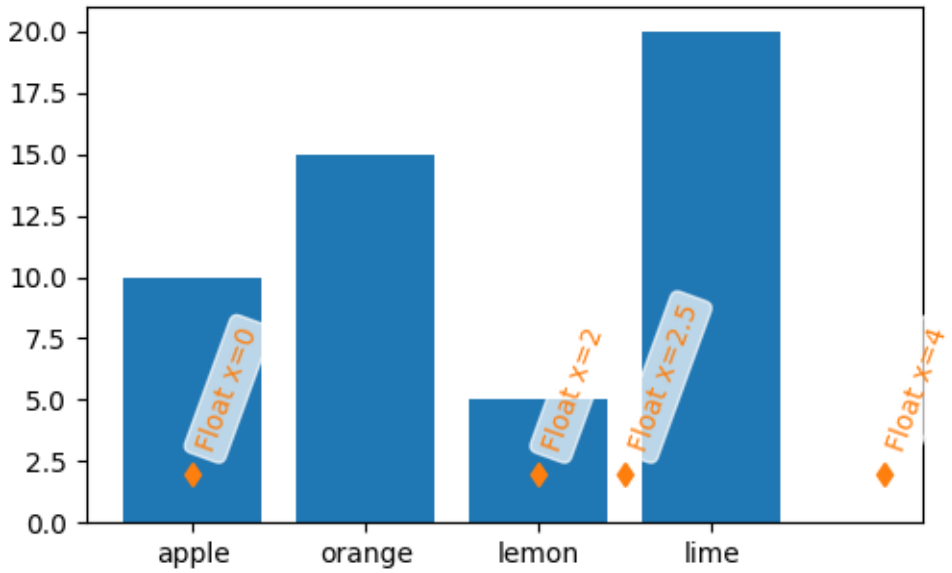
The category converter maps from categories to integers, starting at zero. So data can also be manually added to the axis using a float. Note that if a float is passed in that does not have a "category" associated with it, the data point can still be plotted, but a tick will not be created. In the following, we plot data at 4.0 and 2.5, but no tick is added there because those are not categories.

```
fig, ax = plt.subplots(figsize=(5, 3), layout='constrained')
ax.bar(names, values)
# arguments for styling the labels below:
args = {'rotation': 70, 'color': 'C1',
        'bbox': {'color': 'white', 'alpha': .7, 'boxstyle': 'round'}}


# 0 gets labeled as "apple"
ax.plot(0, 2, 'd', color='C1')
ax.text(0, 3, 'Float x=0', **args)

# 2 gets labeled as "lemon"
```

(continues on next page)

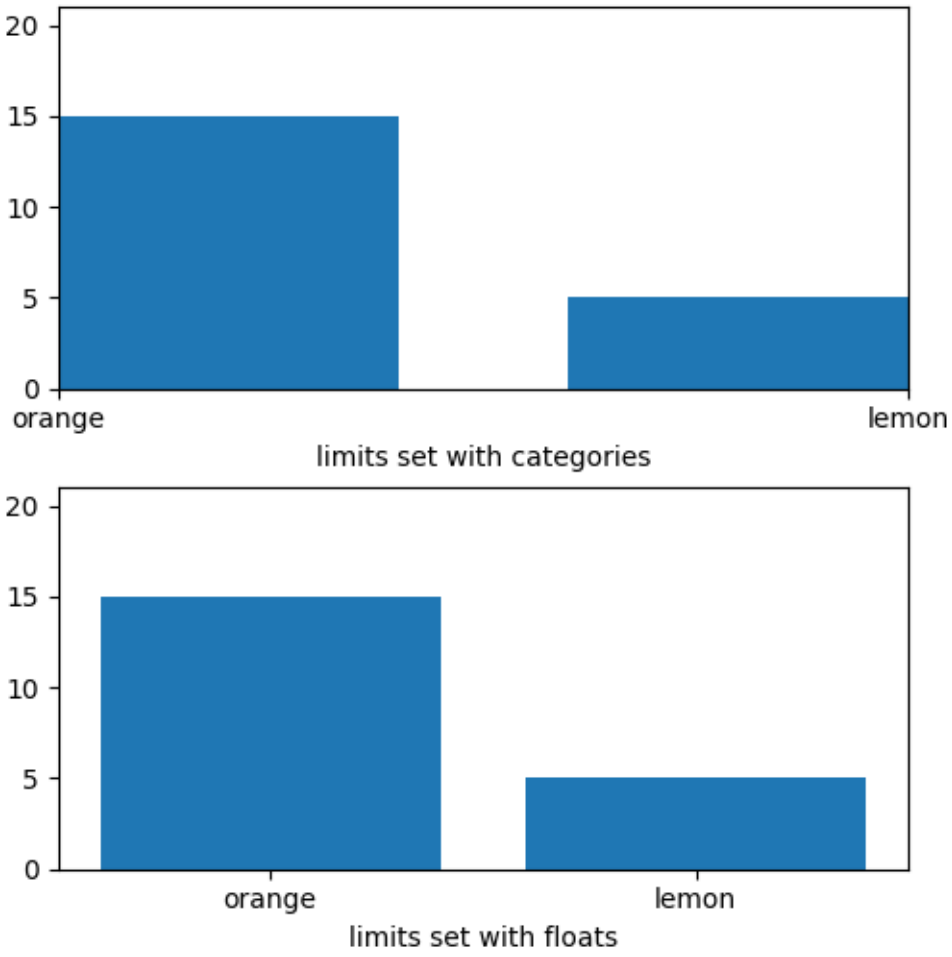
(continued from previous page)

```
ax.plot(2, 2, 'd', color='C1')
ax.text(2, 3, 'Float x=2', **args)

# 4 doesn't get a label
ax.plot(4, 2, 'd', color='C1')
ax.text(4, 3, 'Float x=4', **args)

# 2.5 doesn't get a label
ax.plot(2.5, 2, 'd', color='C1')
ax.text(2.5, 3, 'Float x=2.5', **args)
```



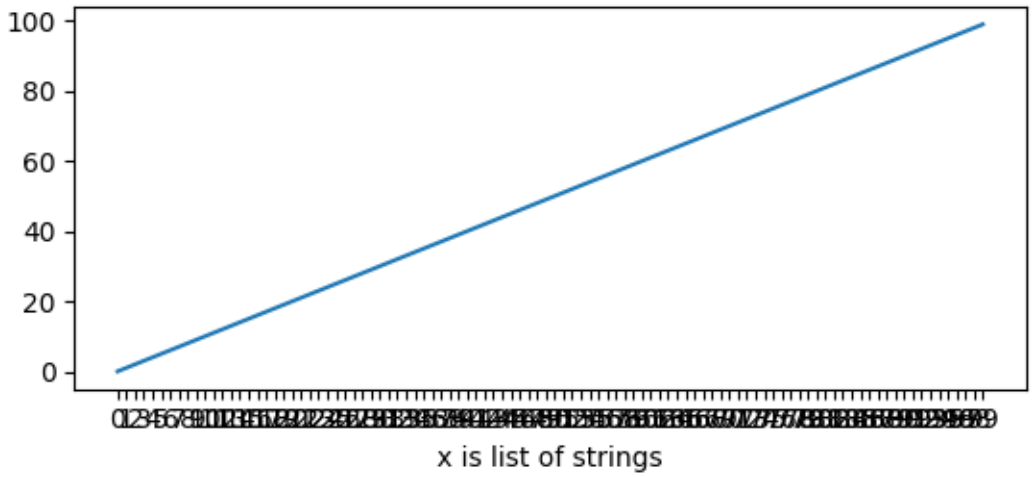

Setting the limits for a category axis can be done by specifying the categories, or by specifying floating point numbers:

```
fig, axs = plt.subplots(2, 1, figsize=(5, 5), layout='constrained')
ax = axs[0]
ax.bar(names, values)
ax.set_xlim('orange', 'lemon')
ax.set_xlabel('limits set with categories')
ax = axs[1]
ax.bar(names, values)
ax.set_xlim(0.5, 2.5)
ax.set_xlabel('limits set with floats')
```



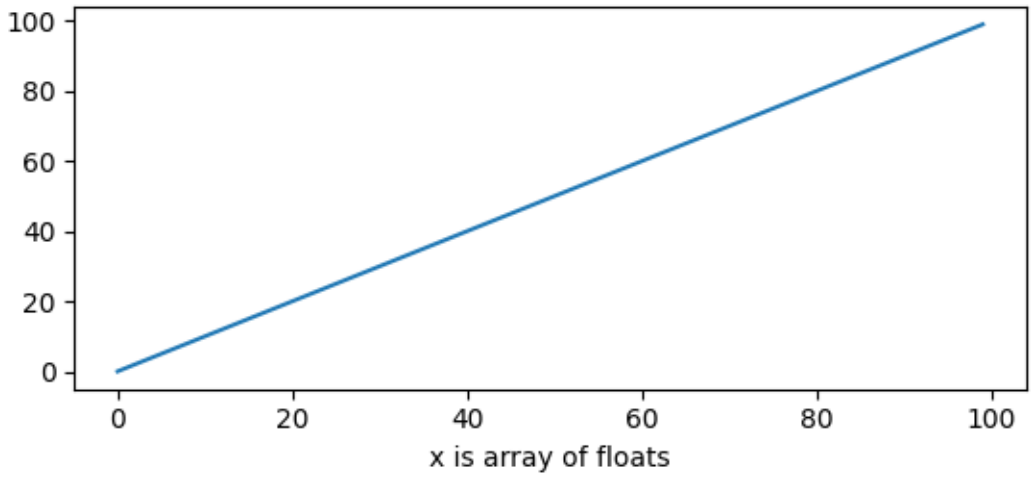

The category axes are helpful for some plot types, but can lead to confusion if data is read in as a list of strings, even if it is meant to be a list of floats or dates. This sometimes happens when reading comma-separated value (CSV) files. The categorical locator and formatter will put a tick at every string value and label each one as well:

```
fig, ax = plt.subplots(figsize=(5.4, 2.5), layout='constrained')
x = [str(xx) for xx in np.arange(100)]  # list of strings
ax.plot(x, np.arange(100))
ax.set_xlabel('x is list of strings')
```




If this is not desired, then simply convert the data to floats before plotting:

```
fig, ax = plt.subplots(figsize=(5.4, 2.5), layout='constrained')
x = np.asarray(x, dtype='float')  # array of float.
ax.plot(x, np.arange(100))
ax.set_xlabel('x is array of floats')
```




### Determine converter, formatter, and locator on an axis

Sometimes it is helpful to be able to debug what Matplotlib is using to convert the incoming data. We can do that by querying the `converter` property on the axis. We can also query the formatters and locators using *`get_major_locator`* and *`get_major_formatter`*.

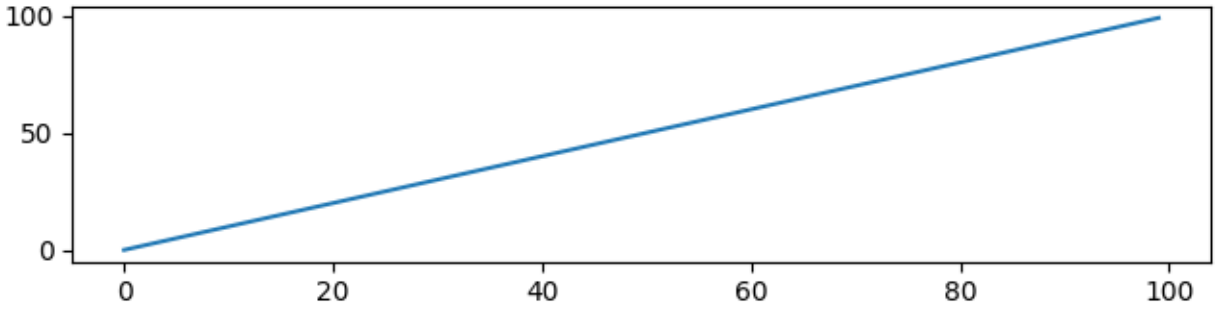
Note that by default the converter is *None*.

```
fig, axs = plt.subplots(3, 1, figsize=(6.4, 7), layout='constrained')
x = np.arange(100)
ax = axs[0]
ax.plot(x, x)
label = f'Converter: {ax.xaxis.converter}\n '
```

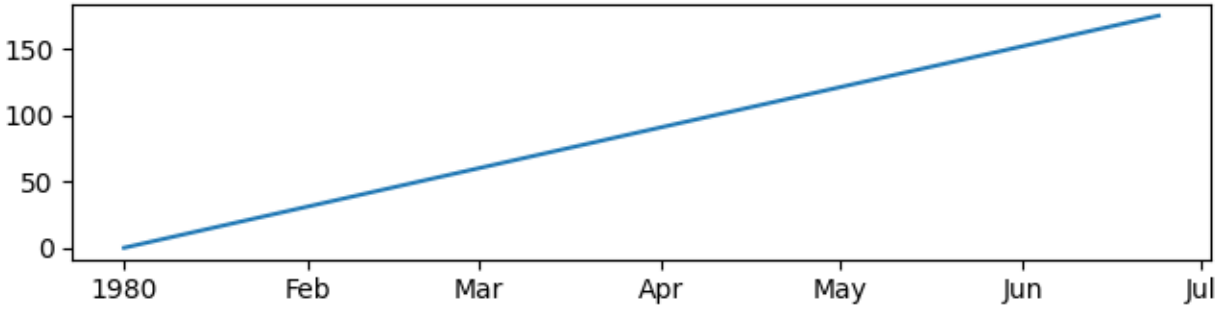
(continues on next page)

(continued from previous page)

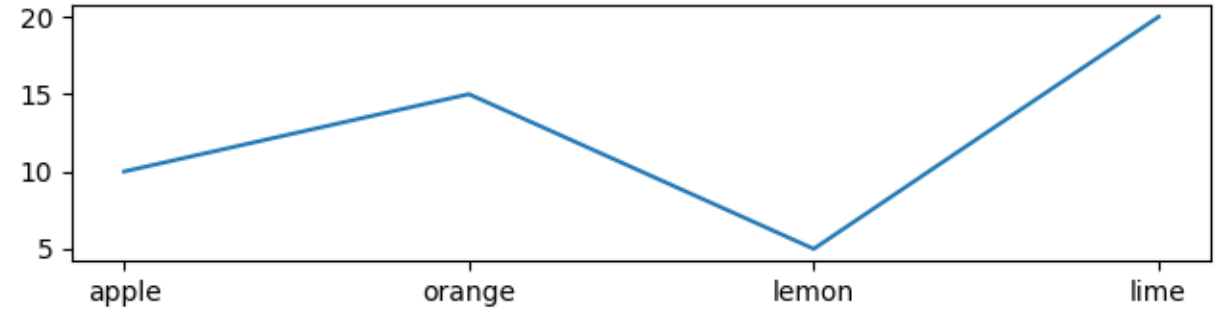
```
label += f'Locator: {ax.xaxis.get_major_locator()}\n'
label += f'Formatter: {ax.xaxis.get_major_formatter()}\n'
ax.set_xlabel(label)

ax = axs[1]
time = np.arange('1980-01-01', '1980-06-25', dtype='datetime64[D]')
x = np.arange(len(time))
ax.plot(time, x)
label = f'Converter: {ax.xaxis.converter}\n '
label += f'Locator: {ax.xaxis.get_major_locator()}\n'
label += f'Formatter: {ax.xaxis.get_major_formatter()}\n'
ax.set_xlabel(label)

ax = axs[2]
data = {'apple': 10, 'orange': 15, 'lemon': 5, 'lime': 20}
names = list(data.keys())
values = list(data.values())
ax.plot(names, values)
label = f'Converter: {ax.xaxis.converter}\n '
label += f'Locator: {ax.xaxis.get_major_locator()}\n'
label += f'Formatter: {ax.xaxis.get_major_formatter()}\n'
ax.set_xlabel(label)
```




Converter: None
Locator: <matplotlib.ticker.AutoLocator object at 0x7f2bbb599700>
Formatter: <matplotlib.ticker.ScalarFormatter object at 0x7f2bc4499a30>




Converter: <matplotlib.dates._SwitchableDateConverter object at 0x7f2bba956180>
Locator: <matplotlib.dates.AutoDateLocator object at 0x7f2bc47e7290>
Formatter: <matplotlib.dates.ConciseDateFormatter object at 0x7f2bbbc790a0>




Converter: <matplotlib.category.StrCategoryConverter object at 0x7f2bc47466c0>
Locator: <matplotlib.category.StrCategoryLocator object at 0x7f2bc50e9220>
Formatter: <matplotlib.category.StrCategoryFormatter object at 0x7f2bbaca0920>

### More about "unit" support

The support for dates and categories is part of "units" support that is built into Matplotlib. This is described at *matplotlib.units* and in the *Basic Units* example.

Unit support works by querying the type of data passed to the plotting function and dispatching to the first converter in a list that accepts that type of data. So below, if `x` has `datetime` objects in it, the converter will be `_SwitchableDateConverter`; if it has has strings in it, it will be sent to the `StrCategoryConverter`.

```
for k, v in munits.registry.items():
    print(f"type: {k};\n    converter: {type(v)}")
```

```
type: <class 'decimal.Decimal'>;
    converter: <class 'matplotlib.units.DecimalConverter'>
type: <class 'numpy.datetime64'>;
    converter: <class 'matplotlib.dates._SwitchableDateConverter'>
type: <class 'datetime.date'>;
    converter: <class 'matplotlib.dates._SwitchableDateConverter'>
type: <class 'datetime.datetime'>;
    converter: <class 'matplotlib.dates._SwitchableDateConverter'>
type: <class 'str'>;
    converter: <class 'matplotlib.category.StrCategoryConverter'>
type: <class 'numpy.str_'>;
    converter: <class 'matplotlib.category.StrCategoryConverter'>
type: <class 'bytes'>;
    converter: <class 'matplotlib.category.StrCategoryConverter'>
type: <class 'numpy.bytes_'>;
    converter: <class 'matplotlib.category.StrCategoryConverter'>
```

There are a number of downstream libraries that provide their own converters with locators and formatters. Physical unit support is provided by astropy, pint, and unyt, among others.

High level libraries like pandas and nc-time-axis (and thus xarray) provide their own datetime support. This support can sometimes be incompatible with Matplotlib native datetime support, so care should be taken when using Matplotlib locators and formatters if these libraries are being used.

**Total running time of the script:** (0 minutes 3.607 seconds)

## 3.3.8 Legend guide

This legend guide extends the *legend* docstring - please read it before proceeding with this guide.

This guide makes use of some common terms, which are documented here for clarity:

**legend entry**

> A legend is made up of one or more legend entries. An entry is made up of exactly one key and one label.

**legend key**

> The colored/patterned marker to the left of each legend label.

**legend label**

The text which describes the handle represented by the key.

**legend handle**

The original object which is used to generate an appropriate entry in the legend.

### Controlling the legend entries

Calling *legend()* with no arguments automatically fetches the legend handles and their associated labels. This functionality is equivalent to:

```
handles, labels = ax.get_legend_handles_labels()
ax.legend(handles, labels)
```

The *get_legend_handles_labels()* function returns a list of handles/artists which exist on the Axes which can be used to generate entries for the resulting legend - it is worth noting however that not all artists can be added to a legend, at which point a "proxy" will have to be created (see *Creating artists specifically for adding to the legend (aka. Proxy artists)* for further details).

**Note:** Artists with an empty string as label or with a label starting with an underscore, "_", will be ignored.

For full control of what is being added to the legend, it is common to pass the appropriate handles directly to *legend()*:

```
fig, ax = plt.subplots()
line_up, = ax.plot([1, 2, 3], label='Line 2')
line_down, = ax.plot([3, 2, 1], label='Line 1')
ax.legend(handles=[line_up, line_down])
```

### Renaming legend entries

When the labels cannot directly be set on the handles, they can be directly passed to *Axes.legend*:

```
fig, ax = plt.subplots()
line_up, = ax.plot([1, 2, 3], label='Line 2')
line_down, = ax.plot([3, 2, 1], label='Line 1')
ax.legend([line_up, line_down], ['Line Up', 'Line Down'])
```

If the handles are not directly accessible, for example when using some Third-party packages, they can be accessed via *Axes.get_legend_handles_labels*. Here we use a dictionary to rename existing labels:

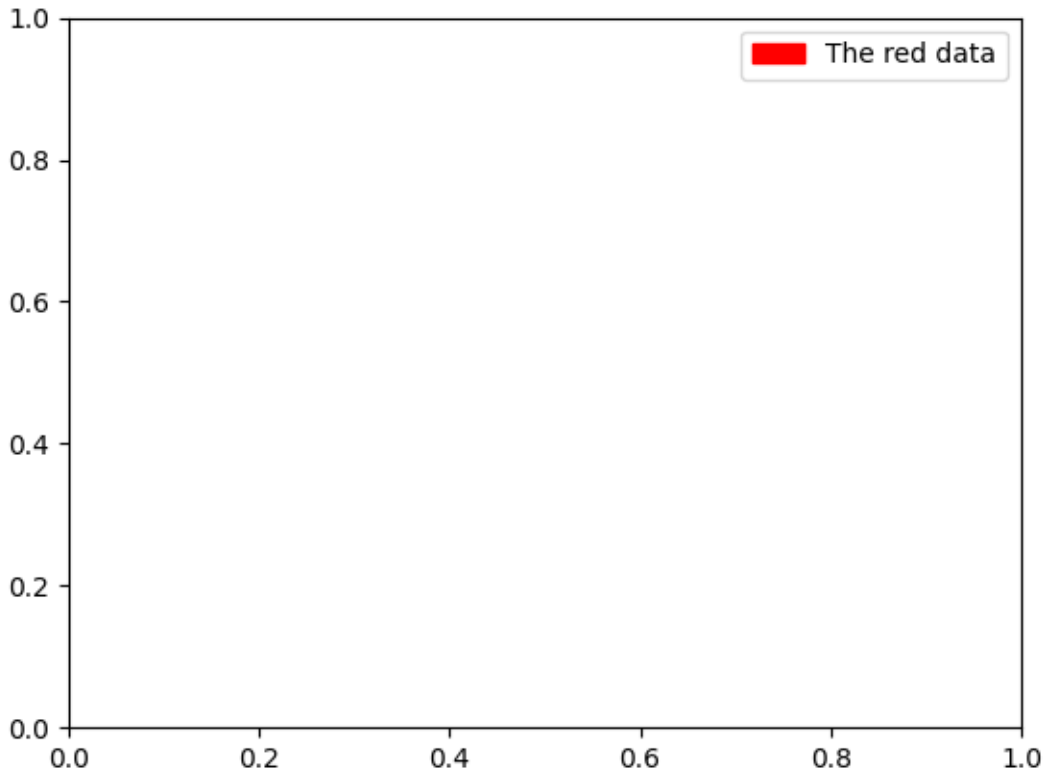
```
my_map = {'Line Up':'Up', 'Line Down':'Down'}

handles, labels = ax.get_legend_handles_labels()
ax.legend(handles, [my_map[l] for l in labels])
```

### Creating artists specifically for adding to the legend (aka. Proxy artists)

Not all handles can be turned into legend entries automatically, so it is often necessary to create an artist which *can*. Legend handles don't have to exist on the Figure or Axes in order to be used.

Suppose we wanted to create a legend which has an entry for some data which is represented by a red color:

```
import matplotlib.pyplot as plt

import matplotlib.patches as mpatches

fig, ax = plt.subplots()
red_patch = mpatches.Patch(color='red', label='The red data')
ax.legend(handles=[red_patch])

plt.show()
```



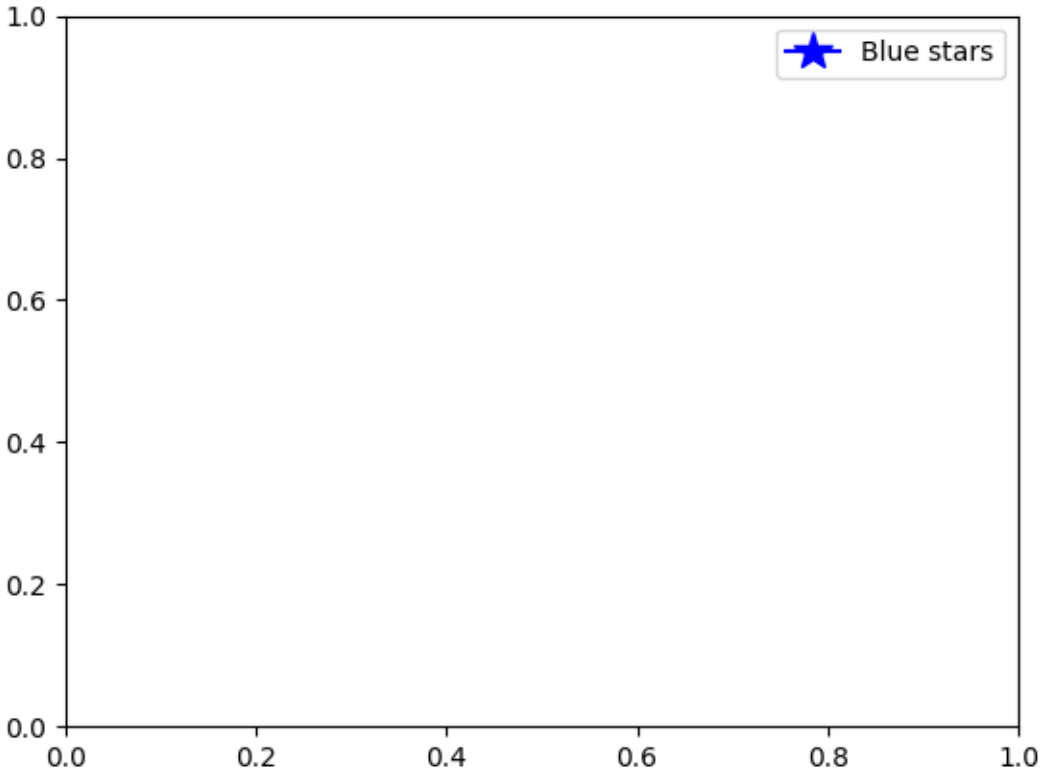

There are many supported legend handles. Instead of creating a patch of color we could have created a line with a marker:

```
import matplotlib.lines as mlines

fig, ax = plt.subplots()
```

(continues on next page)

(continued from previous page)

```
blue_line = mlines.Line2D([], [], color='blue', marker='*',
                          markersize=15, label='Blue stars')
ax.legend(handles=[blue_line])

plt.show()
```




### Legend location

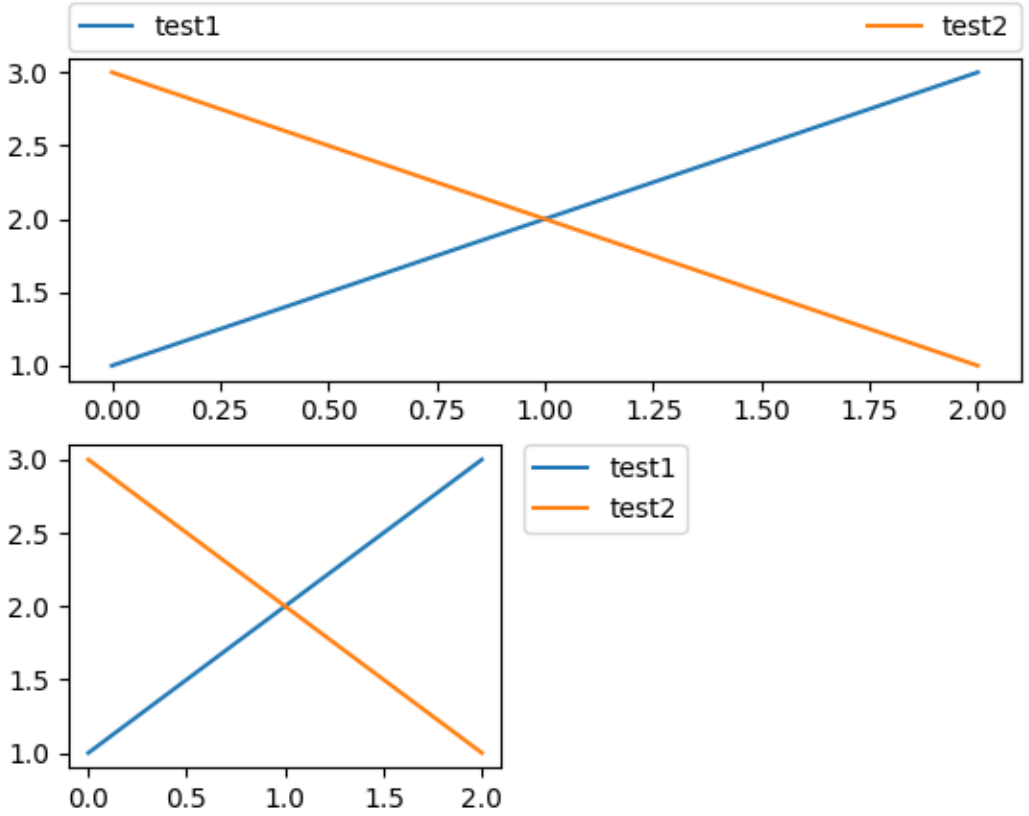
The location of the legend can be specified by the keyword argument *loc*. Please see the documentation at `legend()` for more details.

The `bbox_to_anchor` keyword gives a great degree of control for manual legend placement. For example, if you want your Axes legend located at the figure's top right-hand corner instead of the Axes' corner, simply specify the corner's location and the coordinate system of that location:

```
ax.legend(bbox_to_anchor=(1, 1),
          bbox_transform=fig.transFigure)
```

More examples of custom legend placement:

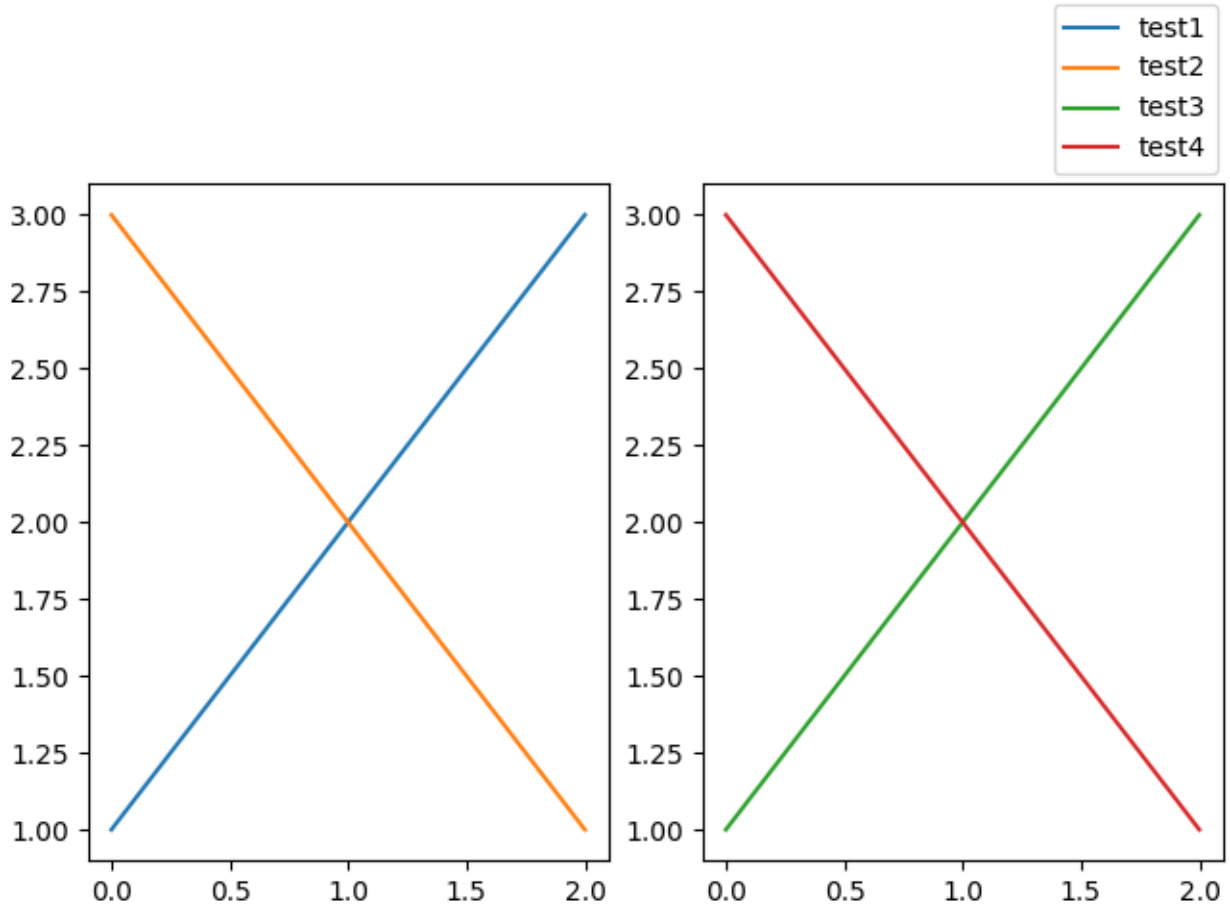
```
fig, ax_dict = plt.subplot_mosaic([['top', 'top'], ['bottom', 'BLANK']],
                                  empty_sentinel="BLANK")
ax_dict['top'].plot([1, 2, 3], label="test1")
ax_dict['top'].plot([3, 2, 1], label="test2")
# Place a legend above this subplot, expanding itself to
# fully use the given bounding box.
ax_dict['top'].legend(bbox_to_anchor=(0., 1.02, 1., .102), loc='lower left',
                      ncols=2, mode="expand", borderaxespad=0.)

ax_dict['bottom'].plot([1, 2, 3], label="test1")
ax_dict['bottom'].plot([3, 2, 1], label="test2")
# Place a legend to the right of this smaller subplot.
ax_dict['bottom'].legend(bbox_to_anchor=(1.05, 1),
                         loc='upper left', borderaxespad=0.)
```



### Figure legends

Sometimes it makes more sense to place a legend relative to the (sub)figure rather than individual Axes. By using *constrained layout* and specifying "outside" at the beginning of the *loc* keyword argument, the legend is drawn outside the Axes on the (sub)figure.

```
fig, axs = plt.subplot_mosaic([['left', 'right']], layout='constrained')

axs['left'].plot([1, 2, 3], label="test1")
axs['left'].plot([3, 2, 1], label="test2")

axs['right'].plot([1, 2, 3], 'C2', label="test3")
axs['right'].plot([3, 2, 1], 'C3', label="test4")
# Place a legend to the right of this smaller subplot.
fig.legend(loc='outside upper right')
```



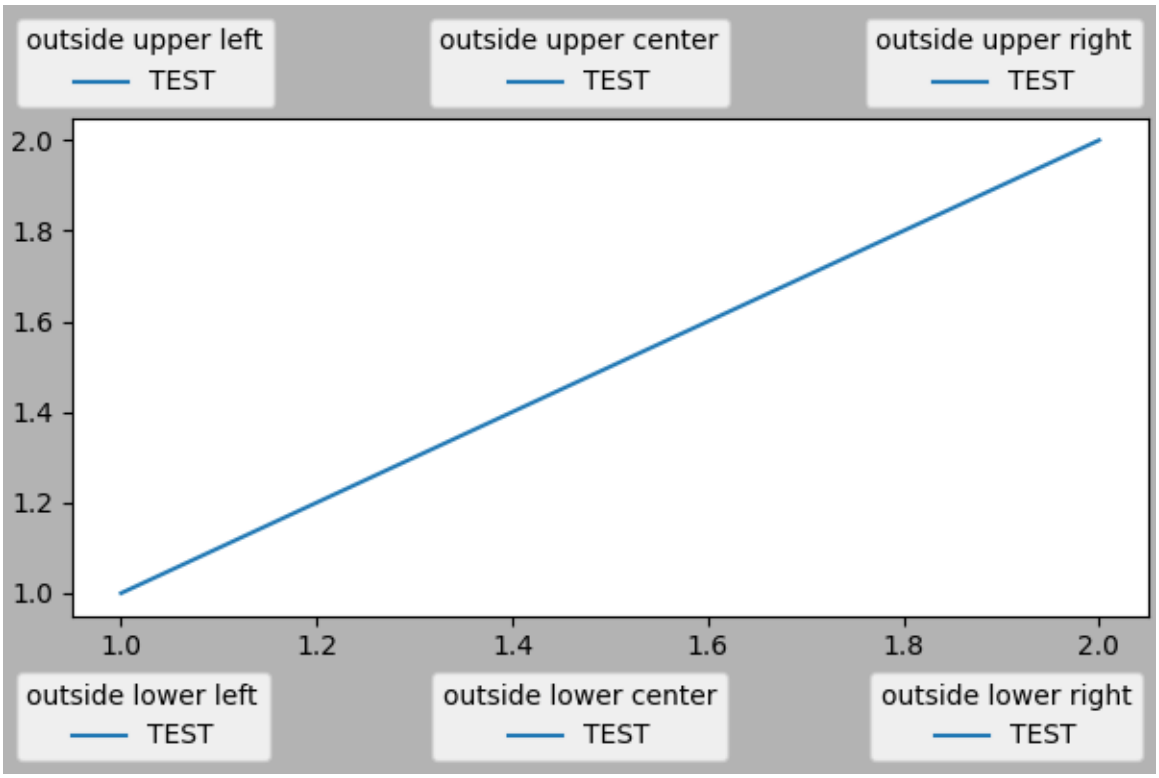

This accepts a slightly different grammar than the normal *loc* keyword, where "outside right upper" is different from "outside upper right".

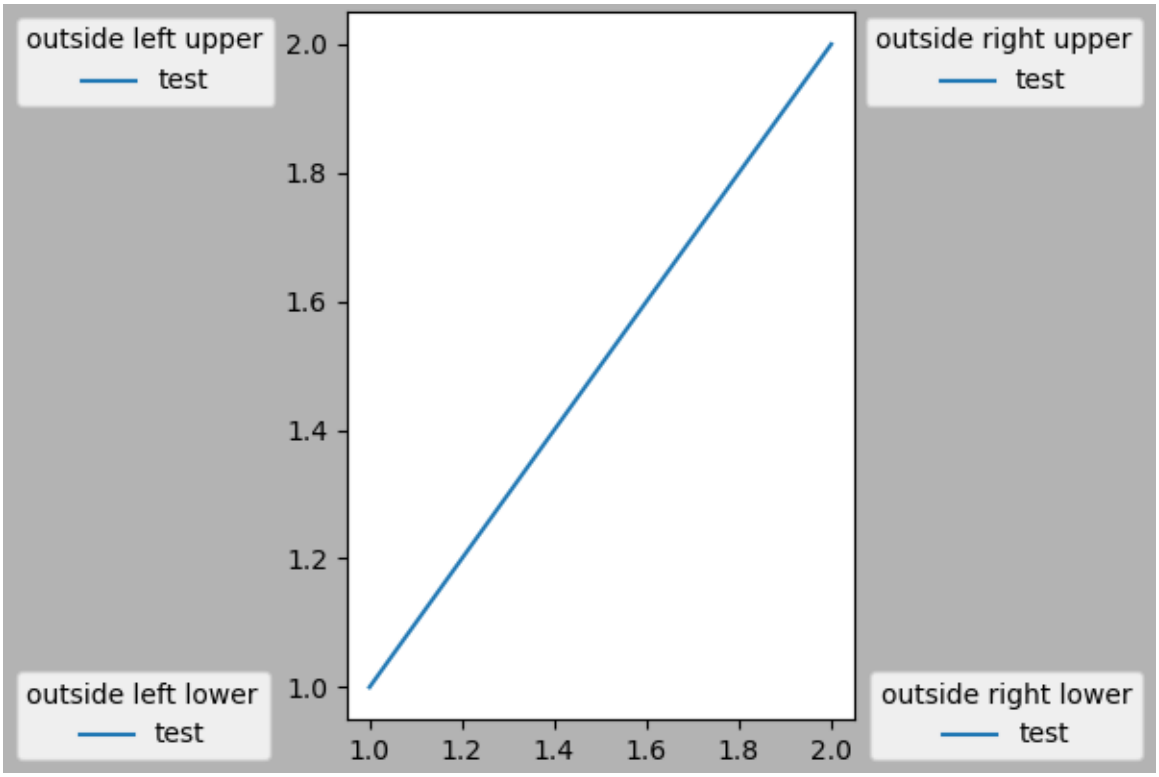
```
ucl = ['upper', 'center', 'lower']
lcr = ['left', 'center', 'right']
fig, ax = plt.subplots(figsize=(6, 4), layout='constrained', facecolor='0.7')
```

(continues on next page)

(continued from previous page)

```
ax.plot([1, 2], [1, 2], label='TEST')
# Place a legend to the right of this smaller subplot.
for loc in [
        'outside upper left',
        'outside upper center',
        'outside upper right',
        'outside lower left',
        'outside lower center',
        'outside lower right']:
    fig.legend(loc=loc, title=loc)

fig, ax = plt.subplots(figsize=(6, 4), layout='constrained', facecolor='0.7')
ax.plot([1, 2], [1, 2], label='test')

for loc in [
        'outside left upper',
        'outside right upper',
        'outside left lower',
        'outside right lower']:
    fig.legend(loc=loc, title=loc)
```

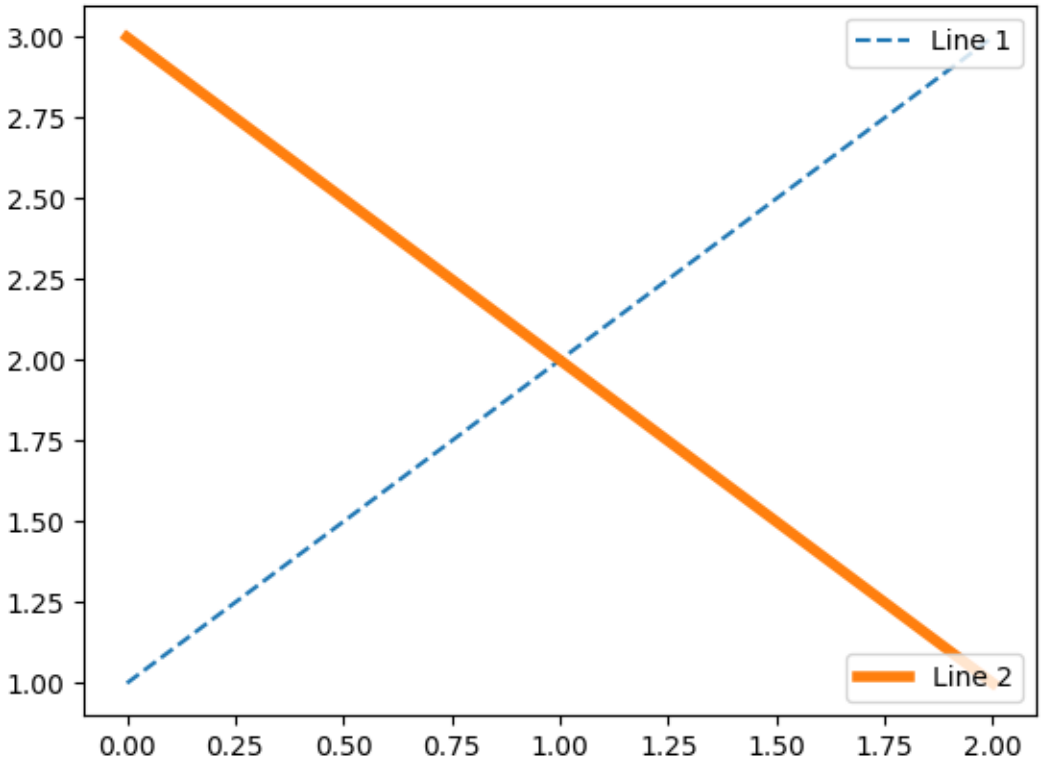
- 

- 


### Multiple legends on the same Axes

Sometimes it is more clear to split legend entries across multiple legends. Whilst the instinctive approach to doing this might be to call the *legend()* function multiple times, you will find that only one legend ever exists on the Axes. This has been done so that it is possible to call *legend()* repeatedly to update the legend to the latest handles on the Axes. To keep old legend instances, we must add them manually to the Axes:

```
fig, ax = plt.subplots()
line1, = ax.plot([1, 2, 3], label="Line 1", linestyle='--')
line2, = ax.plot([3, 2, 1], label="Line 2", linewidth=4)

# Create a legend for the first line.
first_legend = ax.legend(handles=[line1], loc='upper right')

# Add the legend manually to the Axes.
ax.add_artist(first_legend)

# Create another legend for the second line.
ax.legend(handles=[line2], loc='lower right')

plt.show()
```




### Legend handlers

In order to create legend entries, handles are given as an argument to an appropriate *HandlerBase* subclass. The choice of handler subclass is determined by the following rules:

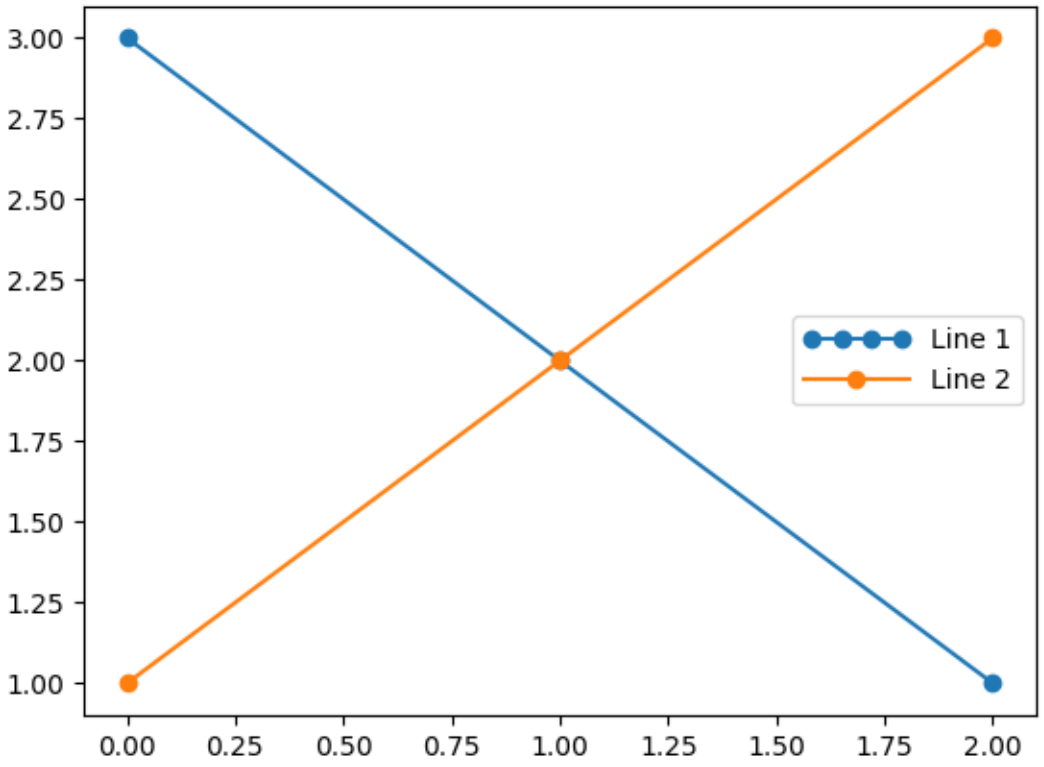
1. Update *get_legend_handler_map()* with the value in the handler_map keyword.
2. Check if the handle is in the newly created handler_map.
3. Check if the type of handle is in the newly created handler_map.
4. Check if any of the types in the handle's mro is in the newly created handler_map.

For completeness, this logic is mostly implemented in *get_legend_handler()*.

All of this flexibility means that we have the necessary hooks to implement custom handlers for our own type of legend key.

The simplest example of using custom handlers is to instantiate one of the existing *legend_handler.HandlerBase* subclasses. For the sake of simplicity, let's choose *legend_handler.HandlerLine2D* which accepts a *numpoints* argument (numpoints is also a keyword on the *legend()* function for convenience). We can then pass the mapping of instance to Handler as a keyword to legend.

```
from matplotlib.legend_handler import HandlerLine2D

fig, ax = plt.subplots()
line1, = ax.plot([3, 2, 1], marker='o', label='Line 1')
line2, = ax.plot([1, 2, 3], marker='o', label='Line 2')

ax.legend(handler_map={line1: HandlerLine2D(numpoints=4)}, handlelength=4)
```




As you can see, "Line 1" now has 4 marker points, where "Line 2" has 2 (the default). We have also increased the length of the handles with the `handlelength` keyword to fit the larger legend entry. Try the above code, only change the map's key from `line1` to `type(line1)`. Notice how now both *`Line2D`* instances get 4 markers.

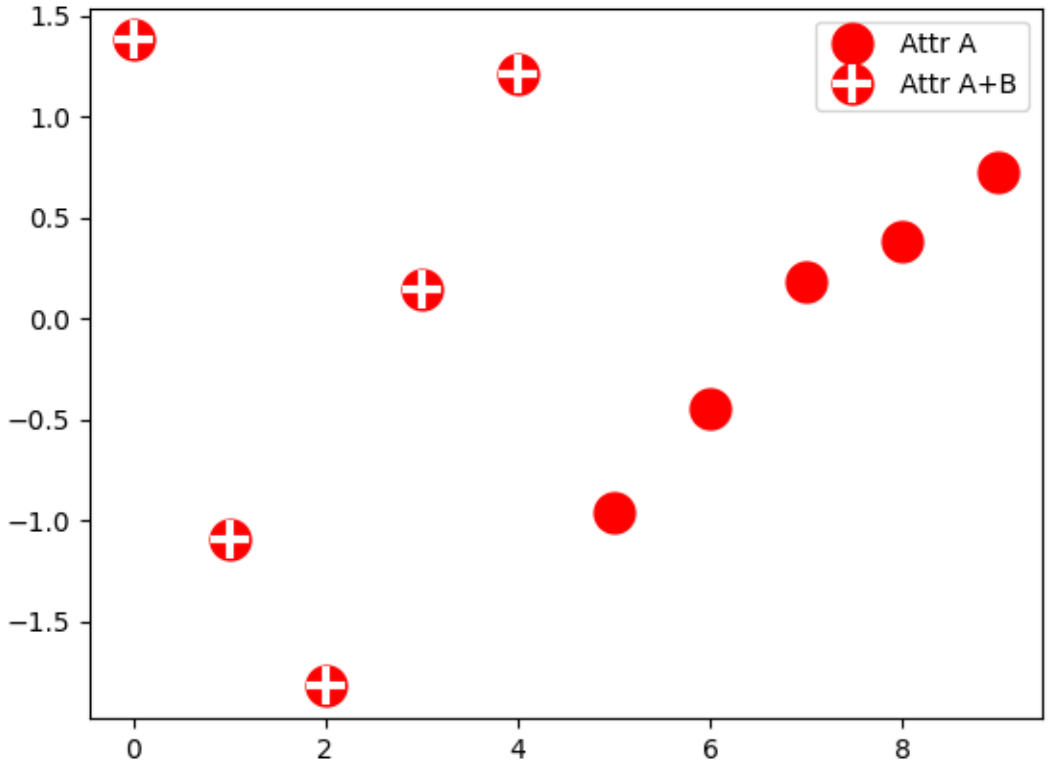
Along with handlers for complex plot types such as errorbars, stem plots and histograms, the default `handler_map` has a special `tuple` handler (*`legend_handler.HandlerTuple`*) which simply plots the handles on top of one another for each item in the given tuple. The following example demonstrates combining two legend keys on top of one another:

```
from numpy.random import randn

z = randn(10)
```

(continues on next page)

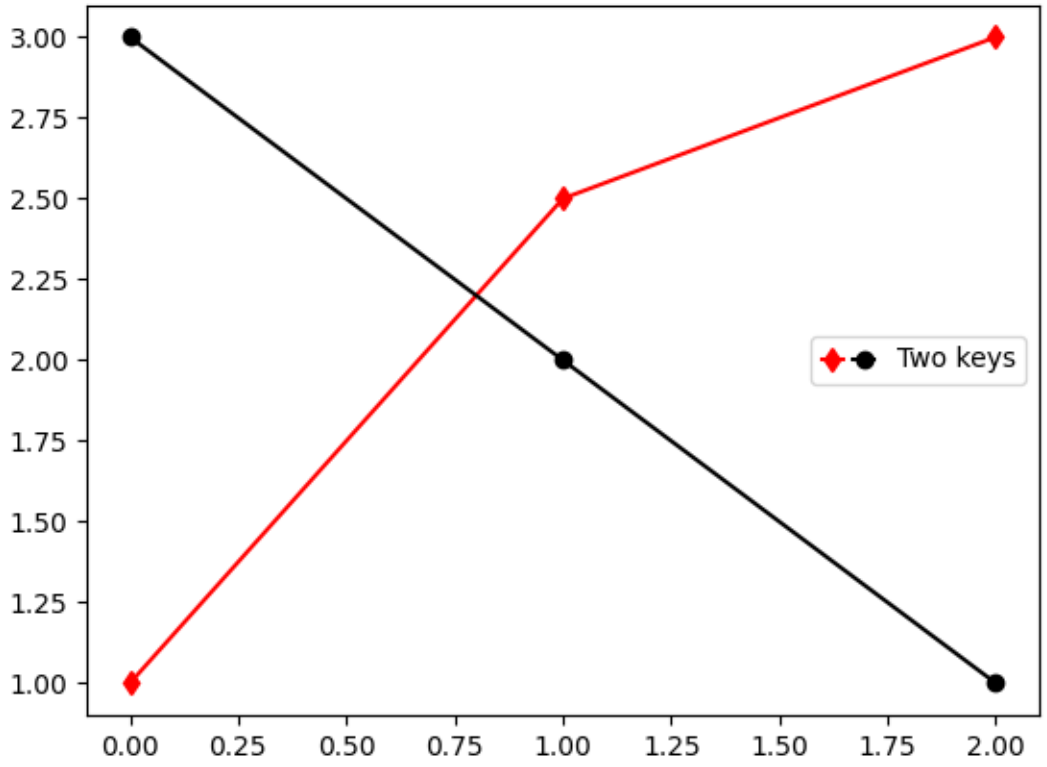
(continued from previous page)

```
fig, ax = plt.subplots()
red_dot, = ax.plot(z, "ro", markersize=15)
# Put a white cross over some of the data.
white_cross, = ax.plot(z[:5], "w+", markeredgewidth=3, markersize=15)

ax.legend([red_dot, (red_dot, white_cross)], ["Attr A", "Attr A+B"])
```




The *legend_handler.HandlerTuple* class can also be used to assign several legend keys to the same entry:

```
from matplotlib.legend_handler import HandlerLine2D, HandlerTuple

fig, ax = plt.subplots()
p1, = ax.plot([1, 2.5, 3], 'r-d')
p2, = ax.plot([3, 2, 1], 'k-o')

l = ax.legend([(p1, p2)], ['Two keys'], numpoints=1,
              handler_map={tuple: HandlerTuple(ndivide=None)})
```




### Implementing a custom legend handler

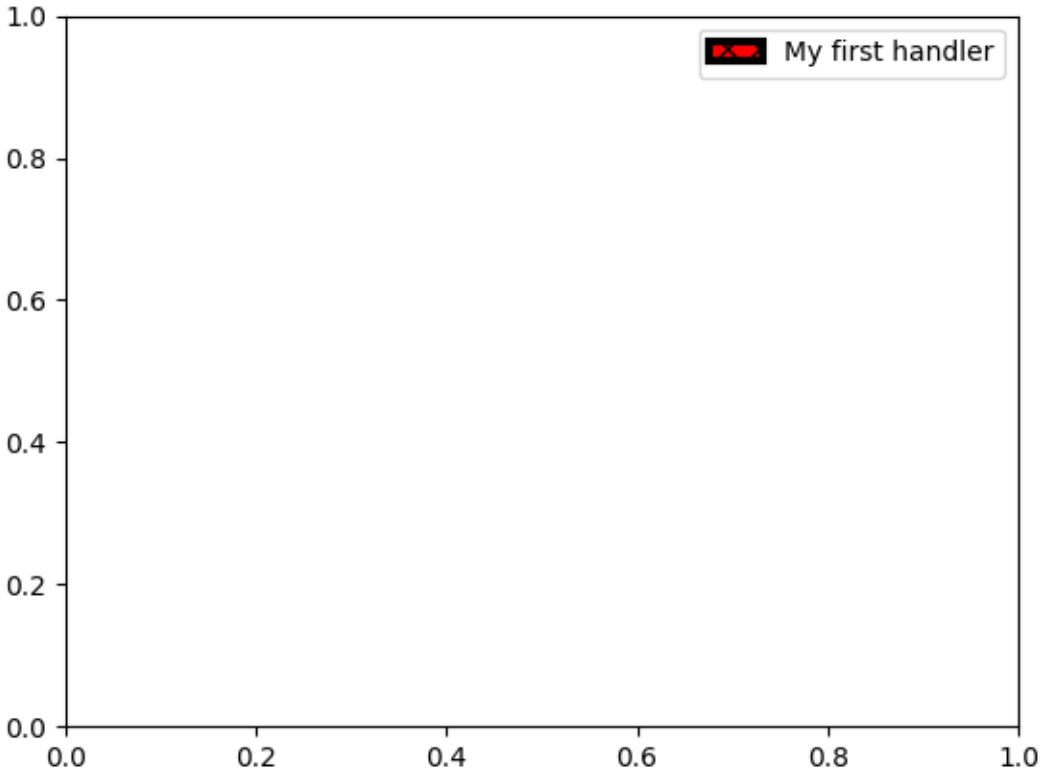
A custom handler can be implemented to turn any handle into a legend key (handles don't necessarily need to be matplotlib artists). The handler must implement a `legend_artist` method which returns a single artist for the legend to use. The required signature for `legend_artist` is documented at *legend_artist*.

```
import matplotlib.patches as mpatches


class AnyObject:
    pass


class AnyObjectHandler:
    def legend_artist(self, legend, orig_handle, fontsize, handlebox):
        x0, y0 = handlebox.xdescent, handlebox.ydescent
        width, height = handlebox.width, handlebox.height
        patch = mpatches.Rectangle([x0, y0], width, height, facecolor='red',
                                   edgecolor='black', hatch='xx', lw=3,
                                   transform=handlebox.get_transform())
        handlebox.add_artist(patch)
```

(continues on next page)

(continued from previous page)

```
        return patch

fig, ax = plt.subplots()

ax.legend([AnyObject()], ['My first handler'],
          handler_map={AnyObject: AnyObjectHandler()})
```




Alternatively, had we wanted to globally accept `AnyObject` instances without needing to manually set the *handler_map* keyword all the time, we could have registered the new handler with:

```
from matplotlib.legend import Legend
Legend.update_default_handler_map({AnyObject: AnyObjectHandler()})
```

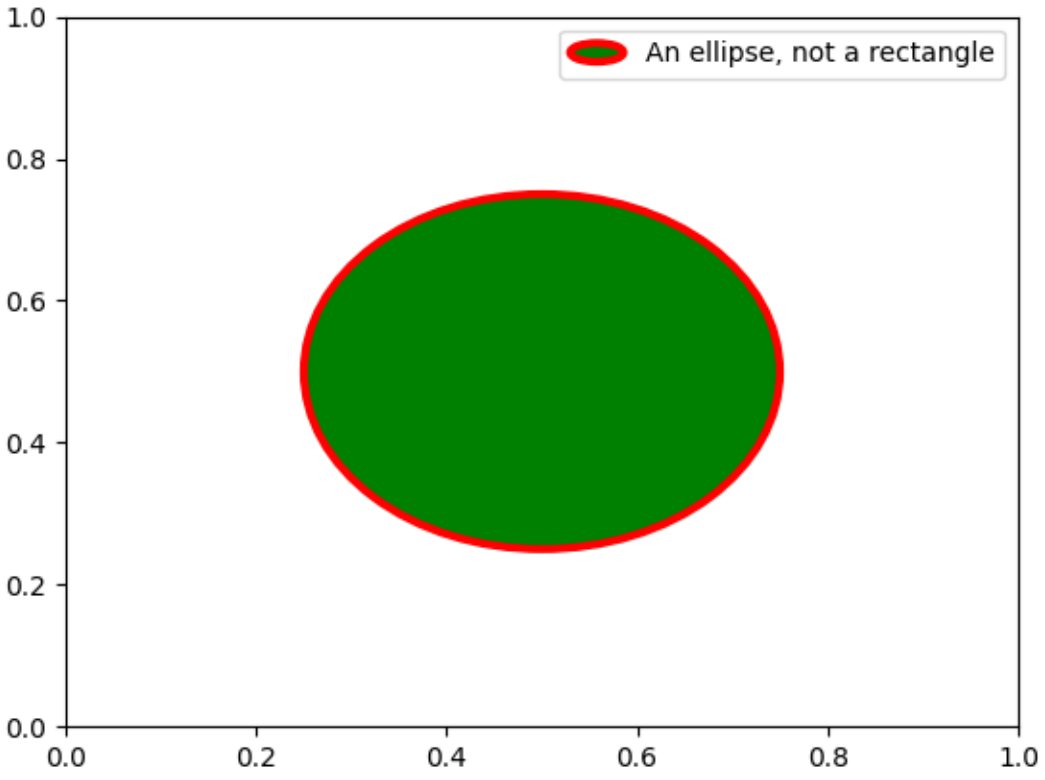
Whilst the power here is clear, remember that there are already many handlers implemented and what you want to achieve may already be easily possible with existing classes. For example, to produce elliptical legend keys, rather than rectangular ones:

```
from matplotlib.legend_handler import HandlerPatch


class HandlerEllipse(HandlerPatch):
    def create_artists(self, legend, orig_handle,
```

(continues on next page)

(continued from previous page)

```
                        xdescent, ydescent, width, height, fontsize, trans):
        center = 0.5 * width - 0.5 * xdescent, 0.5 * height - 0.5 * ydescent
        p = mpatches.Ellipse(xy=center, width=width + xdescent,
                             height=height + ydescent)
        self.update_prop(p, orig_handle, legend)
        p.set_transform(trans)
        return [p]


c = mpatches.Circle((0.5, 0.5), 0.25, facecolor="green",
                    edgecolor="red", linewidth=3)

fig, ax = plt.subplots()

ax.add_patch(c)
ax.legend([c], ["An ellipse, not a rectangle"],
          handler_map={mpatches.Circle: HandlerEllipse()})
```




**Total running time of the script:** (0 minutes 3.584 seconds)

### 3.3.9 Complex and semantic figure composition (subplot_mosaic)

Laying out Axes in a Figure in a non-uniform grid can be both tedious and verbose. For dense, even grids we have *Figure.subplots* but for more complex layouts, such as Axes that span multiple columns / rows of the layout or leave some areas of the Figure blank, you can use *gridspec.GridSpec* (see *Arranging multiple Axes in a Figure*) or manually place your Axes. *Figure.subplot_mosaic* aims to provide an interface to visually lay out your Axes (as either ASCII art or nested lists) to streamline this process.

This interface naturally supports naming your Axes. *Figure.subplot_mosaic* returns a dictionary keyed on the labels used to lay out the Figure. By returning data structures with names, it is easier to write plotting code that is independent of the Figure layout.

This is inspired by a proposed MEP and the patchwork library for R. While we do not implement the operator overloading style, we do provide a Pythonic API for specifying (nested) Axes layouts.

```
import matplotlib.pyplot as plt
import numpy as np


# Helper function used for visualization in the following examples
def identify_axes(ax_dict, fontsize=48):
    """
    Helper to identify the Axes in the examples below.

    Draws the label in a large font in the center of the Axes.

    Parameters
    ----------
    ax_dict : dict[str, Axes]
        Mapping between the title / label and the Axes.
    fontsize : int, optional
        How big the label should be.
    """
    kw = dict(ha="center", va="center", fontsize=fontsize, color="darkgrey")
    for k, ax in ax_dict.items():
        ax.text(0.5, 0.5, k, transform=ax.transAxes, **kw)
```

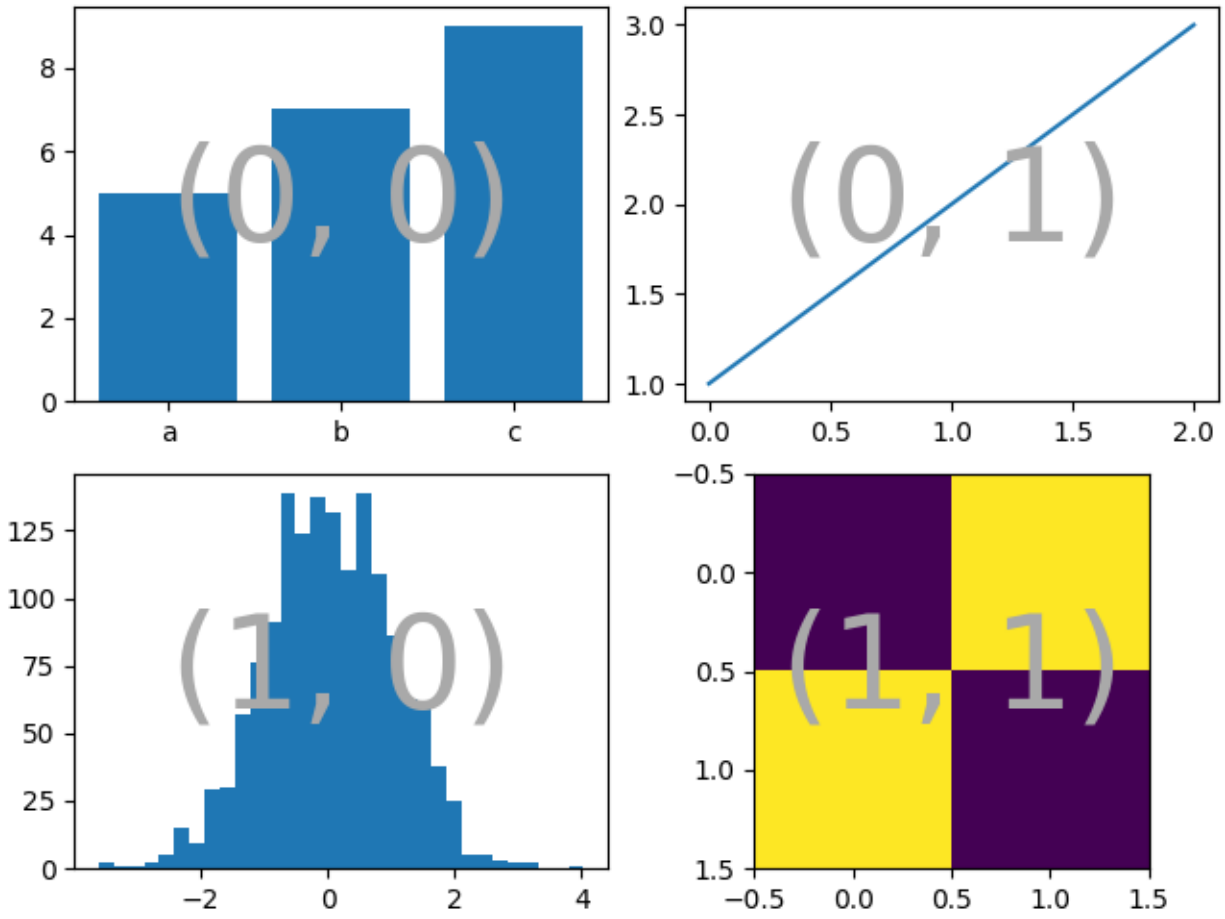
If we want a 2x2 grid we can use *Figure.subplots* which returns a 2D array of *axes.Axes* which we can index into to do our plotting.

```
np.random.seed(19680801)
hist_data = np.random.randn(1_500)


fig = plt.figure(layout="constrained")
ax_array = fig.subplots(2, 2, squeeze=False)

ax_array[0, 0].bar(["a", "b", "c"], [5, 7, 9])
ax_array[0, 1].plot([1, 2, 3])
ax_array[1, 0].hist(hist_data, bins="auto")
ax_array[1, 1].imshow([[1, 2], [2, 1]])
```

(continues on next page)

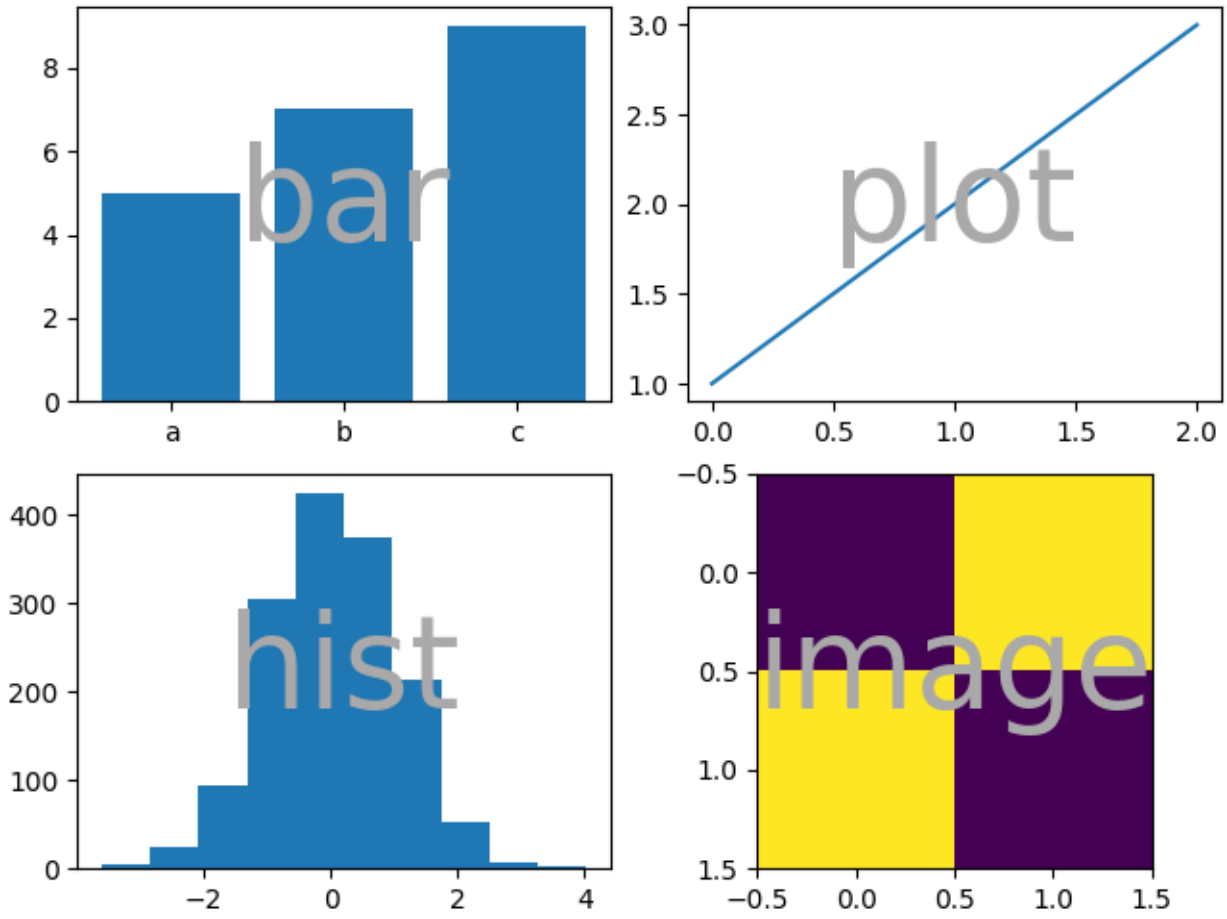
(continued from previous page)

```
identify_axes(
    {(j, k): a for j, r in enumerate(ax_array) for k, a in enumerate(r)},
)
```




Using *Figure.subplot_mosaic* we can produce the same mosaic but give the Axes semantic names

```
fig = plt.figure(layout="constrained")
ax_dict = fig.subplot_mosaic(
    [
        ["bar", "plot"],
        ["hist", "image"],
    ],
)
ax_dict["bar"].bar(["a", "b", "c"], [5, 7, 9])
ax_dict["plot"].plot([1, 2, 3])
ax_dict["hist"].hist(hist_data)
ax_dict["image"].imshow([[1, 2], [2, 1]])
identify_axes(ax_dict)
```




A key difference between *Figure.subplots* and *Figure.subplot_mosaic* is the return value. While the former returns an array for index access, the latter returns a dictionary mapping the labels to the *axes.Axes* instances created

```
print(ax_dict)
```

```
{'bar': <Axes: label='bar'>, 'plot': <Axes: label='plot'>, 'hist': <Axes:␣
↪label='hist'>, 'image': <Axes: label='image'>}
```

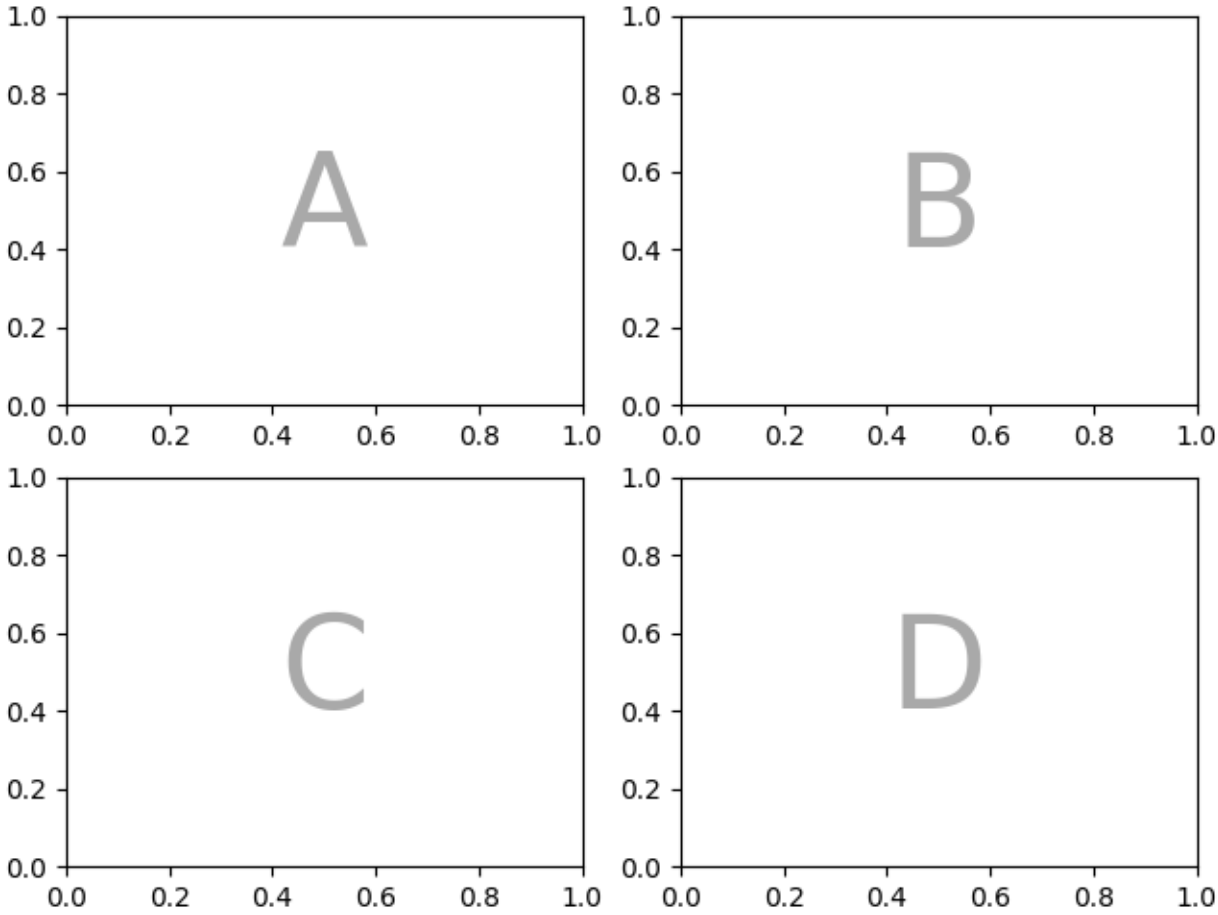
### String short-hand

By restricting our Axes labels to single characters we can "draw" the Axes we want as "ASCII art". The following

```
mosaic = """
    AB
    CD
    """
```

will give us 4 Axes laid out in a 2x2 grid and generates the same figure mosaic as above (but now labeled with `{"A", "B", "C", "D"}` rather than `{"bar", "plot", "hist", "image"}`).

```
fig = plt.figure(layout="constrained")
ax_dict = fig.subplot_mosaic(mosaic)
identify_axes(ax_dict)
```

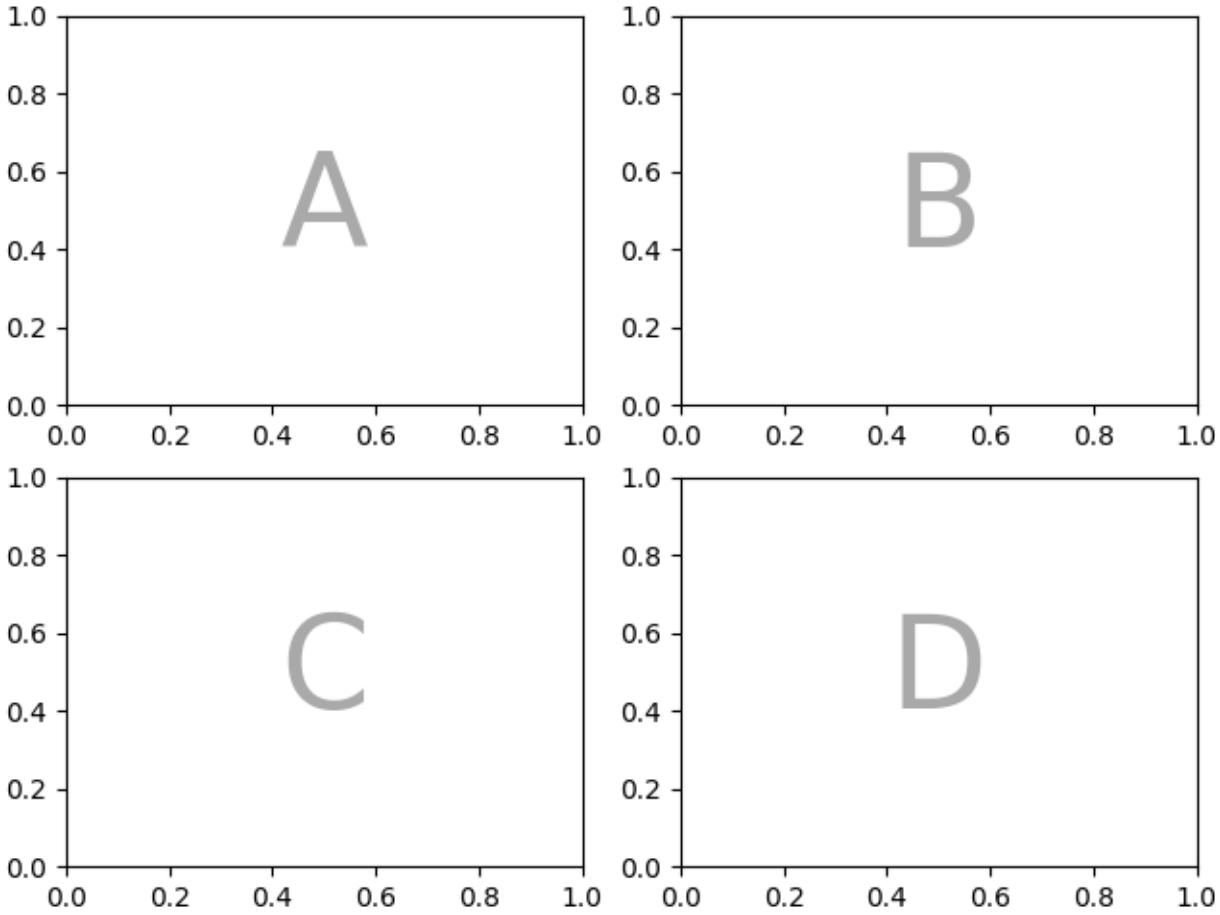



Alternatively, you can use the more compact string notation

```
mosaic = "AB;CD"
```

will give you the same composition, where the `";"` is used as the row separator instead of newline.

```
fig = plt.figure(layout="constrained")
ax_dict = fig.subplot_mosaic(mosaic)
identify_axes(ax_dict)
```

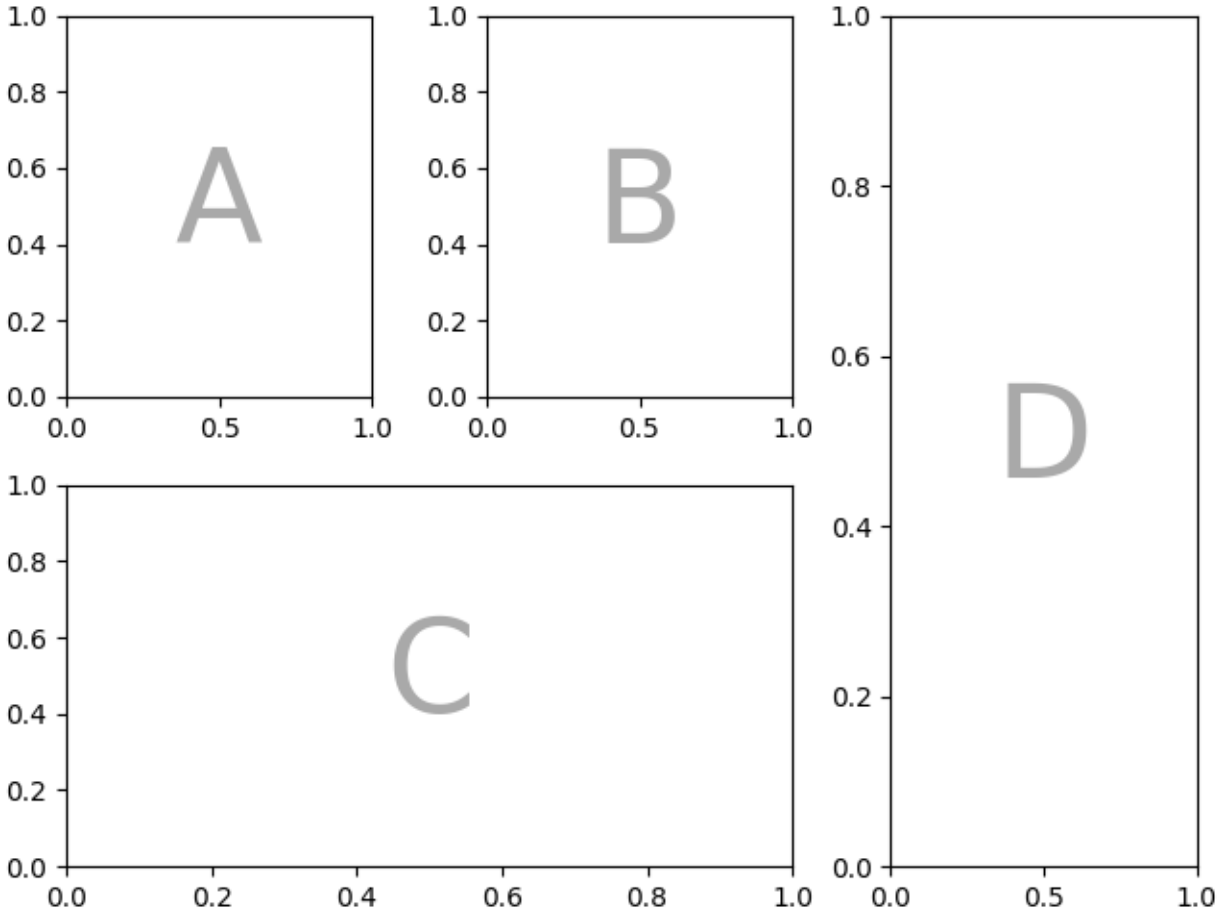



### Axes spanning multiple rows/columns

Something we can do with *Figure.subplot_mosaic*, that we cannot do with *Figure.subplots*, is to specify that an Axes should span several rows or columns.

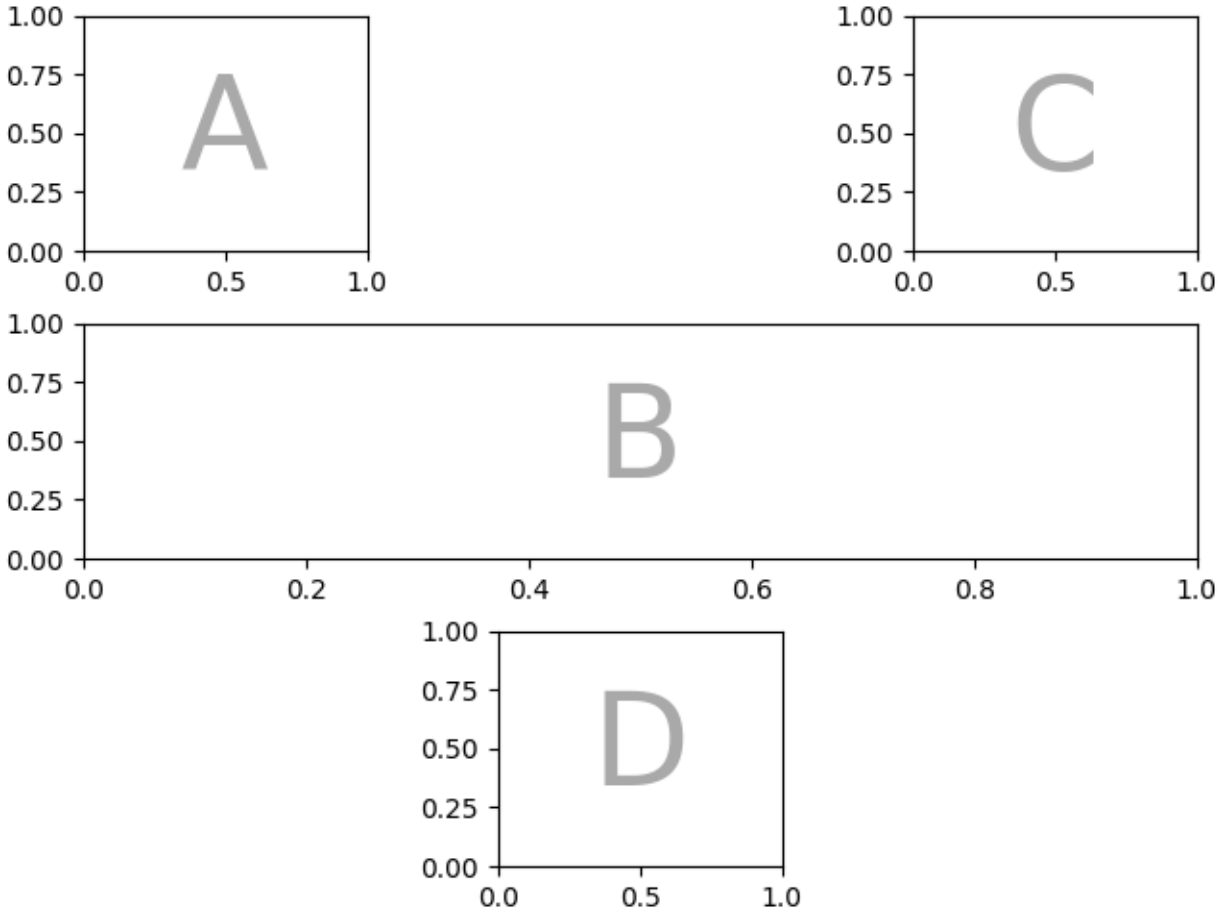
If we want to re-arrange our four Axes to have "C" be a horizontal span on the bottom and "D" be a vertical span on the right we would do

```
axd = plt.figure(layout="constrained").subplot_mosaic(
    """
    ABD
    CCD
    """
)
identify_axes(axd)
```



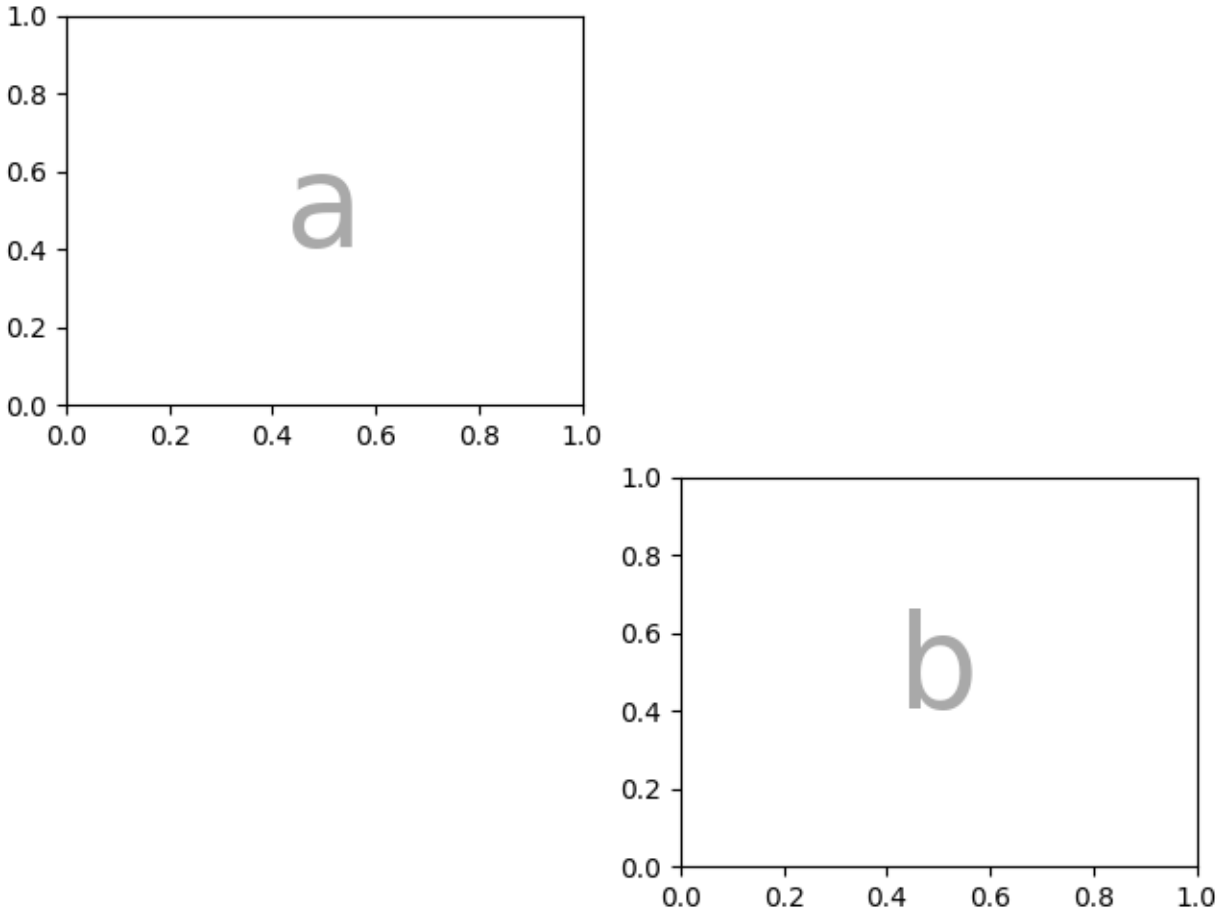

If we do not want to fill in all the spaces in the Figure with Axes, we can specify some spaces in the grid to be blank

```
axd = plt.figure(layout="constrained").subplot_mosaic(
    """
    A.C
    BBB
    .D.
    """
)
identify_axes(axd)
```



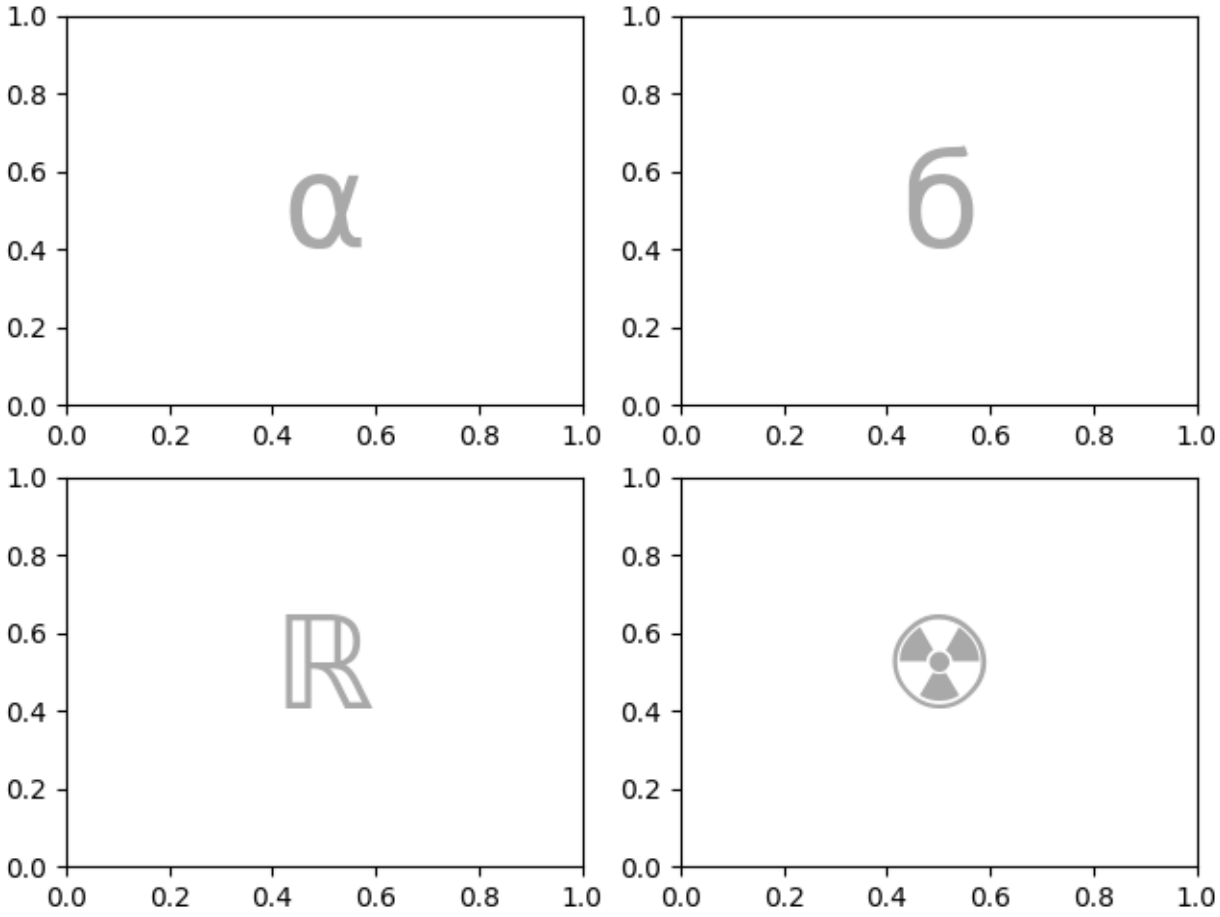

If we prefer to use another character (rather than a period `"."`) to mark the empty space, we can use *empty_sentinel* to specify the character to use.

```
axd = plt.figure(layout="constrained").subplot_mosaic(
    """
    aX
    Xb
    """,
    empty_sentinel="X",
)
identify_axes(axd)
```




Internally there is no meaning attached to the letters we use, any Unicode code point is valid!

```
axd = plt.figure(layout="constrained").subplot_mosaic(
    """аб
       R□"""
)
identify_axes(axd)
```




It is not recommended to use white space as either a label or an empty sentinel with the string shorthand because it may be stripped while processing the input.

### Controlling mosaic creation

This feature is built on top of *gridspec* and you can pass the keyword arguments through to the underlying *gridspec.GridSpec* (the same as *Figure.subplots*).

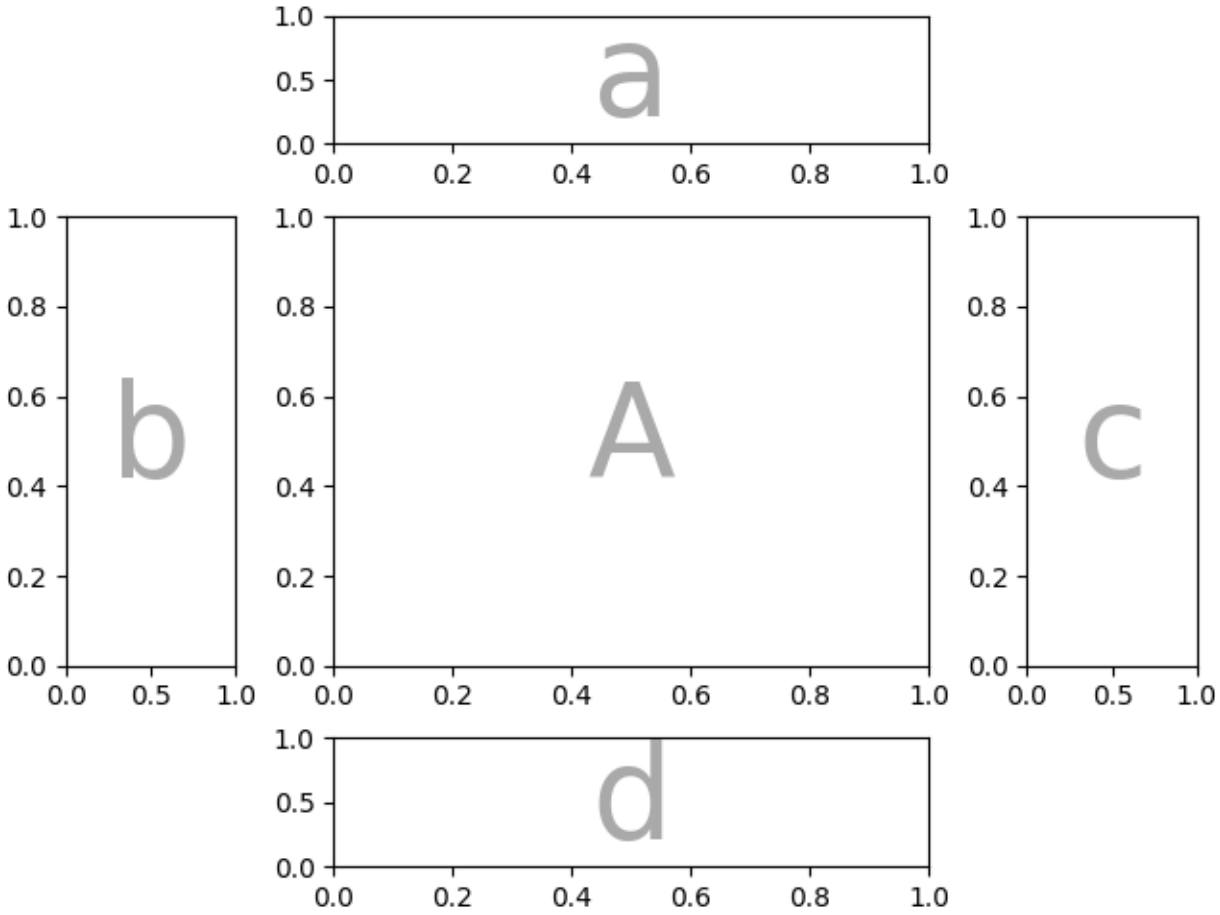
In this case we want to use the input to specify the arrangement, but set the relative widths of the rows / columns. For convenience, *gridspec.GridSpec*'s *height_ratios* and *width_ratios* are exposed in the *Figure.subplot_mosaic* calling sequence.

```
axd = plt.figure(layout="constrained").subplot_mosaic(
    """
    .a.
    bAc
    .d.
    """,
    # set the height ratios between the rows
    height_ratios=[1, 3.5, 1],
    # set the width ratios between the columns
    width_ratios=[1, 3.5, 1],
```

(continues on next page)

(continued from previous page)

```
)
identify_axes(axd)
```



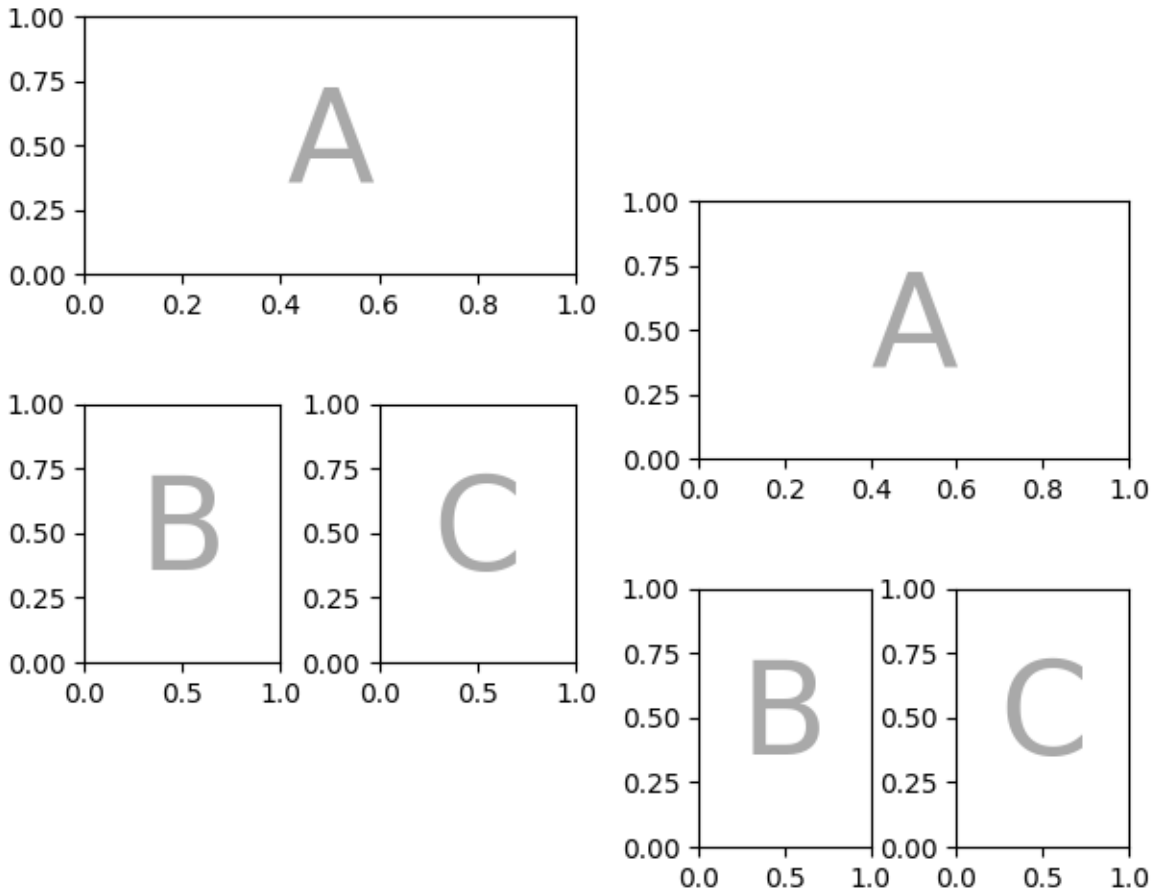

Other *gridspec.GridSpec* keywords can be passed via *gridspec_kw*. For example, use the {*left*, *right*, *bottom*, *top*} keyword arguments to position the overall mosaic to put multiple versions of the same mosaic in a figure.

```
mosaic = """AA
            BC"""
fig = plt.figure()
axd = fig.subplot_mosaic(
    mosaic,
    gridspec_kw={
        "bottom": 0.25,
        "top": 0.95,
        "left": 0.1,
        "right": 0.5,
        "wspace": 0.5,
        "hspace": 0.5,
    },
)
identify_axes(axd)
```

(continues on next page)

(continued from previous page)

```
axd = fig.subplot_mosaic(
    mosaic,
    gridspec_kw={
        "bottom": 0.05,
        "top": 0.75,
        "left": 0.6,
        "right": 0.95,
        "wspace": 0.5,
        "hspace": 0.5,
    },
)
identify_axes(axd)
```



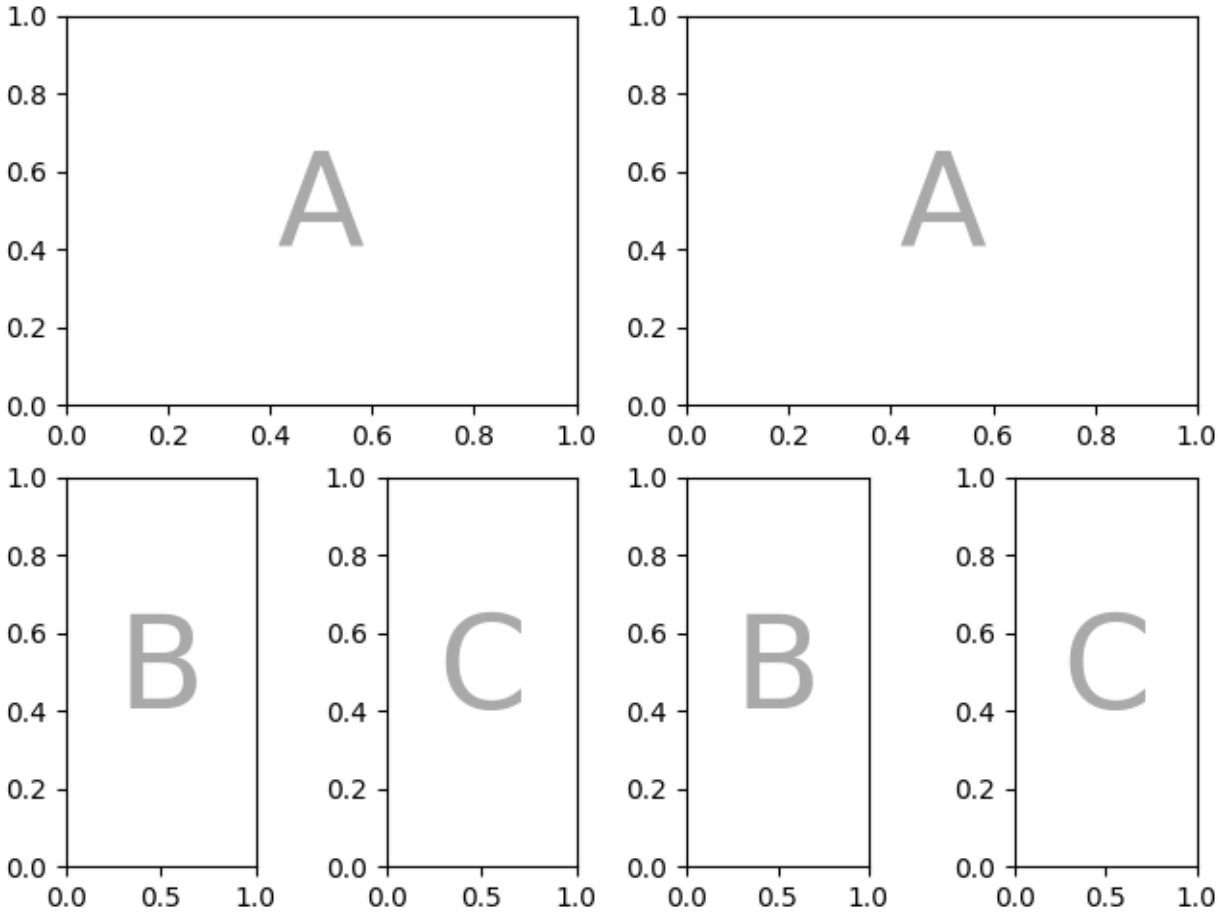

Alternatively, you can use the sub-Figure functionality:

```
mosaic = """AA
            BC"""
fig = plt.figure(layout="constrained")
left, right = fig.subfigures(nrows=1, ncols=2)
axd = left.subplot_mosaic(mosaic)
identify_axes(axd)
```

(continues on next page)

(continued from previous page)

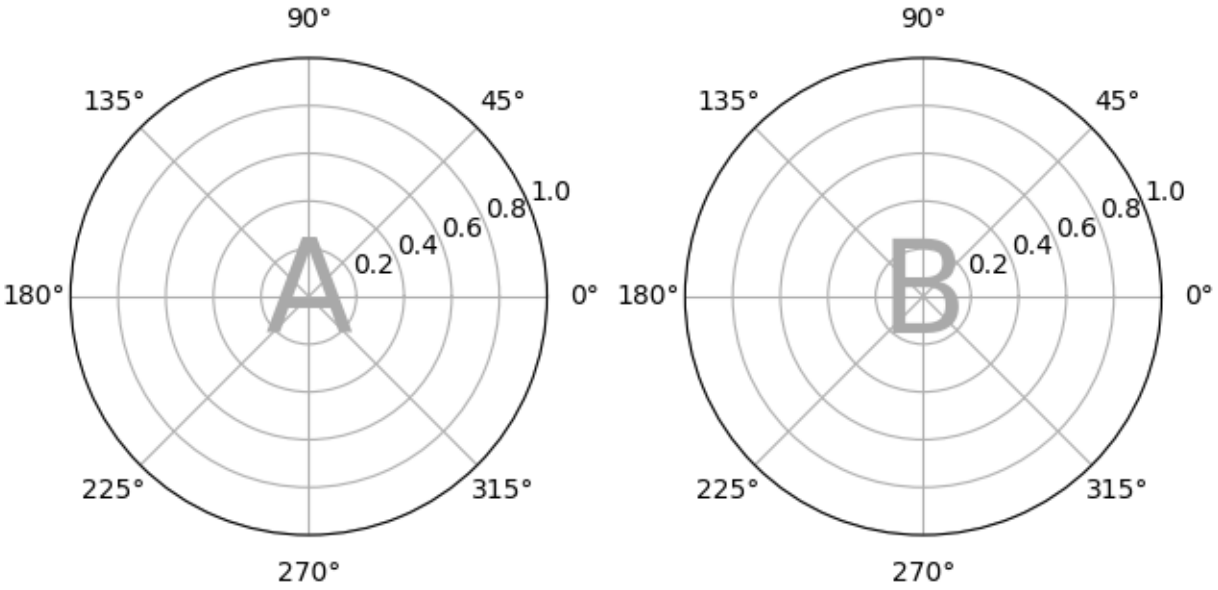
```
axd = right.subplot_mosaic(mosaic)
identify_axes(axd)
```




### Controlling subplot creation

We can also pass through arguments used to create the subplots (again, the same as *Figure.subplots*) which will apply to all of the Axes created.

```
axd = plt.figure(layout="constrained").subplot_mosaic(
    "AB", subplot_kw={"projection": "polar"}
)
identify_axes(axd)
```

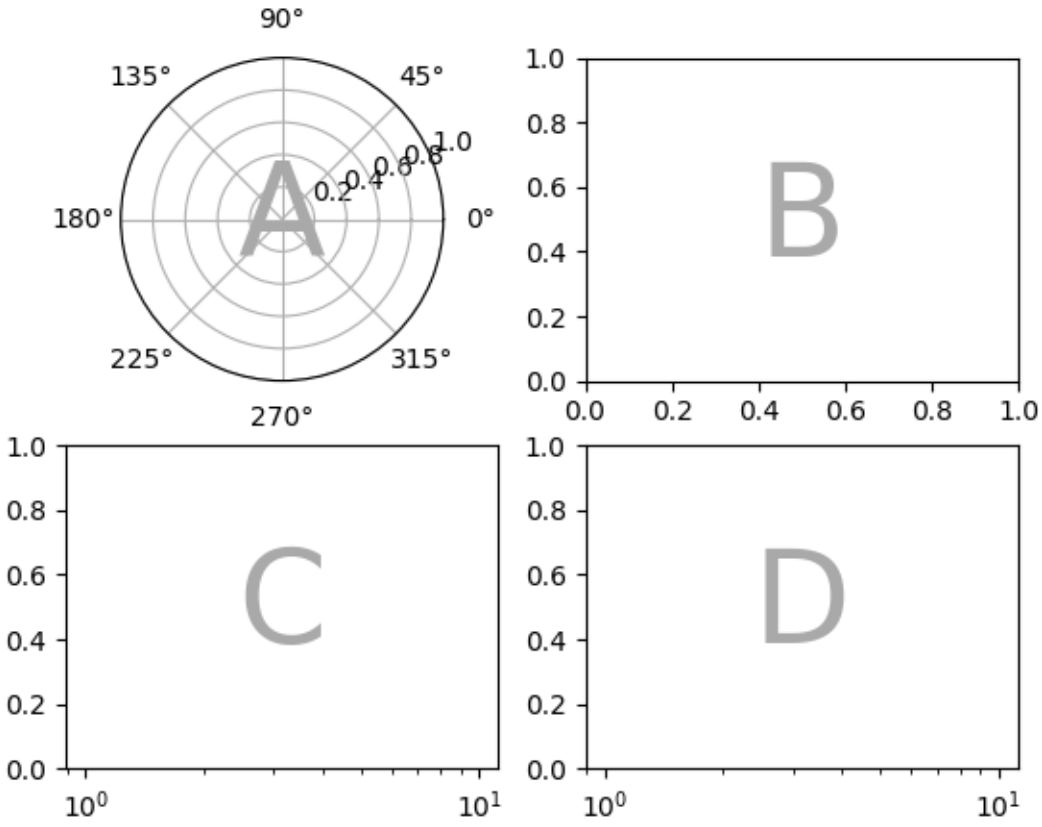



### Per-Axes subplot keyword arguments

If you need to control the parameters passed to each subplot individually use *per_subplot_kw* to pass a mapping between the Axes identifiers (or tuples of Axes identifiers) to dictionaries of keywords to be passed.

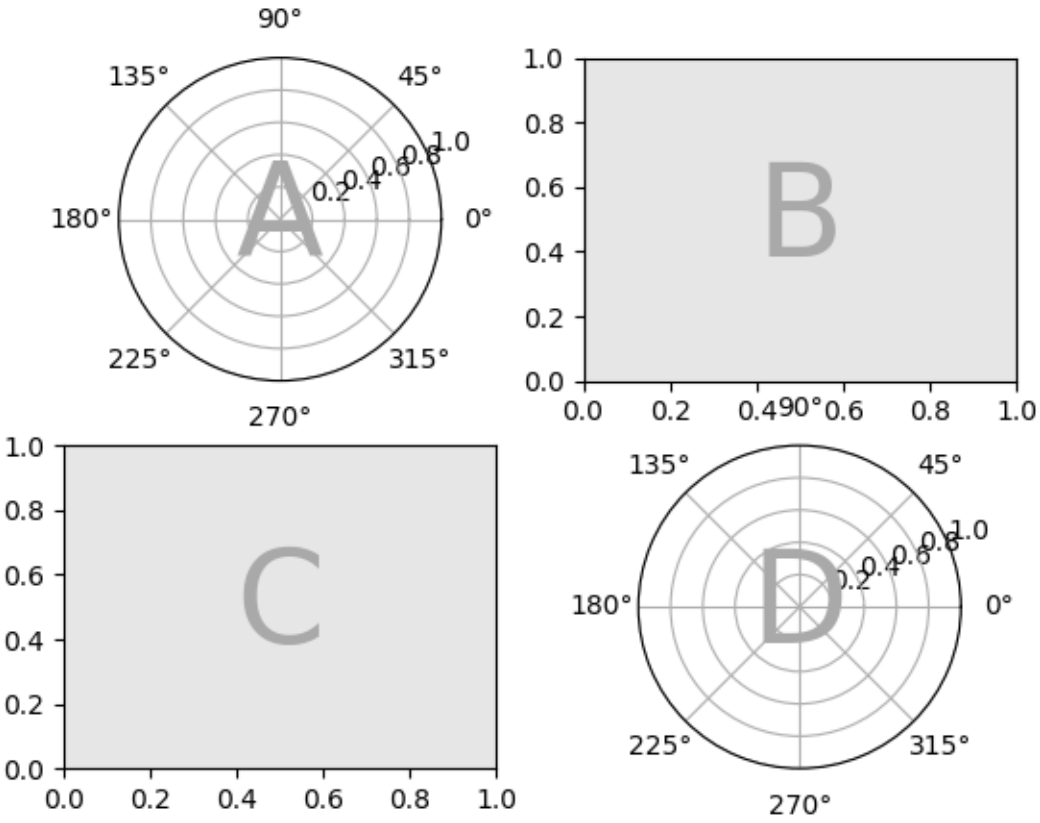
New in version 3.7.

```
fig, axd = plt.subplot_mosaic(
    "AB;CD",
    per_subplot_kw={
        "A": {"projection": "polar"},
        ("C", "D"): {"xscale": "log"}
    },
)
identify_axes(axd)
```



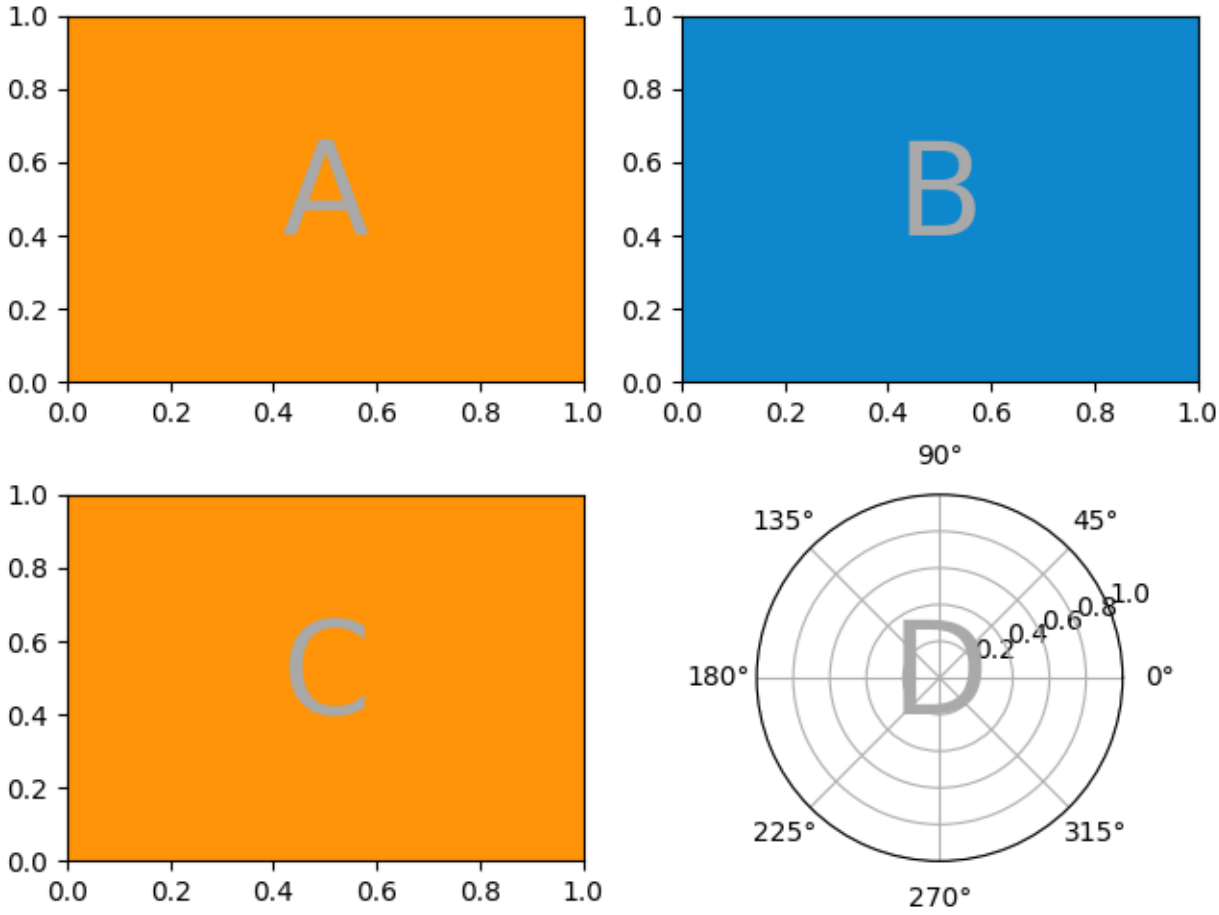

If the layout is specified with the string short-hand, then we know the Axes labels will be one character and can unambiguously interpret longer strings in *per_subplot_kw* to specify a set of Axes to apply the keywords to:

```
fig, axd = plt.subplot_mosaic(
    "AB;CD",
    per_subplot_kw={
        "AD": {"projection": "polar"},
        "BC": {"facecolor": ".9"}
    },
)
identify_axes(axd)
```




If *subplot_kw* and *per_subplot_kw* are used together, then they are merged with *per_subplot_kw* taking priority:

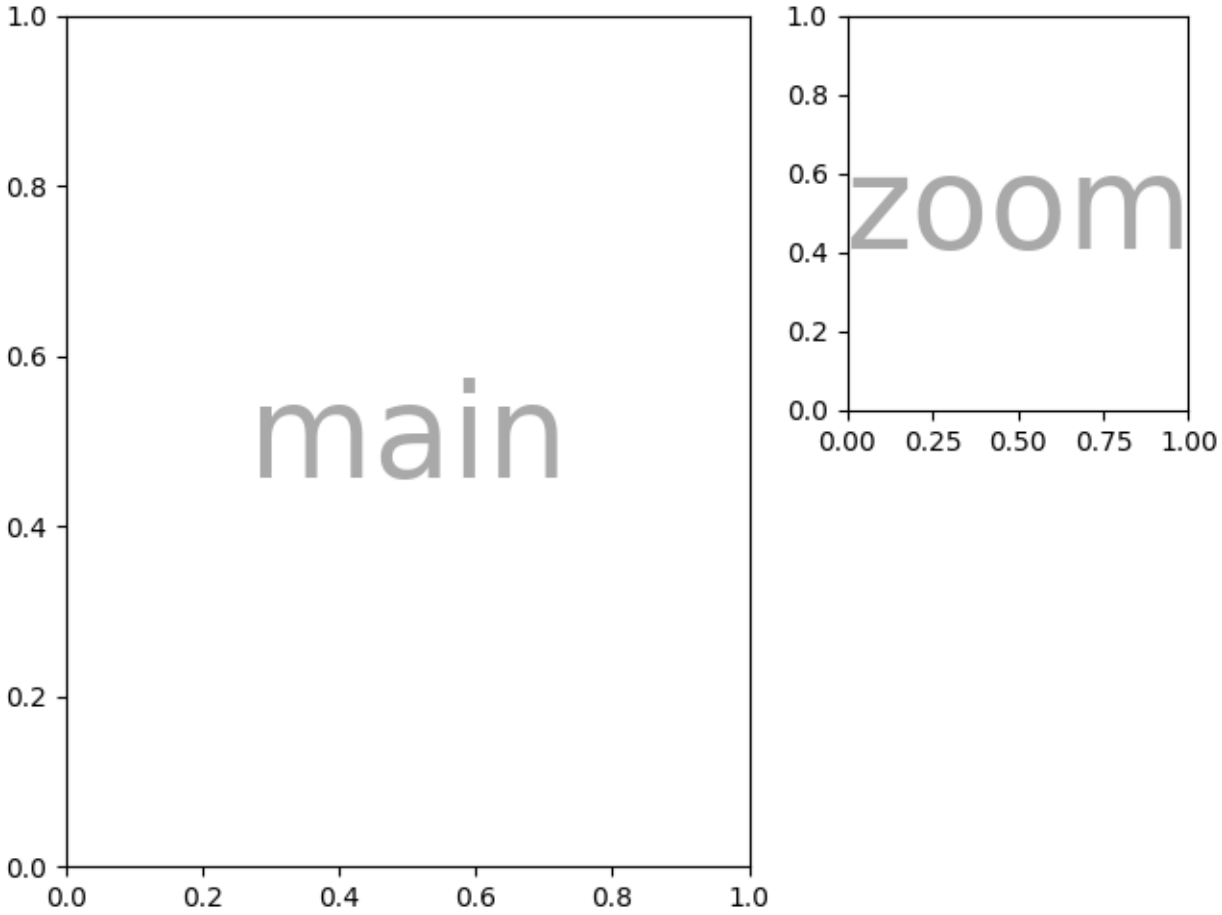
```
axd = plt.figure(layout="constrained").subplot_mosaic(
    "AB;CD",
    subplot_kw={"facecolor": "xkcd:tangerine"},
    per_subplot_kw={
        "B": {"facecolor": "xkcd:water blue"},
        "D": {"projection": "polar", "facecolor": "w"},
    }
)
identify_axes(axd)
```




### Nested list input

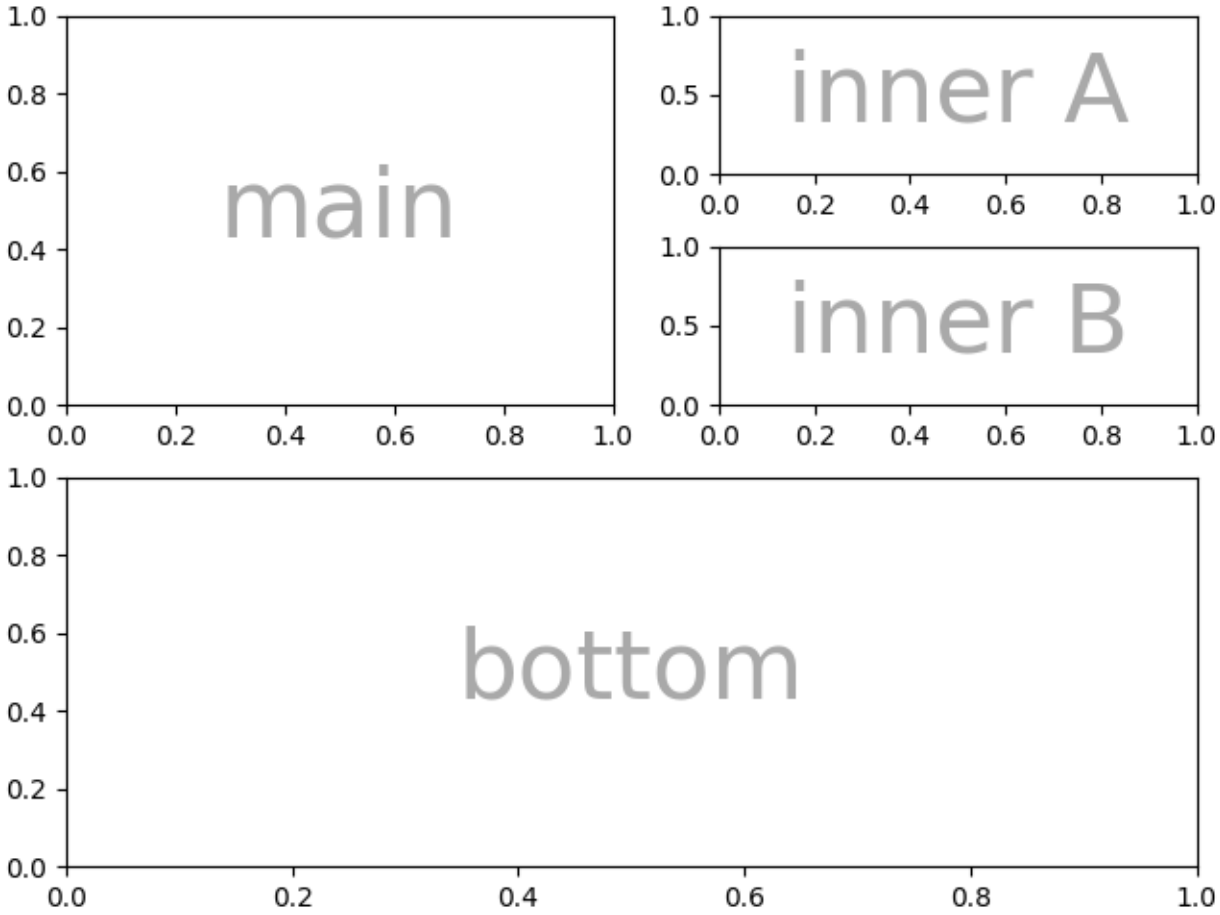
Everything we can do with the string shorthand we can also do when passing in a list (internally we convert the string shorthand to a nested list), for example using spans, blanks, and *gridspec_kw*:

```
axd = plt.figure(layout="constrained").subplot_mosaic(
    [
        ["main", "zoom"],
        ["main", "BLANK"],
    ],
    empty_sentinel="BLANK",
    width_ratios=[2, 1],
)
identify_axes(axd)
```



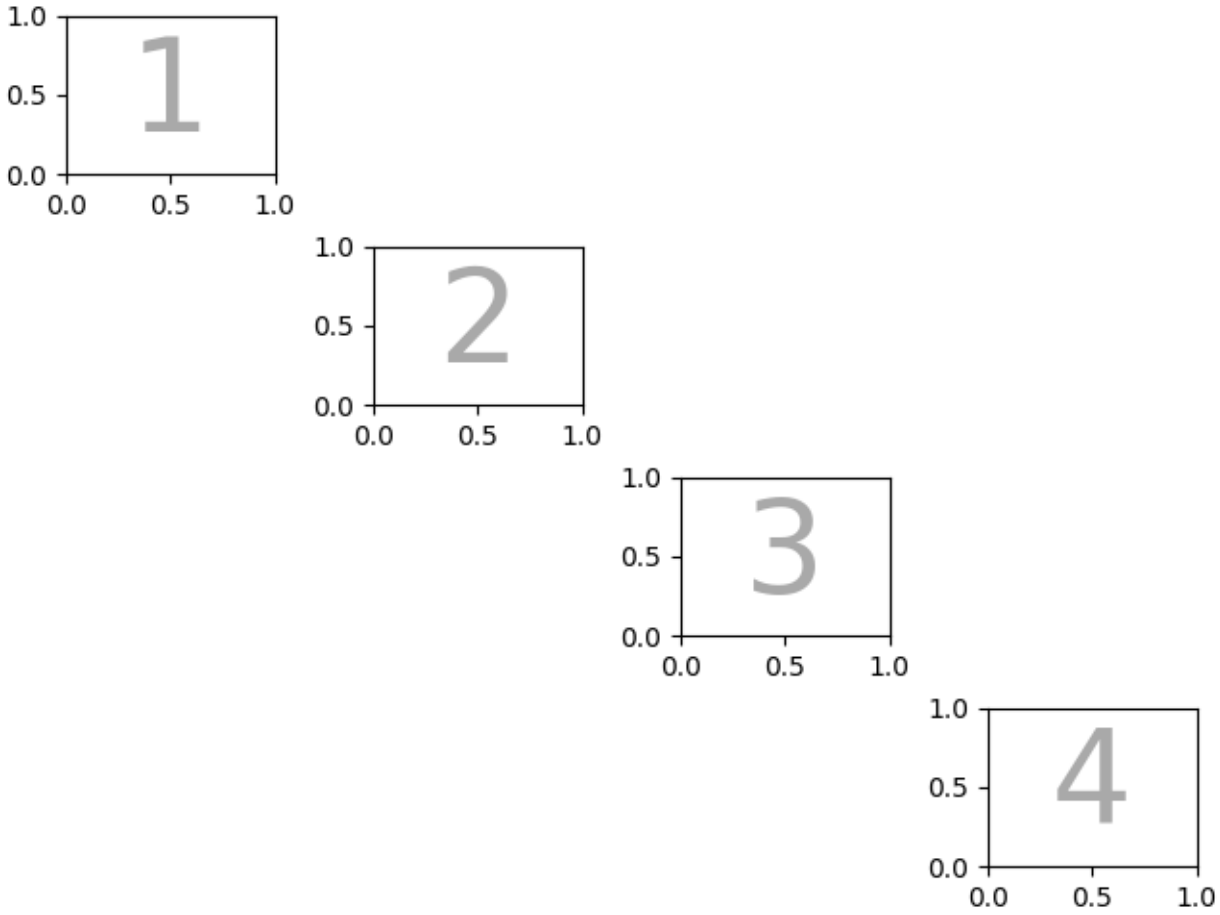

In addition, using the list input we can specify nested mosaics. Any element of the inner list can be another set of nested lists:

```
inner = [
    ["inner A"],
    ["inner B"],
]

outer_nested_mosaic = [
    ["main", inner],
    ["bottom", "bottom"],
]
axd = plt.figure(layout="constrained").subplot_mosaic(
    outer_nested_mosaic, empty_sentinel=None
)
identify_axes(axd, fontsize=36)
```




We can also pass in a 2D NumPy array to do things like

```
mosaic = np.zeros((4, 4), dtype=int)
for j in range(4):
    mosaic[j, j] = j + 1
axd = plt.figure(layout="constrained").subplot_mosaic(
    mosaic,
    empty_sentinel=0,
)
identify_axes(axd)
```




**Total running time of the script:** (0 minutes 8.000 seconds)

### 3.3.10 Constrained layout guide

Use *constrained layout* to fit plots within your figure cleanly.

*Constrained layout* automatically adjusts subplots so that decorations like tick labels, legends, and colorbars do not overlap, while still preserving the logical layout requested by the user.

*Constrained layout* is similar to *Tight layout*, but is substantially more flexible. It handles colorbars placed on multiple Axes (*Placing colorbars*) nested layouts (`subfigures`) and Axes that span rows or columns (`subplot_mosaic`), striving to align spines from Axes in the same row or column. In addition, *Compressed layout* will try and move fixed aspect-ratio Axes closer together. These features are described in this document, as well as some *implementation details* discussed at the end.

*Constrained layout* typically needs to be activated before any Axes are added to a figure. Two ways of doing so are

- using the respective argument to `subplots`, `figure`, `subplot_mosaic` e.g.:

```
plt.subplots(layout="constrained")
```

- activate it via *rcParams*, like:

```
plt.rcParams['figure.constrained_layout.use'] = True
```

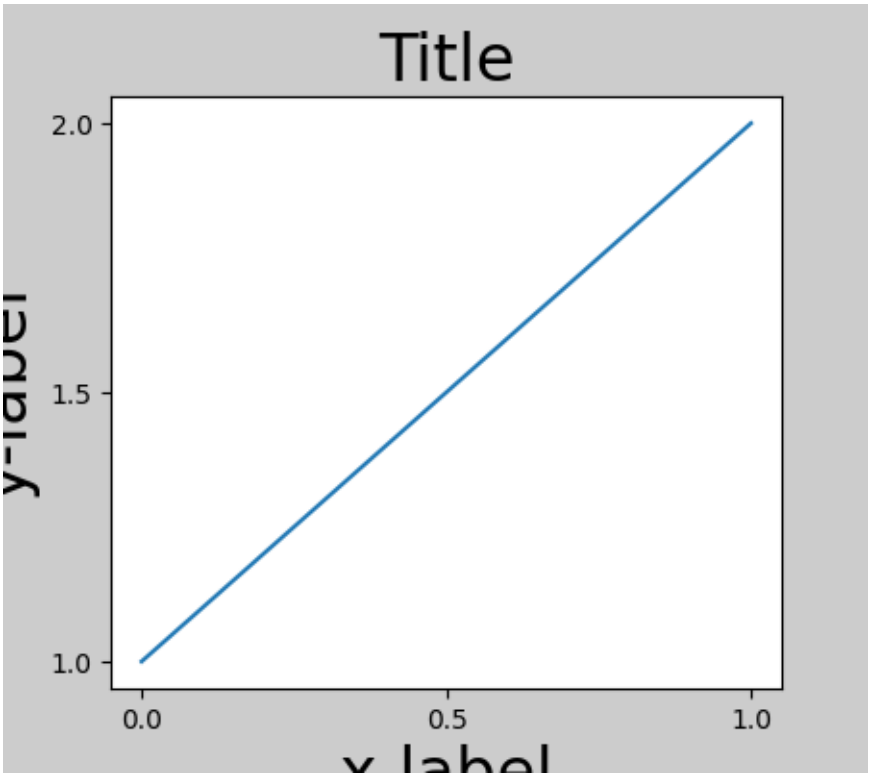
Those are described in detail throughout the following sections.

**Warning:** Calling *tight_layout* will turn off *constrained layout*!

### Simple example

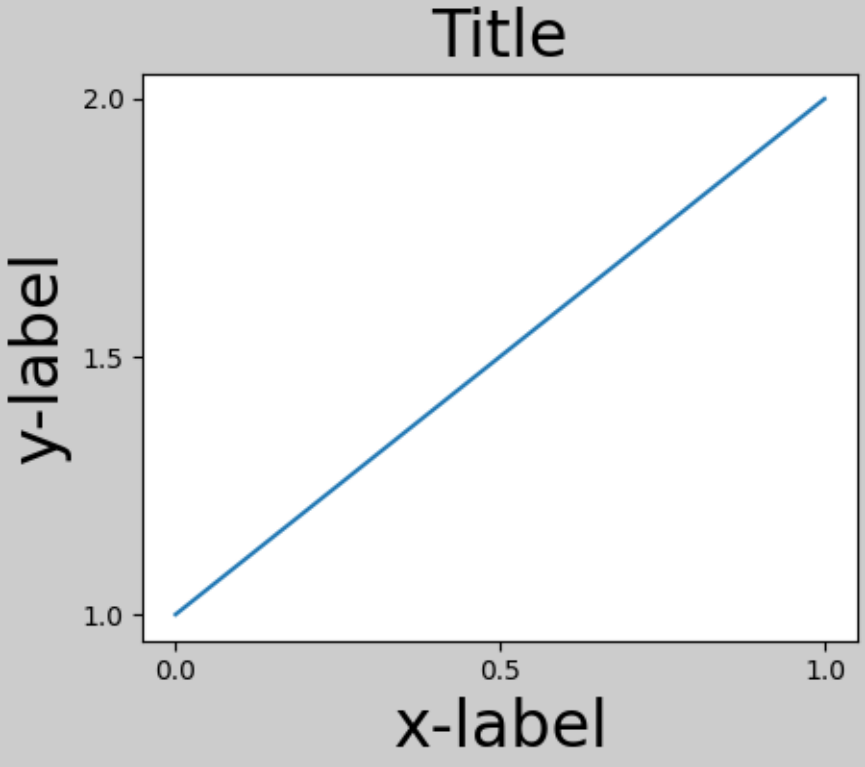
With the default Axes positioning, the axes title, axis labels, or tick labels can sometimes go outside the figure area, and thus get clipped.

```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib.colors as mcolors
import matplotlib.gridspec as gridspec

plt.rcParams['savefig.facecolor'] = "0.8"
plt.rcParams['figure.figsize'] = 4.5, 4.
plt.rcParams['figure.max_open_warning'] = 50


def example_plot(ax, fontsize=12, hide_labels=False):
    ax.plot([1, 2])

    ax.locator_params(nbins=3)
    if hide_labels:
        ax.set_xticklabels([])
        ax.set_yticklabels([])
    else:
        ax.set_xlabel('x-label', fontsize=fontsize)
        ax.set_ylabel('y-label', fontsize=fontsize)
        ax.set_title('Title', fontsize=fontsize)

fig, ax = plt.subplots(layout=None)
example_plot(ax, fontsize=24)
```



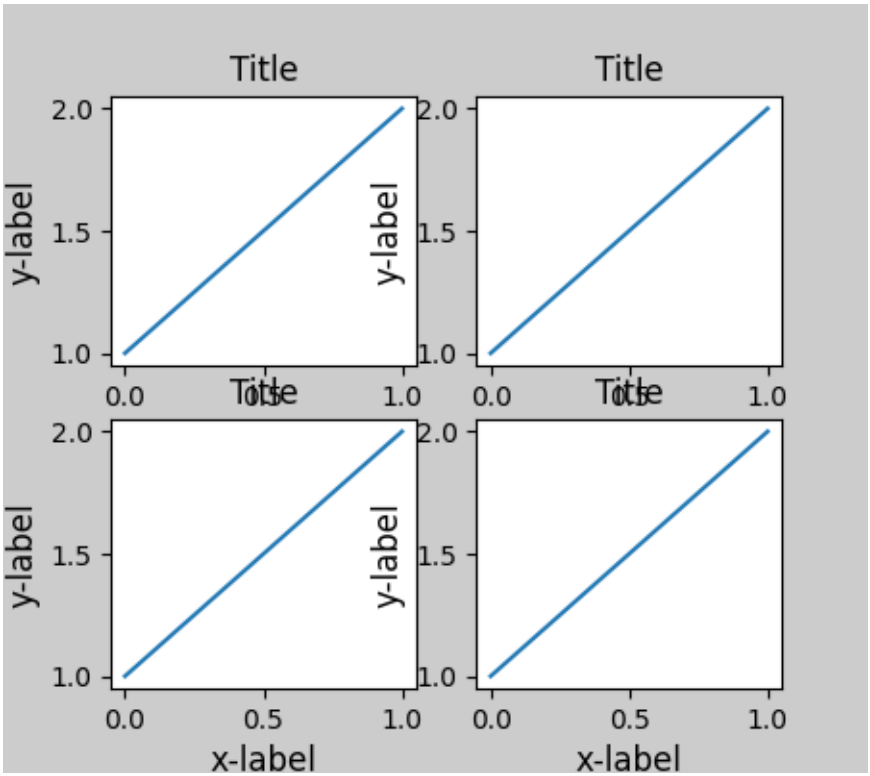

To prevent this, the location of Axes needs to be adjusted. For subplots, this can be done manually by adjusting the subplot parameters using *Figure.subplots_adjust*. However, specifying your figure with the `layout="constrained"` keyword argument will do the adjusting automatically.

```
fig, ax = plt.subplots(layout="constrained")
example_plot(ax, fontsize=24)
```



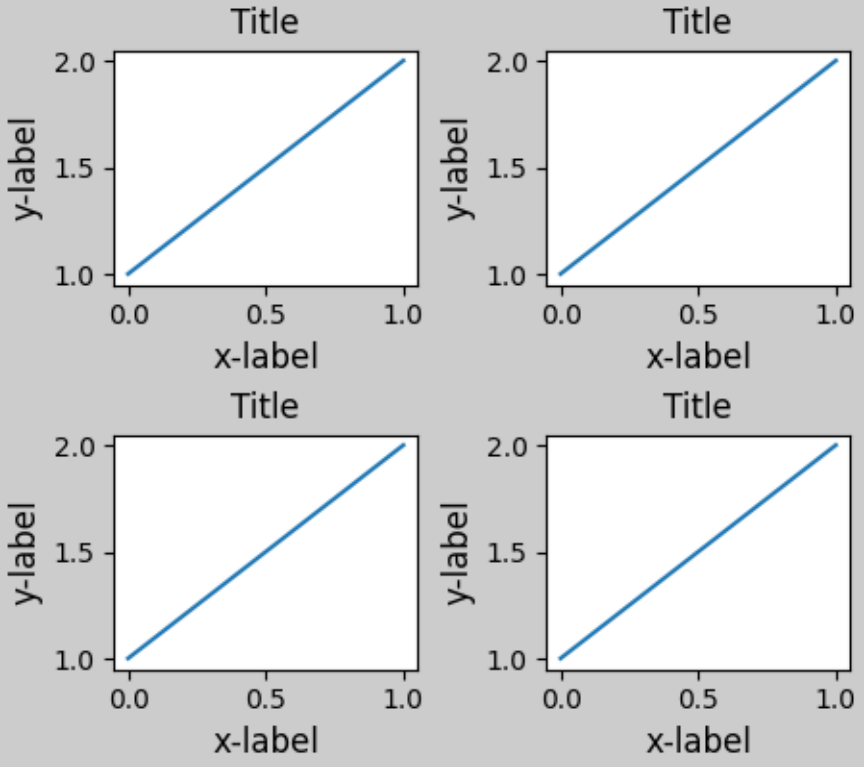

When you have multiple subplots, often you see labels of different Axes overlapping each other.

```
fig, axs = plt.subplots(2, 2, layout=None)
for ax in axs.flat:
    example_plot(ax)
```




Specifying `layout="constrained"` in the call to `plt.subplots` causes the layout to be properly constrained.

```
fig, axs = plt.subplots(2, 2, layout="constrained")
for ax in axs.flat:
    example_plot(ax)
```




### Colorbars

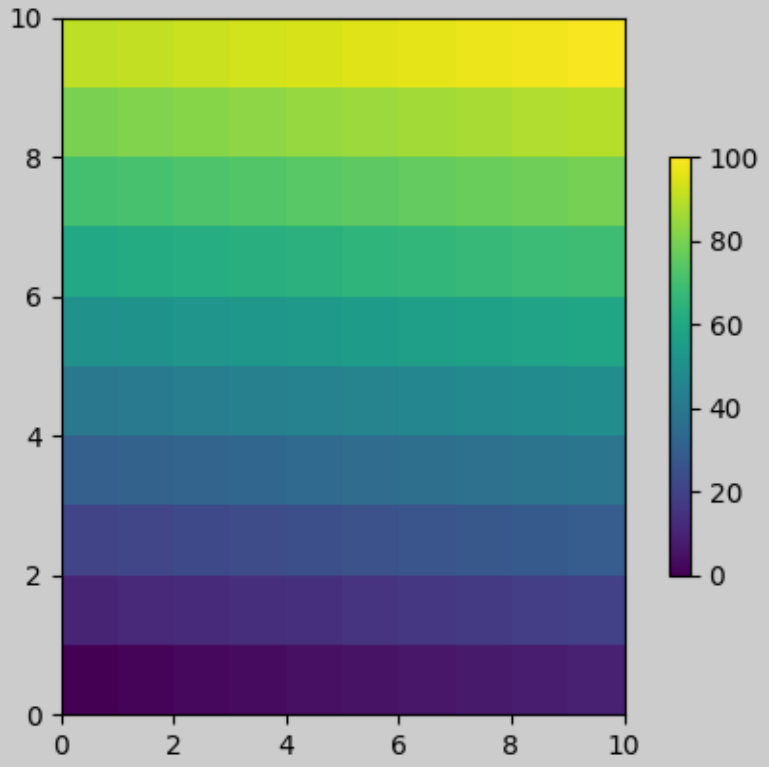
If you create a colorbar with *Figure.colorbar*, you need to make room for it. *Constrained layout* does this automatically. Note that if you specify `use_gridspec=True` it will be ignored because this option is made for improving the layout via `tight_layout`.

---

**Note:** For the *pcolormesh* keyword arguments (`pc_kwargs`) we use a dictionary to keep the calls consistent across this document.

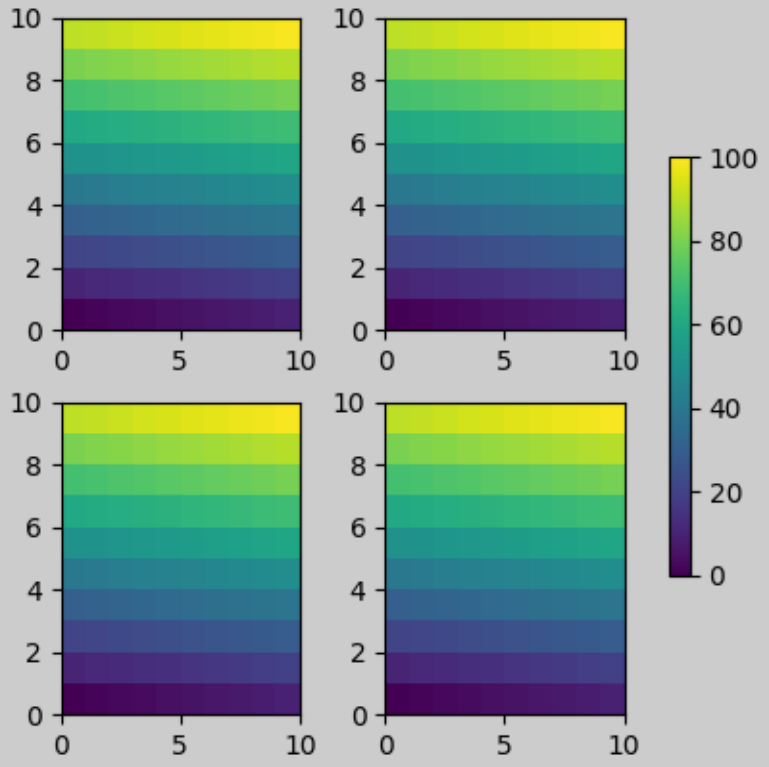
---

```
arr = np.arange(100).reshape((10, 10))
norm = mcolors.Normalize(vmin=0., vmax=100.)
# see note above: this makes all pcolormesh calls consistent:
pc_kwargs = {'rasterized': True, 'cmap': 'viridis', 'norm': norm}
fig, ax = plt.subplots(figsize=(4, 4), layout="constrained")
im = ax.pcolormesh(arr, **pc_kwargs)
fig.colorbar(im, ax=ax, shrink=0.6)
```



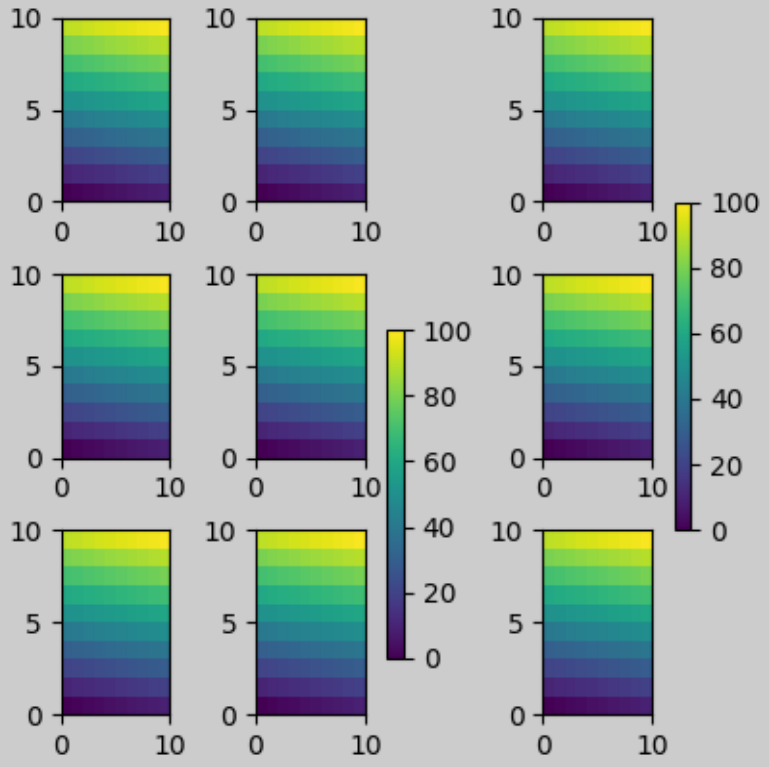

If you specify a list of Axes (or other iterable container) to the `ax` argument of `colorbar`, *constrained layout* will take space from the specified Axes.

```
fig, axs = plt.subplots(2, 2, figsize=(4, 4), layout="constrained")
for ax in axs.flat:
    im = ax.pcolormesh(arr, **pc_kwargs)
fig.colorbar(im, ax=axs, shrink=0.6)
```




If you specify a list of Axes from inside a grid of Axes, the colorbar will steal space appropriately, and leave a gap, but all subplots will still be the same size.

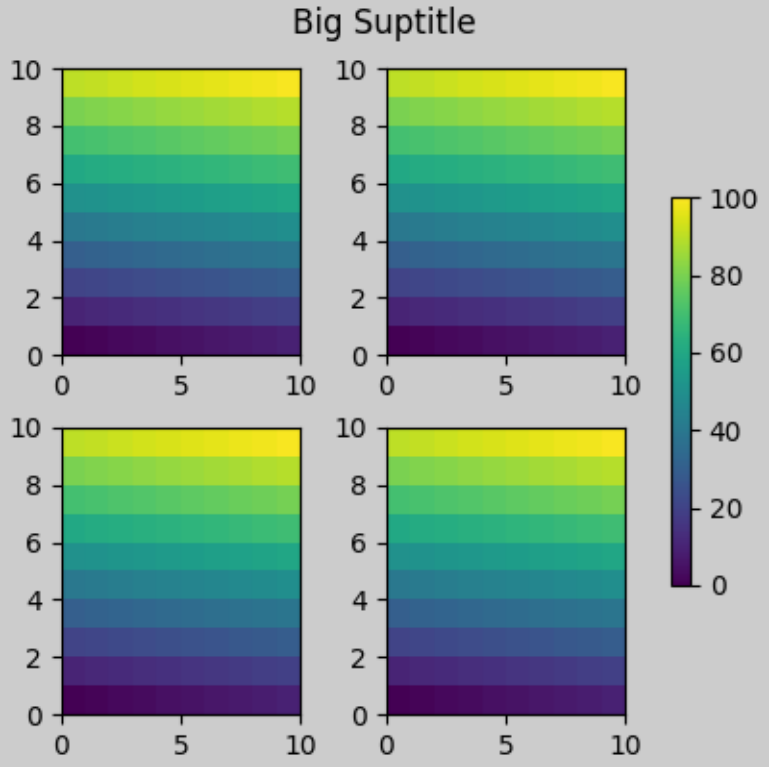
```
fig, axs = plt.subplots(3, 3, figsize=(4, 4), layout="constrained")
for ax in axs.flat:
    im = ax.pcolormesh(arr, **pc_kwargs)
fig.colorbar(im, ax=axs[1:, 1], shrink=0.8)
fig.colorbar(im, ax=axs[:, -1], shrink=0.6)
```




### Suptitle

*Constrained layout* can also make room for `suptitle`.

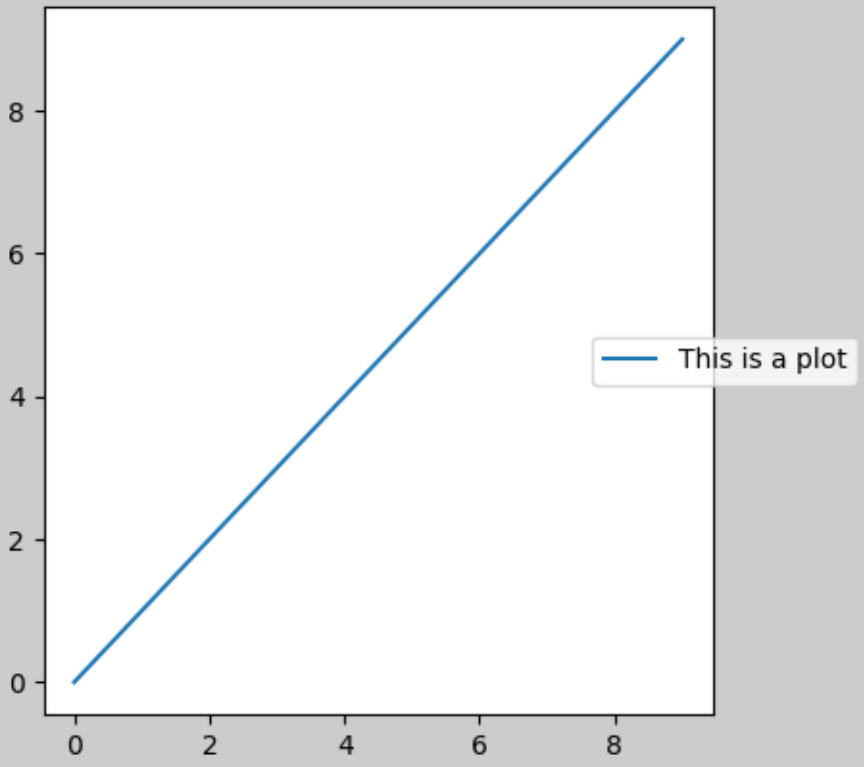
```
fig, axs = plt.subplots(2, 2, figsize=(4, 4), layout="constrained")
for ax in axs.flat:
    im = ax.pcolormesh(arr, **pc_kwargs)
fig.colorbar(im, ax=axs, shrink=0.6)
fig.suptitle('Big Suptitle')
```




### Legends

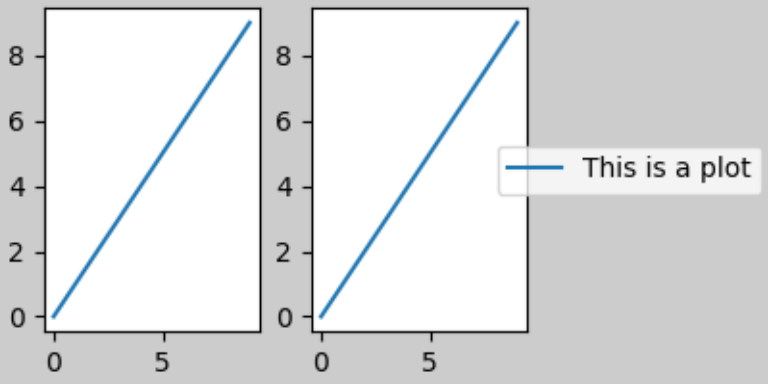
Legends can be placed outside of their parent axis. *Constrained layout* is designed to handle this for `Axes.legend()`. However, *constrained layout* does *not* handle legends being created via `Figure.legend()` (yet).

```
fig, ax = plt.subplots(layout="constrained")
ax.plot(np.arange(10), label='This is a plot')
ax.legend(loc='center left', bbox_to_anchor=(0.8, 0.5))
```



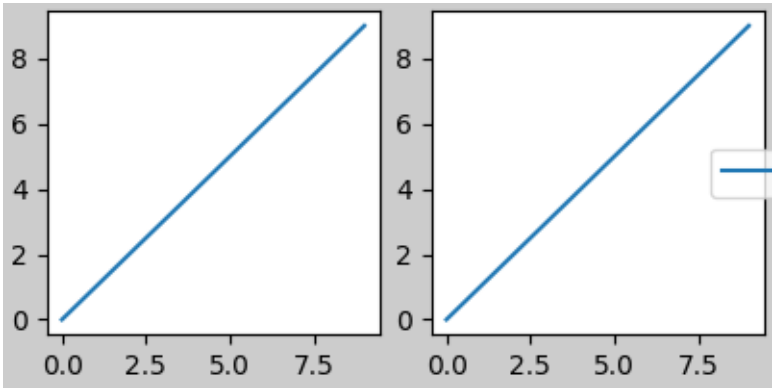

However, this will steal space from a subplot layout:

```
fig, axs = plt.subplots(1, 2, figsize=(4, 2), layout="constrained")
axs[0].plot(np.arange(10))
axs[1].plot(np.arange(10), label='This is a plot')
axs[1].legend(loc='center left', bbox_to_anchor=(0.8, 0.5))
```

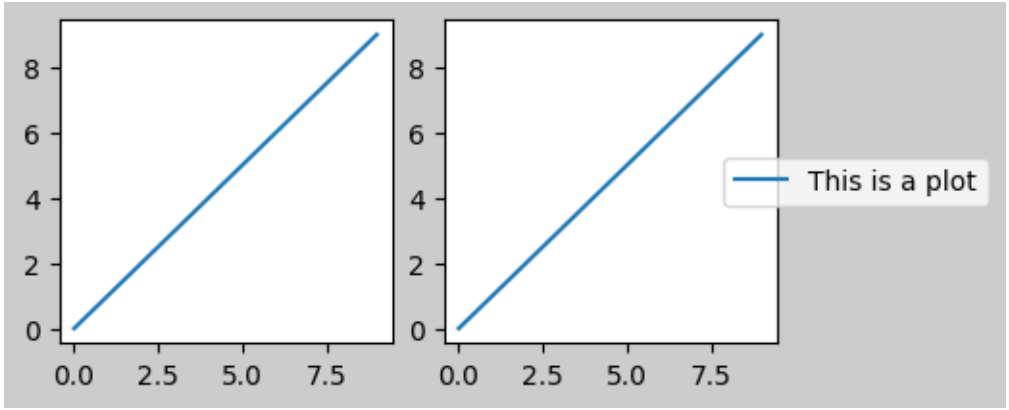



In order for a legend or other artist to *not* steal space from the subplot layout, we can `leg.set_in_layout(False)`. Of course this can mean the legend ends up cropped, but can be useful if the plot is subsequently called with `fig.savefig('outname.png', bbox_inches='tight')`. Note, however, that the legend's `get_in_layout` status will have to be toggled again to make the saved file work, and we must manually trigger a draw if we want *constrained layout* to adjust the size of the Axes before printing.

```
fig, axs = plt.subplots(1, 2, figsize=(4, 2), layout="constrained")

axs[0].plot(np.arange(10))
axs[1].plot(np.arange(10), label='This is a plot')
leg = axs[1].legend(loc='center left', bbox_to_anchor=(0.8, 0.5))
leg.set_in_layout(False)
# trigger a draw so that constrained layout is executed once
# before we turn it off when printing....
fig.canvas.draw()
# we want the legend included in the bbox_inches='tight' calcs.
leg.set_in_layout(True)
# we don't want the layout to change at this point.
fig.set_layout_engine('none')
try:
    fig.savefig('../../../doc/_static/constrained_layout_1b.png',
                bbox_inches='tight', dpi=100)
except FileNotFoundError:
    # this allows the script to keep going if run interactively and
    # the directory above doesn't exist
    pass
```

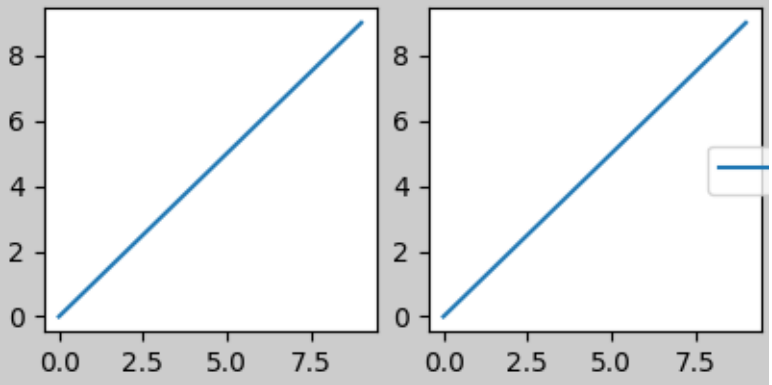



The saved file looks like:

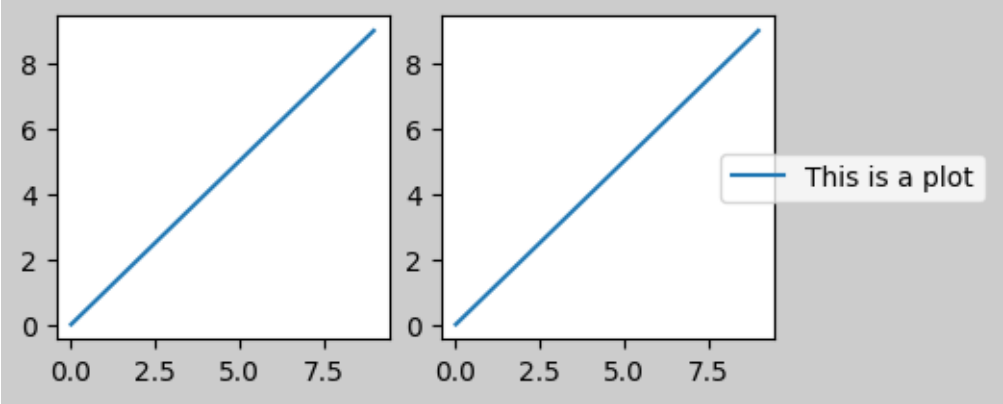



A better way to get around this awkwardness is to simply use the legend method provided by *Figure.legend*:

```
fig, axs = plt.subplots(1, 2, figsize=(4, 2), layout="constrained")
axs[0].plot(np.arange(10))
lines = axs[1].plot(np.arange(10), label='This is a plot')
labels = [l.get_label() for l in lines]
leg = fig.legend(lines, labels, loc='center left',
                 bbox_to_anchor=(0.8, 0.5), bbox_transform=axs[1].transAxes)
try:
    fig.savefig('../../../doc/_static/constrained_layout_2b.png',
                bbox_inches='tight', dpi=100)
except FileNotFoundError:
    # this allows the script to keep going if run interactively and
    # the directory above doesn't exist
    pass
```




The saved file looks like:




### Padding and spacing

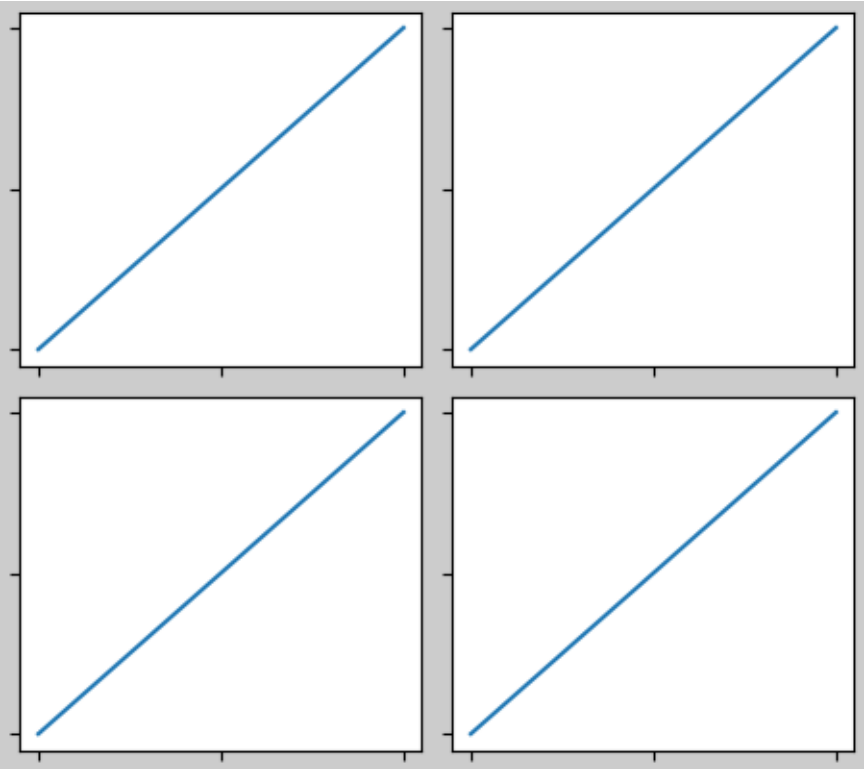
Padding between Axes is controlled in the horizontal by *w_pad* and *wspace*, and vertical by *h_pad* and *hspace*. These can be edited via `set`. *w/h_pad* are the minimum space around the Axes in units of inches:

```
fig, axs = plt.subplots(2, 2, layout="constrained")
for ax in axs.flat:
    example_plot(ax, hide_labels=True)
```

(continues on next page)

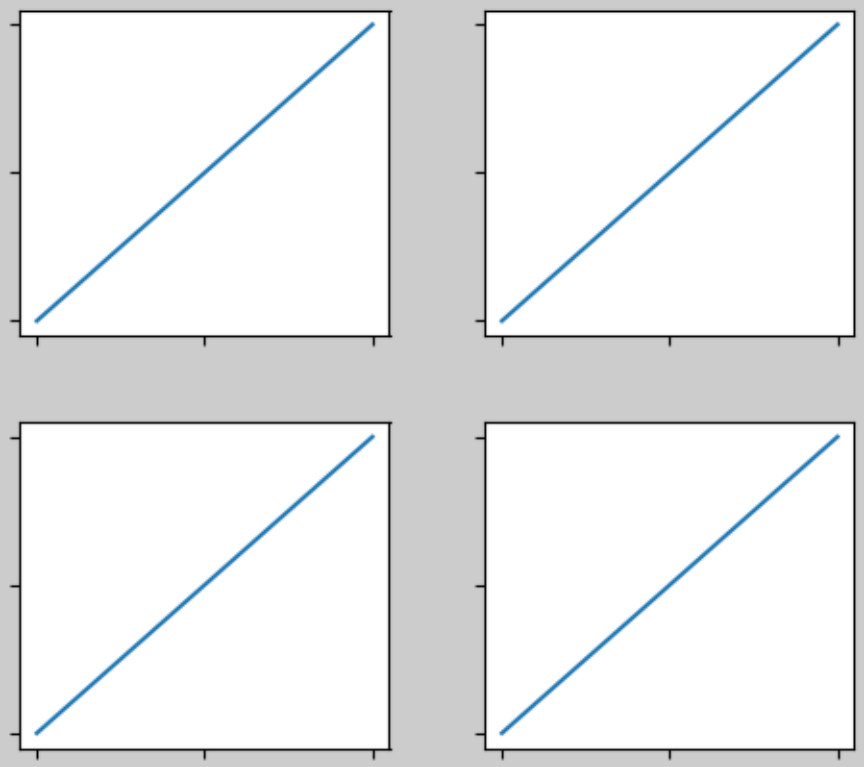
(continued from previous page)

```
fig.get_layout_engine().set(w_pad=4 / 72, h_pad=4 / 72, hspace=0,
                            wspace=0)
```



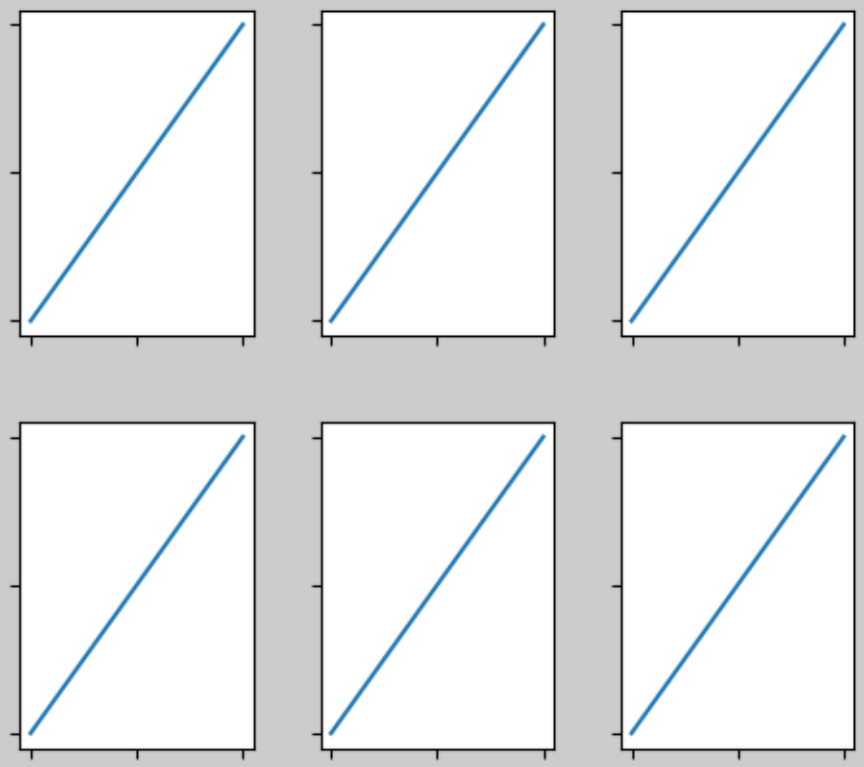
Spacing between subplots is further set by *wspace* and *hspace*. These are specified as a fraction of the size of the subplot group as a whole. If these values are smaller than *w_pad* or *h_pad*, then the fixed pads are used instead. Note in the below how the space at the edges doesn't change from the above, but the space between subplots does.

```
fig, axs = plt.subplots(2, 2, layout="constrained")
for ax in axs.flat:
    example_plot(ax, hide_labels=True)
fig.get_layout_engine().set(w_pad=4 / 72, h_pad=4 / 72, hspace=0.2,
                            wspace=0.2)
```



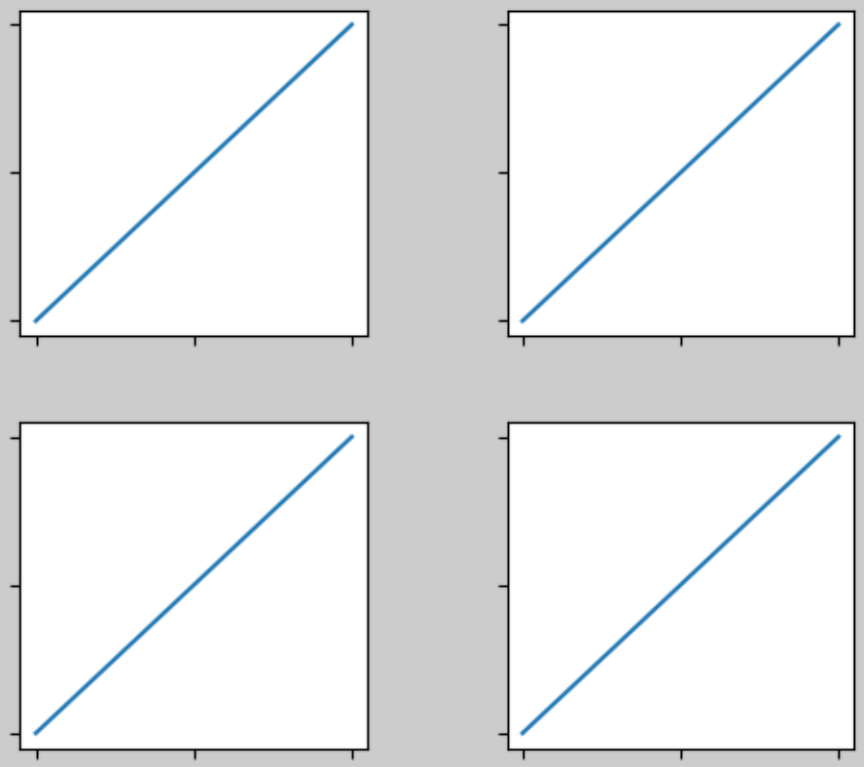
If there are more than two columns, the *wspace* is shared between them, so here the wspace is divided in two, with a *wspace* of 0.1 between each column:

```
fig, axs = plt.subplots(2, 3, layout="constrained")
for ax in axs.flat:
    example_plot(ax, hide_labels=True)
fig.get_layout_engine().set(w_pad=4 / 72, h_pad=4 / 72, hspace=0.2,
                            wspace=0.2)
```



GridSpecs also have optional *hspace* and *wspace* keyword arguments, that will be used instead of the pads set by *constrained layout*:

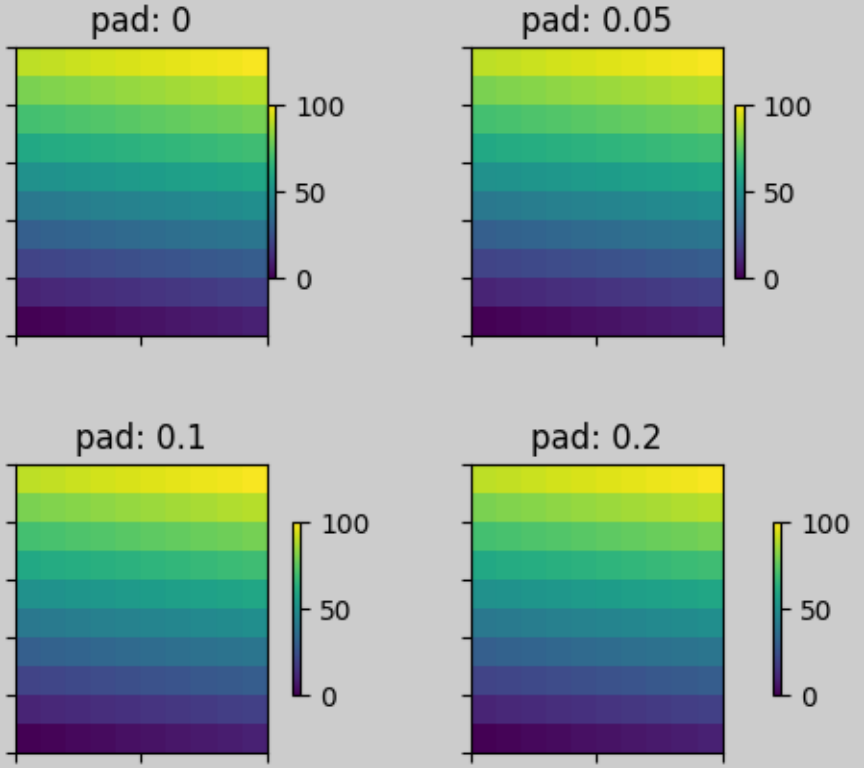
```
fig, axs = plt.subplots(2, 2, layout="constrained",
                        gridspec_kw={'wspace': 0.3, 'hspace': 0.2})
for ax in axs.flat:
    example_plot(ax, hide_labels=True)
# this has no effect because the space set in the gridspec trumps the
# space set in *constrained layout*.
fig.get_layout_engine().set(w_pad=4 / 72, h_pad=4 / 72, hspace=0.0,
                            wspace=0.0)
```



### Spacing with colorbars

Colorbars are placed a distance *pad* from their parent, where *pad* is a fraction of the width of the parent(s). The spacing to the next subplot is then given by *w/hspace*.

```
fig, axs = plt.subplots(2, 2, layout="constrained")
pads = [0, 0.05, 0.1, 0.2]
for pad, ax in zip(pads, axs.flat):
    pc = ax.pcolormesh(arr, **pc_kwargs)
    fig.colorbar(pc, ax=ax, shrink=0.6, pad=pad)
    ax.set_xticklabels([])
    ax.set_yticklabels([])
    ax.set_title(f'pad: {pad}')
fig.get_layout_engine().set(w_pad=2 / 72, h_pad=2 / 72, hspace=0.2,
                            wspace=0.2)
```

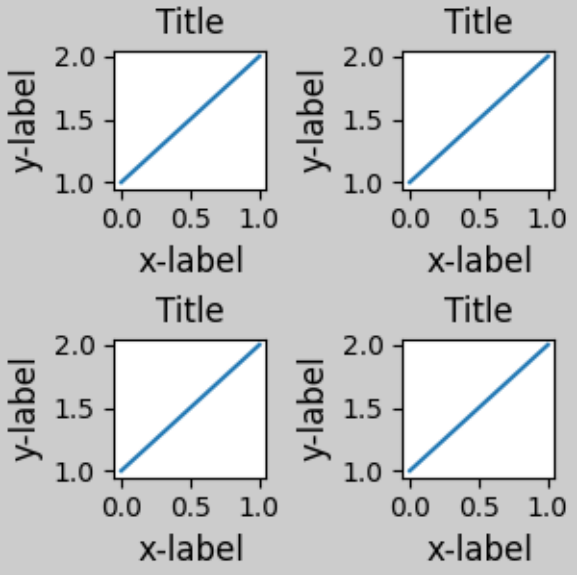



### rcParams

There are five *rcParams* that can be set, either in a script or in the `matplotlibrc` file. They all have the prefix `figure.constrained_layout`:

- *use*: Whether to use *constrained layout*. Default is False
- *w_pad*, *h_pad*: Padding around Axes objects. Float representing inches. Default is 3./72. inches (3 pts)
- *wspace*, *hspace*: Space between subplot groups. Float representing a fraction of the subplot widths being separated. Default is 0.02.

```
plt.rcParams['figure.constrained_layout.use'] = True
fig, axs = plt.subplots(2, 2, figsize=(3, 3))
for ax in axs.flat:
    example_plot(ax)
```

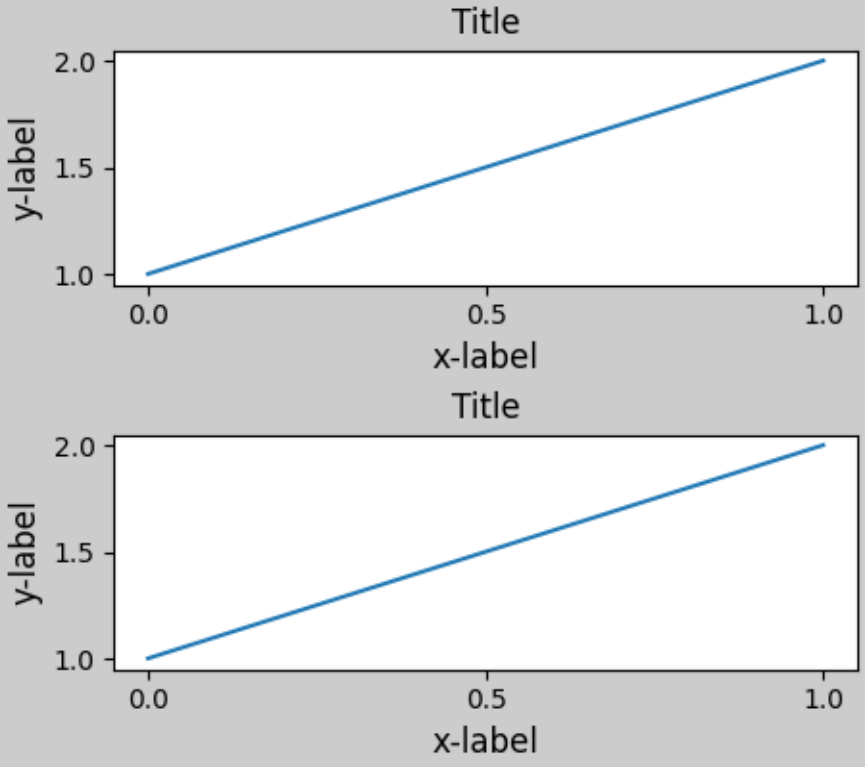



### Use with GridSpec

*Constrained layout* is meant to be used with *subplots()*, *subplot_mosaic()*, or *GridSpec()* with *add_subplot()*.

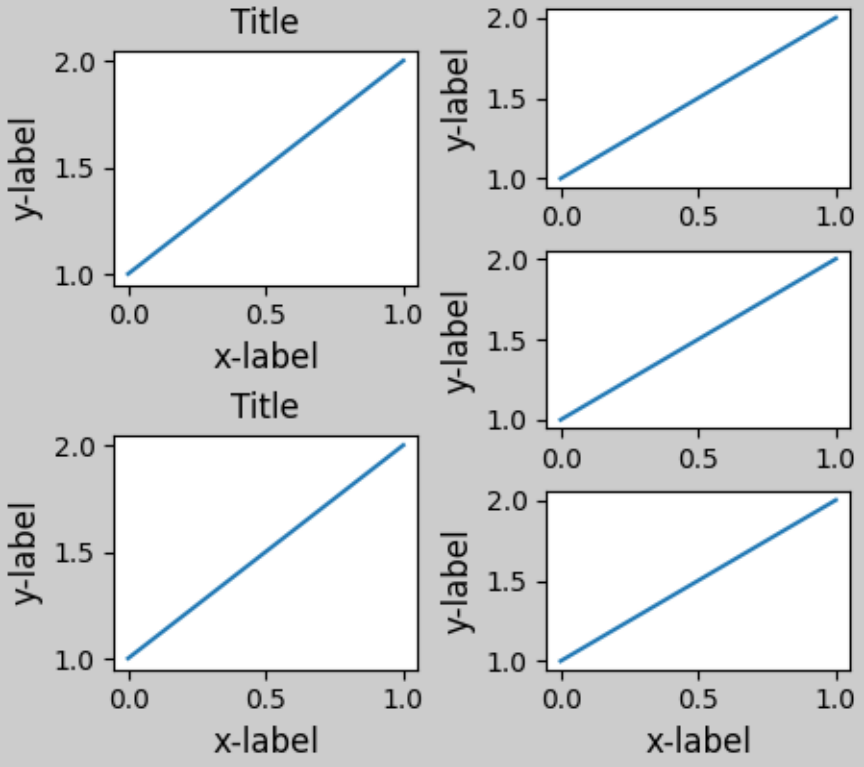
Note that in what follows `layout="constrained"`

```
plt.rcParams['figure.constrained_layout.use'] = False
fig = plt.figure(layout="constrained")

gs1 = gridspec.GridSpec(2, 1, figure=fig)
ax1 = fig.add_subplot(gs1[0])
ax2 = fig.add_subplot(gs1[1])

example_plot(ax1)
example_plot(ax2)
```



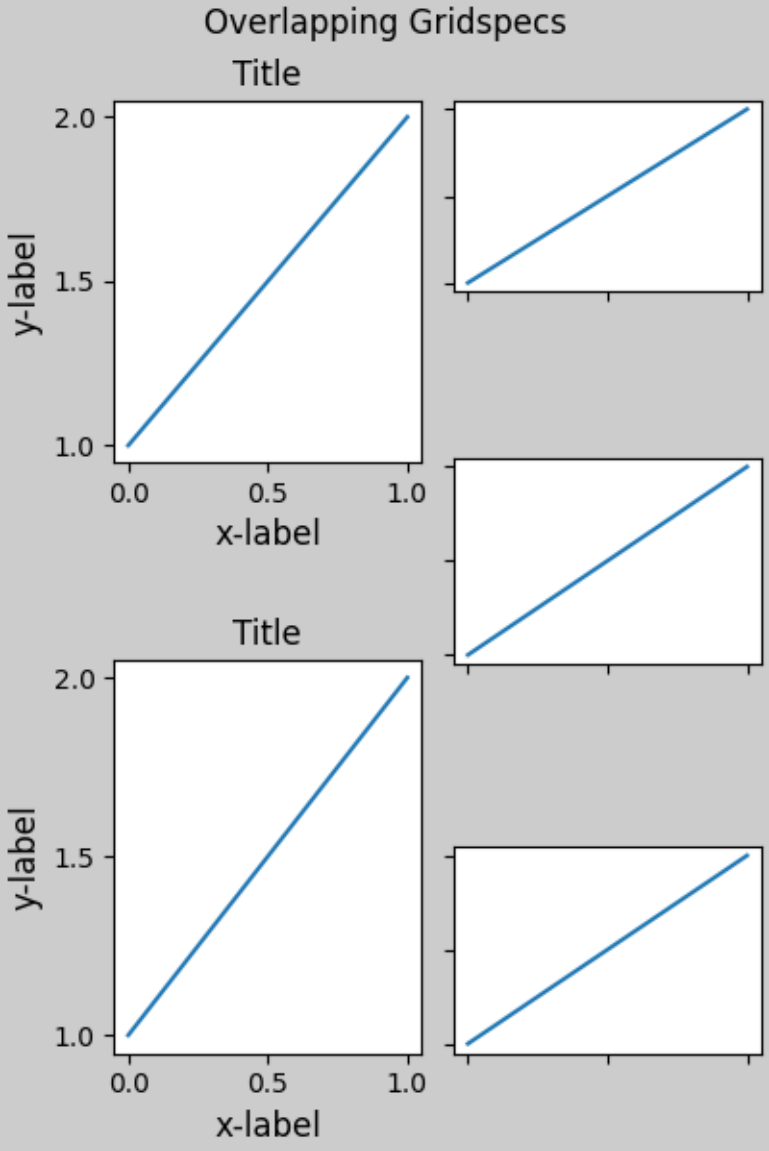

More complicated gridspec layouts are possible. Note here we use the convenience functions *add_gridspec* and *subgridspec*.

```
fig = plt.figure(layout="constrained")

gs0 = fig.add_gridspec(1, 2)

gs1 = gs0[0].subgridspec(2, 1)
ax1 = fig.add_subplot(gs1[0])
ax2 = fig.add_subplot(gs1[1])

example_plot(ax1)
example_plot(ax2)

gs2 = gs0[1].subgridspec(3, 1)

for ss in gs2:
    ax = fig.add_subplot(ss)
    example_plot(ax)
    ax.set_title("")
    ax.set_xlabel("")

ax.set_xlabel("x-label", fontsize=12)
```




Note that in the above the left and right columns don't have the same vertical extent. If we want the top and bottom of the two grids to line up then they need to be in the same gridspec. We need to make this figure larger as well in order for the Axes not to collapse to zero height:

```
fig = plt.figure(figsize=(4, 6), layout="constrained")

gs0 = fig.add_gridspec(6, 2)

ax1 = fig.add_subplot(gs0[:3, 0])
ax2 = fig.add_subplot(gs0[3:, 0])

example_plot(ax1)
example_plot(ax2)

ax = fig.add_subplot(gs0[0:2, 1])
example_plot(ax, hide_labels=True)
ax = fig.add_subplot(gs0[2:4, 1])
example_plot(ax, hide_labels=True)
ax = fig.add_subplot(gs0[4:, 1])
example_plot(ax, hide_labels=True)
fig.suptitle('Overlapping Gridspecs')
```



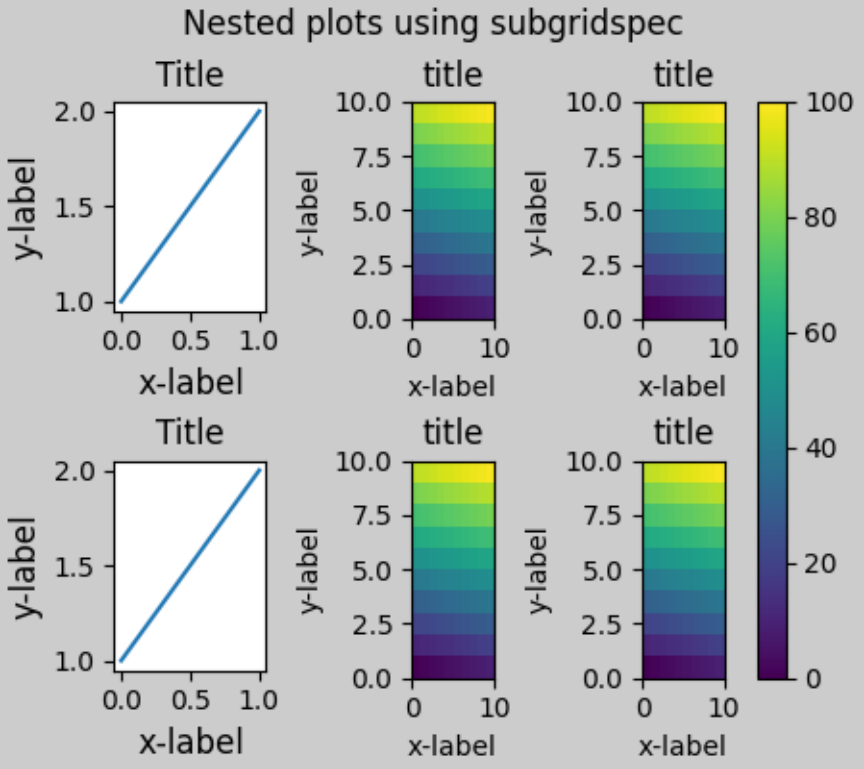

This example uses two gridspecs to have the colorbar only pertain to one set of pcolors. Note how the left column is wider than the two right-hand columns because of this. Of course, if you wanted the subplots to be the same size you only needed one gridspec. Note that the same effect can be achieved using *subfigures*.

```
fig = plt.figure(layout="constrained")
gs0 = fig.add_gridspec(1, 2, figure=fig, width_ratios=[1, 2])
gs_left = gs0[0].subgridspec(2, 1)
gs_right = gs0[1].subgridspec(2, 2)

for gs in gs_left:
    ax = fig.add_subplot(gs)
    example_plot(ax)
axs = []
for gs in gs_right:
    ax = fig.add_subplot(gs)
    pcm = ax.pcolormesh(arr, **pc_kwargs)
```

(continues on next page)

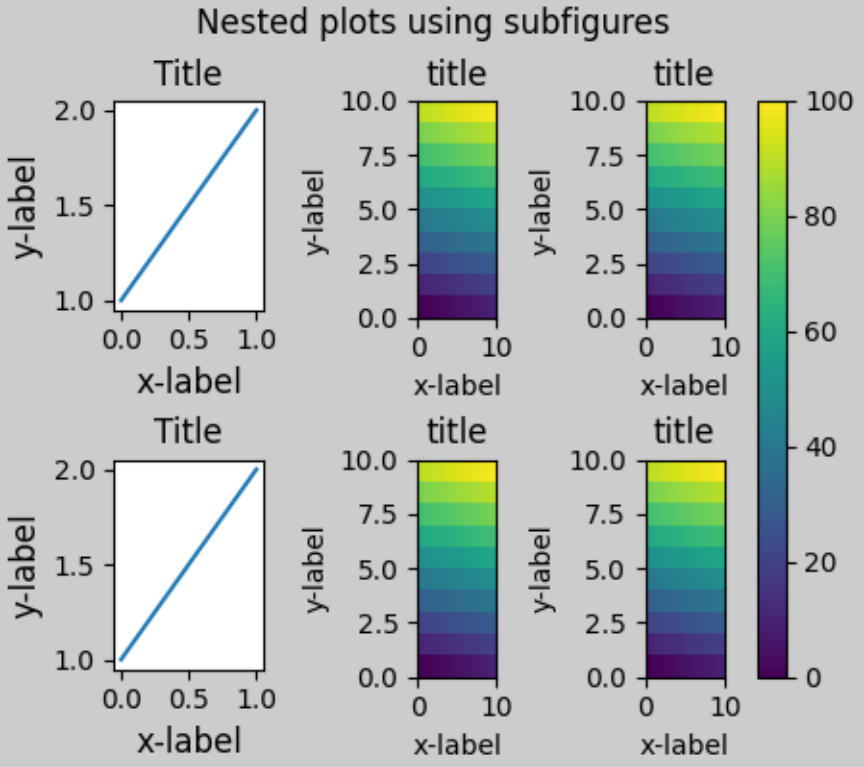
(continued from previous page)

```
    ax.set_xlabel('x-label')
    ax.set_ylabel('y-label')
    ax.set_title('title')
    axs += [ax]
fig.suptitle('Nested plots using subgridspec')
fig.colorbar(pcm, ax=axs)
```




Rather than using subgridspecs, Matplotlib now provides *subfigures* which also work with *constrained layout*:

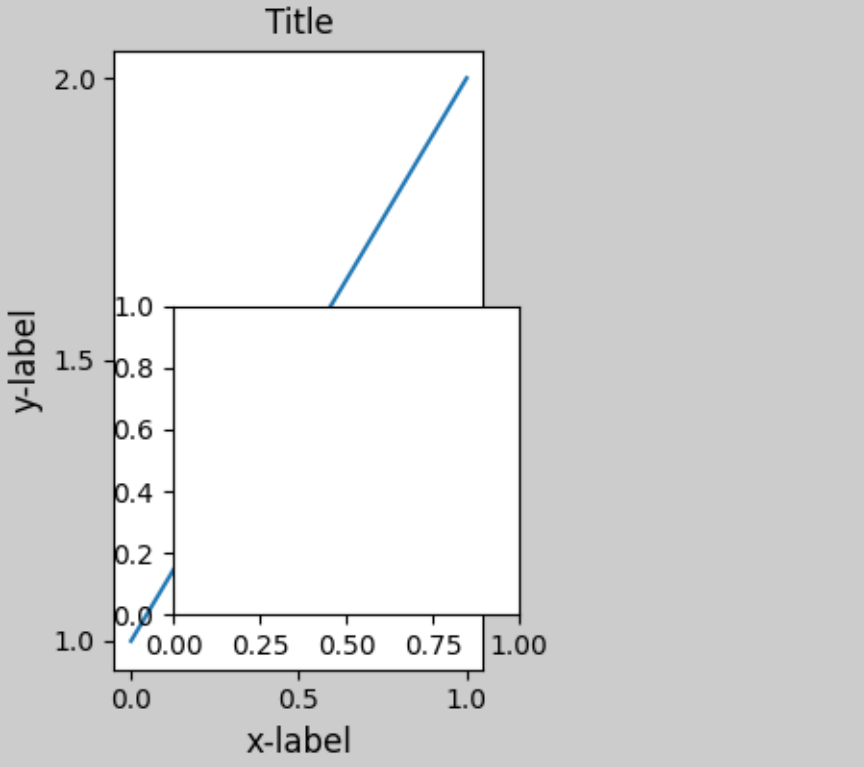
```
fig = plt.figure(layout="constrained")
sfigs = fig.subfigures(1, 2, width_ratios=[1, 2])

axs_left = sfigs[0].subplots(2, 1)
for ax in axs_left.flat:
    example_plot(ax)

axs_right = sfigs[1].subplots(2, 2)
for ax in axs_right.flat:
    pcm = ax.pcolormesh(arr, **pc_kwargs)
    ax.set_xlabel('x-label')
    ax.set_ylabel('y-label')
    ax.set_title('title')
fig.colorbar(pcm, ax=axs_right)
fig.suptitle('Nested plots using subfigures')
```




### Manually setting Axes positions

There can be good reasons to manually set an Axes position. A manual call to `set_position` will set the Axes so *constrained layout* has no effect on it anymore. (Note that *constrained layout* still leaves the space for the Axes that is moved).

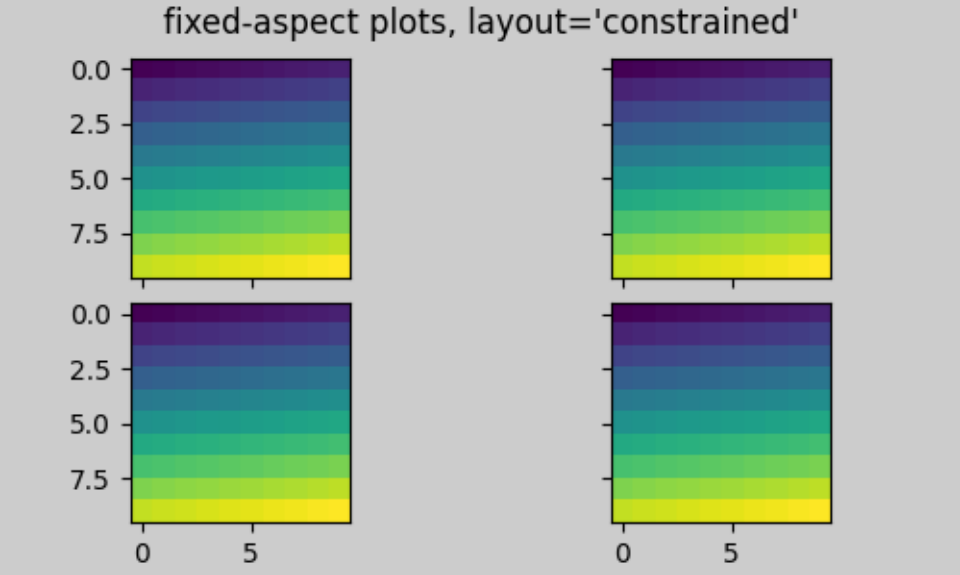
```
fig, axs = plt.subplots(1, 2, layout="constrained")
example_plot(axs[0], fontsize=12)
axs[1].set_position([0.2, 0.2, 0.4, 0.4])
```




### Grids of fixed aspect-ratio Axes: "compressed" layout

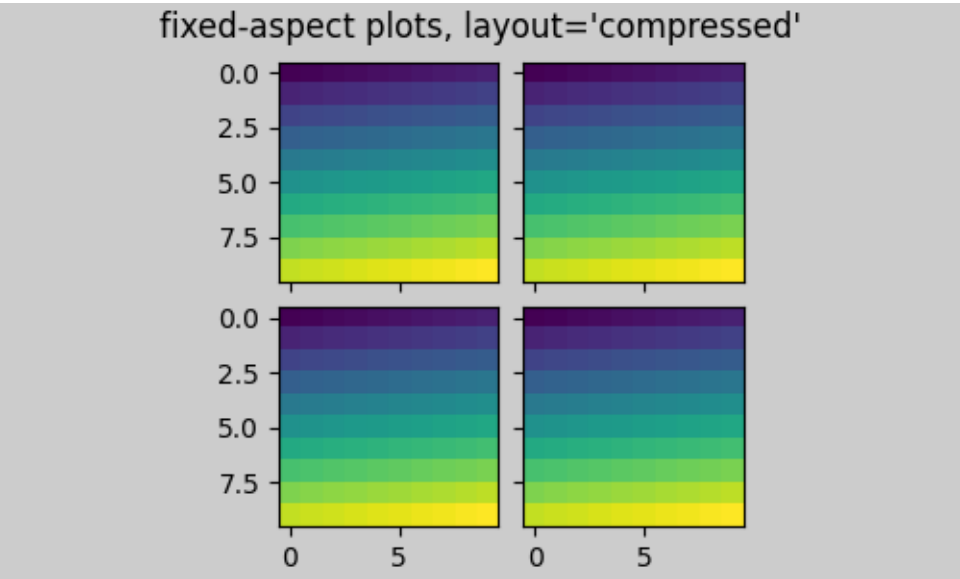
*Constrained layout* operates on the grid of "original" positions for Axes. However, when Axes have fixed aspect ratios, one side is usually made shorter, and leaves large gaps in the shortened direction. In the following, the Axes are square, but the figure quite wide so there is a horizontal gap:

```
fig, axs = plt.subplots(2, 2, figsize=(5, 3),
                        sharex=True, sharey=True, layout="constrained")
for ax in axs.flat:
    ax.imshow(arr)
fig.suptitle("fixed-aspect plots, layout='constrained'")
```




One obvious way of fixing this is to make the figure size more square, however, closing the gaps exactly requires trial and error. For simple grids of Axes we can use `layout="compressed"` to do the job for us:

```
fig, axs = plt.subplots(2, 2, figsize=(5, 3),
                        sharex=True, sharey=True, layout='compressed')
for ax in axs.flat:
    ax.imshow(arr)
fig.suptitle("fixed-aspect plots, layout='compressed'")
```



### Manually turning off *constrained layout*

*Constrained layout* usually adjusts the Axes positions on each draw of the figure. If you want to get the spacing provided by *constrained layout* but not have it update, then do the initial draw and then call `fig.set_layout_engine('none')`. This is potentially useful for animations where the tick labels may change length.

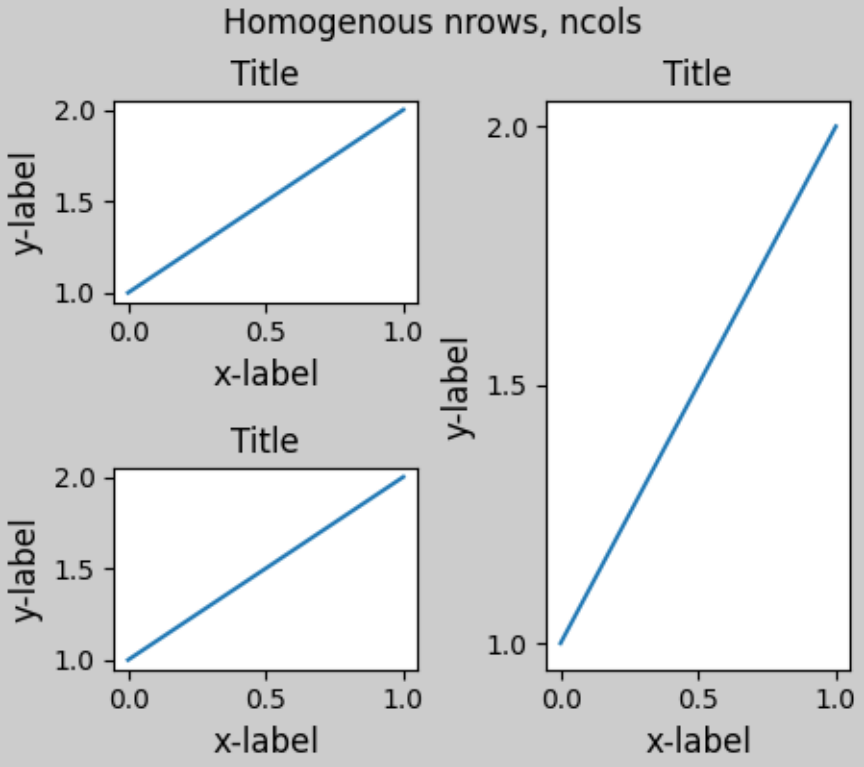
Note that *constrained layout* is turned off for `ZOOM` and `PAN` GUI events for the backends that use the toolbar. This prevents the Axes from changing position during zooming and panning.

### Limitations

### Incompatible functions

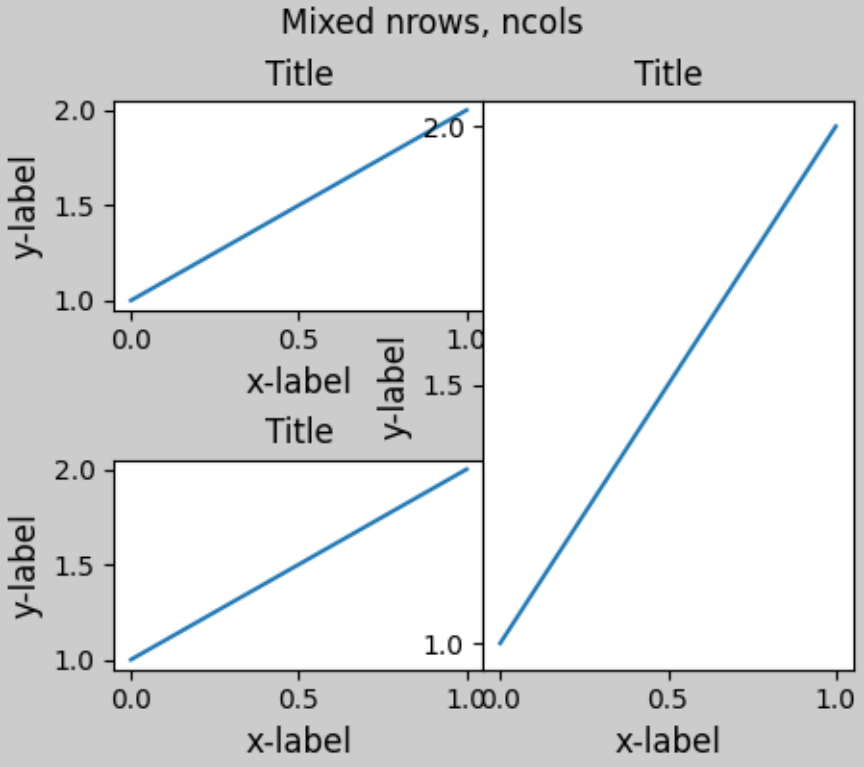
*Constrained layout* will work with *pyplot.subplot*, but only if the number of rows and columns is the same for each call. The reason is that each call to *pyplot.subplot* will create a new *GridSpec* instance if the geometry is not the same, and *constrained layout*. So the following works fine:

```
fig = plt.figure(layout="constrained")

ax1 = plt.subplot(2, 2, 1)
ax2 = plt.subplot(2, 2, 3)
# third Axes that spans both rows in second column:
ax3 = plt.subplot(2, 2, (2, 4))

example_plot(ax1)
example_plot(ax2)
example_plot(ax3)
plt.suptitle('Homogenous nrows, ncols')
```



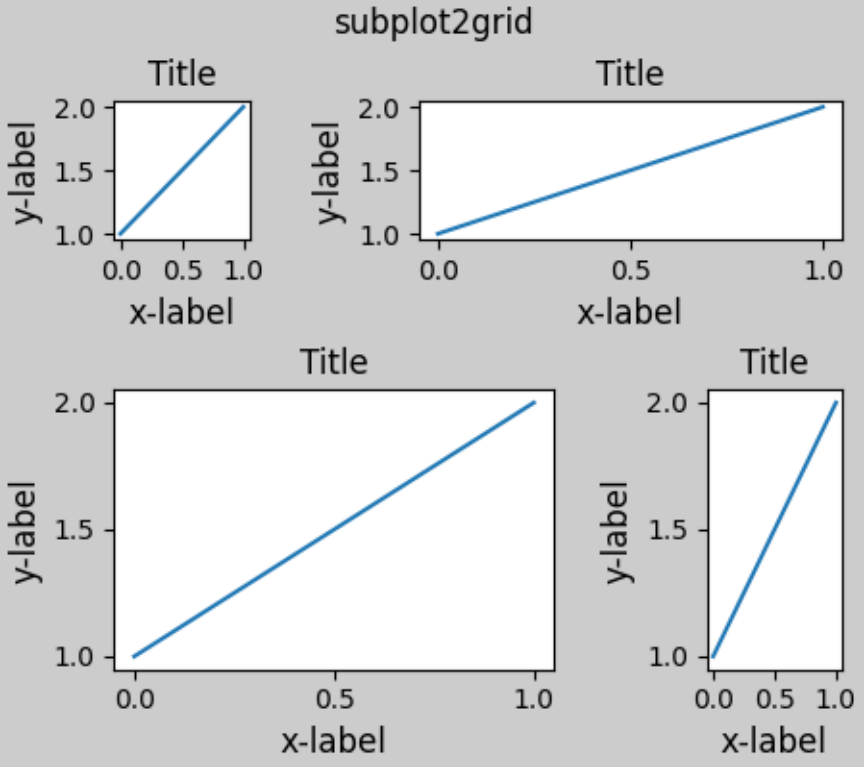

but the following leads to a poor layout:

```
fig = plt.figure(layout="constrained")

ax1 = plt.subplot(2, 2, 1)
ax2 = plt.subplot(2, 2, 3)
ax3 = plt.subplot(1, 2, 2)

example_plot(ax1)
example_plot(ax2)
example_plot(ax3)
plt.suptitle('Mixed nrows, ncols')
```




Similarly, *subplot2grid* works with the same limitation that nrows and ncols cannot change for the layout to look good.

```
fig = plt.figure(layout="constrained")

ax1 = plt.subplot2grid((3, 3), (0, 0))
ax2 = plt.subplot2grid((3, 3), (0, 1), colspan=2)
ax3 = plt.subplot2grid((3, 3), (1, 0), colspan=2, rowspan=2)
ax4 = plt.subplot2grid((3, 3), (1, 2), rowspan=2)

example_plot(ax1)
example_plot(ax2)
example_plot(ax3)
example_plot(ax4)
fig.suptitle('subplot2grid')
```




### Other caveats

- *Constrained layout* only considers ticklabels, axis labels, titles, and legends. Thus, other artists may be clipped and also may overlap.
- It assumes that the extra space needed for ticklabels, axis labels, and titles is independent of original location of Axes. This is often true, but there are rare cases where it is not.
- There are small differences in how the backends handle rendering fonts, so the results will not be pixel-identical.
- An artist using Axes coordinates that extend beyond the Axes boundary will result in unusual layouts when added to an Axes. This can be avoided by adding the artist directly to the `Figure` using `add_artist()`. See `ConnectionPatch` for an example.

### Debugging

*Constrained layout* can fail in somewhat unexpected ways. Because it uses a constraint solver the solver can find solutions that are mathematically correct, but that aren't at all what the user wants. The usual failure mode is for all sizes to collapse to their smallest allowable value. If this happens, it is for one of two reasons:

1. There was not enough room for the elements you were requesting to draw.
2. There is a bug - in which case open an issue at https://github.com/matplotlib/matplotlib/issues.

If there is a bug, please report with a self-contained example that does not require outside data or dependencies (other than numpy).

### Notes on the algorithm

The algorithm for the constraint is relatively straightforward, but has some complexity due to the complex ways we can lay out a figure.

Layout in Matplotlib is carried out with gridspecs via the *GridSpec* class. A gridspec is a logical division of the figure into rows and columns, with the relative width of the Axes in those rows and columns set by *width_ratios* and *height_ratios*.

In *constrained layout*, each gridspec gets a *layoutgrid* associated with it. The *layoutgrid* has a series of `left` and `right` variables for each column, and `bottom` and `top` variables for each row, and further it has a margin for each of left, right, bottom and top. In each row, the bottom/top margins are widened until all the decorators in that row are accommodated. Similarly, for columns and the left/right margins.

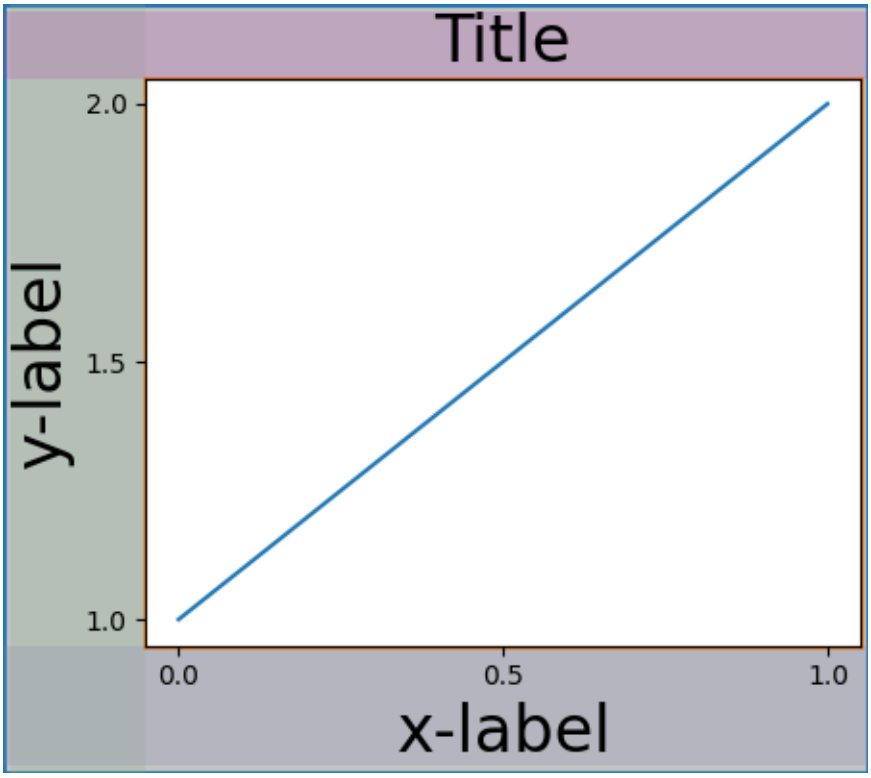
### Simple case: one Axes

For a single Axes the layout is straight forward. There is one parent layoutgrid for the figure consisting of one column and row, and a child layoutgrid for the gridspec that contains the Axes, again consisting of one row and column. Space is made for the "decorations" on each side of the Axes. In the code, this is accomplished by the entries in `do_constrained_layout()` like:

```
gridspec._layoutgrid[0, 0].edit_margin_min('left',
      -bbox.x0 + pos.x0 + w_pad)
```

where `bbox` is the tight bounding box of the Axes, and `pos` its position. Note how the four margins encompass the Axes decorations.

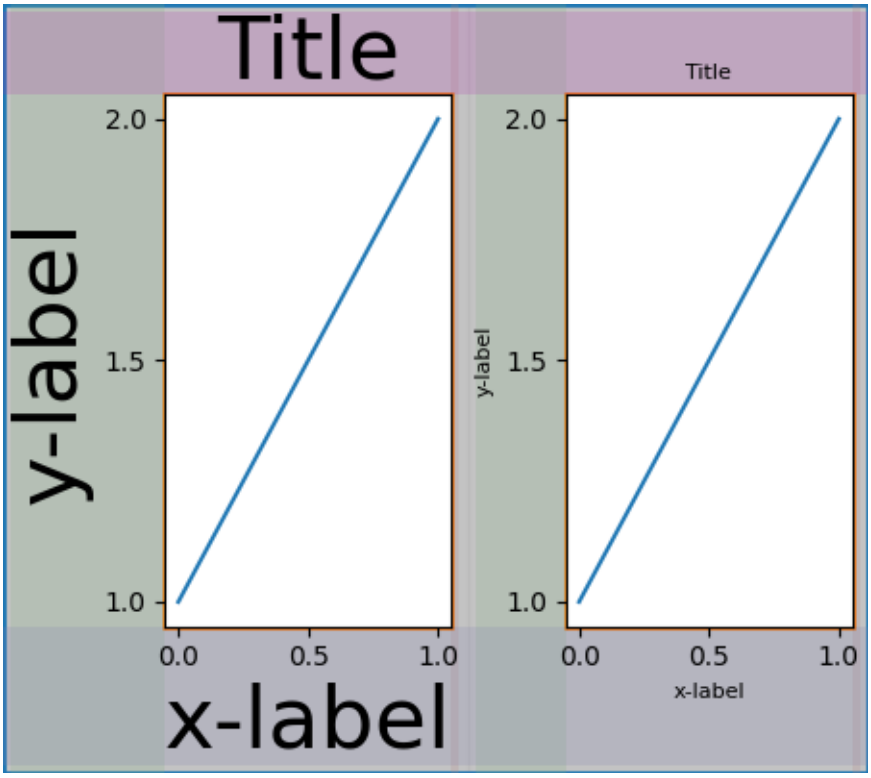
```
from matplotlib._layoutgrid import plot_children

fig, ax = plt.subplots(layout="constrained")
example_plot(ax, fontsize=24)
plot_children(fig)
```




### Simple case: two Axes

When there are multiple Axes they have their layouts bound in simple ways. In this example the left Axes has much larger decorations than the right, but they share a bottom margin, which is made large enough to accommodate the larger xlabel. Same with the shared top margin. The left and right margins are not shared, and hence are allowed to be different.

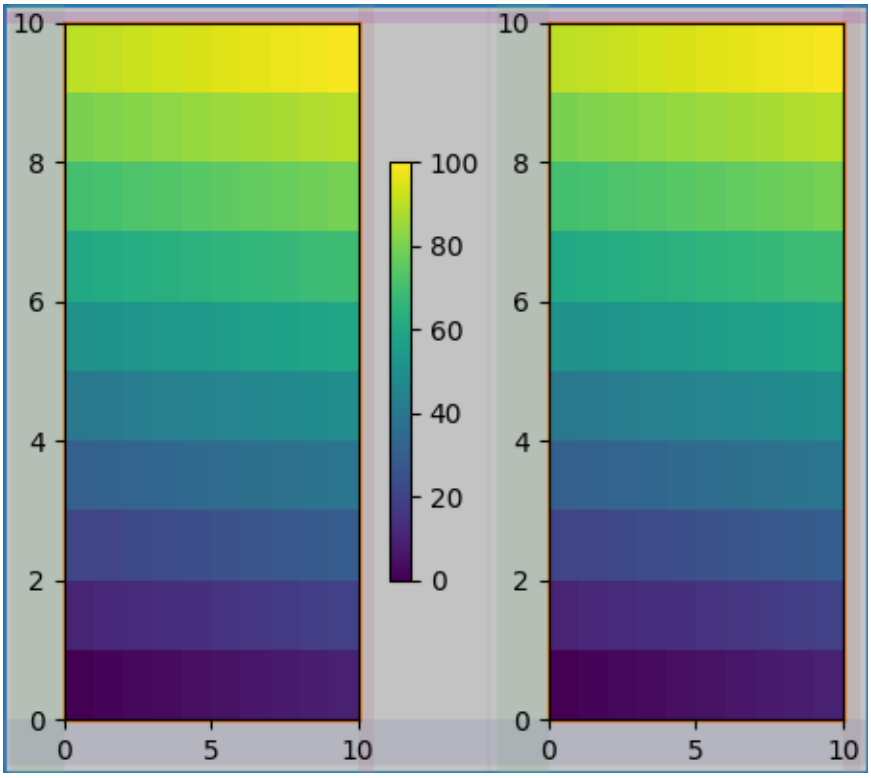
```
fig, ax = plt.subplots(1, 2, layout="constrained")
example_plot(ax[0], fontsize=32)
example_plot(ax[1], fontsize=8)
plot_children(fig)
```




### Two Axes and colorbar

A colorbar is simply another item that expands the margin of the parent layoutgrid cell:

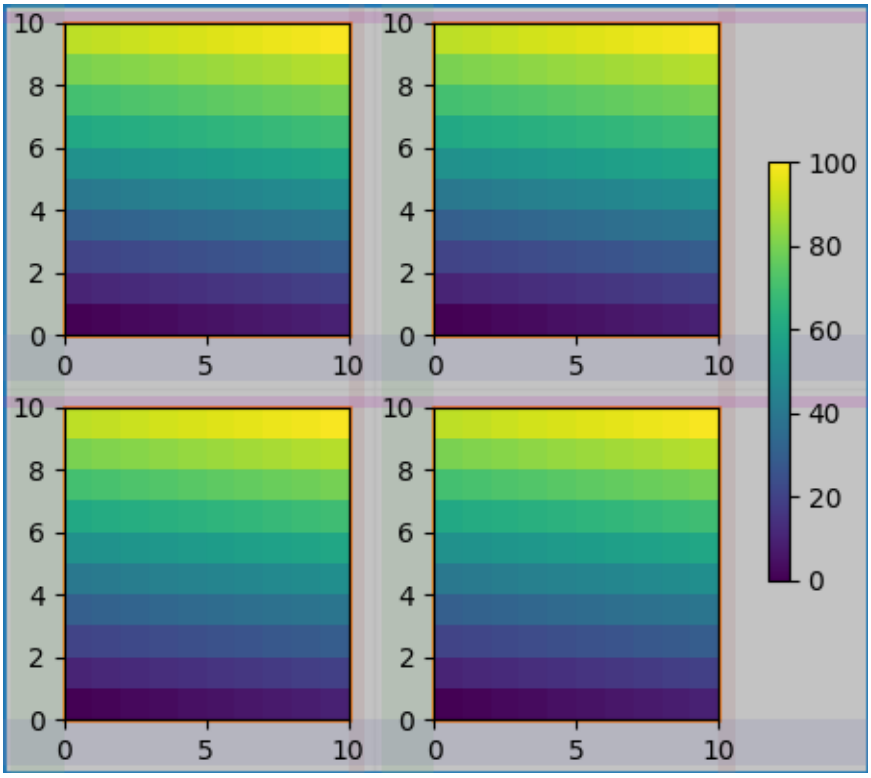
```
fig, ax = plt.subplots(1, 2, layout="constrained")
im = ax[0].pcolormesh(arr, **pc_kwargs)
fig.colorbar(im, ax=ax[0], shrink=0.6)
im = ax[1].pcolormesh(arr, **pc_kwargs)
plot_children(fig)
```




### Colorbar associated with a Gridspec

If a colorbar belongs to more than one cell of the grid, then it makes a larger margin for each:

```
fig, axs = plt.subplots(2, 2, layout="constrained")
for ax in axs.flat:
    im = ax.pcolormesh(arr, **pc_kwargs)
fig.colorbar(im, ax=axs, shrink=0.6)
plot_children(fig)
```

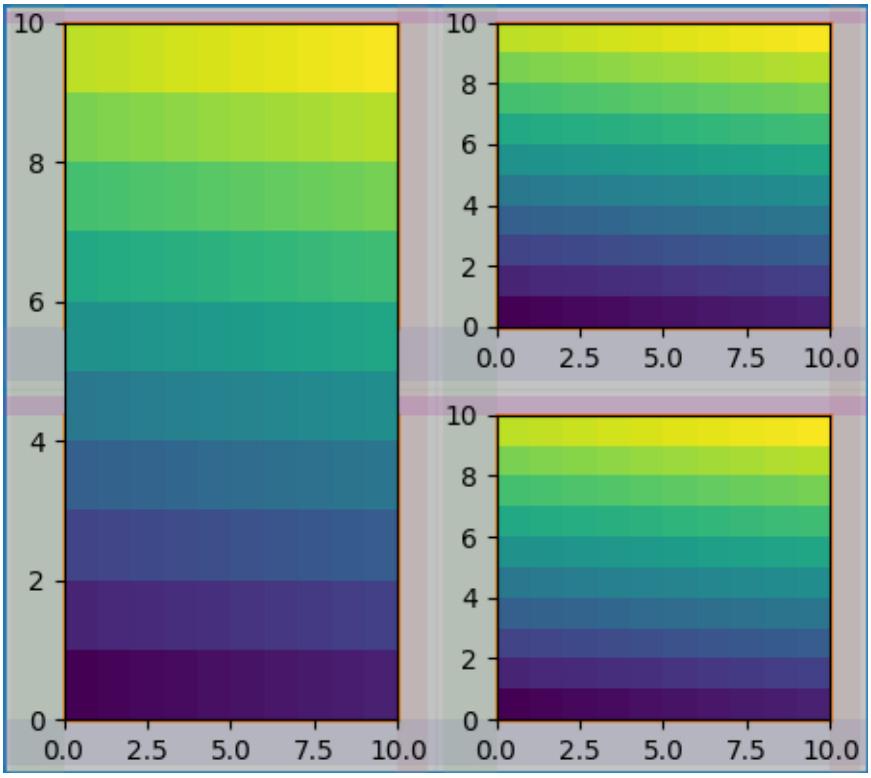



### Uneven sized Axes

There are two ways to make Axes have an uneven size in a Gridspec layout, either by specifying them to cross Gridspecs rows or columns, or by specifying width and height ratios.

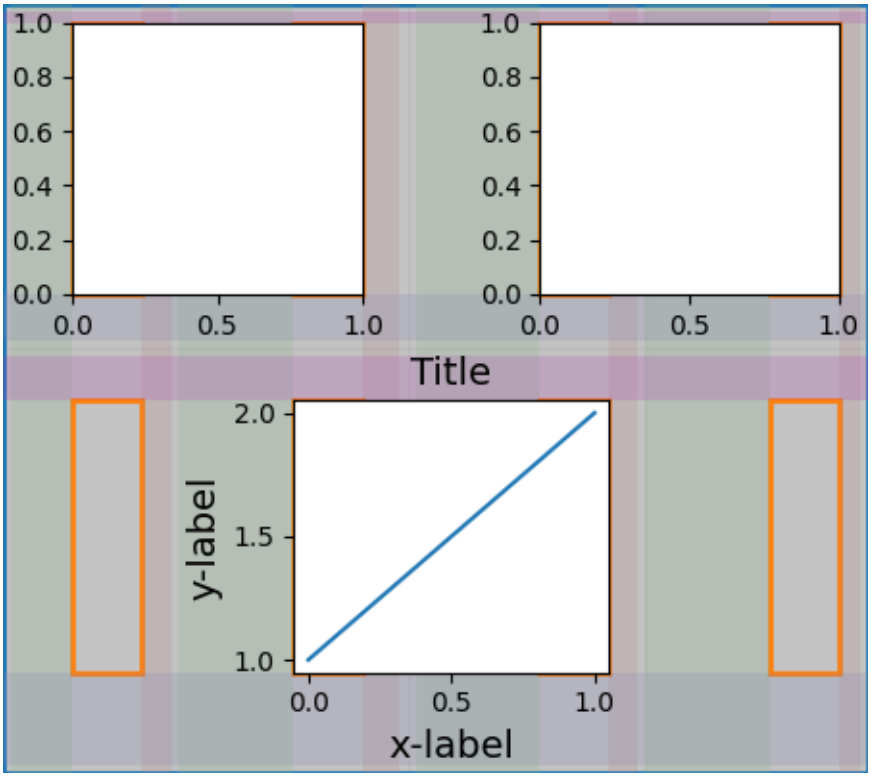
The first method is used here. Note that the middle `top` and `bottom` margins are not affected by the left-hand column. This is a conscious decision of the algorithm, and leads to the case where the two right-hand Axes have the same height, but it is not 1/2 the height of the left-hand Axes. This is consistent with how `gridspec` works without *constrained layout*.

```
fig = plt.figure(layout="constrained")
gs = gridspec.GridSpec(2, 2, figure=fig)
ax = fig.add_subplot(gs[:, 0])
im = ax.pcolormesh(arr, **pc_kwargs)
ax = fig.add_subplot(gs[0, 1])
im = ax.pcolormesh(arr, **pc_kwargs)
ax = fig.add_subplot(gs[1, 1])
im = ax.pcolormesh(arr, **pc_kwargs)
plot_children(fig)
```




One case that requires finessing is if margins do not have any artists constraining their width. In the case below, the right margin for column 0 and the left margin for column 3 have no margin artists to set their width, so we take the maximum width of the margin widths that do have artists. This makes all the Axes have the same size:

```
fig = plt.figure(layout="constrained")
gs = fig.add_gridspec(2, 4)
ax00 = fig.add_subplot(gs[0, 0:2])
ax01 = fig.add_subplot(gs[0, 2:])
ax10 = fig.add_subplot(gs[1, 1:3])
example_plot(ax10, fontsize=14)
plot_children(fig)
plt.show()
```




**Total running time of the script:** (0 minutes 14.161 seconds)

### 3.3.11 Tight layout guide

How to use tight-layout to fit plots within your figure cleanly.

*tight_layout* automatically adjusts subplot params so that the subplot(s) fits in to the figure area. This is an experimental feature and may not work for some cases. It only checks the extents of ticklabels, axis labels, and titles.

An alternative to *tight_layout* is *constrained_layout*.

#### Simple example

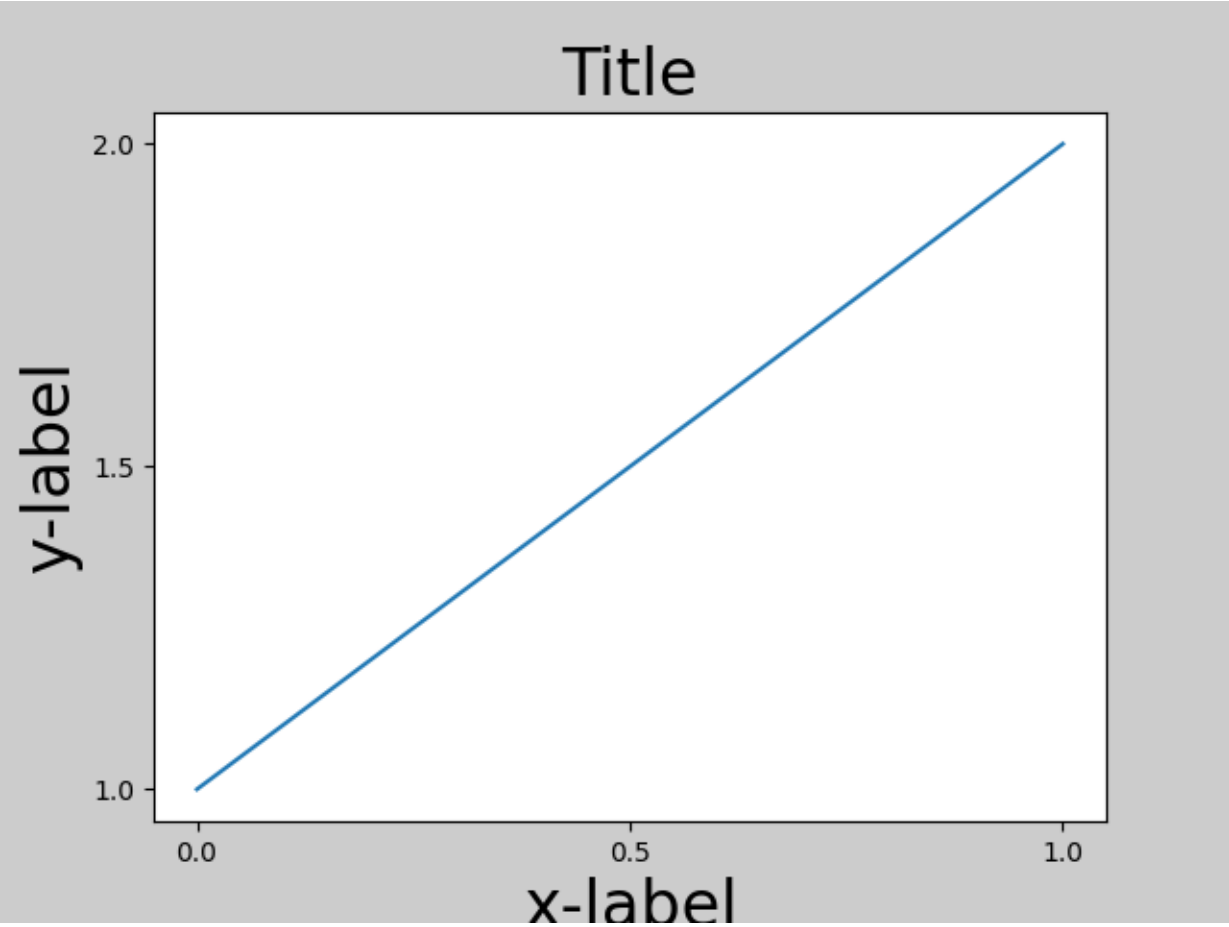
With the default Axes positioning, the axes title, axis labels, or tick labels can sometimes go outside the figure area, and thus get clipped.

```
import matplotlib.pyplot as plt
import numpy as np

plt.rcParams['savefig.facecolor'] = "0.8"


def example_plot(ax, fontsize=12):
    ax.plot([1, 2])
```

(continues on next page)

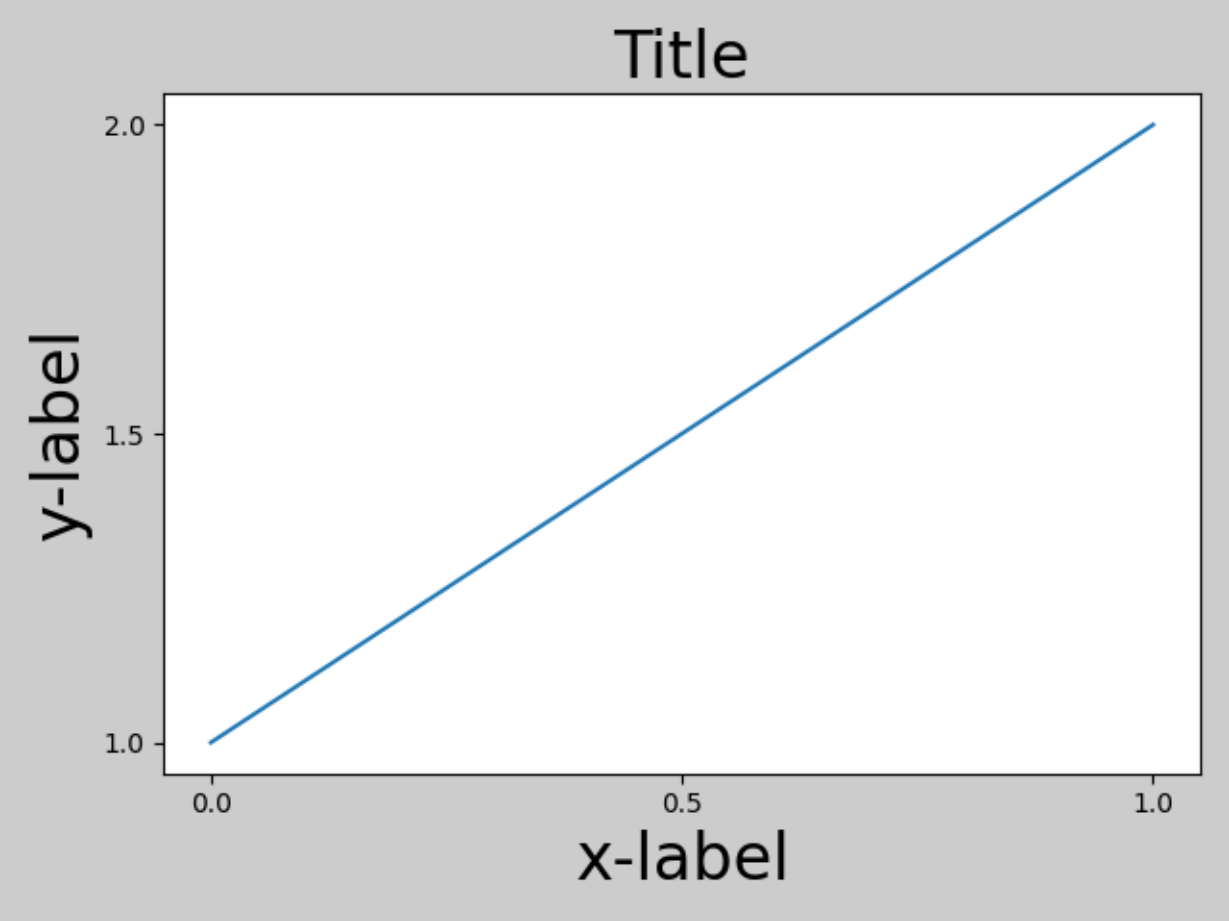
(continued from previous page)

```
    ax.locator_params(nbins=3)
    ax.set_xlabel('x-label', fontsize=fontsize)
    ax.set_ylabel('y-label', fontsize=fontsize)
    ax.set_title('Title', fontsize=fontsize)

plt.close('all')
fig, ax = plt.subplots()
example_plot(ax, fontsize=24)
```




To prevent this, the location of Axes needs to be adjusted. For subplots, this can be done manually by adjusting the subplot parameters using *Figure.subplots_adjust*. *Figure.tight_layout* does this automatically.

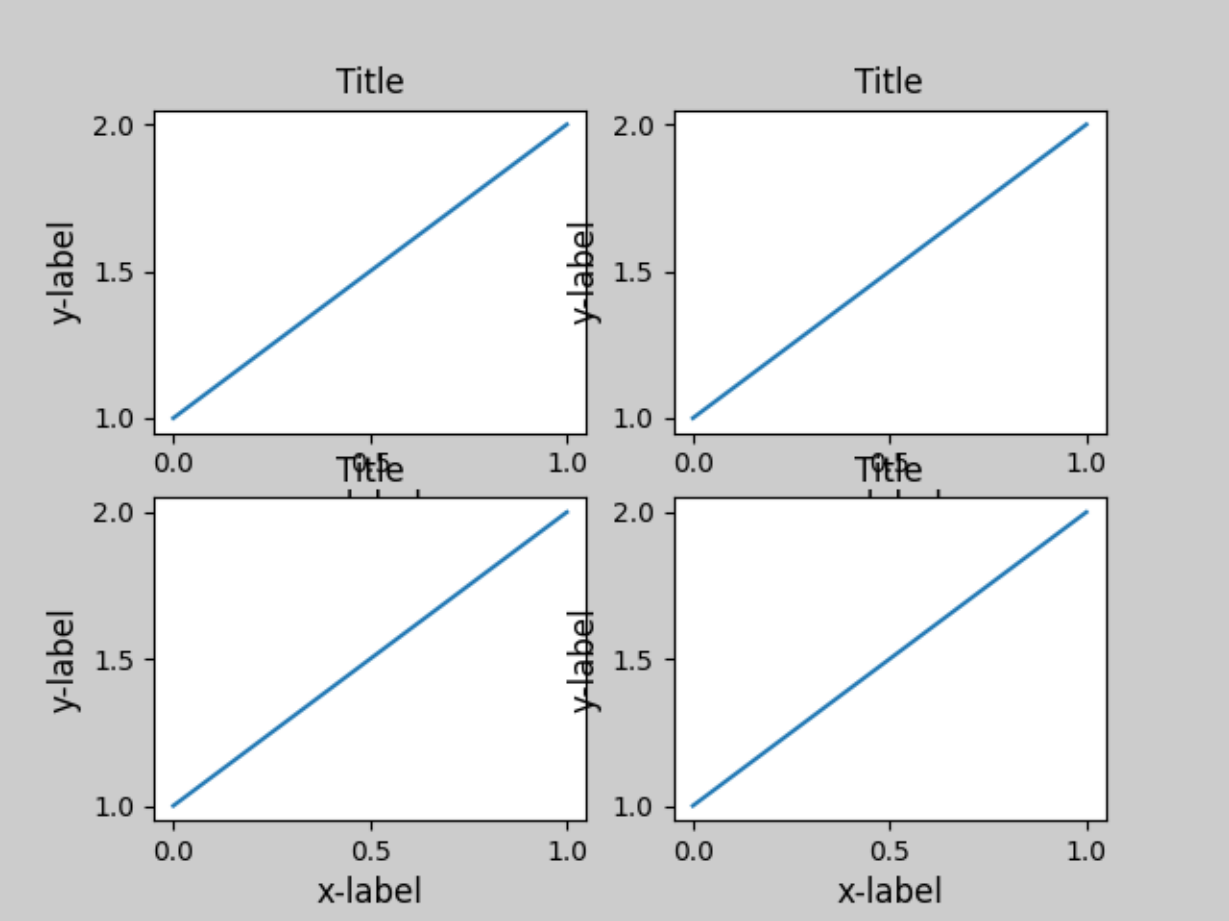
```
fig, ax = plt.subplots()
example_plot(ax, fontsize=24)
plt.tight_layout()
```




Note that *matplotlib.pyplot.tight_layout()* will only adjust the subplot params when it is called. In order to perform this adjustment each time the figure is redrawn, you can call `fig.set_tight_layout(True)`, or, equivalently, set *rcParams["figure.autolayout"]* (default: `False`) to `True`.

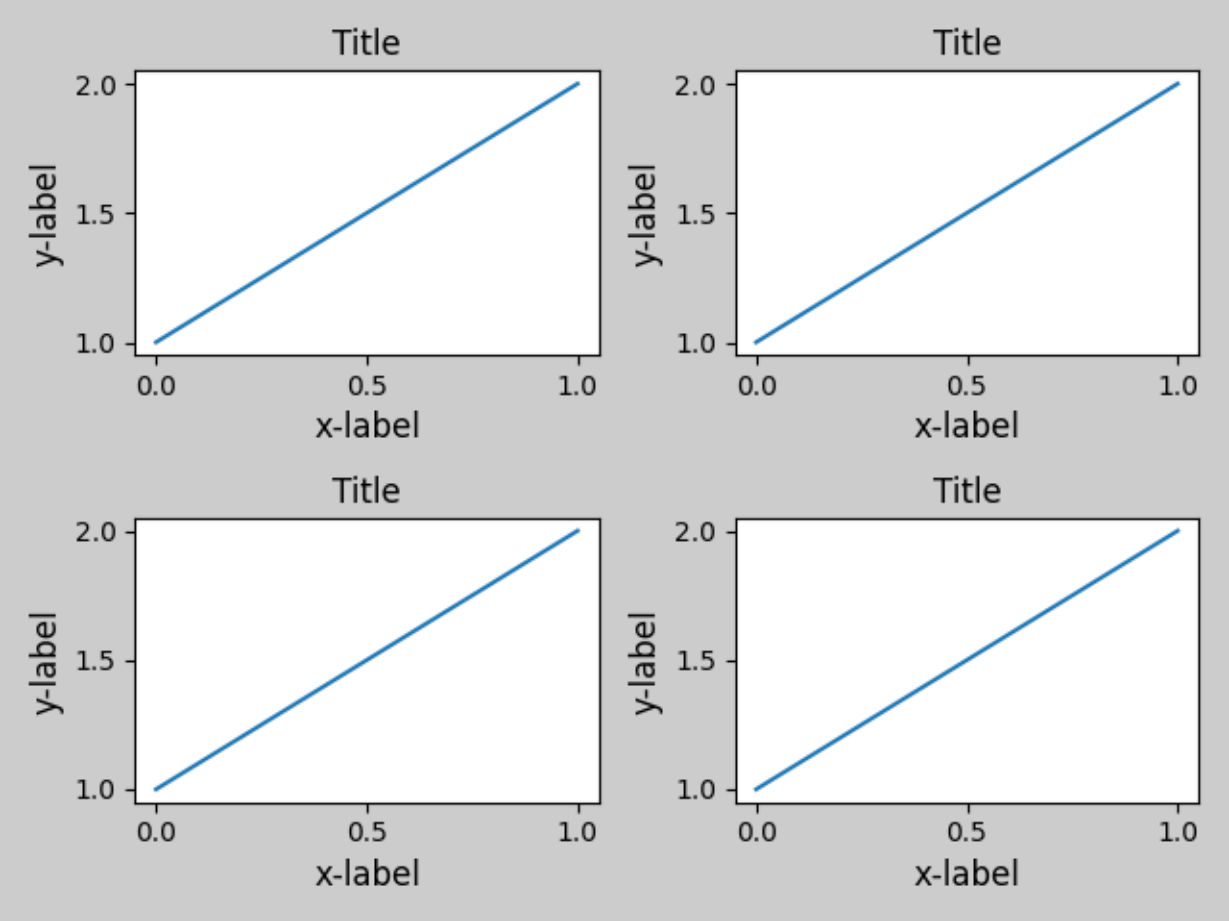
When you have multiple subplots, often you see labels of different Axes overlapping each other.

```
plt.close('all')

fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(nrows=2, ncols=2)
example_plot(ax1)
example_plot(ax2)
example_plot(ax3)
example_plot(ax4)
```



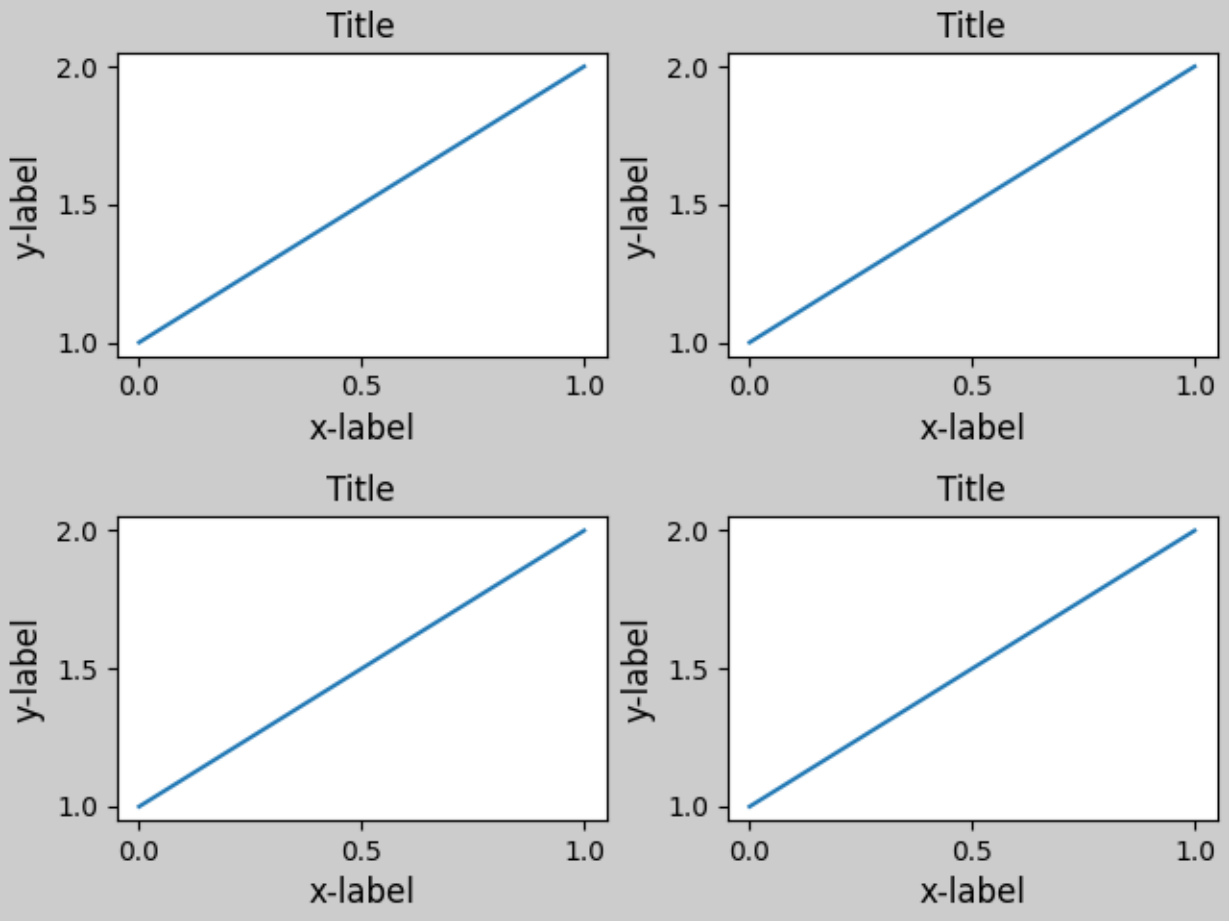

*tight_layout()* will also adjust spacing between subplots to minimize the overlaps.

```
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(nrows=2, ncols=2)
example_plot(ax1)
example_plot(ax2)
example_plot(ax3)
example_plot(ax4)
plt.tight_layout()
```



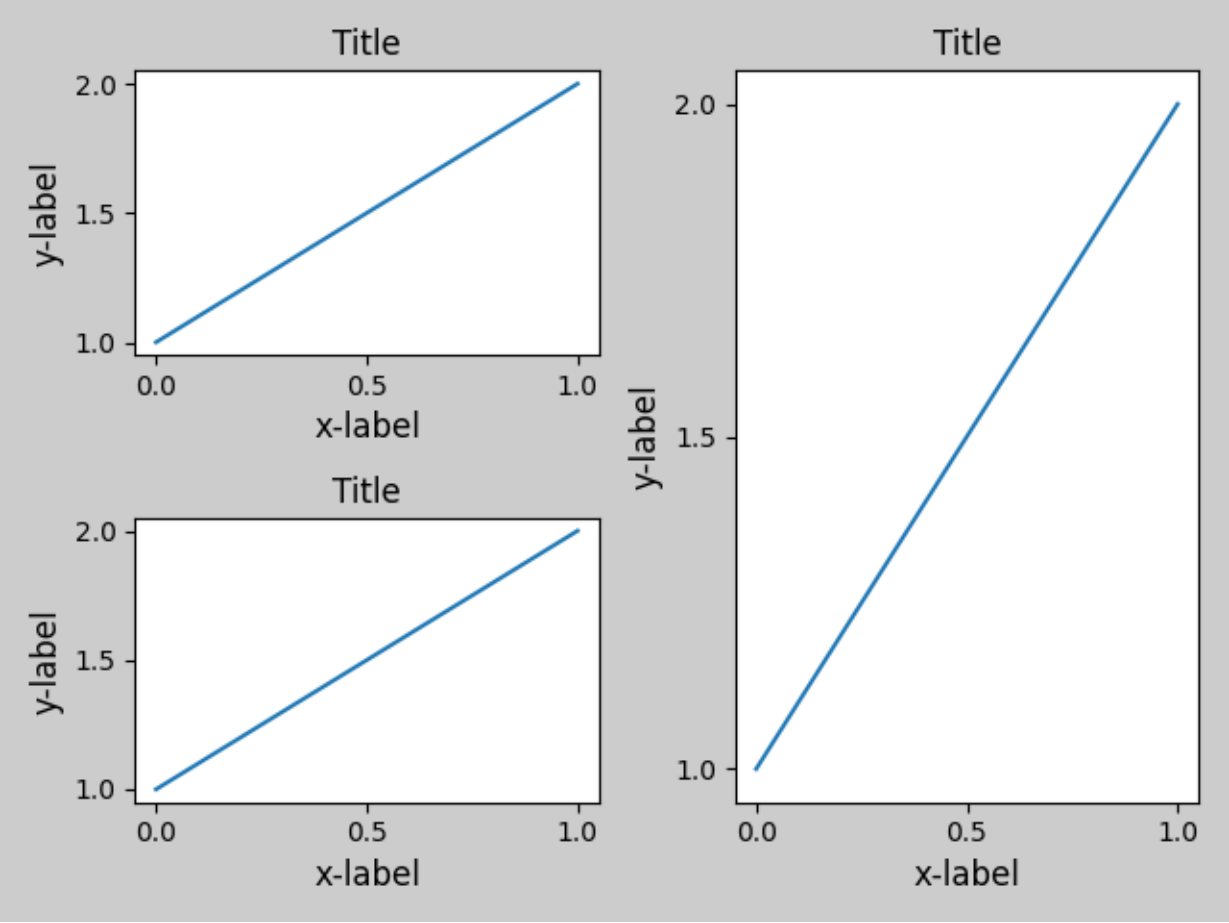

`tight_layout()` can take keyword arguments of *pad*, *w_pad* and *h_pad*. These control the extra padding around the figure border and between subplots. The pads are specified in fraction of fontsize.

```
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(nrows=2, ncols=2)
example_plot(ax1)
example_plot(ax2)
example_plot(ax3)
example_plot(ax4)
plt.tight_layout(pad=0.4, w_pad=0.5, h_pad=1.0)
```



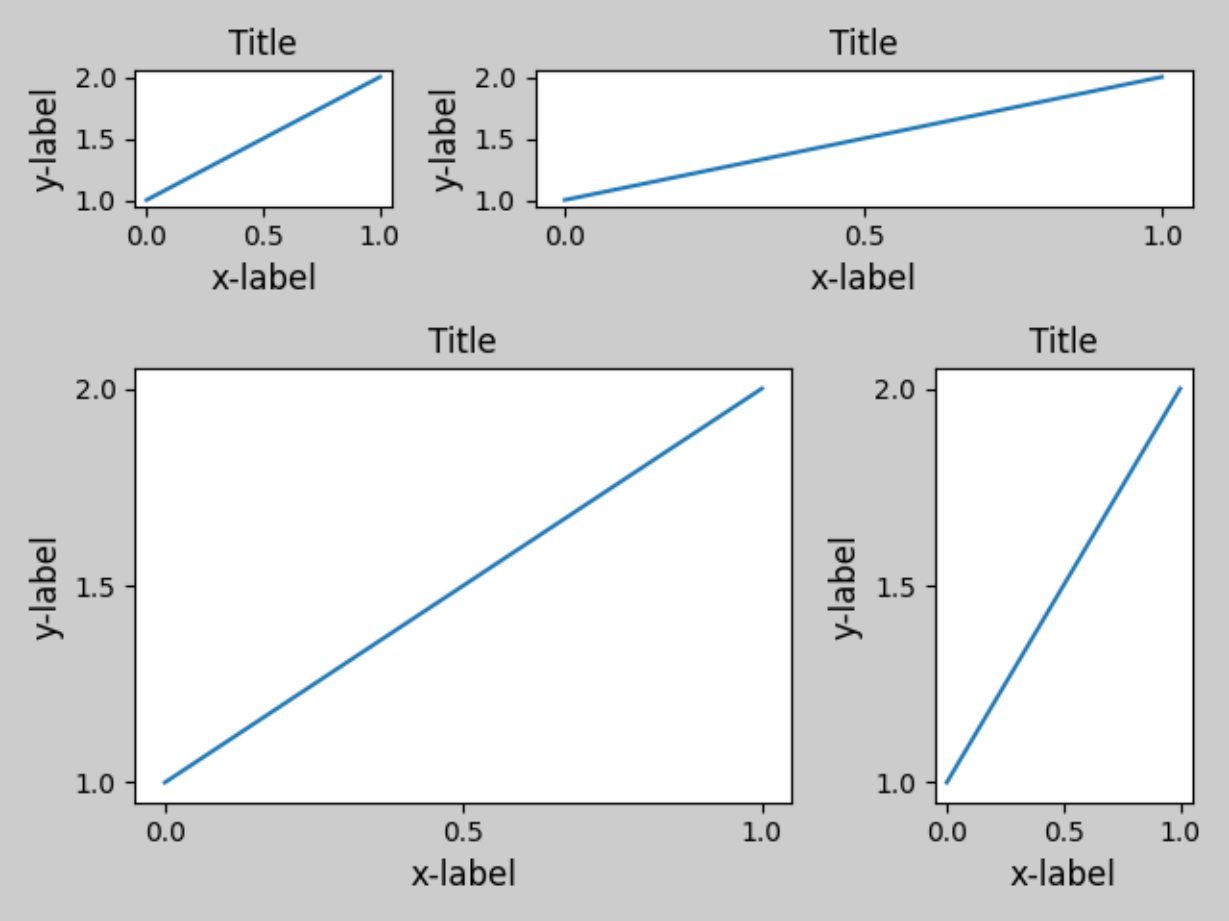

*tight_layout()* will work even if the sizes of subplots are different as far as their grid specification is compatible. In the example below, *ax1* and *ax2* are subplots of a 2x2 grid, while *ax3* is of a 1x2 grid.

```
plt.close('all')
fig = plt.figure()

ax1 = plt.subplot(221)
ax2 = plt.subplot(223)
ax3 = plt.subplot(122)

example_plot(ax1)
example_plot(ax2)
example_plot(ax3)

plt.tight_layout()
```



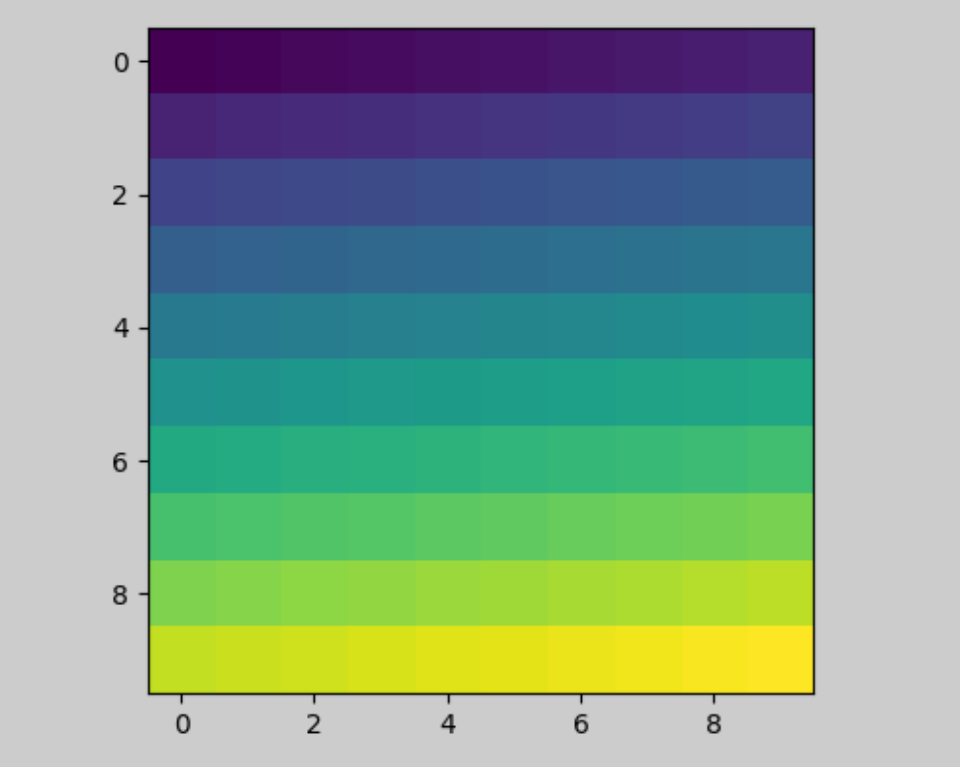

It works with subplots created with *subplot2grid()*. In general, subplots created from the gridspec (*Arranging multiple Axes in a Figure*) will work.

```
plt.close('all')
fig = plt.figure()

ax1 = plt.subplot2grid((3, 3), (0, 0))
ax2 = plt.subplot2grid((3, 3), (0, 1), colspan=2)
ax3 = plt.subplot2grid((3, 3), (1, 0), colspan=2, rowspan=2)
ax4 = plt.subplot2grid((3, 3), (1, 2), rowspan=2)

example_plot(ax1)
example_plot(ax2)
example_plot(ax3)
example_plot(ax4)

plt.tight_layout()
```




Although not thoroughly tested, it seems to work for subplots with aspect != "auto" (e.g., Axes with images).

```
arr = np.arange(100).reshape((10, 10))

plt.close('all')
fig = plt.figure(figsize=(5, 4))

ax = plt.subplot()
im = ax.imshow(arr, interpolation="none")

plt.tight_layout()
```




### Caveats

- *tight_layout* considers all artists on the Axes by default. To remove an artist from the layout calculation you can call *Artist.set_in_layout*.
- `tight_layout` assumes that the extra space needed for artists is independent of the original location of Axes. This is often true, but there are rare cases where it is not.
- `pad=0` can clip some texts by a few pixels. This may be a bug or a limitation of the current algorithm, and it is not clear why it happens. Meanwhile, use of pad larger than 0.3 is recommended.

### Use with GridSpec

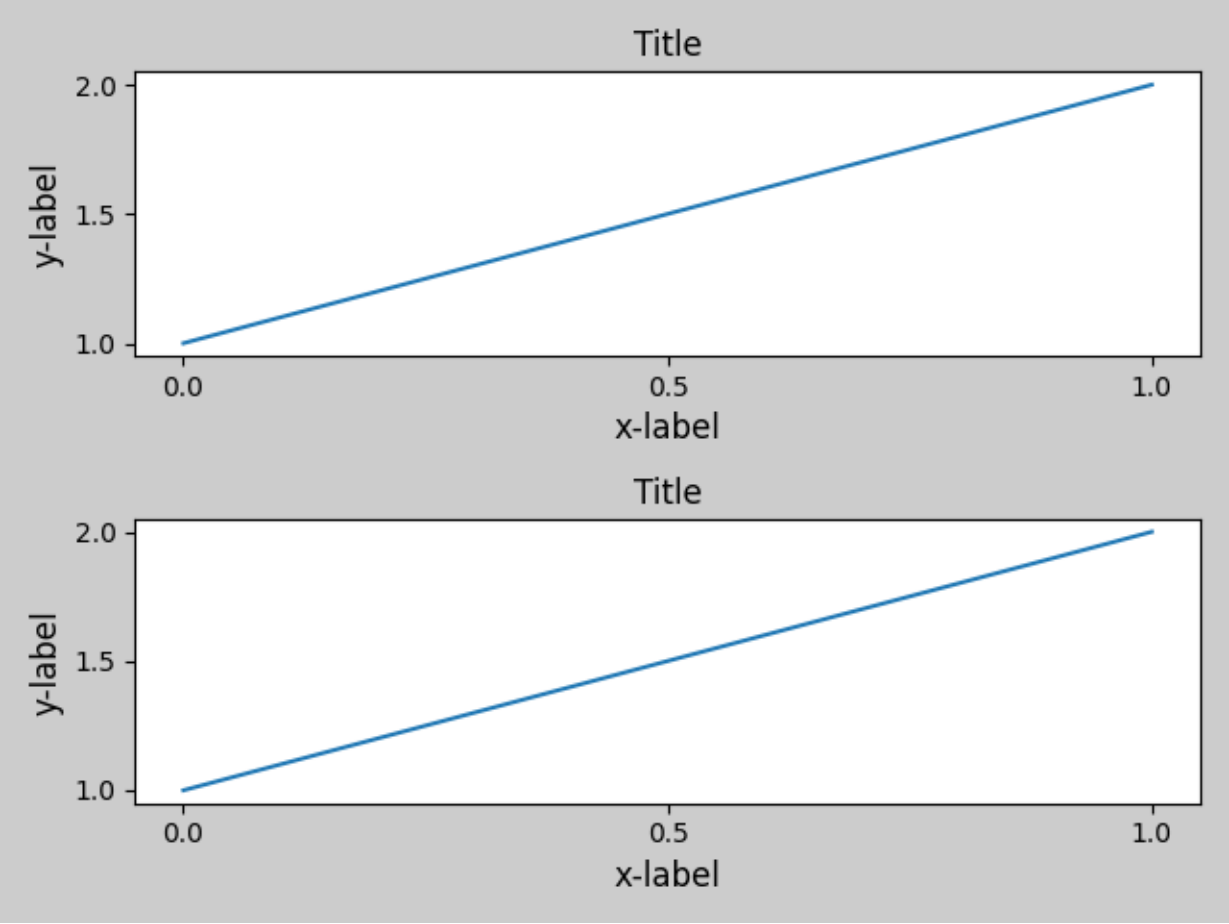
GridSpec has its own *GridSpec.tight_layout* method (the pyplot api *pyplot.tight_layout* also works).

```
import matplotlib.gridspec as gridspec

plt.close('all')
fig = plt.figure()

gs1 = gridspec.GridSpec(2, 1)
ax1 = fig.add_subplot(gs1[0])
ax2 = fig.add_subplot(gs1[1])
```

(continues on next page)

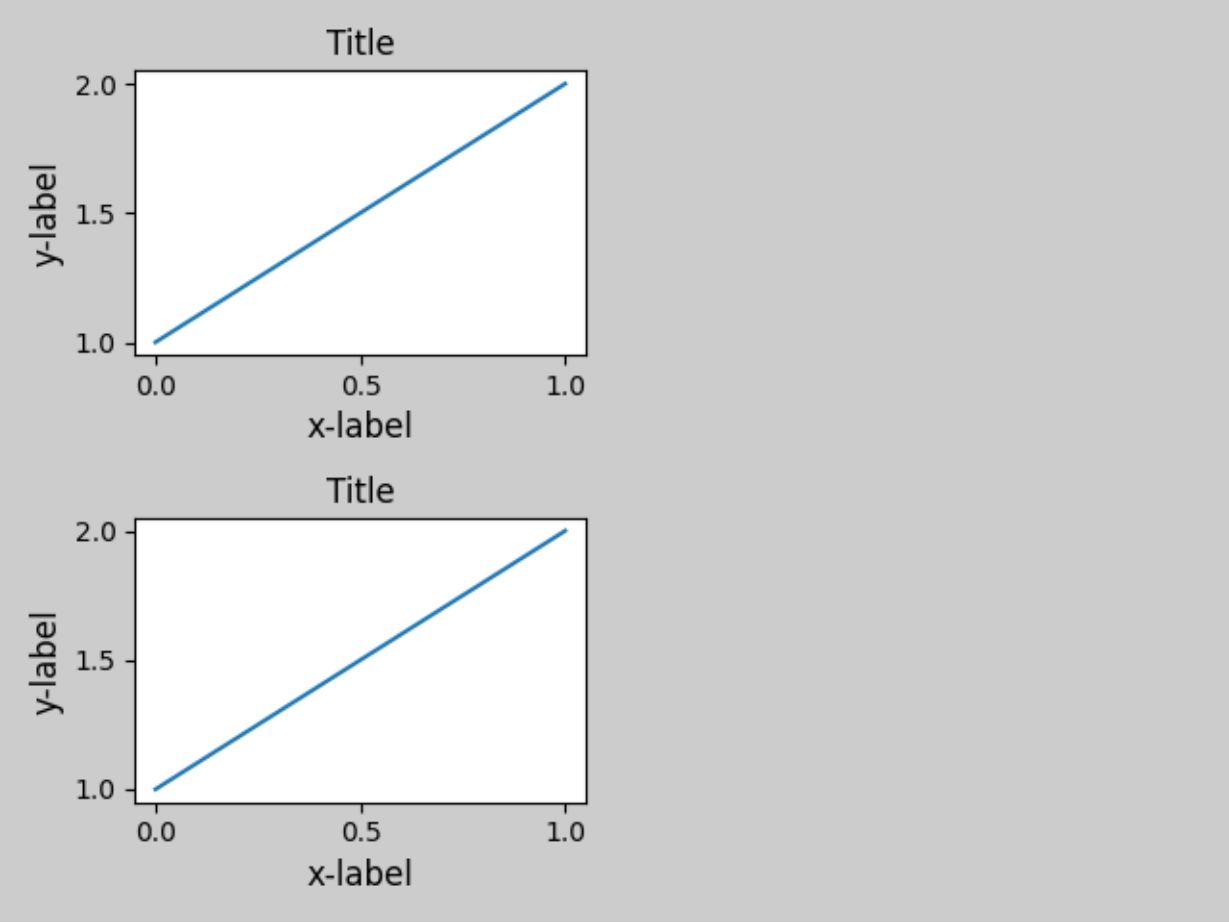
(continued from previous page)

```
example_plot(ax1)
example_plot(ax2)

gs1.tight_layout(fig)
```




You may provide an optional *rect* parameter, which specifies the bounding box that the subplots will be fit inside. The coordinates are in normalized figure coordinates and default to (0, 0, 1, 1) (the whole figure).

```
fig = plt.figure()

gs1 = gridspec.GridSpec(2, 1)
ax1 = fig.add_subplot(gs1[0])
ax2 = fig.add_subplot(gs1[1])

example_plot(ax1)
example_plot(ax2)

gs1.tight_layout(fig, rect=[0, 0, 0.5, 1.0])
```

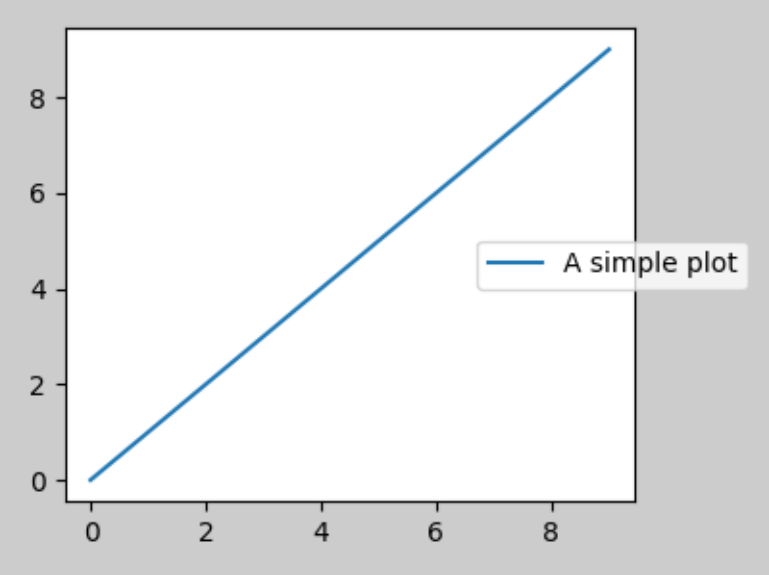



However, we do not recommend that this be used to manually construct more complicated layouts, like having one GridSpec in the left and one in the right side of the figure. For these use cases, one should instead take advantage of *Nested Gridspecs*, or the *Figure subfigures*.

### Legends and annotations

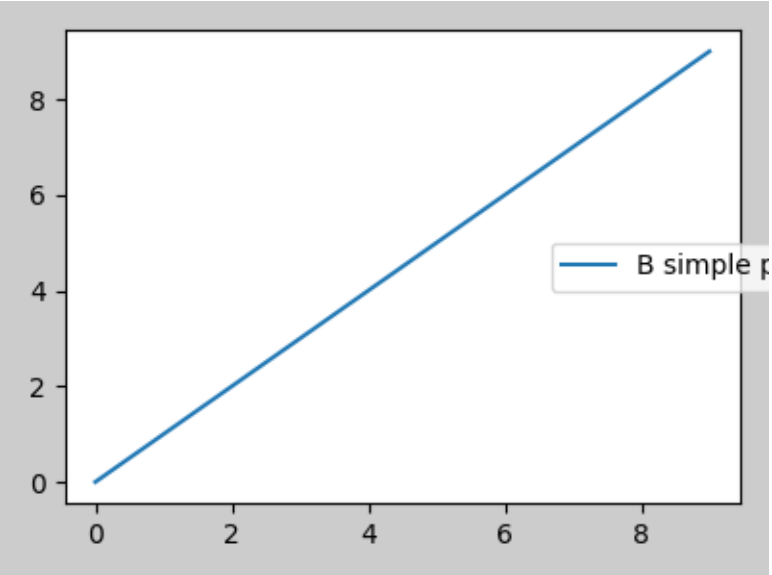
Pre Matplotlib 2.2, legends and annotations were excluded from the bounding box calculations that decide the layout. Subsequently, these artists were added to the calculation, but sometimes it is undesirable to include them. For instance in this case it might be good to have the Axes shrink a bit to make room for the legend:

```
fig, ax = plt.subplots(figsize=(4, 3))
lines = ax.plot(range(10), label='A simple plot')
ax.legend(bbox_to_anchor=(0.7, 0.5), loc='center left',)
fig.tight_layout()
plt.show()
```




However, sometimes this is not desired (quite often when using `fig.savefig('outname.png', bbox_inches='tight')`). In order to remove the legend from the bounding box calculation, we simply set its bounding `leg.set_in_layout(False)` and the legend will be ignored.

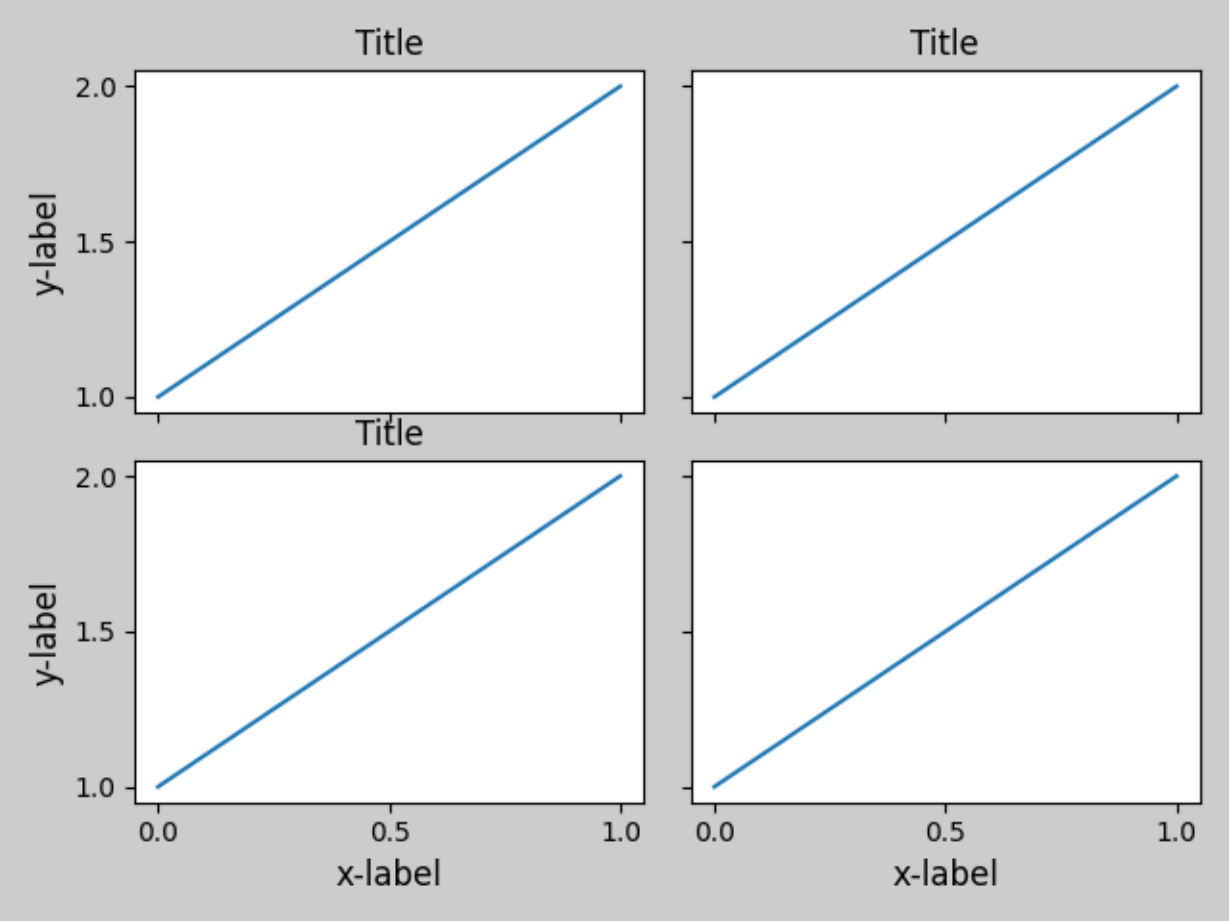
```
fig, ax = plt.subplots(figsize=(4, 3))
lines = ax.plot(range(10), label='B simple plot')
leg = ax.legend(bbox_to_anchor=(0.7, 0.5), loc='center left',)
leg.set_in_layout(False)
fig.tight_layout()
plt.show()
```



### Use with AxesGrid1

Limited support for *mpl_toolkits.axes_grid1* is provided.

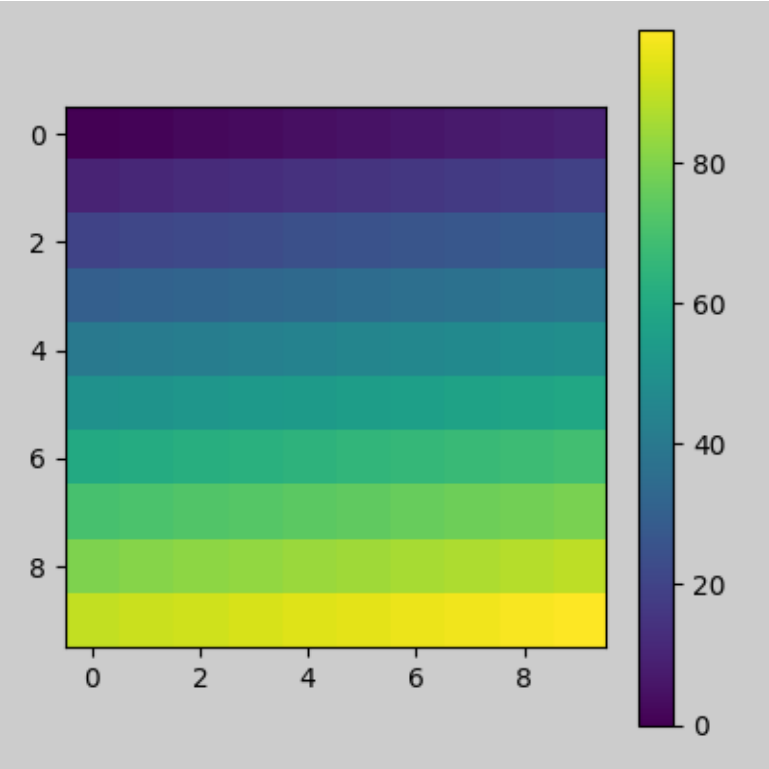
```
from mpl_toolkits.axes_grid1 import Grid

plt.close('all')
fig = plt.figure()
grid = Grid(fig, rect=111, nrows_ncols=(2, 2),
            axes_pad=0.25, label_mode='L',
            )

for ax in grid:
    example_plot(ax)
ax.title.set_visible(False)

plt.tight_layout()
```



### Colorbar

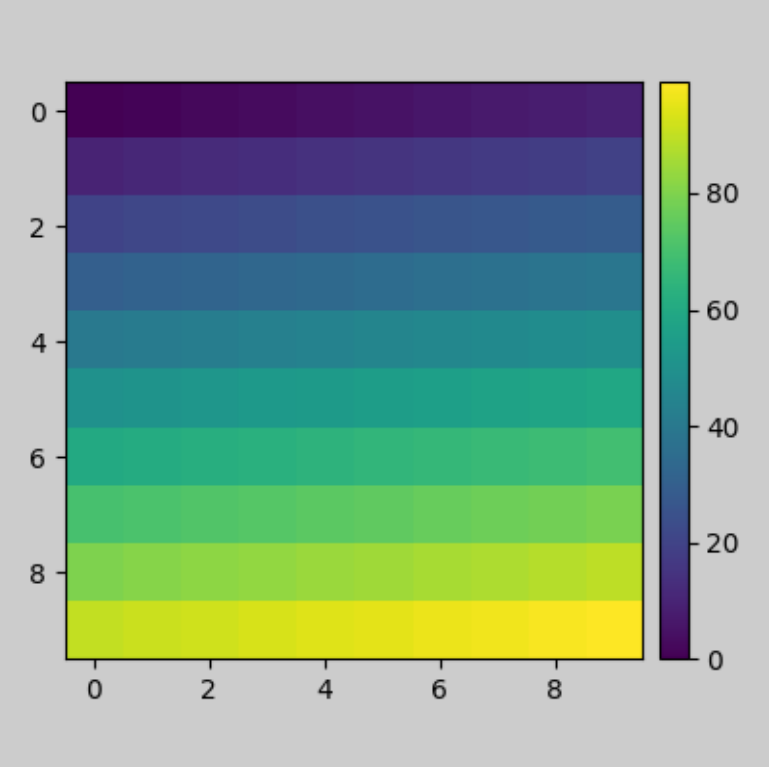
If you create a colorbar with *Figure.colorbar*, the created colorbar is drawn in a Subplot as long as the parent Axes is also a Subplot, so *Figure.tight_layout* will work.

```
plt.close('all')
arr = np.arange(100).reshape((10, 10))
fig = plt.figure(figsize=(4, 4))
im = plt.imshow(arr, interpolation="none")

plt.colorbar(im)

plt.tight_layout()
```




Another option is to use the AxesGrid1 toolkit to explicitly create an Axes for the colorbar.

```
from mpl_toolkits.axes_grid1 import make_axes_locatable

plt.close('all')
arr = np.arange(100).reshape((10, 10))
fig = plt.figure(figsize=(4, 4))
im = plt.imshow(arr, interpolation="none")

divider = make_axes_locatable(plt.gca())
cax = divider.append_axes("right", "5%", pad="3%")
plt.colorbar(im, cax=cax)

plt.tight_layout()
```




**Total running time of the script:** (0 minutes 4.780 seconds)

## 3.4 Artists

Almost all objects you interact with on a Matplotlib plot are called "Artist" (and are subclasses of the `Artist` class). *Figure* and *Axes* are Artists, and generally contain `Axis` Artists and Artists that contain data or annotation information.

### 3.4.1 Introduction to Artists

Almost all objects you interact with on a Matplotlib plot are called "Artist" (and are subclasses of the `Artist` class). *Figure* and *Axes* are Artists, and generally contain `Axis` Artists and Artists that contain data or annotation information.

#### Creating Artists

Usually we do not instantiate Artists directly, but rather use a plotting method on `Axes`. Some examples of plotting methods and the Artist object they create is given below:

| Axes helper method | Artist |
|---|---|
| *annotate* - text annotations | *Annotation* |
| *bar* - bar charts | *Rectangle* |
| *errorbar* - error bar plots | *Line2D* and *Rectangle* |
| *fill* - shared area | *Polygon* |
| *hist* - histograms | *Rectangle* |
| *imshow* - image data | *AxesImage* |
| *legend* - Axes legend | *Legend* |
| *plot* - xy plots | *Line2D* |
| *scatter* - scatter charts | *PolyCollection* |
| *text* - text | *Text* |

As an example, we can save the Line2D Artist returned from *axes.Axes.plot*:

```
In [209]: import matplotlib.pyplot as plt
In [210]: import matplotlib.artist as martist
In [211]: import numpy as np

In [212]: fig, ax = plt.subplots()
In [213]: x, y = np.random.rand(2, 100)
In [214]: lines = ax.plot(x, y, '-', label='example')
In [215]: print(lines)
[<matplotlib.lines.Line2D at 0xd378b0c>]
```

Note that `plot` returns a _list_ of lines because you can pass in multiple x, y pairs to plot. The line has been added to the Axes, and we can retrieve the Artist via *get_lines()*:

```
In [216]: print(ax.get_lines())
<a list of 1 Line2D objects>
In [217]: print(ax.get_lines()[0])
Line2D(example)
```

### Changing Artist properties

Getting the `lines` object gives us access to all the properties of the Line2D object. So if we want to change the *linewidth* after the fact, we can do so using *Artist.set*.

```
fig, ax = plt.subplots(figsize=(4, 2.5))
x = np.arange(0, 13, 0.2)
y = np.sin(x)
lines = ax.plot(x, y, '-', label='example', linewidth=0.2, color='blue')
lines[0].set(color='green', linewidth=2)
```

We can interrogate the full list of settable properties with *matplotlib.artist.getp*:

```
In [218]: martist.getp(lines[0])
agg_filter = None
alpha = None
animated = False
antialiased or aa = True
bbox = Bbox(x0=0.004013842290585101, y0=0.013914221641967...
children = []
clip_box = TransformedBbox(     Bbox(x0=0.0, y0=0.0, x1=1.0, ...
clip_on = True
clip_path = None
color or c = blue
dash_capstyle = butt
dash_joinstyle = round
data = (array([0.91377845, 0.58456834, 0.36492019, 0.0379...
drawstyle or ds = default
figure = Figure(550x450)
fillstyle = full
gapcolor = None
gid = None
in_layout = True
label = example
linestyle or ls = -
linewidth or lw = 2.0
marker = None
markeredgecolor or mec = blue
markeredgewidth or mew = 1.0
markerfacecolor or mfc = blue
markerfacecoloralt or mfcalt = none
markersize or ms = 6.0
markevery = None
mouseover = False
path = Path(array([[0.91377845, 0.51224793],        [0.58...
path_effects = []
picker = None
pickradius = 5
rasterized = False
sketch_params = None
snap = None
solid_capstyle = projecting
solid_joinstyle = round
tightbbox = Bbox(x0=70.4609002763619, y0=54.321277798941786, x...
transform = CompositeGenericTransform(     TransformWrapper(  ...
transformed_clip_path_and_affine = (None, None)
url = None
visible = True
window_extent = Bbox(x0=70.4609002763619, y0=54.321277798941786, x...
xdata = [0.91377845 0.58456834 0.36492019 0.03796664 0.884...
xydata = [[0.91377845 0.51224793]  [0.58456834 0.9820474 ] ...
ydata = [0.51224793 0.9820474  0.24469912 0.61647032 0.483...
zorder = 2
```

Note most Artists also have a distinct list of setters; e.g. *Line2D.set_color* or *Line2D.set_linewidth*.

### Changing Artist data

In addition to styling properties like *color* and *linewidth*, the Line2D object has a *data* property. You can set the data after the line has been created using *Line2D.set_data*. This is often used for Animations, where the same line is shown evolving over time (see *Animations using Matplotlib*)

```
fig, ax = plt.subplots(figsize=(4, 2.5))
x = np.arange(0, 13, 0.2)
y = np.sin(x)
lines = ax.plot(x, y, '-', label='example')
lines[0].set_data([x, np.cos(x)])
```

### Manually adding Artists

Not all Artists have helper methods, or you may want to use a low-level method for some reason. For example the *patches.Circle* Artist does not have a helper, but we can still create and add to an Axes using the *axes.Axes.add_artist* method:

```
import matplotlib.patches as mpatches

fig, ax = plt.subplots(figsize=(4, 2.5))
circle = mpatches.Circle((0.5, 0.5), 0.25, ec="none")
ax.add_artist(circle)
clipped_circle = mpatches.Circle((1, 0.5), 0.125, ec="none", facecolor='C1')
ax.add_artist(clipped_circle)
ax.set_aspect(1)
```

The Circle takes the center and radius of the Circle as arguments to its constructor; optional arguments are passed as keyword arguments.

Note that when we add an Artist manually like this, it doesn't necessarily adjust the axis limits like most of the helper methods do, so the Artists can be clipped, as is the case above for the `clipped_circle` patch.

See *Reference for Matplotlib artists* for other patches.

### Removing Artists

Sometimes we want to remove an Artist from a figure without re-specifying the whole figure from scratch. Most Artists have a usable *remove* method that will remove the Artist from its Axes list. For instance `lines[0].remove()` would remove the *Line2D* artist created in the example above.

### 3.4.2 Styling with cycler

Demo of custom property-cycle settings to control colors and other style properties for multi-line plots.

**Note:** More complete documentation of the `cycler` API can be found here.

This example demonstrates two different APIs:

1. Setting the rc parameter specifying the default property cycle. This affects all subsequent Axes (but not Axes already created).
2. Setting the property cycle for a single pair of Axes.

```
from cycler import cycler

import matplotlib.pyplot as plt
import numpy as np
```

First we'll generate some sample data, in this case, four offset sine curves.

```
x = np.linspace(0, 2 * np.pi, 50)
offsets = np.linspace(0, 2 * np.pi, 4, endpoint=False)
yy = np.transpose([np.sin(x + phi) for phi in offsets])
```

Now `yy` has shape

```
print(yy.shape)
```

```
(50, 4)
```

So `yy[:, i]` will give you the `i`-th offset sine curve. Let's set the default `prop_cycle` using *`matplotlib.pyplot.rc()`*. We'll combine a color cycler and a linestyle cycler by adding (+) two `cycler`'s together. See the bottom of this tutorial for more information about combining different cyclers.

```
default_cycler = (cycler(color=['r', 'g', 'b', 'y']) +
                  cycler(linestyle=['-', '--', ':', '-.']))

plt.rc('lines', linewidth=4)
plt.rc('axes', prop_cycle=default_cycler)
```

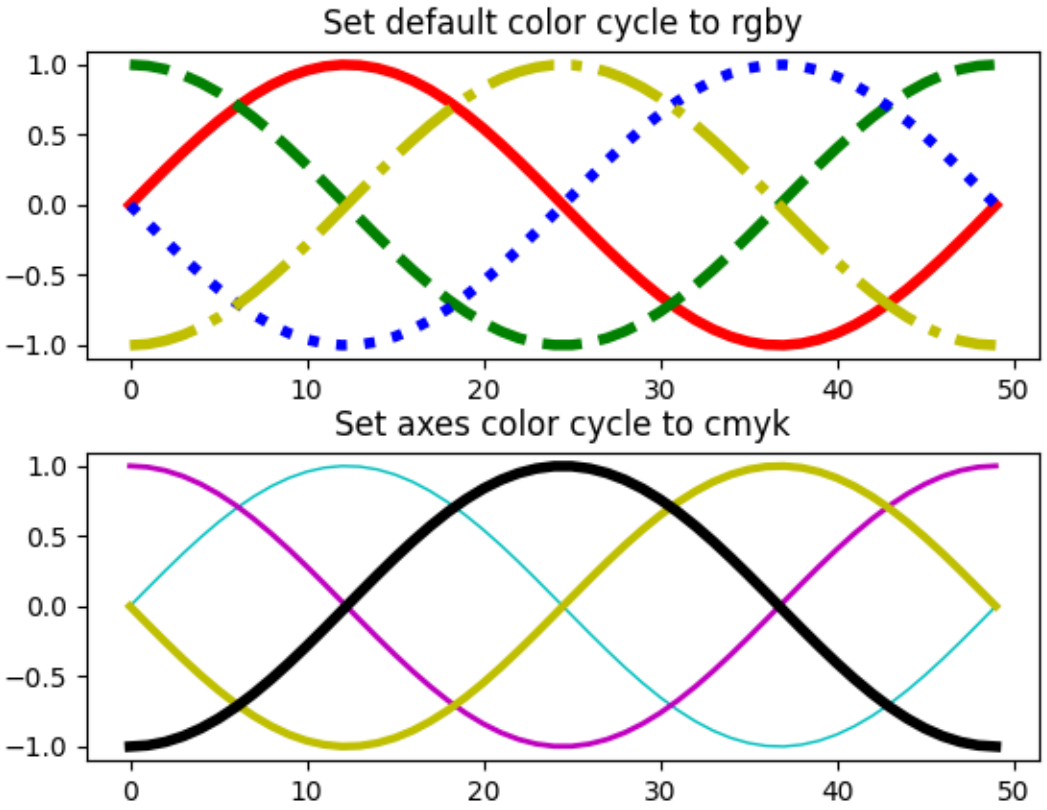
Now we'll generate a figure with two Axes, one on top of the other. On the first axis, we'll plot with the default cycler. On the second axis, we'll set the `prop_cycle` using *`matplotlib.axes.Axes.set_prop_cycle()`*, which will only set the `prop_cycle` for this *`matplotlib.axes.Axes`* instance. We'll use a second `cycler` that combines a color cycler and a linewidth cycler.

```
custom_cycler = (cycler(color=['c', 'm', 'y', 'k']) +
                 cycler(lw=[1, 2, 3, 4]))

fig, (ax0, ax1) = plt.subplots(nrows=2)
ax0.plot(yy)
```

(continues on next page)

(continued from previous page)

```
ax0.set_title('Set default color cycle to rgby')
ax1.set_prop_cycle(custom_cycler)
ax1.plot(yy)
ax1.set_title('Set axes color cycle to cmyk')

# Add a bit more space between the two plots.
fig.subplots_adjust(hspace=0.3)
plt.show()
```




### Setting `prop_cycle` in the `matplotlibrc` file or style files

Remember, a custom cycler can be set in your `matplotlibrc` file or a style file (`style.mplstyle`) under `axes.prop_cycle`:

```
axes.prop_cycle : cycler(color='bgrcmyk')
```

### Cycling through multiple properties

You can add cyclers:

```
from cycler import cycler
cc = (cycler(color=list('rgb')) +
      cycler(linestyle=['-', '--', '-.']))
for d in cc:
    print(d)
```

Results in:

```
{'color': 'r', 'linestyle': '-'}
{'color': 'g', 'linestyle': '--'}
{'color': 'b', 'linestyle': '-.'}
```

You can multiply cyclers:

```
from cycler import cycler
cc = (cycler(color=list('rgb')) *
      cycler(linestyle=['-', '--', '-.']))
for d in cc:
    print(d)
```

Results in:

```
{'color': 'r', 'linestyle': '-'}
{'color': 'r', 'linestyle': '--'}
{'color': 'r', 'linestyle': '-.'}
{'color': 'g', 'linestyle': '-'}
{'color': 'g', 'linestyle': '--'}
{'color': 'g', 'linestyle': '-.'}
{'color': 'b', 'linestyle': '-'}
{'color': 'b', 'linestyle': '--'}
{'color': 'b', 'linestyle': '-.'}
```

### 3.4.3 Performance

Whether exploring data in interactive mode or programmatically saving lots of plots, rendering performance can be a challenging bottleneck in your pipeline. Matplotlib provides multiple ways to greatly reduce rendering time at the cost of a slight change (to a settable tolerance) in your plot's appearance. The methods available to reduce rendering time depend on the type of plot that is being created.

#### Line segment simplification

For plots that have line segments (e.g. typical line plots, outlines of polygons, etc.), rendering performance can be controlled by *rcParams["path.simplify"]* (default: `True`) and *rcParams["path.simplify_threshold"]* (default: `0.111111111111`), which can be defined e.g. in the `matplotlibrc` file (see *Customizing Matplotlib with style sheets and rcParams* for more information about the `matplotlibrc` file). *rcParams["path.simplify"]* (default: `True`) is a Boolean indicating whether or not line segments are simplified at all. *rcParams["path.simplify_threshold"]* (default: `0.111111111111`) controls how much line segments are simplified; higher thresholds result in quicker rendering.

The following script will first display the data without any simplification, and then display the same data with simplification. Try interacting with both of them:

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl

# Setup, and create the data to plot
y = np.random.rand(100000)
y[50000:] *= 2
y[np.geomspace(10, 50000, 400).astype(int)] = -1
mpl.rcParams['path.simplify'] = True

mpl.rcParams['path.simplify_threshold'] = 0.0
plt.plot(y)
plt.show()

mpl.rcParams['path.simplify_threshold'] = 1.0
plt.plot(y)
plt.show()
```

Matplotlib currently defaults to a conservative simplification threshold of `1/9`. To change default settings to use a different value, change the `matplotlibrc` file. Alternatively, users can create a new style for interactive plotting (with maximal simplification) and another style for publication quality plotting (with minimal simplification) and activate them as necessary. See *Customizing Matplotlib with style sheets and rcParams* for instructions on how to perform these actions.

The simplification works by iteratively merging line segments into a single vector until the next line segment's perpendicular distance to the vector (measured in display-coordinate space) is greater than the `path.simplify_threshold` parameter.

**Note:** Changes related to how line segments are simplified were made in version 2.1. Rendering time will

still be improved by these parameters prior to 2.1, but rendering time for some kinds of data will be vastly improved in versions 2.1 and greater.

### Marker subsampling

Markers can also be simplified, albeit less robustly than line segments. Marker subsampling is only available to *Line2D* objects (through the `markevery` property). Wherever *Line2D* construction parameters are passed through, such as *pyplot.plot* and *Axes.plot*, the `markevery` parameter can be used:

```
plt.plot(x, y, markevery=10)
```

The `markevery` argument allows for naive subsampling, or an attempt at evenly spaced (along the *x* axis) sampling. See the *Markevery Demo* for more information.

### Splitting lines into smaller chunks

If you are using the Agg backend (see *What is a backend?*), then you can make use of *rcParams["agg.path.chunksize"]* (default: `0`) This allows users to specify a chunk size, and any lines with greater than that many vertices will be split into multiple lines, each of which has no more than `agg.path.chunksize` many vertices. (Unless `agg.path.chunksize` is zero, in which case there is no chunking.) For some kind of data, chunking the line up into reasonable sizes can greatly decrease rendering time.

The following script will first display the data without any chunk size restriction, and then display the same data with a chunk size of 10,000. The difference can best be seen when the figures are large, try maximizing the GUI and then interacting with them:

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
mpl.rcParams['path.simplify_threshold'] = 1.0

# Setup, and create the data to plot
y = np.random.rand(100000)
y[50000:] *= 2
y[np.geomspace(10, 50000, 400).astype(int)] = -1
mpl.rcParams['path.simplify'] = True

mpl.rcParams['agg.path.chunksize'] = 0
plt.plot(y)
plt.show()

mpl.rcParams['agg.path.chunksize'] = 10000
plt.plot(y)
plt.show()
```

### Legends

The default legend behavior for axes attempts to find the location that covers the fewest data points (`loc='best'`). This can be a very expensive computation if there are lots of data points. In this case, you may want to provide a specific location.

### Using the *fast* style

The *fast* style can be used to automatically set simplification and chunking parameters to reasonable settings to speed up plotting large amounts of data. The following code runs it:

```
import matplotlib.style as mplstyle
mplstyle.use('fast')
```

It is very lightweight, so it works well with other styles. Be sure the fast style is applied last so that other styles do not overwrite the settings:

```
mplstyle.use(['dark_background', 'ggplot', 'fast'])
```

## 3.4.4 Path Tutorial

Defining paths in your Matplotlib visualization.

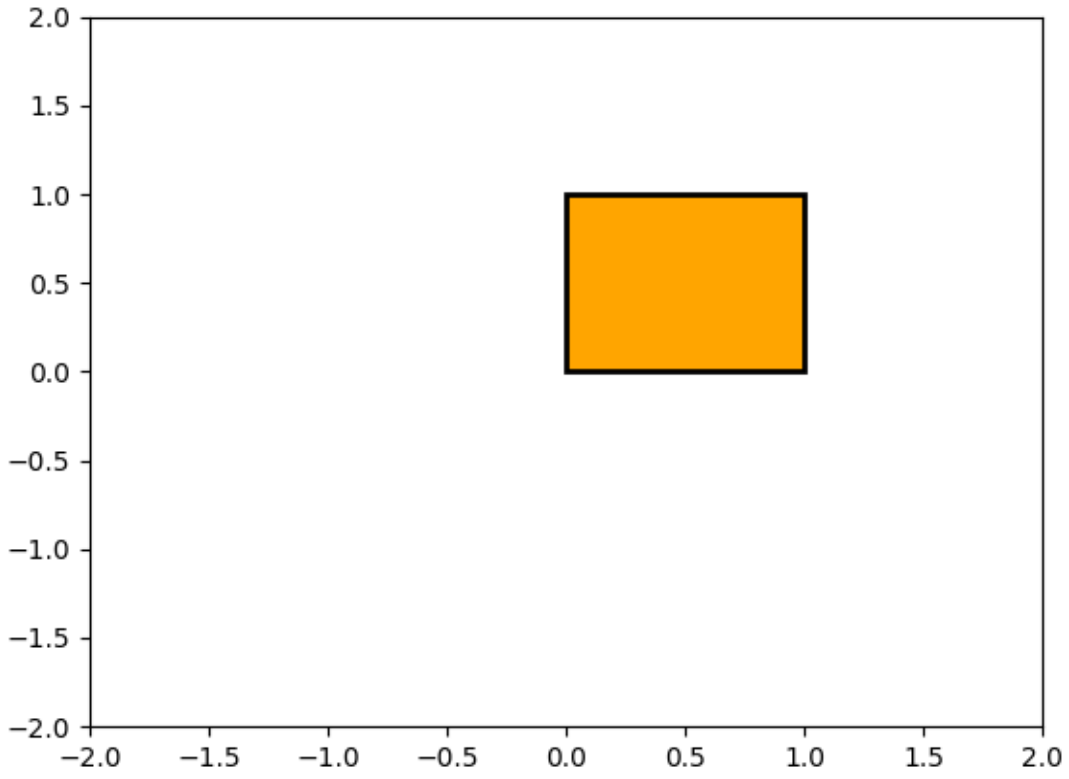
The object underlying all of the *matplotlib.patches* objects is the *Path*, which supports the standard set of moveto, lineto, curveto commands to draw simple and compound outlines consisting of line segments and splines. The `Path` is instantiated with a (N, 2) array of (x, y) vertices, and an N-length array of path codes. For example to draw the unit rectangle from (0, 0) to (1, 1), we could use this code:

```
import numpy as np

import matplotlib.pyplot as plt

import matplotlib.patches as patches
from matplotlib.path import Path

verts = [
   (0., 0.),  # left, bottom
   (0., 1.),  # left, top
   (1., 1.),  # right, top
   (1., 0.),  # right, bottom
   (0., 0.),  # ignored
]

codes = [
    Path.MOVETO,
    Path.LINETO,
    Path.LINETO,
    Path.LINETO,
    Path.CLOSEPOLY,
```

(continues on next page)

(continued from previous page)

```
]

path = Path(verts, codes)

fig, ax = plt.subplots()
patch = patches.PathPatch(path, facecolor='orange', lw=2)
ax.add_patch(patch)
ax.set_xlim(-2, 2)
ax.set_ylim(-2, 2)
plt.show()
```




The following path codes are recognized

| Code | Vertices | Description |
|---|---|---|
| STOP | 1 (ignored) | A marker for the end of the entire path (currently not required and ignored). |
| MOVETO | 1 | Pick up the pen and move to the given vertex. |
| LINETO | 1 | Draw a line from the current position to the given vertex. |
| CURVE3 | 2: 1 control point, 1 end point | Draw a quadratic Bézier curve from the current position, with the given control point, to the given end point. |
| CURVE4 | 3: 2 control points, 1 end point | Draw a cubic Bézier curve from the current position, with the given control points, to the given end point. |
| CLOSEPOLY | 1 (the point is ignored) | Draw a line segment to the start point of the current polyline. |

### Bézier example

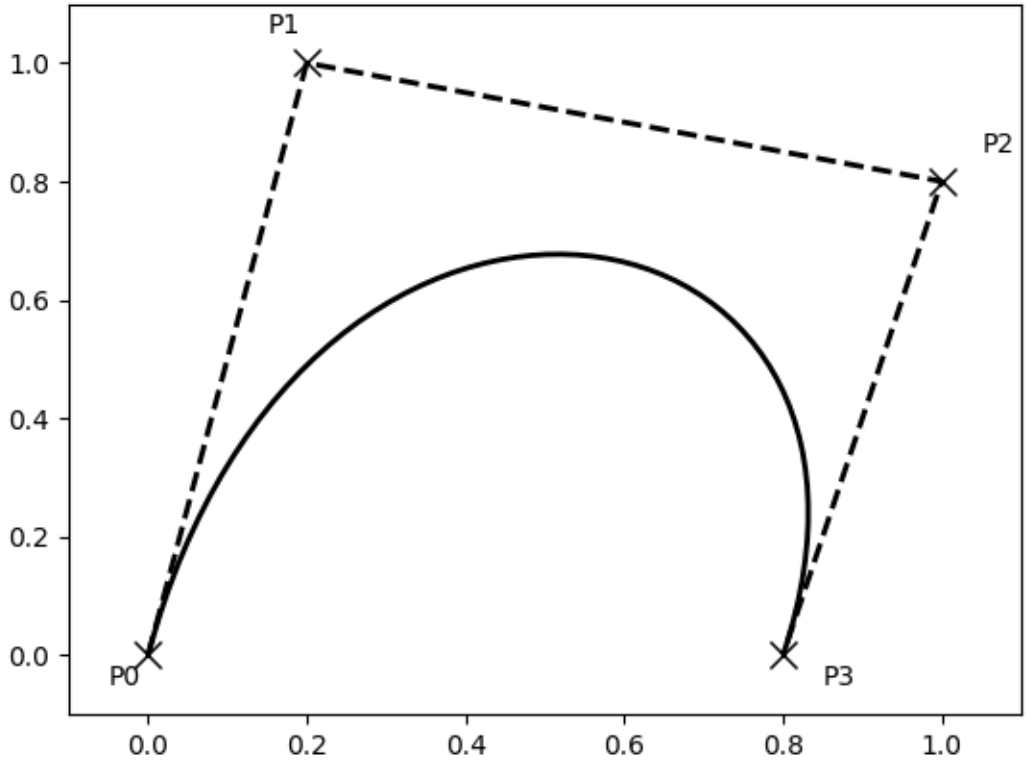
Some of the path components require multiple vertices to specify them: for example CURVE 3 is a Bézier curve with one control point and one end point, and CURVE4 has three vertices for the two control points and the end point. The example below shows a CURVE4 Bézier spline -- the Bézier curve will be contained in the convex hull of the start point, the two control points, and the end point

```
verts = [
   (0., 0.),   # P0
   (0.2, 1.),  # P1
   (1., 0.8),  # P2
   (0.8, 0.),  # P3
]

codes = [
    Path.MOVETO,
    Path.CURVE4,
    Path.CURVE4,
    Path.CURVE4,
]

path = Path(verts, codes)

fig, ax = plt.subplots()
patch = patches.PathPatch(path, facecolor='none', lw=2)
ax.add_patch(patch)

xs, ys = zip(*verts)
ax.plot(xs, ys, 'x--', lw=2, color='black', ms=10)

ax.text(-0.05, -0.05, 'P0')
ax.text(0.15, 1.05, 'P1')
ax.text(1.05, 0.85, 'P2')
ax.text(0.85, -0.05, 'P3')

ax.set_xlim(-0.1, 1.1)
```

(continues on next page)

(continued from previous page)

```
ax.set_ylim(-0.1, 1.1)
plt.show()
```




### Compound paths

All of the simple patch primitives in matplotlib, Rectangle, Circle, Polygon, etc, are implemented with simple path. Plotting functions like *hist()* and *bar()*, which create a number of primitives, e.g., a bunch of Rectangles, can usually be implemented more efficiently using a compound path. The reason bar creates a list of rectangles and not a compound path is largely historical: the *Path* code is comparatively new and bar predates it. While we could change it now, it would break old code, so here we will cover how to create compound paths, replacing the functionality in bar, in case you need to do so in your own code for efficiency reasons, e.g., you are creating an animated bar plot.

We will make the histogram chart by creating a series of rectangles for each histogram bar: the rectangle width is the bin width and the rectangle height is the number of datapoints in that bin. First we'll create some random normally distributed data and compute the histogram. Because NumPy returns the bin edges and not centers, the length of bins is one greater than the length of n in the example below:

```
# histogram our data with numpy
data = np.random.randn(1000)
n, bins = np.histogram(data, 100)
```

We'll now extract the corners of the rectangles. Each of the `left`, `bottom`, etc., arrays below is `len(n)`, where `n` is the array of counts for each histogram bar:

```
# get the corners of the rectangles for the histogram
left = np.array(bins[:-1])
right = np.array(bins[1:])
bottom = np.zeros(len(left))
top = bottom + n
```

Now we have to construct our compound path, which will consist of a series of `MOVETO`, `LINETO` and `CLOSEPOLY` for each rectangle. For each rectangle, we need five vertices: one for the `MOVETO`, three for the `LINETO`, and one for the `CLOSEPOLY`. As indicated in the table above, the vertex for the closepoly is ignored, but we still need it to keep the codes aligned with the vertices:

```
nverts = nrects*(1+3+1)
verts = np.zeros((nverts, 2))
codes = np.ones(nverts, int) * path.Path.LINETO
codes[0::5] = path.Path.MOVETO
codes[4::5] = path.Path.CLOSEPOLY
verts[0::5, 0] = left
verts[0::5, 1] = bottom
verts[1::5, 0] = left
verts[1::5, 1] = top
verts[2::5, 0] = right
verts[2::5, 1] = top
verts[3::5, 0] = right
verts[3::5, 1] = bottom
```

All that remains is to create the path, attach it to a *PathPatch*, and add it to our Axes:

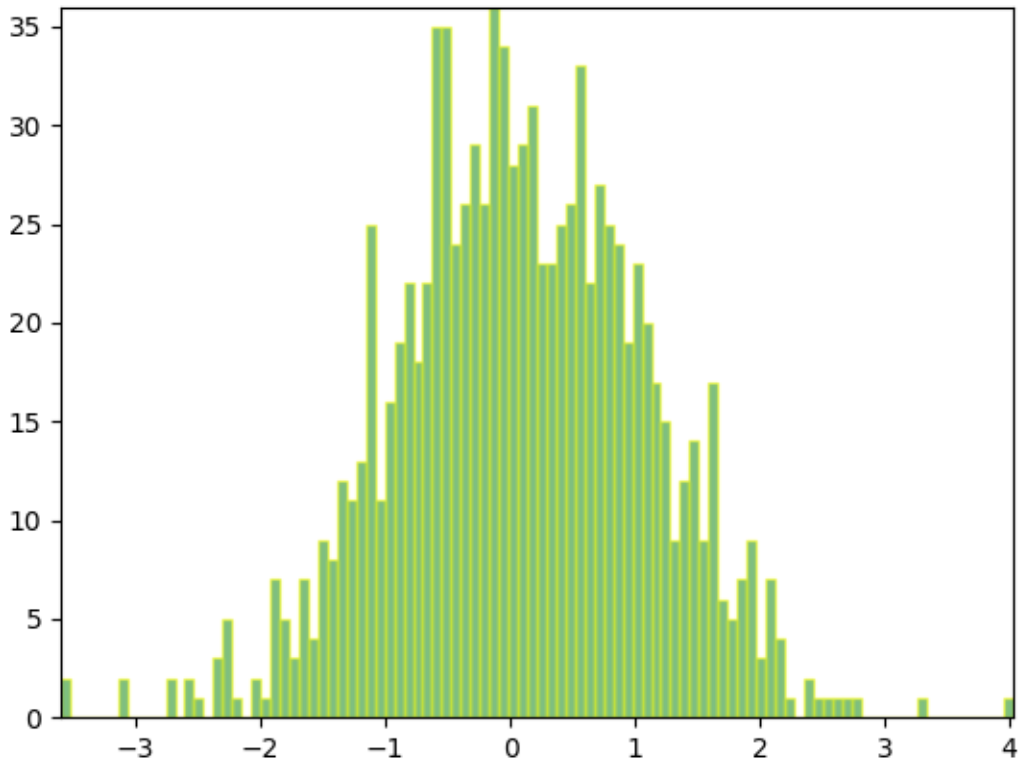
```
barpath = path.Path(verts, codes)
patch = patches.PathPatch(barpath, facecolor='green',
  edgecolor='yellow', alpha=0.5)
ax.add_patch(patch)
```

```
fig, ax = plt.subplots()
# Fixing random state for reproducibility
np.random.seed(19680801)

# histogram our data with numpy
data = np.random.randn(1000)
n, bins = np.histogram(data, 100)

# get the corners of the rectangles for the histogram
left = np.array(bins[:-1])
right = np.array(bins[1:])
bottom = np.zeros(len(left))
```

(continues on next page)

(continued from previous page)

```
top = bottom + n
nrects = len(left)

nverts = nrects*(1+3+1)
verts = np.zeros((nverts, 2))
codes = np.full(nverts, Path.LINETO, dtype=int)
codes[0::5] = Path.MOVETO
codes[4::5] = Path.CLOSEPOLY
verts[0::5, 0] = left
verts[0::5, 1] = bottom
verts[1::5, 0] = left
verts[1::5, 1] = top
verts[2::5, 0] = right
verts[2::5, 1] = top
verts[3::5, 0] = right
verts[3::5, 1] = bottom

barpath = Path(verts, codes)
patch = patches.PathPatch(barpath, facecolor='green',
                          edgecolor='yellow', alpha=0.5)
ax.add_patch(patch)

ax.set_xlim(left[0], right[-1])
ax.set_ylim(bottom.min(), top.max())

plt.show()
```




### 3.4.5 Path effects guide

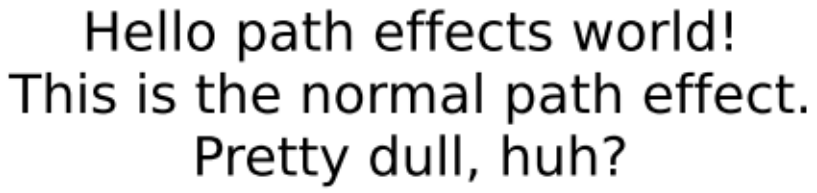
Defining paths that objects follow on a canvas.

Matplotlib's *patheffects* module provides functionality to apply a multiple draw stage to any Artist which can be rendered via a *path.Path*.

Artists which can have a path effect applied to them include *patches.Patch*, *lines.Line2D*, *collections.Collection* and even *text.Text*. Each artist's path effects can be controlled via the *Artist.set_path_effects* method, which takes an iterable of *AbstractPathEffect* instances.

The simplest path effect is the *Normal* effect, which simply draws the artist without any effect:

```
import matplotlib.pyplot as plt

import matplotlib.patheffects as path_effects

fig = plt.figure(figsize=(5, 1.5))
text = fig.text(0.5, 0.5, 'Hello path effects world!\nThis is the normal '
                          'path effect.\nPretty dull, huh?',
                ha='center', va='center', size=20)
text.set_path_effects([path_effects.Normal()])
plt.show()
```

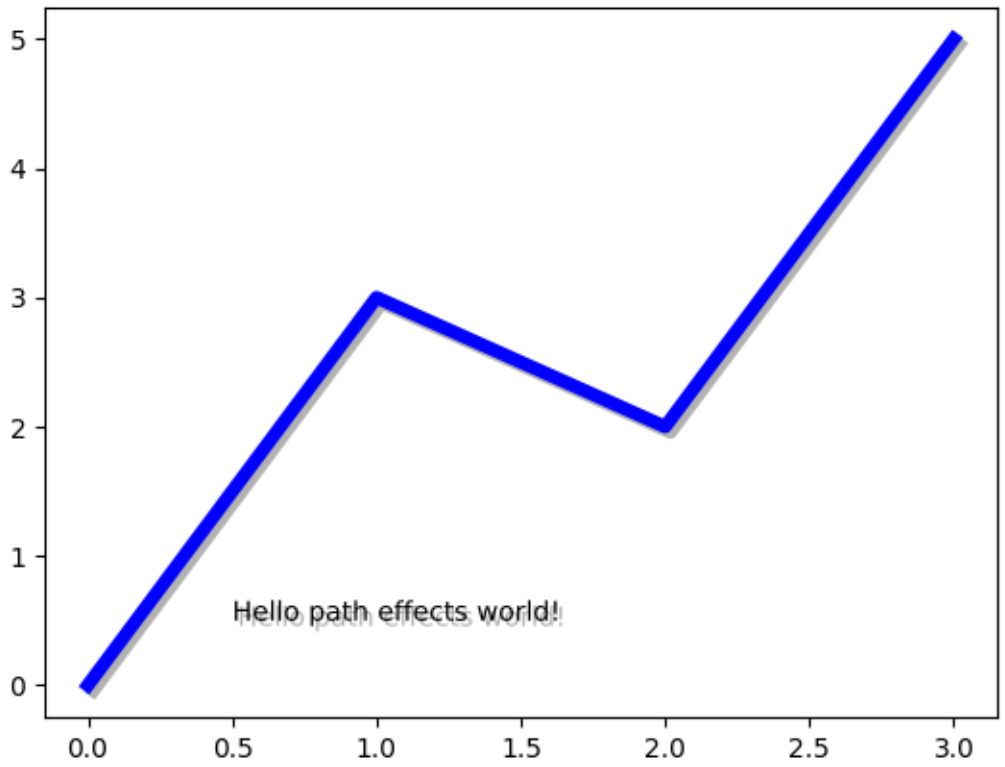



Whilst the plot doesn't look any different to what you would expect without any path effects, the drawing of the text has now been changed to use the path effects framework, opening up the possibilities for more interesting examples.

### Adding a shadow

A far more interesting path effect than *Normal* is the drop-shadow, which we can apply to any of our path based artists. The classes *SimplePatchShadow* and *SimpleLineShadow* do precisely this by drawing either a filled patch or a line patch below the original artist:

```
import matplotlib.patheffects as path_effects

text = plt.text(0.5, 0.5, 'Hello path effects world!',
                path_effects=[path_effects.withSimplePatchShadow()])

plt.plot([0, 3, 2, 5], linewidth=5, color='blue',
         path_effects=[path_effects.SimpleLineShadow(),
                       path_effects.Normal()])
plt.show()
```




Notice the two approaches to setting the path effects in this example. The first uses the `with*` classes to include the desired functionality automatically followed with the "normal" effect, whereas the latter explicitly defines the two path effects to draw.

### Making an Artist stand out

One nice way of making artists visually stand out is to draw an outline in a bold color below the actual artist. The *Stroke* path effect makes this a relatively simple task:

```
fig = plt.figure(figsize=(7, 1))
text = fig.text(0.5, 0.5, 'This text stands out because of\n'
                          'its black border.', color='white',
                          ha='center', va='center', size=30)
text.set_path_effects([path_effects.Stroke(linewidth=3, foreground='black'),
                       path_effects.Normal()])
plt.show()
```

It is important to note that this effect only works because we have drawn the text path twice; once with a thick black line, and then once with the original text path on top.

You may have noticed that the keywords to *Stroke* and *SimplePatchShadow* and *SimpleLineShadow* are not the usual Artist keywords (*facecolor edgecolor*, etc.). This is because with these path effects we are operating at lower level of Matplotlib. In fact, the keywords which are accepted are those for a *matplotlib.backend_bases.GraphicsContextBase* instance, which have been designed for making it easy to create new backends - and not for its user interface.

#### Greater control of the path effect Artist

As already mentioned, some of the path effects operate at a lower level than most users will be used to, meaning that setting keywords such as *facecolor* and *edgecolor* raise an AttributeError. Luckily there is a generic *PathPatchEffect* path effect which creates a *patches.PathPatch* class with the original path. The keywords to this effect are identical to those of *patches.PathPatch*:

```
fig = plt.figure(figsize=(8.5, 1))
t = fig.text(0.02, 0.5, 'Hatch shadow', fontsize=75, weight=1000, va='center')
t.set_path_effects([
    path_effects.PathPatchEffect(
        offset=(4, -4), hatch='xxxx', facecolor='gray'),
    path_effects.PathPatchEffect(
        edgecolor='white', linewidth=1.1, facecolor='black')])
plt.show()
```

### 3.4.6 *origin* and *extent* in `imshow`

*imshow()* allows you to render an image (either a 2D array which will be color-mapped (based on *norm* and *cmap*) or a 3D RGB(A) array which will be used as-is) to a rectangular region in data space. The orientation of the image in the final rendering is controlled by the *origin* and *extent* keyword arguments (and attributes on the resulting *AxesImage* instance) and the data limits of the Axes.

The *extent* keyword arguments controls the bounding box in data coordinates that the image will fill specified as `(left, right, bottom, top)` in **data coordinates**, the *origin* keyword argument controls how the image fills that bounding box, and the orientation in the final rendered image is also affected by the axes limits.

---

**Hint:** Most of the code below is used for adding labels and informative text to the plots. The described effects of *origin* and *extent* can be seen in the plots without the need to follow all code details.

For a quick understanding, you may want to skip the code details below and directly continue with the discussion of the results.

```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.gridspec import GridSpec


def index_to_coordinate(index, extent, origin):
    """Return the pixel center of an index."""
    left, right, bottom, top = extent

    hshift = 0.5 * np.sign(right - left)
    left, right = left + hshift, right - hshift
    vshift = 0.5 * np.sign(top - bottom)
    bottom, top = bottom + vshift, top - vshift

    if origin == 'upper':
        bottom, top = top, bottom

    return {
        "[0, 0]": (left, bottom),
        "[M', 0]": (left, top),
        "[0, N']": (right, bottom),
        "[M', N']": (right, top),
    }[index]


def get_index_label_pos(index, extent, origin, inverted_xindex):
    """
    Return the desired position and horizontal alignment of an index label.
    """
    if extent is None:
        extent = lookup_extent(origin)
    left, right, bottom, top = extent
    x, y = index_to_coordinate(index, extent, origin)

    is_x0 = index[-2:] == "0]"
    halign = 'left' if is_x0 ^ inverted_xindex else 'right'
    hshift = 0.5 * np.sign(left - right)
    x += hshift * (1 if is_x0 else -1)
    return x, y, halign


def get_color(index, data, cmap):
    """Return the data color of an index."""
    val = {
        "[0, 0]": data[0, 0],
        "[0, N']": data[0, -1],
        "[M', 0]": data[-1, 0],
        "[M', N']": data[-1, -1],
```

(continues on next page)

(continued from previous page)

```
    }[index]
    return cmap(val / data.max())


def lookup_extent(origin):
    """Return extent for label positioning when not given explicitly."""
    if origin == 'lower':
        return (-0.5, 6.5, -0.5, 5.5)
    else:
        return (-0.5, 6.5, 5.5, -0.5)


def set_extent_None_text(ax):
    ax.text(3, 2.5, 'equals\nextent=None', size='large',
            ha='center', va='center', color='w')


def plot_imshow_with_labels(ax, data, extent, origin, xlim, ylim):
    """Actually run ``imshow()`` and add extent and index labels."""
    im = ax.imshow(data, origin=origin, extent=extent)

    # extent labels (left, right, bottom, top)
    left, right, bottom, top = im.get_extent()
    if xlim is None or top > bottom:
        upper_string, lower_string = 'top', 'bottom'
    else:
        upper_string, lower_string = 'bottom', 'top'
    if ylim is None or left < right:
        port_string, starboard_string = 'left', 'right'
        inverted_xindex = False
    else:
        port_string, starboard_string = 'right', 'left'
        inverted_xindex = True
    bbox_kwargs = {'fc': 'w', 'alpha': .75, 'boxstyle': "round4"}
    ann_kwargs = {'xycoords': 'axes fraction',
                  'textcoords': 'offset points',
                  'bbox': bbox_kwargs}
    ax.annotate(upper_string, xy=(.5, 1), xytext=(0, -1),
                ha='center', va='top', **ann_kwargs)
    ax.annotate(lower_string, xy=(.5, 0), xytext=(0, 1),
                ha='center', va='bottom', **ann_kwargs)
    ax.annotate(port_string, xy=(0, .5), xytext=(1, 0),
                ha='left', va='center', rotation=90,
                **ann_kwargs)
    ax.annotate(starboard_string, xy=(1, .5), xytext=(-1, 0),
                ha='right', va='center', rotation=-90,
                **ann_kwargs)
    ax.set_title(f'origin: {origin}')

    # index labels
    for index in ["[0, 0]", "[0, N']", "[M', 0]", "[M', N']"]:
        tx, ty, halign = get_index_label_pos(index, extent, origin,
```

(continues on next page)

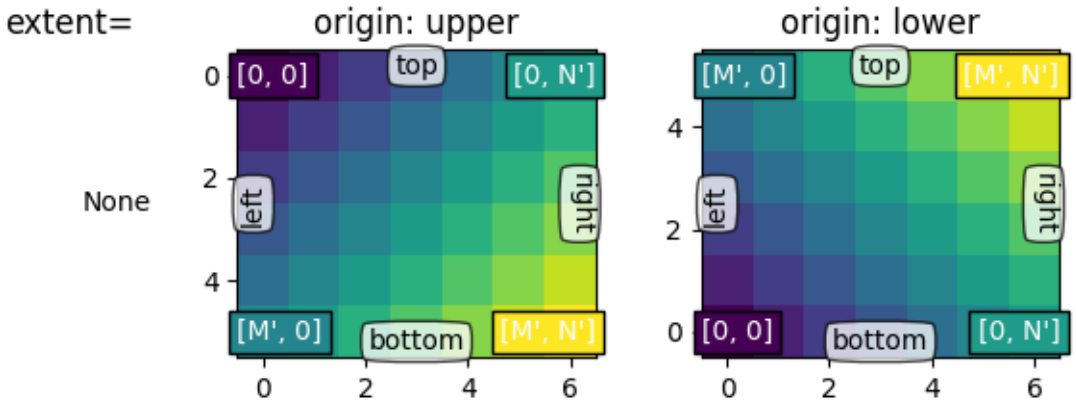
(continued from previous page)

```
                                                inverted_xindex)
        facecolor = get_color(index, data, im.get_cmap())
        ax.text(tx, ty, index, color='white', ha=halign, va='center',
                bbox={'boxstyle': 'square', 'facecolor': facecolor})
    if xlim:
        ax.set_xlim(*xlim)
    if ylim:
        ax.set_ylim(*ylim)


def generate_imshow_demo_grid(extents, xlim=None, ylim=None):
    N = len(extents)
    fig = plt.figure(tight_layout=True)
    fig.set_size_inches(6, N * (11.25) / 5)
    gs = GridSpec(N, 5, figure=fig)

    columns = {'label': [fig.add_subplot(gs[j, 0]) for j in range(N)],
               'upper': [fig.add_subplot(gs[j, 1:3]) for j in range(N)],
               'lower': [fig.add_subplot(gs[j, 3:5]) for j in range(N)]}
    x, y = np.ogrid[0:6, 0:7]
    data = x + y

    for origin in ['upper', 'lower']:
        for ax, extent in zip(columns[origin], extents):
            plot_imshow_with_labels(ax, data, extent, origin, xlim, ylim)

    columns['label'][0].set_title('extent=')
    for ax, extent in zip(columns['label'], extents):
        if extent is None:
            text = 'None'
        else:
            left, right, bottom, top = extent
            text = (f'left: {left:0.1f}\nright: {right:0.1f}\n'
                    f'bottom: {bottom:0.1f}\ntop: {top:0.1f}\n')
        ax.text(1., .5, text, transform=ax.transAxes, ha='right', va='center')
        ax.axis('off')
    return columns
```

### Default extent

First, let's have a look at the default `extent=None`

```
generate_imshow_demo_grid(extents=[None])
```




Generally, for an array of shape (M, N), the first index runs along the vertical, the second index runs along the horizontal. The pixel centers are at integer positions ranging from 0 to `N' = N - 1` horizontally and from 0 to `M' = M - 1` vertically. *origin* determines how the data is filled in the bounding box.

For `origin='lower'`:

- [0, 0] is at (left, bottom)
- [M', 0] is at (left, top)
- [0, N'] is at (right, bottom)
- [M', N'] is at (right, top)

`origin='upper'` reverses the vertical axes direction and filling:

- [0, 0] is at (left, top)
- [M', 0] is at (left, bottom)
- [0, N'] is at (right, top)
- [M', N'] is at (right, bottom)

In summary, the position of the [0, 0] index as well as the extent are influenced by *origin*:

| origin | [0, 0] position | extent |
|---|---|---|
| upper | top left | `(-0.5, numcols-0.5, numrows-0.5, -0.5)` |
| lower | bottom left | `(-0.5, numcols-0.5, -0.5, numrows-0.5)` |

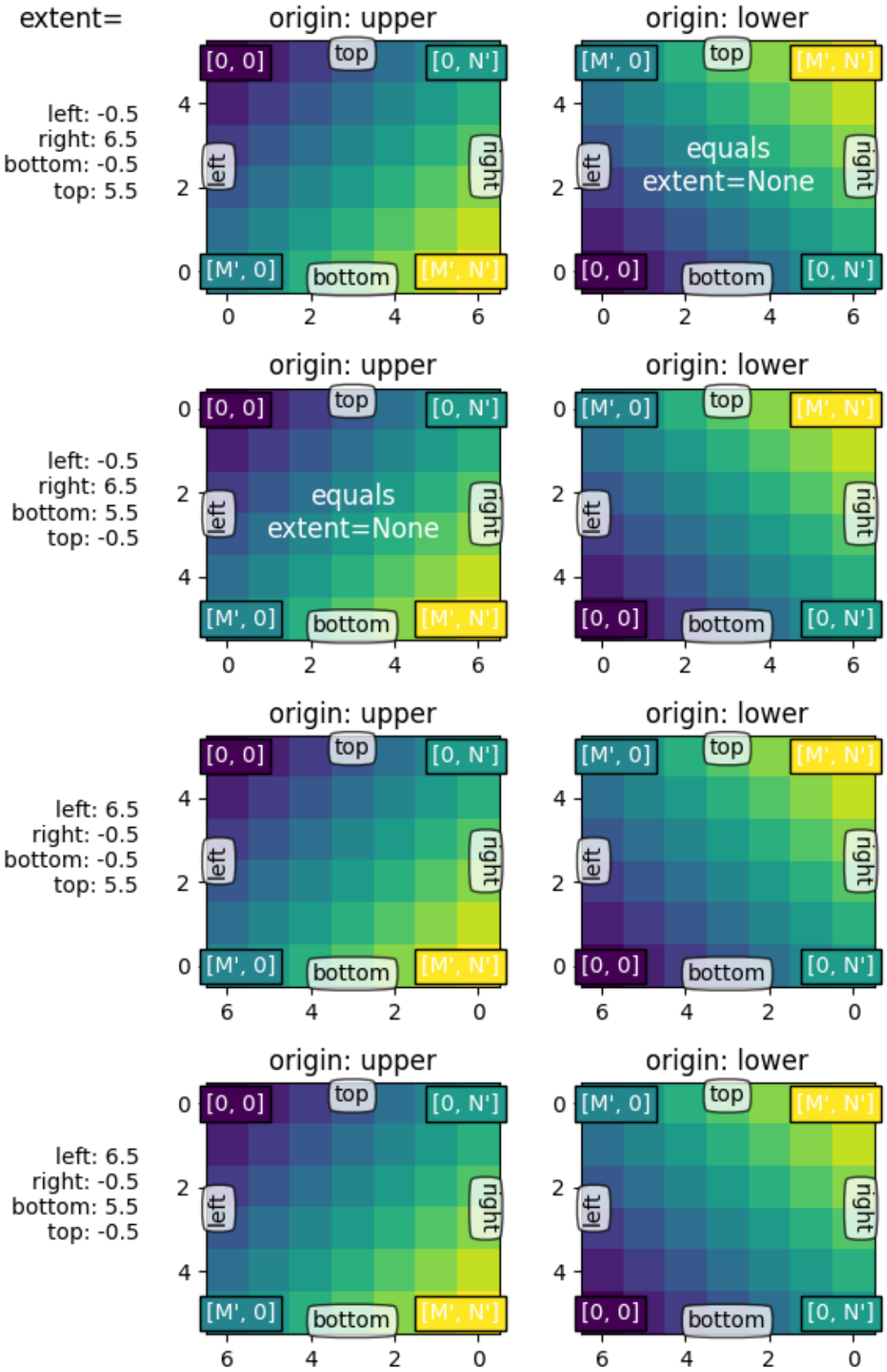
The default value of *origin* is set by *`rcParams["image.origin"]`* (default: `'upper'`) which defaults to `'upper'` to match the matrix indexing conventions in math and computer graphics image indexing conventions.

### Explicit extent

By setting *extent* we define the coordinates of the image area. The underlying image data is interpolated/resampled to fill that area.

If the Axes is set to autoscale, then the view limits of the Axes are set to match the *extent* which ensures that the coordinate set by `(left, bottom)` is at the bottom left of the Axes! However, this may invert the axis so they do not increase in the 'natural' direction.

```
extents = [(-0.5, 6.5, -0.5, 5.5),
           (-0.5, 6.5, 5.5, -0.5),
           (6.5, -0.5, -0.5, 5.5),
           (6.5, -0.5, 5.5, -0.5)]

columns = generate_imshow_demo_grid(extents)
set_extent_None_text(columns['upper'][1])
set_extent_None_text(columns['lower'][0])
```


extent=
origin: upper
origin: lower
left: -0.5
right: 6.5
bottom: -0.5
top: 5.5
left: -0.5
right: 6.5
bottom: 5.5
top: -0.5
left: 6.5
right: -0.5
bottom: -0.5
top: 5.5
left: 6.5
right: -0.5
bottom: 5.5
top: -0.5
[0, 0]
[0, N']
[M', 0]
[M', N']
top
bottom
left
right
equals
extent=None

### Explicit extent and axes limits

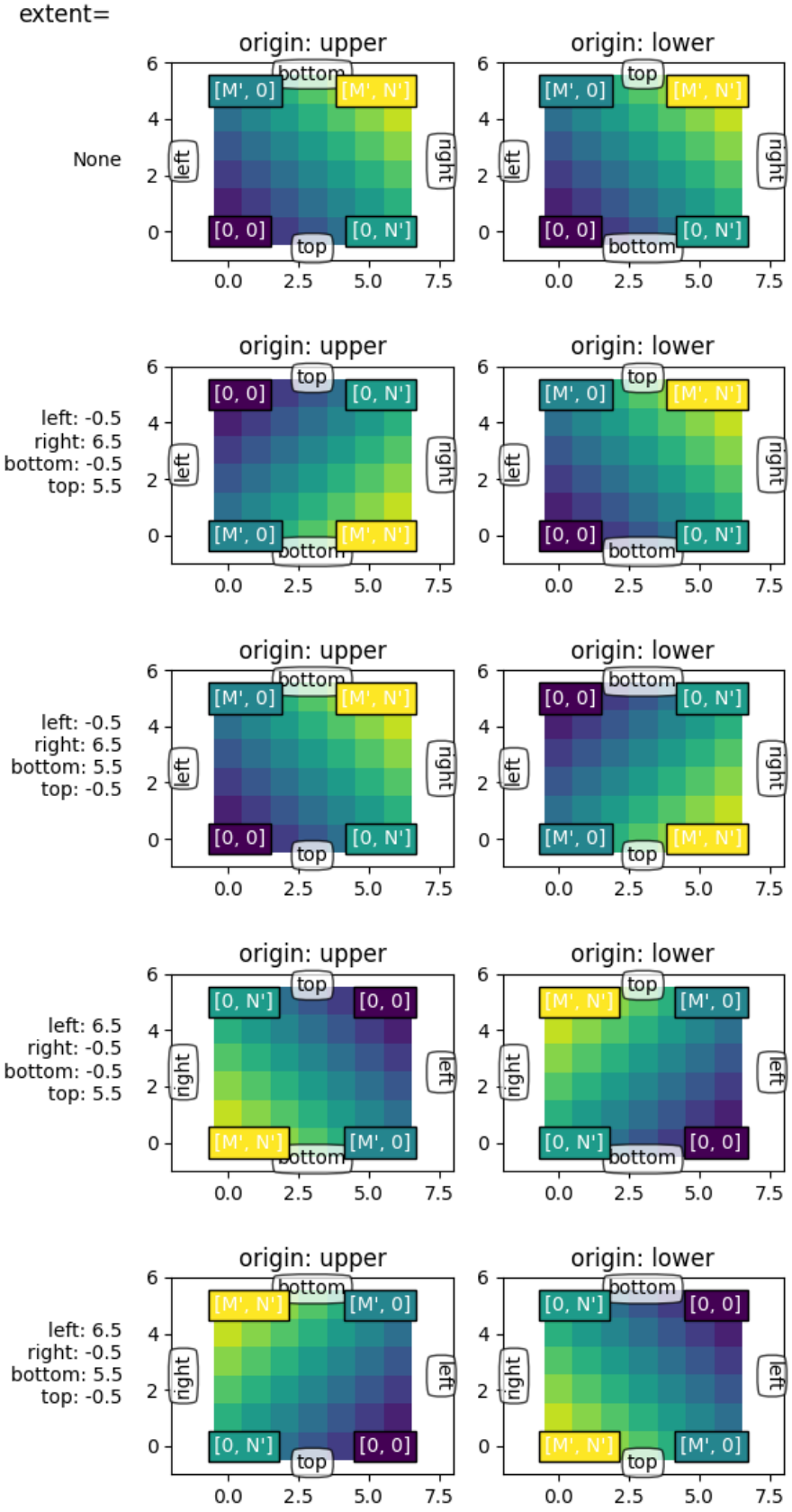
If we fix the axes limits by explicitly setting *set_xlim* / *set_ylim*, we force a certain size and orientation of the Axes. This can decouple the 'left-right' and 'top-bottom' sense of the image from the orientation on the screen.

In the example below we have chosen the limits slightly larger than the extent (note the white areas within the Axes).

While we keep the extents as in the examples before, the coordinate (0, 0) is now explicitly put at the bottom left and values increase to up and to the right (from the viewer's point of view). We can see that:

- The coordinate `(left, bottom)` anchors the image which then fills the box going towards the `(right, top)` point in data space.
- The first column is always closest to the 'left'.
- *origin* controls if the first row is closest to 'top' or 'bottom'.
- The image may be inverted along either direction.
- The 'left-right' and 'top-bottom' sense of the image may be uncoupled from the orientation on the screen.

```
generate_imshow_demo_grid(extents=[None] + extents,
                          xlim=(-2, 8), ylim=(-1, 6))

plt.show()
```


extent=
origin: upper
origin: lower
None
left: -0.5
right: 6.5
bottom: -0.5
top: 5.5
left: -0.5
right: 6.5
bottom: 5.5
top: -0.5
left: 6.5
right: -0.5
bottom: -0.5
top: 5.5
left: 6.5
right: -0.5
bottom: 5.5
top: -0.5
[M', 0]
[M', N']
[0, 0]
[0, N']
top
bottom
left
right
0.0
2.5
5.0
7.5
0
2
4
6

**Total running time of the script:** (0 minutes 3.362 seconds)

### 3.4.7 Transformations Tutorial

Like any graphics packages, Matplotlib is built on top of a transformation framework to easily move between coordinate systems, the userland *data* coordinate system, the *axes* coordinate system, the *figure* coordinate system, and the *display* coordinate system. In 95% of your plotting, you won't need to think about this, as it happens under the hood, but as you push the limits of custom figure generation, it helps to have an understanding of these objects, so you can reuse the existing transformations Matplotlib makes available to you, or create your own (see *matplotlib.transforms*). The table below summarizes some useful coordinate systems, a description of each system, and the transformation object for going from each coordinate system to the *display* coordinates. In the "Transformation Object" column, `ax` is a *Axes* instance, `fig` is a *Figure* instance, and `subfigure` is a *SubFigure* instance.

| Coordinate system | Description | Transformation object from system to display |
|---|---|---|
| "data" | The coordinate system of the data in the Axes. | `ax.transData` |
| "axes" | The coordinate system of the *Axes*; (0, 0) is bottom left of the Axes, and (1, 1) is top right of the Axes. | `ax.transAxes` |
| "subfigure" | The coordinate system of the *SubFigure*; (0, 0) is bottom left of the subfigure, and (1, 1) is top right of the subfigure. If a figure has no subfigures, this is the same as `transFigure`. | `subfigure.transSubfigure` |
| "figure" | The coordinate system of the *Figure*; (0, 0) is bottom left of the figure, and (1, 1) is top right of the figure. | `fig.transFigure` |
| "figure inches' | The coordinate system of the *Figure* in inches; (0, 0) is bottom left of the figure, and (width, height) is the top right of the figure in inches. | `fig.dpi_scale_trans` |
| "xaxis" "yaxis" | Blended coordinate systems, using data coordinates on one direction and axes coordinates on the other. | `ax.get_xaxis_transform()`, `ax.get_yaxis_transform()` |
| "display" | The native coordinate system of the output ; (0, 0) is the bottom left of the window, and (width, height) is top right of the output in "display units". The exact interpretation of the units depends on the back end. For example it is pixels for Agg and points for svg/pdf. | `None`, or *IdentityTransform()* |

The *Transform* objects are naive to the source and destination coordinate systems, however the objects referred to in the table above are constructed to take inputs in their coordinate system, and transform the input to the *display* coordinate system. That is why the *display* coordinate system has `None` for the "Transformation Object" column -- it already is in *display* coordinates. The naming and destination conventions are an aid to keeping track of the available "standard" coordinate systems and transforms.

The transformations also know how to invert themselves (via *Transform.inverted*) to generate a transform from output coordinate system back to the input coordinate system. For example, `ax.transData`

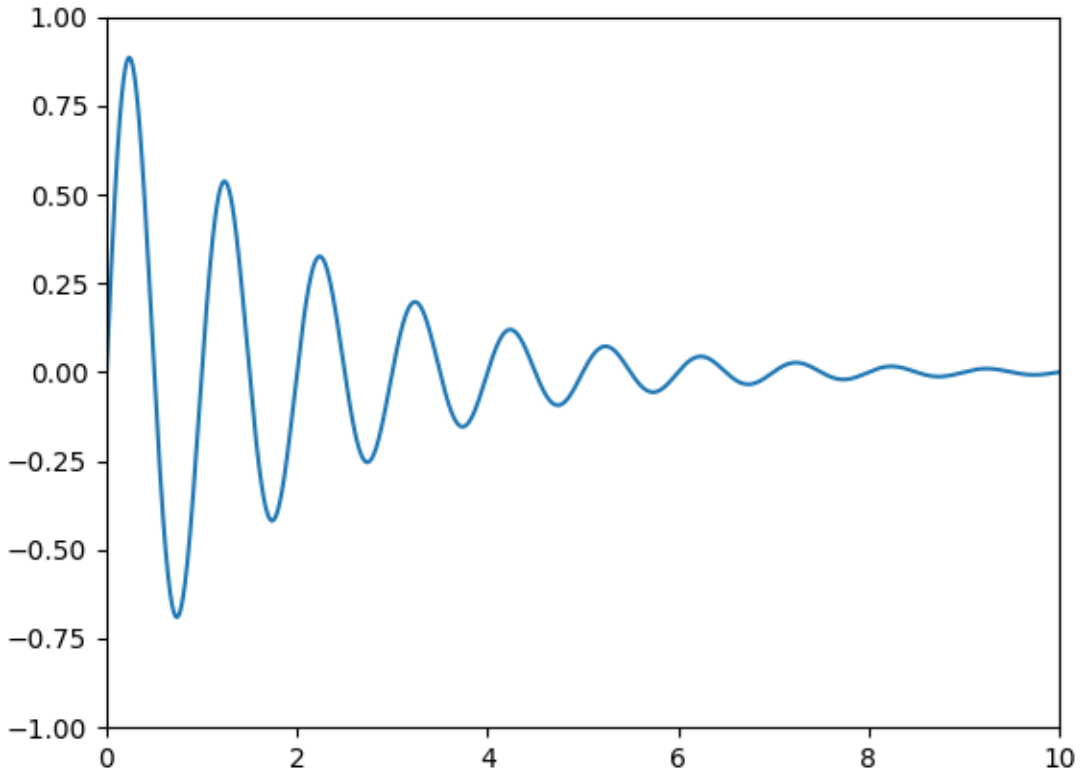
converts values in data coordinates to display coordinates and `ax.transData.inverted()` is a *matplotlib.transforms.Transform* that goes from display coordinates to data coordinates. This is particularly useful when processing events from the user interface, which typically occur in display space, and you want to know where the mouse click or key-press occurred in your *data* coordinate system.

Note that specifying the position of Artists in *display* coordinates may change their relative location if the `dpi` or size of the figure changes. This can cause confusion when printing or changing screen resolution, because the object can change location and size. Therefore, it is most common for artists placed in an Axes or figure to have their transform set to something *other* than the *IdentityTransform()*; the default when an artist is added to an Axes using *add_artist* is for the transform to be `ax.transData` so that you can work and think in *data* coordinates and let Matplotlib take care of the transformation to *display*.

### Data coordinates

Let's start with the most commonly used coordinate, the *data* coordinate system. Whenever you add data to the Axes, Matplotlib updates the datalimits, most commonly updated with the *set_xlim()* and *set_ylim()* methods. For example, in the figure below, the data limits stretch from 0 to 10 on the x-axis, and -1 to 1 on the y-axis.

```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib.patches as mpatches

x = np.arange(0, 10, 0.005)
y = np.exp(-x/2.) * np.sin(2*np.pi*x)

fig, ax = plt.subplots()
ax.plot(x, y)
ax.set_xlim(0, 10)
ax.set_ylim(-1, 1)

plt.show()
```




You can use the `ax.transData` instance to transform from your *data* to your *display* coordinate system, either a single point or a sequence of points as shown below:

```
In [14]: type(ax.transData)
Out[14]: <class 'matplotlib.transforms.CompositeGenericTransform'>

In [15]: ax.transData.transform((5, 0))
Out[15]: array([ 335.175,  247.   ])

In [16]: ax.transData.transform([(5, 0), (1, 2)])
Out[16]:
array([[ 335.175,  247.   ],
       [ 132.435,  642.2  ]])
```

You can use the *`inverted()`* method to create a transform which will take you from *display* to *data* coordinates:

```
In [41]: inv = ax.transData.inverted()

In [42]: type(inv)
Out[42]: <class 'matplotlib.transforms.CompositeGenericTransform'>

In [43]: inv.transform((335.175,  247.))
```

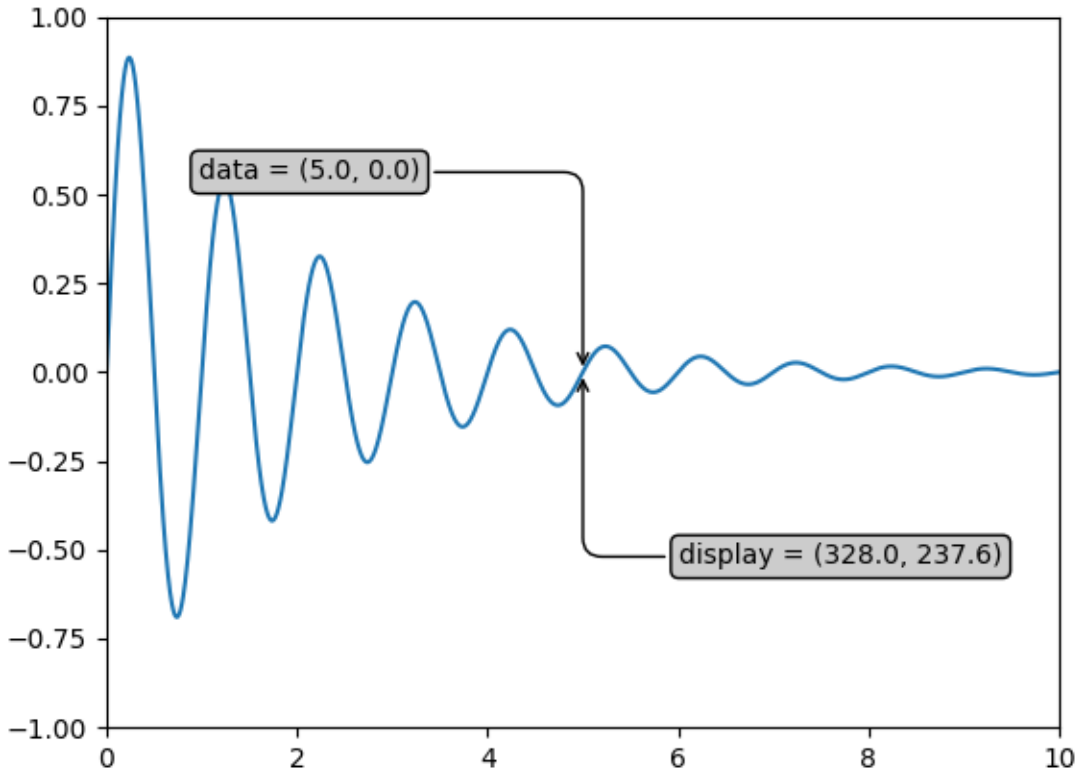
(continues on next page)

(continued from previous page)

```
Out[43]: array([ 5.,  0.])
```

If your are typing along with this tutorial, the exact values of the *display* coordinates may differ if you have a different window size or dpi setting. Likewise, in the figure below, the display labeled points are probably not the same as in the ipython session because the documentation figure size defaults are different.

```
x = np.arange(0, 10, 0.005)
y = np.exp(-x/2.) * np.sin(2*np.pi*x)

fig, ax = plt.subplots()
ax.plot(x, y)
ax.set_xlim(0, 10)
ax.set_ylim(-1, 1)

xdata, ydata = 5, 0
# This computing the transform now, if anything
# (figure size, dpi, axes placement, data limits, scales..)
# changes re-calling transform will get a different value.
xdisplay, ydisplay = ax.transData.transform((xdata, ydata))

bbox = dict(boxstyle="round", fc="0.8")
arrowprops = dict(
    arrowstyle="->",
    connectionstyle="angle,angleA=0,angleB=90,rad=10")

offset = 72
ax.annotate(f'data = ({xdata:.1f}, {ydata:.1f})',
            (xdata, ydata), xytext=(-2*offset, offset), textcoords='offset␣
 ↪points',
            bbox=bbox, arrowprops=arrowprops)

disp = ax.annotate(f'display = ({xdisplay:.1f}, {ydisplay:.1f})',
                   (xdisplay, ydisplay), xytext=(0.5*offset, -offset),
                   xycoords='figure pixels',
                   textcoords='offset points',
                   bbox=bbox, arrowprops=arrowprops)

plt.show()
```




**Warning:** If you run the source code in the example above in a GUI backend, you may also find that the two arrows for the *data* and *display* annotations do not point to exactly the same point. This is because the display point was computed before the figure was displayed, and the GUI backend may slightly resize the figure when it is created. The effect is more pronounced if you resize the figure yourself. This is one good reason why you rarely want to work in *display* space, but you can connect to the `'on_draw'` `Event` to update *figure* coordinates on figure draws; see *Event handling and picking*.

When you change the x or y limits of your axes, the data limits are updated so the transformation yields a new display point. Note that when we just change the ylim, only the y-display coordinate is altered, and when we change the xlim too, both are altered. More on this later when we talk about the `Bbox`.

```
In [54]: ax.transData.transform((5, 0))
Out[54]: array([ 335.175,  247.   ])

In [55]: ax.set_ylim(-1, 2)
Out[55]: (-1, 2)

In [56]: ax.transData.transform((5, 0))
Out[56]: array([ 335.175     ,  181.13333333])
```

(continues on next page)

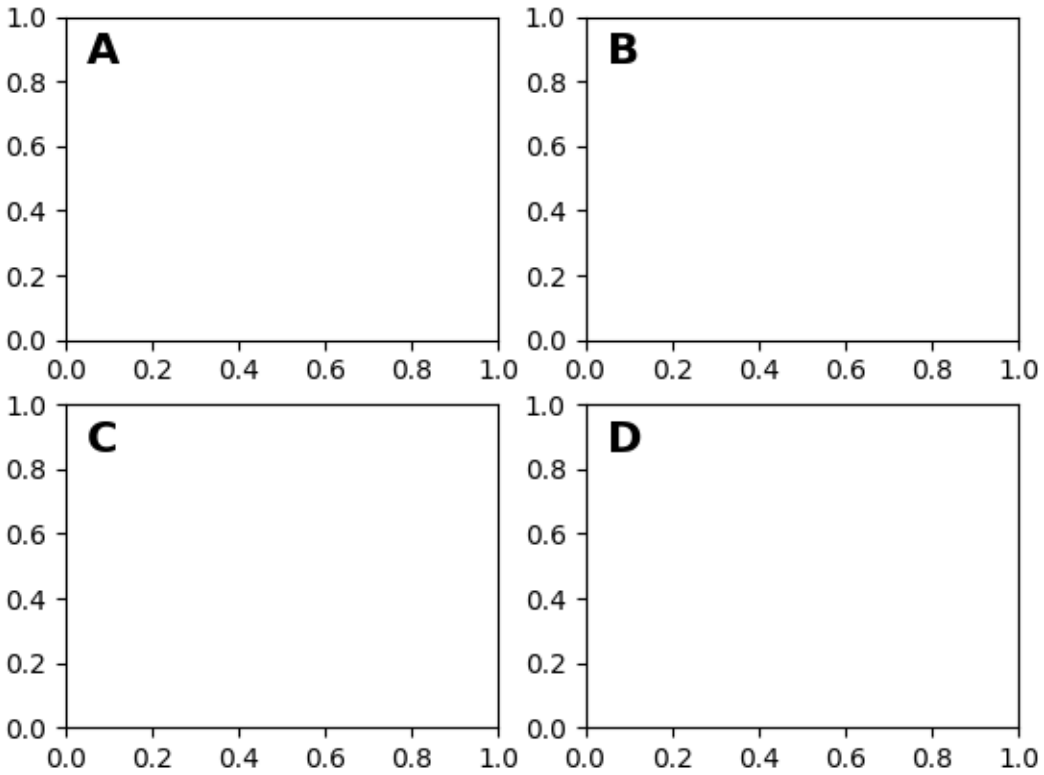
(continued from previous page)

```
In [57]: ax.set_xlim(10, 20)
Out[57]: (10, 20)

In [58]: ax.transData.transform((5, 0))
Out[58]: array([-171.675     ,  181.13333333])
```

### Axes coordinates

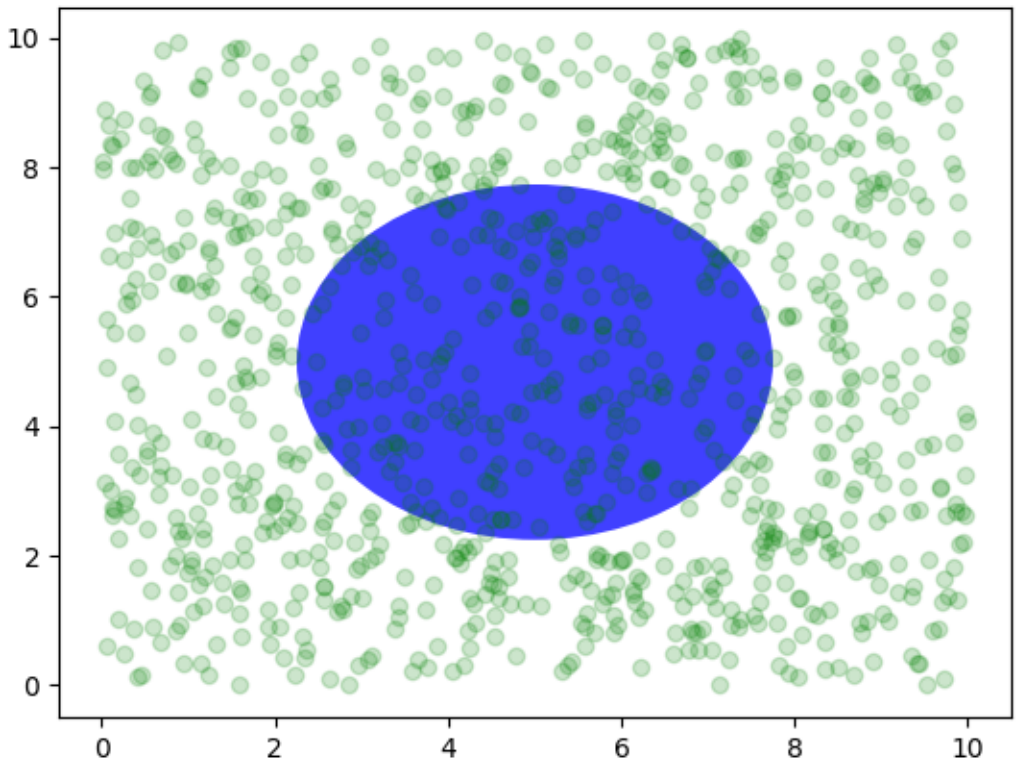
After the *data* coordinate system, *axes* is probably the second most useful coordinate system. Here the point (0, 0) is the bottom left of your Axes or subplot, (0.5, 0.5) is the center, and (1.0, 1.0) is the top right. You can also refer to points outside the range, so (-0.1, 1.1) is to the left and above your Axes. This coordinate system is extremely useful when placing text in your Axes, because you often want a text bubble in a fixed, location, e.g., the upper left of the Axes pane, and have that location remain fixed when you pan or zoom. Here is a simple example that creates four panels and labels them 'A', 'B', 'C', 'D' as you often see in journals. A more sophisticated approach for such labeling is presented at *Labelling subplots*.

```
fig = plt.figure()
for i, label in enumerate(('A', 'B', 'C', 'D')):
    ax = fig.add_subplot(2, 2, i+1)
    ax.text(0.05, 0.95, label, transform=ax.transAxes,
            fontsize=16, fontweight='bold', va='top')

plt.show()
```




You can also make lines or patches in the *axes* coordinate system, but this is less useful in my experience than using `ax.transAxes` for placing text. Nonetheless, here is a silly example which plots some random dots in data space, and overlays a semi-transparent *`Circle`* centered in the middle of the Axes with a radius one quarter of the Axes -- if your Axes does not preserve aspect ratio (see *`set_aspect()`*), this will look like an ellipse. Use the pan/zoom tool to move around, or manually change the data xlim and ylim, and you will see the data move, but the circle will remain fixed because it is not in *data* coordinates and will always remain at the center of the Axes.

```
fig, ax = plt.subplots()
x, y = 10*np.random.rand(2, 1000)
ax.plot(x, y, 'go', alpha=0.2)  # plot some data in data coordinates

circ = mpatches.Circle((0.5, 0.5), 0.25, transform=ax.transAxes,
                       facecolor='blue', alpha=0.75)
ax.add_patch(circ)
plt.show()
```




### Blended transformations

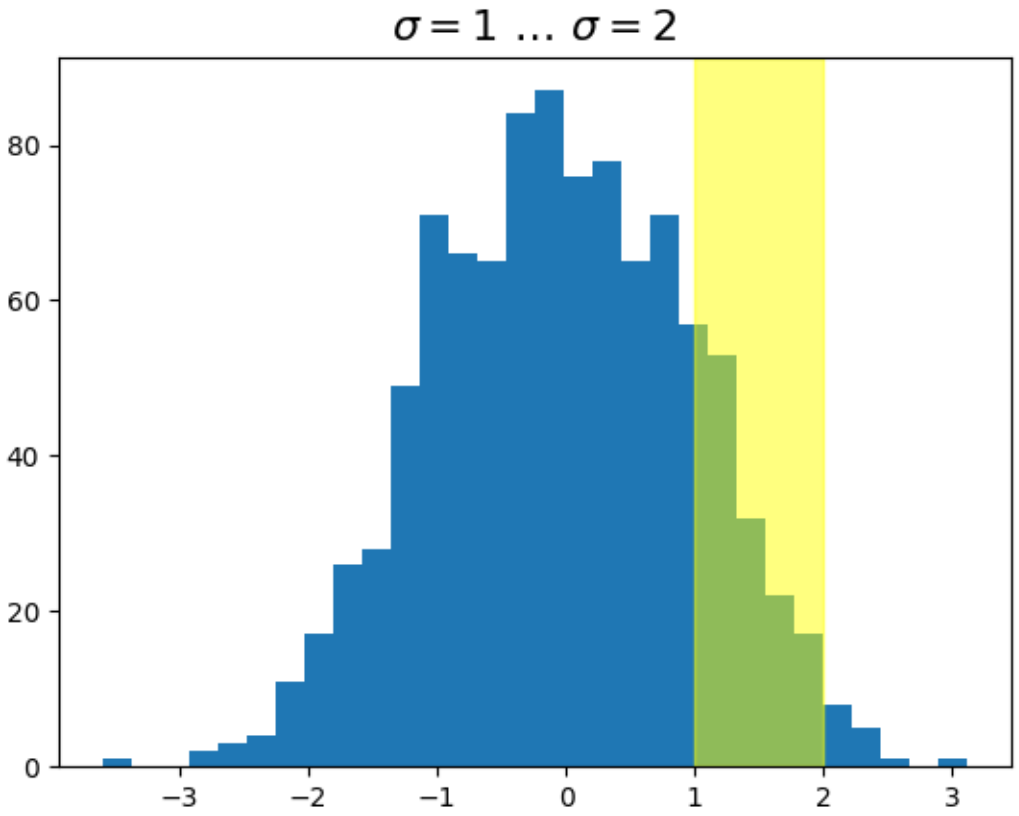
Drawing in *blended* coordinate spaces which mix *axes* with *data* coordinates is extremely useful, for example to create a horizontal span which highlights some region of the y-data but spans across the x-axis regardless of the data limits, pan or zoom level, etc. In fact these blended lines and spans are so useful, we have built-in functions to make them easy to plot (see *axhline()*, *axvline()*, *axhspan()*, *axvspan()*) but for didactic purposes we will implement the horizontal span here using a blended transformation. This trick only works for separable transformations, like you see in normal Cartesian coordinate systems, but not on inseparable transformations like the *PolarTransform*.

```
import matplotlib.transforms as transforms

fig, ax = plt.subplots()
x = np.random.randn(1000)

ax.hist(x, 30)
ax.set_title(r'$\sigma=1 \/ \dots \/ \sigma=2$', fontsize=16)

# the x coords of this transformation are data, and the y coord are axes
trans = transforms.blended_transform_factory(
    ax.transData, ax.transAxes)
```

(continues on next page)

(continued from previous page)

```
# highlight the 1..2 stddev region with a span.
# We want x to be in data coordinates and y to span from 0..1 in axes coords.
rect = mpatches.Rectangle((1, 0), width=1, height=1, transform=trans,
                          color='yellow', alpha=0.5)
ax.add_patch(rect)

plt.show()
```



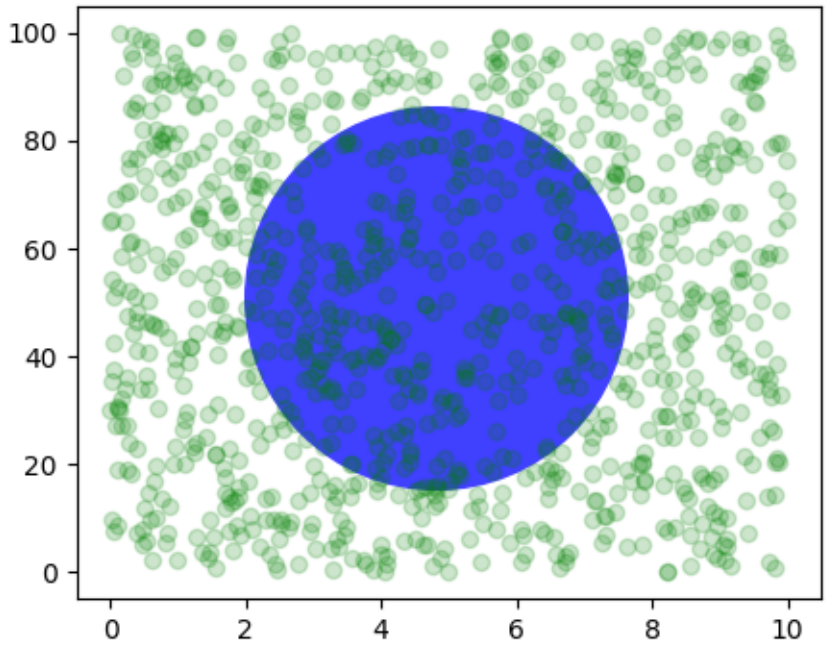

**Note:** The blended transformations where x is in *data* coords and y in *axes* coordinates is so useful that we have helper methods to return the versions Matplotlib uses internally for drawing ticks, ticklabels, etc. The methods are *matplotlib.axes.Axes.get_xaxis_transform()* and *matplotlib.axes.Axes.get_yaxis_transform()*. So in the example above, the call to *blended_transform_factory()* can be replaced by `get_xaxis_transform`:

```
trans = ax.get_xaxis_transform()
```

### Plotting in physical coordinates

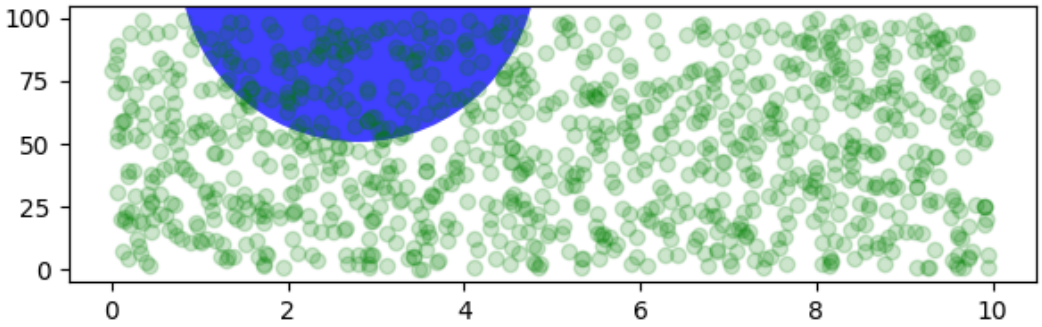
Sometimes we want an object to be a certain physical size on the plot. Here we draw the same circle as above, but in physical coordinates. If done interactively, you can see that changing the size of the figure does not change the offset of the circle from the lower-left corner, does not change its size, and the circle remains a circle regardless of the aspect ratio of the Axes.

```
fig, ax = plt.subplots(figsize=(5, 4))
x, y = 10*np.random.rand(2, 1000)
ax.plot(x, y*10., 'go', alpha=0.2)  # plot some data in data coordinates
# add a circle in fixed-coordinates
circ = mpatches.Circle((2.5, 2), 1.0, transform=fig.dpi_scale_trans,
                       facecolor='blue', alpha=0.75)
ax.add_patch(circ)
plt.show()
```




If we change the figure size, the circle does not change its absolute position and is cropped.

```
fig, ax = plt.subplots(figsize=(7, 2))
x, y = 10*np.random.rand(2, 1000)
ax.plot(x, y*10., 'go', alpha=0.2)  # plot some data in data coordinates
# add a circle in fixed-coordinates
circ = mpatches.Circle((2.5, 2), 1.0, transform=fig.dpi_scale_trans,
                       facecolor='blue', alpha=0.75)
ax.add_patch(circ)
plt.show()
```




Another use is putting a patch with a set physical dimension around a data point on the Axes. Here we add together two transforms. The first sets the scaling of how large the ellipse should be and the second sets its position. The ellipse is then placed at the origin, and then we use the helper transform *ScaledTranslation* to move it to the right place in the `ax.transData` coordinate system. This helper is instantiated with:

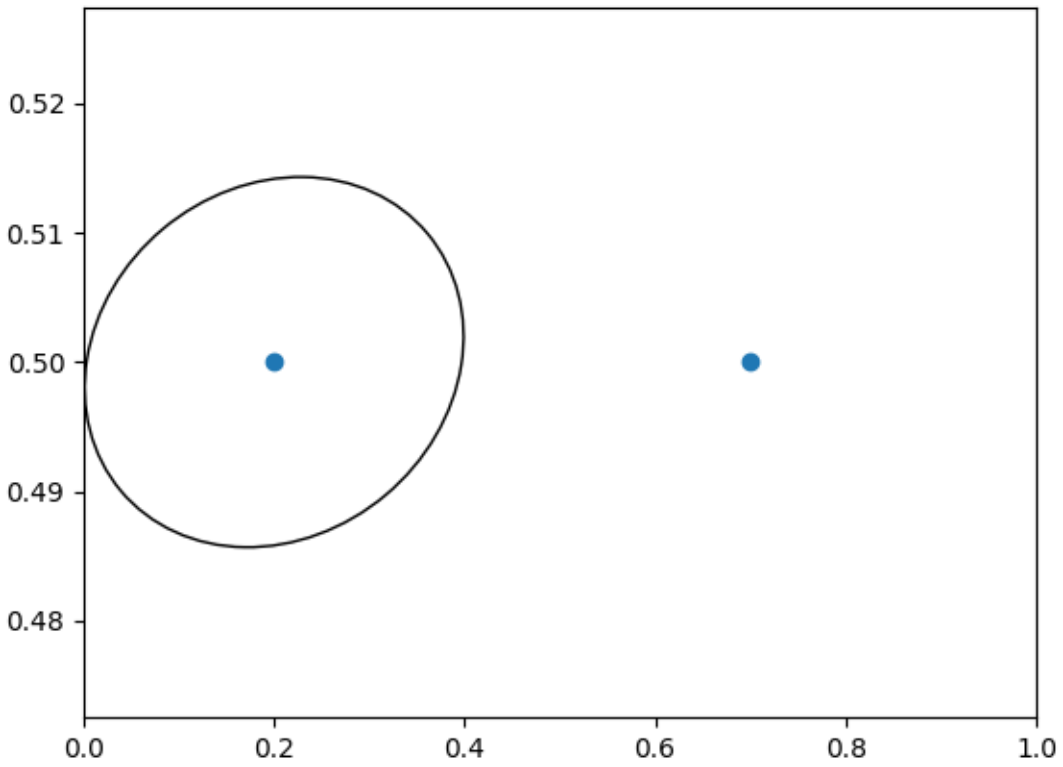
```
trans = ScaledTranslation(xt, yt, scale_trans)
```

where *xt* and *yt* are the translation offsets, and *scale_trans* is a transformation which scales *xt* and *yt* at transformation time before applying the offsets.

Note the use of the plus operator on the transforms below. This code says: first apply the scale transformation `fig.dpi_scale_trans` to make the ellipse the proper size, but still centered at (0, 0), and then translate the data to `xdata[0]` and `ydata[0]` in data space.

In interactive use, the ellipse stays the same size even if the axes limits are changed via zoom.

```
fig, ax = plt.subplots()
xdata, ydata = (0.2, 0.7), (0.5, 0.5)
ax.plot(xdata, ydata, "o")
ax.set_xlim((0, 1))

trans = (fig.dpi_scale_trans +
         transforms.ScaledTranslation(xdata[0], ydata[0], ax.transData))

# plot an ellipse around the point that is 150 x 130 points in diameter...
circle = mpatches.Ellipse((0, 0), 150/72, 130/72, angle=40,
                          fill=None, transform=trans)
ax.add_patch(circle)
plt.show()
```




**Note:** The order of transformation matters. Here the ellipse is given the right dimensions in display space *first* and then moved in data space to the correct spot. If we had done the `ScaledTranslation` first, then `xdata[0]` and `ydata[0]` would first be transformed to *display* coordinates (`[ 358.4 475.2]` on a 200-dpi monitor) and then those coordinates would be scaled by `fig.dpi_scale_trans` pushing the center of the ellipse well off the screen (i.e. `[ 71680. 95040.]`).

### Using offset transforms to create a shadow effect

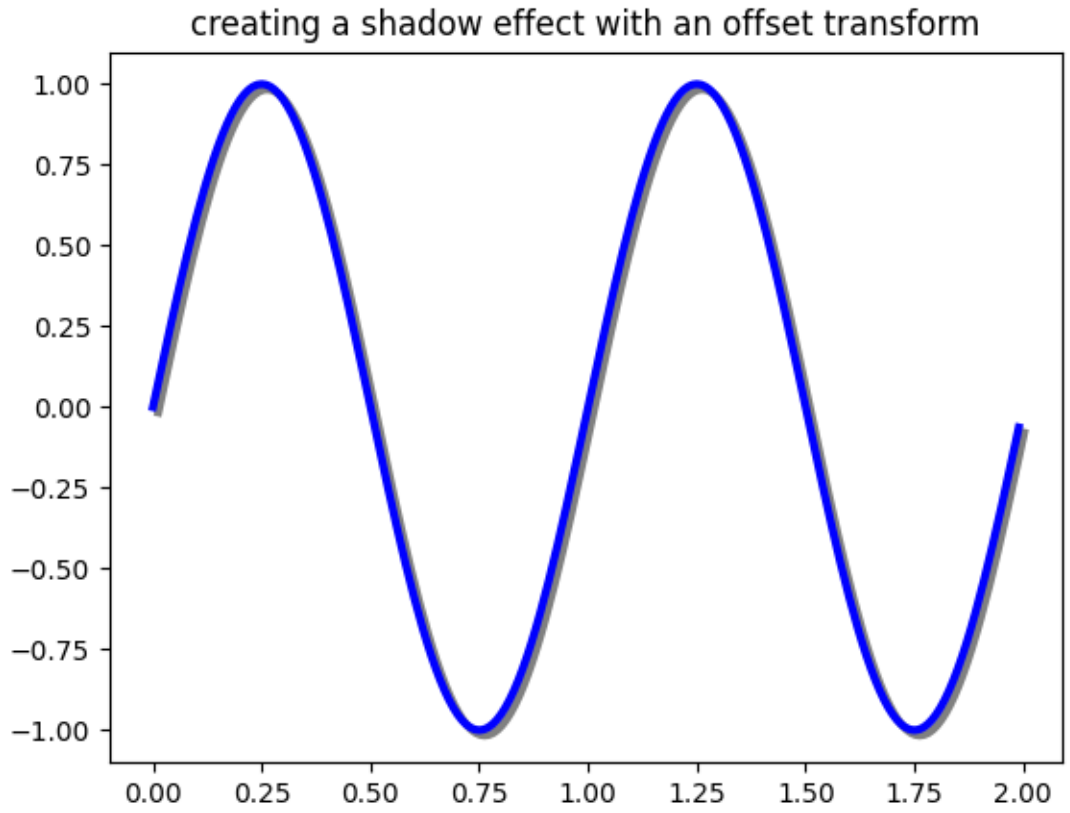
Another use of *`ScaledTranslation`* is to create a new transformation that is offset from another transformation, e.g., to place one object shifted a bit relative to another object. Typically, you want the shift to be in some physical dimension, like points or inches rather than in *data* coordinates, so that the shift effect is constant at different zoom levels and dpi settings.

One use for an offset is to create a shadow effect, where you draw one object identical to the first just to the right of it, and just below it, adjusting the zorder to make sure the shadow is drawn first and then the object it is shadowing above it.

Here we apply the transforms in the *opposite* order to the use of *`ScaledTranslation`* above. The plot is first made in data coordinates (`ax.transData`) and then shifted by `dx` and `dy` points using `fig.dpi_scale_trans`. (In typography, a point is 1/72 inches, and by specifying your offsets in points, your

figure will look the same regardless of the dpi resolution it is saved in.)

```
fig, ax = plt.subplots()

# make a simple sine wave
x = np.arange(0., 2., 0.01)
y = np.sin(2*np.pi*x)
line, = ax.plot(x, y, lw=3, color='blue')

# shift the object over 2 points, and down 2 points
dx, dy = 2/72., -2/72.
offset = transforms.ScaledTranslation(dx, dy, fig.dpi_scale_trans)
shadow_transform = ax.transData + offset

# now plot the same data with our offset transform;
# use the zorder to make sure we are below the line
ax.plot(x, y, lw=3, color='gray',
        transform=shadow_transform,
        zorder=0.5*line.get_zorder())

ax.set_title('creating a shadow effect with an offset transform')
plt.show()
```



**Note:** The dpi and inches offset is a common-enough use case that we have a special helper function to create it in *matplotlib.transforms.offset_copy()*, which returns a new transform with an added offset. So above we could have done:

```
shadow_transform = transforms.offset_copy(ax.transData,
         fig, dx, dy, units='inches')
```

### The transformation pipeline

The `ax.transData` transform we have been working with in this tutorial is a composite of three different transformations that comprise the transformation pipeline from *data* -> *display* coordinates. Michael Droettboom implemented the transformations framework, taking care to provide a clean API that segregated the nonlinear projections and scales that happen in polar and logarithmic plots, from the linear affine transformations that happen when you pan and zoom. There is an efficiency here, because you can pan and zoom in your Axes which affects the affine transformation, but you may not need to compute the potentially expensive nonlinear scales or projections on simple navigation events. It is also possible to multiply affine transformation matrices together, and then apply them to coordinates in one step. This is not true of all possible transformations.

Here is how the `ax.transData` instance is defined in the basic separable axis *Axes* class:

```
self.transData = self.transScale + (self.transLimits + self.transAxes)
```

We've been introduced to the `transAxes` instance above in *Axes coordinates*, which maps the (0, 0), (1, 1) corners of the Axes or subplot bounding box to *display* space, so let's look at these other two pieces.

`self.transLimits` is the transformation that takes you from *data* to *axes* coordinates; i.e., it maps your view xlim and ylim to the unit space of the Axes (and `transAxes` then takes that unit space to display space). We can see this in action here

```
In [80]: ax = plt.subplot()

In [81]: ax.set_xlim(0, 10)
Out[81]: (0, 10)

In [82]: ax.set_ylim(-1, 1)
Out[82]: (-1, 1)

In [84]: ax.transLimits.transform((0, -1))
Out[84]: array([ 0.,  0.])

In [85]: ax.transLimits.transform((10, -1))
Out[85]: array([ 1.,  0.])

In [86]: ax.transLimits.transform((10, 1))
Out[86]: array([ 1.,  1.])
```

(continues on next page)

(continued from previous page)

```
In [87]: ax.transLimits.transform((5, 0))
Out[87]: array([ 0.5,  0.5])
```

and we can use this same inverted transformation to go from the unit *axes* coordinates back to *data* coordinates.

```
In [90]: inv.transform((0.25, 0.25))
Out[90]: array([ 2.5, -0.5])
```

The final piece is the `self.transScale` attribute, which is responsible for the optional non-linear scaling of the data, e.g., for logarithmic axes. When an Axes is initially setup, this is just set to the identity transform, since the basic Matplotlib axes has linear scale, but when you call a logarithmic scaling function like *`semilogx()`* or explicitly set the scale to logarithmic with *`set_xscale()`*, then the `ax.transScale` attribute is set to handle the nonlinear projection. The scales transforms are properties of the respective `xaxis` and `yaxis` *`Axis`* instances. For example, when you call `ax.set_xscale('log')`, the xaxis updates its scale to a *`matplotlib.scale.LogScale`* instance.

For non-separable axes the PolarAxes, there is one more piece to consider, the projection transformation. The `transData` *`matplotlib.projections.polar.PolarAxes`* is similar to that for the typical separable matplotlib Axes, with one additional piece `transProjection`:

```
self.transData = (
    self.transScale + self.transShift + self.transProjection +
    (self.transProjectionAffine + self.transWedge + self.transAxes))
```

`transProjection` handles the projection from the space, e.g., latitude and longitude for map data, or radius and theta for polar data, to a separable Cartesian coordinate system. There are several projection examples in the *`matplotlib.projections`* package, and the best way to learn more is to open the source for those packages and see how to make your own, since Matplotlib supports extensible axes and projections. Michael Droettboom has provided a nice tutorial example of creating a Hammer projection axes; see *Custom projection*.

**Total running time of the script:** (0 minutes 3.404 seconds)

## 3.5 Customizing Matplotlib with style sheets and rcParams

Tips for customizing the properties and default styles of Matplotlib.

There are three ways to customize Matplotlib:

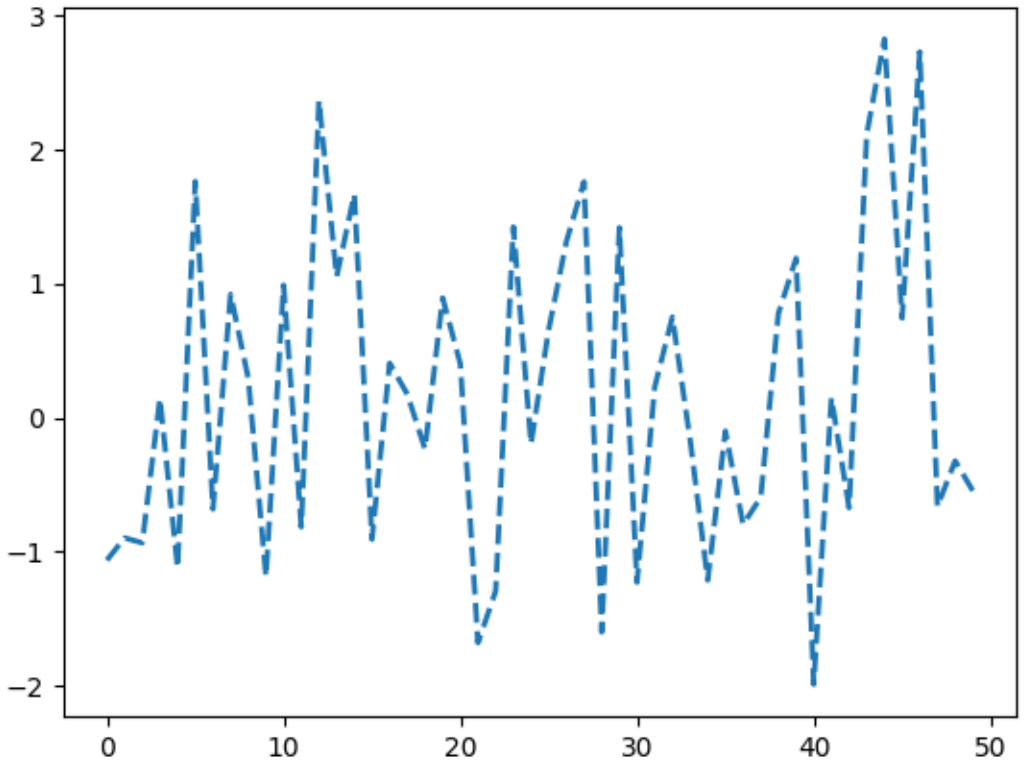
1. *Setting rcParams at runtime*.
2. *Using style sheets*.
3. *Changing your matplotlibrc file*.

Setting rcParams at runtime takes precedence over style sheets, style sheets take precedence over `matplotlibrc` files.

### 3.5.1 Runtime rc settings

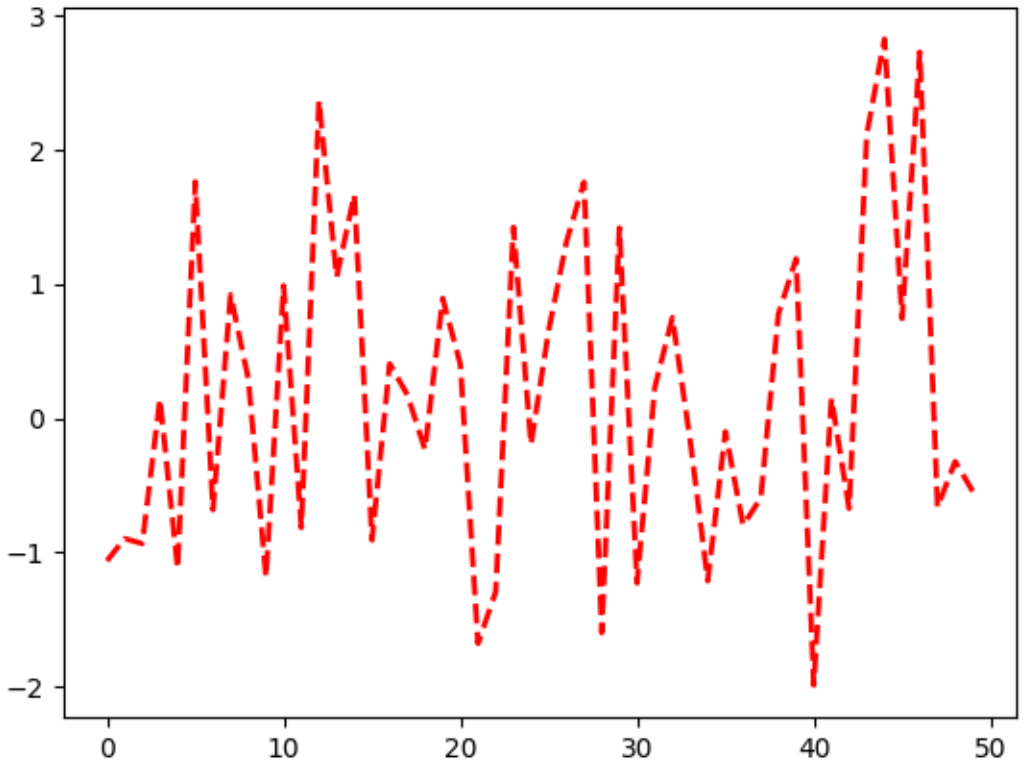
You can dynamically change the default rc (runtime configuration) settings in a python script or interactively from the python shell. All rc settings are stored in a dictionary-like variable called *matplotlib.rcParams*, which is global to the matplotlib package. See *matplotlib.rcParams* for a full list of configurable rcParams. rcParams can be modified directly, for example:

```
from cycler import cycler

import matplotlib.pyplot as plt
import numpy as np

import matplotlib as mpl

mpl.rcParams['lines.linewidth'] = 2
mpl.rcParams['lines.linestyle'] = '--'
data = np.random.randn(50)
plt.plot(data)
```



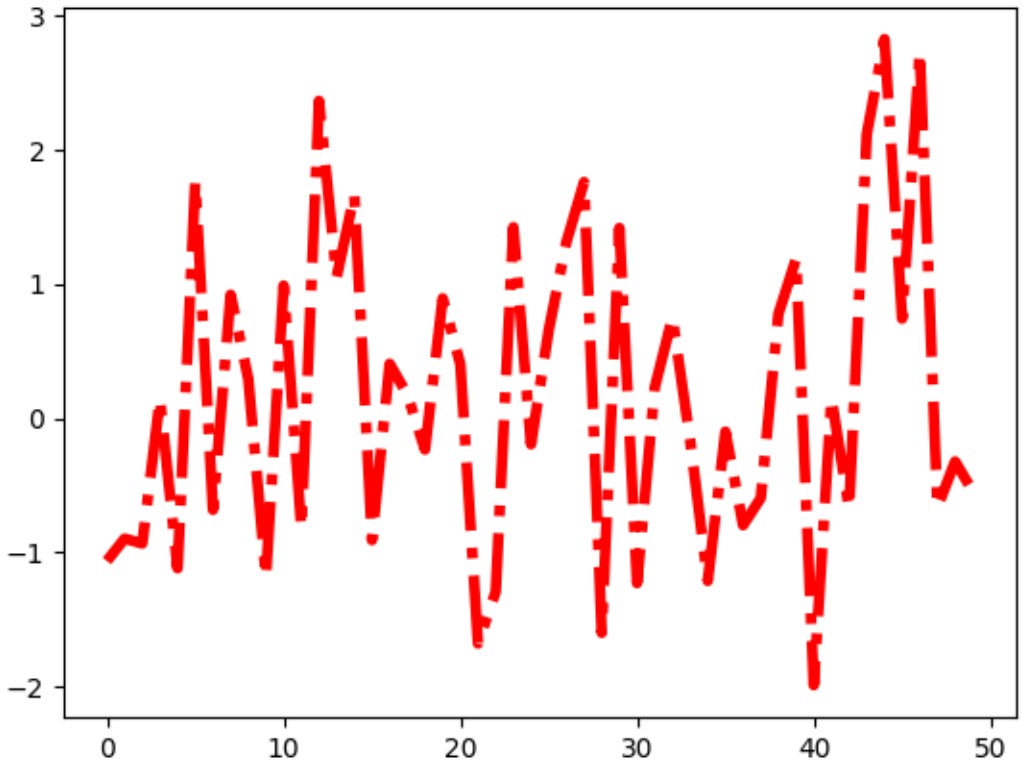

Note, that in order to change the usual *plot* color you have to change the *prop_cycle* property of *axes*:

```
mpl.rcParams['axes.prop_cycle'] = cycler(color=['r', 'g', 'b', 'y'])
plt.plot(data)  # first color is red
```




Matplotlib also provides a couple of convenience functions for modifying rc settings. *matplotlib.rc* can be used to modify multiple settings in a single group at once, using keyword arguments:

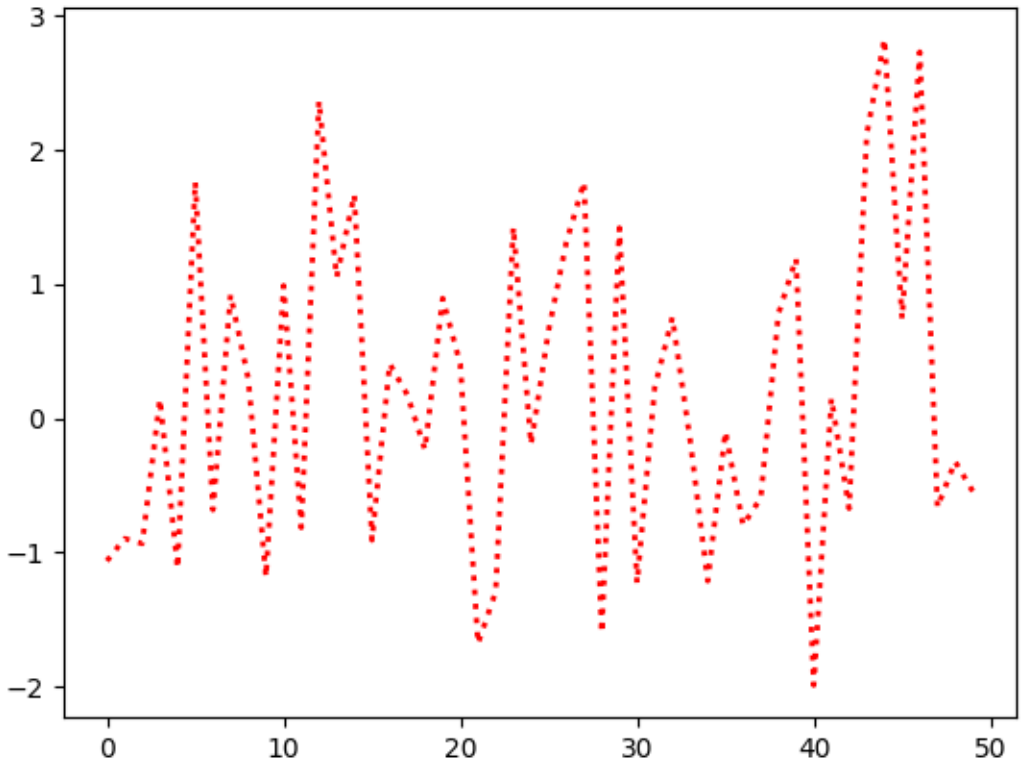
```
mpl.rc('lines', linewidth=4, linestyle='-.')
plt.plot(data)
```




### Temporary rc settings

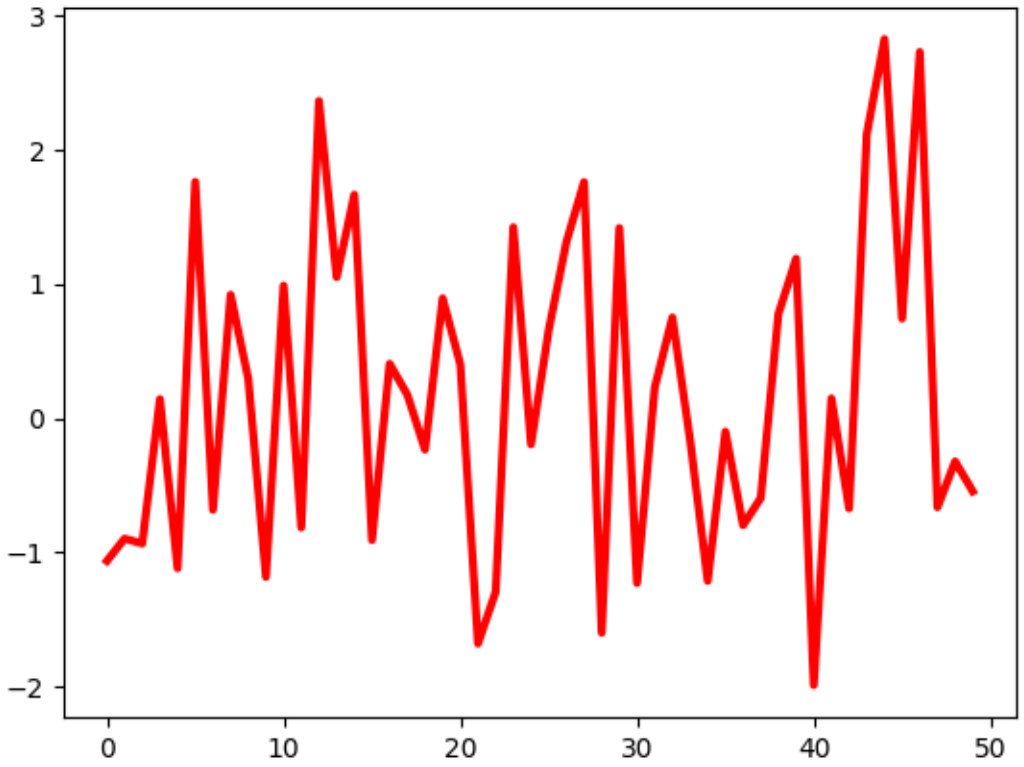
The *matplotlib.rcParams* object can also be changed temporarily using the *matplotlib.rc_context* context manager:

```
with mpl.rc_context({'lines.linewidth': 2, 'lines.linestyle': ':'}):
    plt.plot(data)
```




*matplotlib.rc_context* can also be used as a decorator to modify the defaults within a function:

```
@mpl.rc_context({'lines.linewidth': 3, 'lines.linestyle': '-'})
def plotting_function():
    plt.plot(data)

plotting_function()
```




*matplotlib.rcdefaults* will restore the standard Matplotlib default settings.

There is some degree of validation when setting the values of rcParams, see *matplotlib.rcsetup* for details.

### 3.5.2 Using style sheets

Another way to change the visual appearance of plots is to set the rcParams in a so-called style sheet and import that style sheet with *matplotlib.style.use*. In this way you can switch easily between different styles by simply changing the imported style sheet. A style sheets looks the same as a *matplotlibrc* file, but in a style sheet you can only set rcParams that are related to the actual style of a plot. Other rcParams, like *backend*, will be ignored. `matplotlibrc` files support all rcParams. The rationale behind this is to make style sheets portable between different machines without having to worry about dependencies which might or might not be installed on another machine. For a full list of rcParams see *matplotlib.rcParams*. For a list of rcParams that are ignored in style sheets see *matplotlib.style.use*.

There are a number of pre-defined styles *provided by Matplotlib*. For example, there's a pre-defined style called "ggplot", which emulates the aesthetics of ggplot (a popular plotting package for R). To use this style, add:

```
plt.style.use('ggplot')
```

To list all available styles, use:

```
print(plt.style.available)
```

```
['Solarize_Light2', '_classic_test_patch', '_mpl-gallery', '_mpl-gallery-
 ↪nogrid', 'bmh', 'classic', 'dark_background', 'fast', 'fivethirtyeight',
 ↪'ggplot', 'grayscale', 'seaborn-v0_8', 'seaborn-v0_8-bright', 'seaborn-v0_8-
 ↪colorblind', 'seaborn-v0_8-dark', 'seaborn-v0_8-dark-palette', 'seaborn-v0_
 ↪8-darkgrid', 'seaborn-v0_8-deep', 'seaborn-v0_8-muted', 'seaborn-v0_8-
 ↪notebook', 'seaborn-v0_8-paper', 'seaborn-v0_8-pastel', 'seaborn-v0_8-poster
 ↪', 'seaborn-v0_8-talk', 'seaborn-v0_8-ticks', 'seaborn-v0_8-white',
 ↪'seaborn-v0_8-whitegrid', 'tableau-colorblind10']
```

### Defining your own style

You can create custom styles and use them by calling *style.use* with the path or URL to the style sheet.

For example, you might want to create `./images/presentation.mplstyle` with the following:

```
axes.titlesize : 24
axes.labelsize : 20
lines.linewidth : 3
lines.markersize : 10
xtick.labelsize : 16
ytick.labelsize : 16
```

Then, when you want to adapt a plot designed for a paper to one that looks good in a presentation, you can just add:

```
>>> import matplotlib.pyplot as plt
>>> plt.style.use('./images/presentation.mplstyle')
```

### Distributing styles

You can include style sheets into standard importable Python packages (which can be e.g. distributed on PyPI). If your package is importable as `import mypackage`, with a `mypackage/__init__.py` module, and you add a `mypackage/presentation.mplstyle` style sheet, then it can be used as `plt.style.use("mypackage.presentation")`. Subpackages (e.g. `dotted.package.name`) are also supported.

Alternatively, you can make your style known to Matplotlib by placing your `<style-name>.mplstyle` file into `mpl_configdir/stylelib`. You can then load your custom style sheet with a call to `style.use(<style-name>)`. By default `mpl_configdir` should be `~/.config/matplotlib`, but you can check where yours is with *matplotlib.get_configdir()*; you may need to create this directory. You also can change the directory where Matplotlib looks for the stylelib/ folder by setting the *MPLCONFIGDIR* environment variable, see *matplotlib configuration and cache directory locations*.

Note that a custom style sheet in `mpl_configdir/stylelib` will override a style sheet defined by Matplotlib if the styles have the same name.

Once your `<style-name>.mplstyle` file is in the appropriate `mpl_configdir` you can specify your style with:

```
>>> import matplotlib.pyplot as plt
>>> plt.style.use(<style-name>)
```

### Composing styles

Style sheets are designed to be composed together. So you can have a style sheet that customizes colors and a separate style sheet that alters element sizes for presentations. These styles can easily be combined by passing a list of styles:

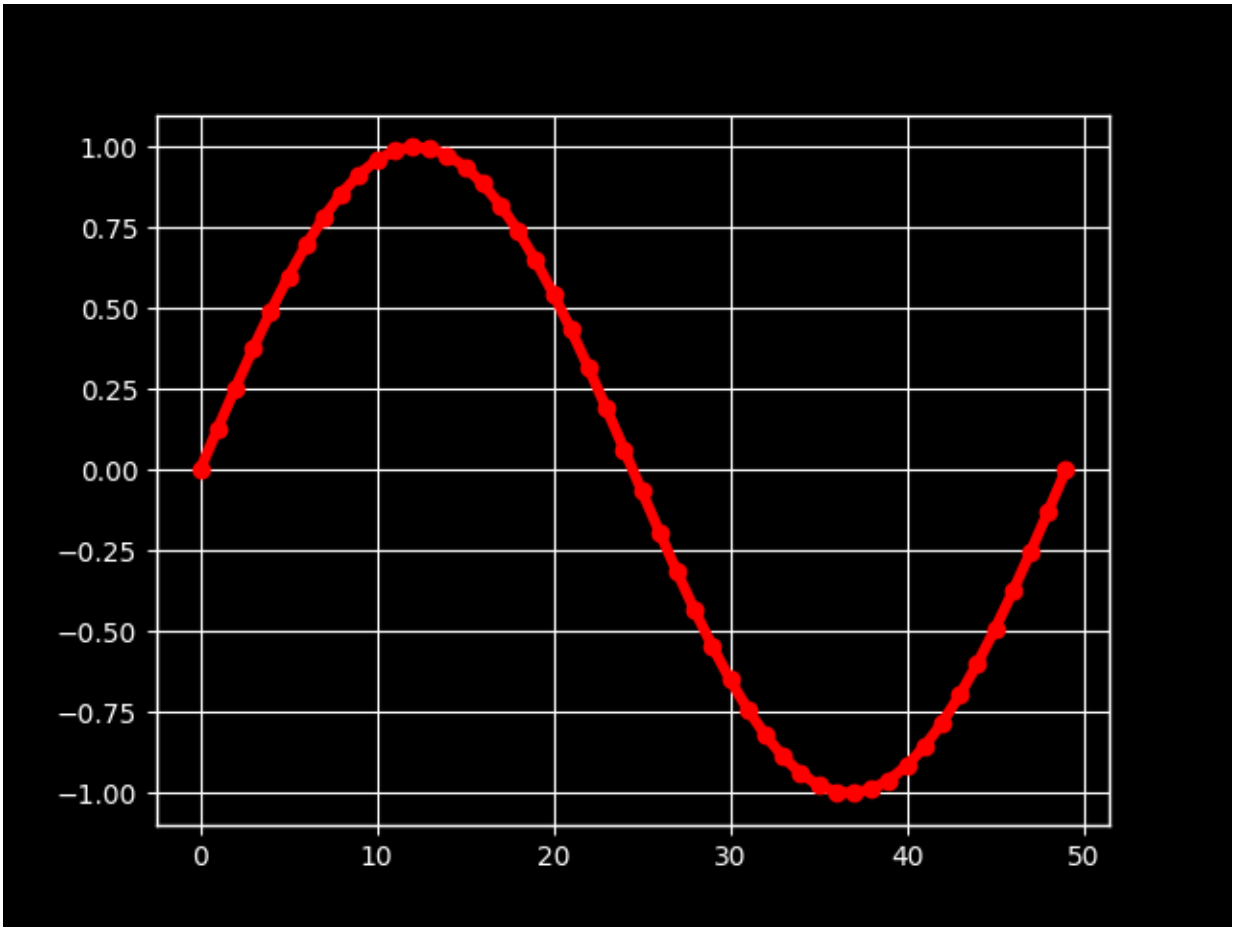
```
>>> import matplotlib.pyplot as plt
>>> plt.style.use(['dark_background', 'presentation'])
```

Note that styles further to the right will overwrite values that are already defined by styles on the left.

### Temporary styling

If you only want to use a style for a specific block of code but don't want to change the global styling, the style package provides a context manager for limiting your changes to a specific scope. To isolate your styling changes, you can write something like the following:

```
with plt.style.context('dark_background'):
    plt.plot(np.sin(np.linspace(0, 2 * np.pi)), 'r-o')
plt.show()
```




### 3.5.3 The `matplotlibrc` file

Matplotlib uses `matplotlibrc` configuration files to customize all kinds of properties, which we call 'rc settings' or 'rc parameters'. You can control the defaults of almost every property in Matplotlib: figure size and DPI, line width, color and style, Axes, axis and grid properties, text and font properties and so on. The `matplotlibrc` is read at startup to configure Matplotlib. Matplotlib looks for `matplotlibrc` in four locations, in the following order:

1. `matplotlibrc` in the current working directory, usually used for specific customizations that you do not want to apply elsewhere.
2. `$MATPLOTLIBRC` if it is a file, else `$MATPLOTLIBRC/matplotlibrc`.
3. It next looks in a user-specific place, depending on your platform:
   - On Linux and FreeBSD, it looks in `.config/matplotlib/matplotlibrc` (or `$XDG_CONFIG_HOME/matplotlib/matplotlibrc`) if you've customized your environment.
   - On other platforms, it looks in `.matplotlib/matplotlibrc`.

   See *matplotlib configuration and cache directory locations*.

4. *INSTALL*/matplotlib/mpl-data/matplotlibrc, where *INSTALL* is something like /usr/lib/python3.9/site-packages on Linux, and maybe C:\Python39\Lib\site-packages on Windows. Every time you install matplotlib, this file will be overwritten, so if you want your customizations to be saved, please move this file to your user-specific matplotlib directory.

Once a matplotlibrc file has been found, it will *not* search any of the other paths. When a *style sheet* is given with style.use('<path>/<style-name>.mplstyle'), settings specified in the style sheet take precedence over settings in the matplotlibrc file.

To display where the currently active matplotlibrc file was loaded from, one can do the following:

```
>>> import matplotlib
>>> matplotlib.matplotlib_fname()
'/home/foo/.config/matplotlib/matplotlibrc'
```

See below for a sample *matplotlibrc file* and see *matplotlib.rcParams* for a full list of configurable rcParams.

### The default matplotlibrc file

```
#### MATPLOTLIBRC FORMAT

## NOTE FOR END USERS: DO NOT EDIT THIS FILE!
##
## This is a sample Matplotlib configuration file - you can find a copy
## of it on your system in site-packages/matplotlib/mpl-data/matplotlibrc
## (relative to your Python installation location).
## DO NOT EDIT IT!
##
## If you wish to change your default style, copy this file to one of the
## following locations:
##     Unix/Linux:
##         $HOME/.config/matplotlib/matplotlibrc OR
##         $XDG_CONFIG_HOME/matplotlib/matplotlibrc (if $XDG_CONFIG_HOME is␣
 ↪set)
##     Other platforms:
##         $HOME/.matplotlib/matplotlibrc
## and edit that copy.
##
## See https://matplotlib.org/stable/users/explain/customizing.html
 ↪#customizing-with-matplotlibrc-files
## for more details on the paths which are checked for the configuration file.
##
## Blank lines, or lines starting with a comment symbol, are ignored, as are
## trailing comments.  Other lines must have the format:
##     key: val  # optional comment
##
## Formatting: Use PEP8-like style (as enforced in the rest of the codebase).
## All lines start with an additional '#', so that removing all leading '#'s
## yields a valid style file.
```

(continues on next page)

(continued from previous page)

```
##
## Colors: for the color values below, you can either use
##     - a Matplotlib color string, such as r, k, or b
##     - an RGB tuple, such as (1.0, 0.5, 0.0)
##     - a double-quoted hex string, such as "#ff00ff".
##       The unquoted string ff00ff is also supported for backward
##       compatibility, but is discouraged.
##     - a scalar grayscale intensity such as 0.75
##     - a legal html color name, e.g., red, blue, darkslategray
##
## String values may optionally be enclosed in double quotes, which allows
## using the comment character # in the string.
##
## This file (and other style files) must be encoded as utf-8.
##
## Matplotlib configuration are currently divided into following parts:
##     - BACKENDS
##     - LINES
##     - PATCHES
##     - HATCHES
##     - BOXPLOT
##     - FONT
##     - TEXT
##     - LaTeX
##     - AXES
##     - DATES
##     - TICKS
##     - GRIDS
##     - LEGEND
##     - FIGURE
##     - IMAGES
##     - CONTOUR PLOTS
##     - ERRORBAR PLOTS
##     - HISTOGRAM PLOTS
##     - SCATTER PLOTS
##     - AGG RENDERING
##     - PATHS
##     - SAVING FIGURES
##     - INTERACTIVE KEYMAPS
##     - ANIMATION

##### CONFIGURATION BEGINS HERE


## ***************************************************************************
## * BACKENDS                                                                *
## ***************************************************************************
## The default backend.  If you omit this parameter, the first working
## backend from the following list is used:
##     MacOSX QtAgg Gtk4Agg Gtk3Agg TkAgg WxAgg Agg
## Other choices include:
##     QtCairo GTK4Cairo GTK3Cairo TkCairo WxCairo Cairo
```

(continues on next page)

(continued from previous page)

```
##     Qt5Agg Qt5Cairo Wx  # deprecated.
##     PS PDF SVG Template
## You can also deploy your own backend outside of Matplotlib by referring to
## the module name (which must be in the PYTHONPATH) as 'module://my_backend'.
##backend: Agg

## The port to use for the web server in the WebAgg backend.
#webagg.port: 8988

## The address on which the WebAgg web server should be reachable
#webagg.address: 127.0.0.1

## If webagg.port is unavailable, a number of other random ports will
## be tried until one that is available is found.
#webagg.port_retries: 50

## When True, open the web browser to the plot that is shown
#webagg.open_in_browser: True

## If you are running pyplot inside a GUI and your backend choice
## conflicts, we will automatically try to find a compatible one for
## you if backend_fallback is True
#backend_fallback: True

#interactive: False
#figure.hooks:          # list of dotted.module.name:dotted.callable.name
#toolbar:     toolbar2  # {None, toolbar2, toolmanager}
#timezone:    UTC       # a pytz timezone string, e.g., US/Central or Europe/
 ↪Paris


## ***************************************************************************
## * LINES                                                                   *
## ***************************************************************************
## See https://matplotlib.org/stable/api/artist_api.html#module-matplotlib.
 ↪lines
## for more information on line properties.
#lines.linewidth: 1.5               # line width in points
#lines.linestyle: -                 # solid line
#lines.color:     C0                # has no affect on plot(); see axes.prop_
 ↪cycle
#lines.marker:          None        # the default marker
#lines.markerfacecolor: auto        # the default marker face color
#lines.markeredgecolor: auto        # the default marker edge color
#lines.markeredgewidth: 1.0         # the line width around the marker symbol
#lines.markersize:      6           # marker size, in points
#lines.dash_joinstyle:  round       # {miter, round, bevel}
#lines.dash_capstyle:   butt        # {butt, round, projecting}
#lines.solid_joinstyle: round       # {miter, round, bevel}
#lines.solid_capstyle:  projecting  # {butt, round, projecting}
#lines.antialiased: True            # render lines in antialiased (no jaggies)
```

(continues on next page)

(continued from previous page)

```
## The three standard dash patterns.  These are scaled by the linewidth.
#lines.dashed_pattern: 3.7, 1.6
#lines.dashdot_pattern: 6.4, 1.6, 1, 1.6
#lines.dotted_pattern: 1, 1.65
#lines.scale_dashes: True

#markers.fillstyle: full  # {full, left, right, bottom, top, none}

#pcolor.shading: auto
#pcolormesh.snap: True  # Whether to snap the mesh to pixel boundaries. This␣
 ↪is
                        # provided solely to allow old test images to remain
                        # unchanged. Set to False to obtain the previous␣
 ↪behavior.

## ***************************************************************************
## * PATCHES                                                                 *
## ***************************************************************************
## Patches are graphical objects that fill 2D space, like polygons or circles.
## See https://matplotlib.org/stable/api/artist_api.html#module-matplotlib.
 ↪patches
## for more information on patch properties.
#patch.linewidth:       1.0     # edge width in points.
#patch.facecolor:       C0
#patch.edgecolor:       black   # if forced, or patch is not filled
#patch.force_edgecolor: False   # True to always use edgecolor
#patch.antialiased:     True    # render patches in antialiased (no jaggies)


## ***************************************************************************
## * HATCHES                                                                 *
## ***************************************************************************
#hatch.color:     black
#hatch.linewidth: 1.0


## ***************************************************************************
## * BOXPLOT                                                                 *
## ***************************************************************************
#boxplot.notch:       False
#boxplot.vertical:    True
#boxplot.whiskers:    1.5
#boxplot.bootstrap:   None
#boxplot.patchartist: False
#boxplot.showmeans:   False
#boxplot.showcaps:    True
#boxplot.showbox:     True
#boxplot.showfliers:  True
#boxplot.meanline:    False

#boxplot.flierprops.color:           black
#boxplot.flierprops.marker:          o
```

(continues on next page)

(continued from previous page)

```
#boxplot.flierprops.markerfacecolor: none
#boxplot.flierprops.markeredgecolor: black
#boxplot.flierprops.markeredgewidth: 1.0
#boxplot.flierprops.markersize:      6
#boxplot.flierprops.linestyle:       none
#boxplot.flierprops.linewidth:       1.0

#boxplot.boxprops.color:     black
#boxplot.boxprops.linewidth: 1.0
#boxplot.boxprops.linestyle: -

#boxplot.whiskerprops.color:     black
#boxplot.whiskerprops.linewidth: 1.0
#boxplot.whiskerprops.linestyle: -

#boxplot.capprops.color:     black
#boxplot.capprops.linewidth: 1.0
#boxplot.capprops.linestyle: -

#boxplot.medianprops.color:     C1
#boxplot.medianprops.linewidth: 1.0
#boxplot.medianprops.linestyle: -

#boxplot.meanprops.color:           C2
#boxplot.meanprops.marker:          ^
#boxplot.meanprops.markerfacecolor: C2
#boxplot.meanprops.markeredgecolor: C2
#boxplot.meanprops.markersize:       6
#boxplot.meanprops.linestyle:       --
#boxplot.meanprops.linewidth:       1.0


## ***************************************************************************
## * FONT                                                                    *
## ***************************************************************************
## The font properties used by `text.Text`.
## See https://matplotlib.org/stable/api/font_manager_api.html for more␣
 ↪information
## on font properties.  The 6 font properties used for font matching are
## given below with their default values.
##
## The font.family property can take either a single or multiple entries of␣
 ↪any
## combination of concrete font names (not supported when rendering text with
## usetex) or the following five generic values:
##     - 'serif' (e.g., Times),
##     - 'sans-serif' (e.g., Helvetica),
##     - 'cursive' (e.g., Zapf-Chancery),
##     - 'fantasy' (e.g., Western), and
##     - 'monospace' (e.g., Courier).
## Each of these values has a corresponding default list of font names
## (font.serif, etc.); the first available font in the list is used.  Note␣
```

(continues on next page)

(continued from previous page)

```
 ↪that
## for font.serif, font.sans-serif, and font.monospace, the first element of
## the list (a DejaVu font) will always be used because DejaVu is shipped with
## Matplotlib and is thus guaranteed to be available; the other entries are
## left as examples of other possible values.
##
## The font.style property has three values: normal (or roman), italic
## or oblique.  The oblique style will be used for italic, if it is not
## present.
##
## The font.variant property has two values: normal or small-caps.  For
## TrueType fonts, which are scalable fonts, small-caps is equivalent
## to using a font size of 'smaller', or about 83 % of the current font
## size.
##
## The font.weight property has effectively 13 values: normal, bold,
## bolder, lighter, 100, 200, 300, ..., 900.  Normal is the same as
## 400, and bold is 700.  bolder and lighter are relative values with
## respect to the current weight.
##
## The font.stretch property has 11 values: ultra-condensed,
## extra-condensed, condensed, semi-condensed, normal, semi-expanded,
## expanded, extra-expanded, ultra-expanded, wider, and narrower.  This
## property is not currently implemented.
##
## The font.size property is the default font size for text, given in points.
## 10 pt is the standard value.
##
## Note that font.size controls default text sizes.  To configure
## special text sizes tick labels, axes, labels, title, etc., see the rc
## settings for axes and ticks.  Special text sizes can be defined
## relative to font.size, using the following values: xx-small, x-small,
## small, medium, large, x-large, xx-large, larger, or smaller

#font.family:  sans-serif
#font.style:   normal
#font.variant: normal
#font.weight:  normal
#font.stretch: normal
#font.size:    10.0

#font.serif:      DejaVu Serif, Bitstream Vera Serif, Computer Modern Roman,␣
 ↪New Century Schoolbook, Century Schoolbook L, Utopia, ITC Bookman, Bookman,␣
 ↪Nimbus Roman No9 L, Times New Roman, Times, Palatino, Charter, serif
#font.sans-serif: DejaVu Sans, Bitstream Vera Sans, Computer Modern Sans␣
 ↪Serif, Lucida Grande, Verdana, Geneva, Lucid, Arial, Helvetica, Avant Garde,
 ↪ sans-serif
#font.cursive:    Apple Chancery, Textile, Zapf Chancery, Sand, Script MT,␣
 ↪Felipa, Comic Neue, Comic Sans MS, cursive
#font.fantasy:    Chicago, Charcoal, Impact, Western, xkcd script, fantasy
#font.monospace:  DejaVu Sans Mono, Bitstream Vera Sans Mono, Computer Modern␣
 ↪Typewriter, Andale Mono, Nimbus Mono L, Courier New, Courier, Fixed,␣
```

(continues on next page)

(continued from previous page)

```
 ↪Terminal, monospace


## ***************************************************************************
## * TEXT                                                                    *
## ***************************************************************************
## The text properties used by `text.Text`.
## See https://matplotlib.org/stable/api/artist_api.html#module-matplotlib.
 ↪text
## for more information on text properties
#text.color: black

## FreeType hinting flag ("foo" corresponds to FT_LOAD_FOO); may be one of the
## following (Proprietary Matplotlib-specific synonyms are given in␣
 ↪parentheses,
## but their use is discouraged):
## - default: Use the font's native hinter if possible, else FreeType's auto-
 ↪hinter.
##            ("either" is a synonym).
## - no_autohint: Use the font's native hinter if possible, else don't hint.
##                ("native" is a synonym.)
## - force_autohint: Use FreeType's auto-hinter.  ("auto" is a synonym.)
## - no_hinting: Disable hinting.  ("none" is a synonym.)
#text.hinting: force_autohint

#text.hinting_factor: 8  # Specifies the amount of softness for hinting in the
                         # horizontal direction.  A value of 1 will hint to␣
 ↪full
                         # pixels.  A value of 2 will hint to half pixels etc.
#text.kerning_factor: 0  # Specifies the scaling factor for kerning values. ␣
 ↪This
                         # is provided solely to allow old test images to␣
 ↪remain
                         # unchanged.  Set to 6 to obtain previous behavior.
                         # Values  other than 0 or 6 have no defined meaning.
#text.antialiased: True  # If True (default), the text will be antialiased.
                         # This only affects raster outputs.
#text.parse_math: True  # Use mathtext if there is an even number of unescaped
                        # dollar signs.


## ***************************************************************************
## * LaTeX                                                                   *
## ***************************************************************************
## For more information on LaTeX properties, see
## https://matplotlib.org/stable/users/explain/text/usetex.html
#text.usetex: False  # use latex for all text handling. The following fonts
                     # are supported through the usual rc parameter settings:
                     # new century schoolbook, bookman, times, palatino,
                     # zapf chancery, charter, serif, sans-serif, helvetica,
                     # avant garde, courier, monospace, computer modern roman,
                     # computer modern sans serif, computer modern typewriter
```

(continues on next page)

(continued from previous page)

```
#text.latex.preamble:    # IMPROPER USE OF THIS FEATURE WILL LEAD TO LATEX␣
 ↪FAILURES
                         # AND IS THEREFORE UNSUPPORTED. PLEASE DO NOT ASK FOR␣
 ↪HELP
                         # IF THIS FEATURE DOES NOT DO WHAT YOU EXPECT IT TO.
                         # text.latex.preamble is a single line of LaTeX code␣
 ↪that
                         # will be passed on to the LaTeX system. It may␣
 ↪contain
                         # any code that is valid for the LaTeX "preamble", i.
 ↪e.
                         # between the "\documentclass" and "\begin{document}"
                         # statements.
                         # Note that it has to be put on a single line, which␣
 ↪may
                         # become quite long.
                         # The following packages are always loaded with␣
 ↪usetex,
                         # so beware of package collisions:
                         #   geometry, inputenc, type1cm.
                         # PostScript (PSNFSS) font packages may also be
                         # loaded, depending on your font settings.

## The following settings allow you to select the fonts in math mode.
#mathtext.fontset: dejavusans  # Should be 'dejavusans' (default),
                               # 'dejavuserif', 'cm' (Computer Modern), 'stix
 ↪',
                               # 'stixsans' or 'custom'
## "mathtext.fontset: custom" is defined by the mathtext.bf, .cal, .it, ...
## settings which map a TeX font name to a fontconfig font pattern.  (These
## settings are not used for other font sets.)
#mathtext.bf:  sans:bold
#mathtext.bfit: sans:italic:bold
#mathtext.cal: cursive
#mathtext.it:  sans:italic
#mathtext.rm:  sans
#mathtext.sf:  sans
#mathtext.tt:  monospace
#mathtext.fallback: cm  # Select fallback font from ['cm' (Computer Modern),
 ↪'stix'
                        # 'stixsans'] when a symbol cannot be found in one of␣
 ↪the
                        # custom math fonts. Select 'None' to not perform␣
 ↪fallback
                        # and replace the missing character by a dummy symbol.
#mathtext.default: it  # The default font to use for math.
                       # Can be any of the LaTeX font names, including
                       # the special name "regular" for the same font
                       # used in regular text.


## ***************************************************************************
```

(continues on next page)

(continued from previous page)

```
## * AXES                                                                     *
## ****************************************************************************
## Following are default face and edge colors, default tick sizes,
## default font sizes for tick labels, and so on.  See
## https://matplotlib.org/stable/api/axes_api.html#module-matplotlib.axes
#axes.facecolor:     white   # axes background color
#axes.edgecolor:     black   # axes edge color
#axes.linewidth:     0.8     # edge line width
#axes.grid:          False   # display grid or not
#axes.grid.axis:     both    # which axis the grid should apply to
#axes.grid.which:    major   # grid lines at {major, minor, both} ticks
#axes.titlelocation: center  # alignment of the title: {left, right, center}
#axes.titlesize:     large   # font size of the axes title
#axes.titleweight:   normal  # font weight of title
#axes.titlecolor:    auto    # color of the axes title, auto falls back to
                             # text.color as default value
#axes.titley:        None    # position title (axes relative units).  None␣
 ↪implies auto
#axes.titlepad:      6.0     # pad between axes and title in points
#axes.labelsize:     medium  # font size of the x and y labels
#axes.labelpad:      4.0     # space between label and axis
#axes.labelweight:   normal  # weight of the x and y labels
#axes.labelcolor:    black
#axes.axisbelow:     line    # draw axis gridlines and ticks:
                             #     - below patches (True)
                             #     - above patches but below lines ('line')
                             #     - above all (False)

#axes.formatter.limits: -5, 6  # use scientific notation if log10
                               # of the axis range is smaller than the
                               # first or larger than the second
#axes.formatter.use_locale: False  # When True, format tick labels
                                   # according to the user's locale.
                                   # For example, use ',' as a decimal
                                   # separator in the fr_FR locale.
#axes.formatter.use_mathtext: False  # When True, use mathtext for scientific
                                     # notation.
#axes.formatter.min_exponent: 0  # minimum exponent to format in scientific␣
 ↪notation
#axes.formatter.useoffset: True  # If True, the tick label formatter
                                 # will default to labeling ticks relative
                                 # to an offset when the data range is
                                 # small compared to the minimum absolute
                                 # value of the data.
#axes.formatter.offset_threshold: 4  # When useoffset is True, the offset
                                     # will be used when it can remove
                                     # at least this number of significant
                                     # digits from tick labels.

#axes.spines.left:   True  # display axis spines
#axes.spines.bottom: True
#axes.spines.top:    True
```

(continues on next page)

(continued from previous page)

```
#axes.spines.right:  True

#axes.unicode_minus: True  # use Unicode for the minus symbol rather than␣
↪hyphen.  See
                           # https://en.wikipedia.org/wiki/Plus_and_minus_
↪signs#Character_codes
#axes.prop_cycle: cycler('color', ['1f77b4', 'ff7f0e', '2ca02c', 'd62728',
↪'9467bd', '8c564b', 'e377c2', '7f7f7f', 'bcbd22', '17becf'])
                  # color cycle for plot lines as list of string color specs:
                  # single letter, long name, or web-style hex
                  # As opposed to all other parameters in this file, the color
                  # values must be enclosed in quotes for this parameter,
                  # e.g. '1f77b4', instead of 1f77b4.
                  # See also https://matplotlib.org/stable/users/explain/
↪artists/color_cycle.html
                  # for more details on prop_cycle usage.
#axes.xmargin:   .05  # x margin.  See `axes.Axes.margins`
#axes.ymargin:   .05  # y margin.  See `axes.Axes.margins`
#axes.zmargin:   .05  # z margin.  See `axes.Axes.margins`
#axes.autolimit_mode: data  # If "data", use axes.xmargin and axes.ymargin as␣
↪is.
                           # If "round_numbers", after application of␣
↪margins, axis
                           # limits are further expanded to the nearest
↪"round" number.
#polaraxes.grid:    True   # display grid on polar axes
#axes3d.grid:       True   # display grid on 3D axes
#axes3d.automargin: False  # automatically add margin when manually setting␣
↪3D axis limits

#axes3d.xaxis.panecolor:    (0.95, 0.95, 0.95, 0.5)  # background pane on 3D␣
↪axes
#axes3d.yaxis.panecolor:    (0.90, 0.90, 0.90, 0.5)  # background pane on 3D␣
↪axes
#axes3d.zaxis.panecolor:    (0.925, 0.925, 0.925, 0.5)  # background pane on␣
↪3D axes

## ***************************************************************************
## * AXIS                                                                    *
## ***************************************************************************
#xaxis.labellocation: center  # alignment of the xaxis label: {left, right,␣
↪center}
#yaxis.labellocation: center  # alignment of the yaxis label: {bottom, top,␣
↪center}


## ***************************************************************************
## * DATES                                                                   *
## ***************************************************************************
## These control the default format strings used in AutoDateFormatter.
## Any valid format datetime format string can be used (see the python
## `datetime` for details).  For example, by using:
```

(continues on next page)

(continued from previous page)

```
##     - '%x' will use the locale date representation
##     - '%X' will use the locale time representation
##     - '%c' will use the full locale datetime representation
## These values map to the scales:
##     {'year': 365, 'month': 30, 'day': 1, 'hour': 1/24, 'minute': 1 / (24 *␣
↪60)}

#date.autoformatter.year:        %Y
#date.autoformatter.month:       %Y-%m
#date.autoformatter.day:         %Y-%m-%d
#date.autoformatter.hour:        %m-%d %H
#date.autoformatter.minute:      %d %H:%M
#date.autoformatter.second:      %H:%M:%S
#date.autoformatter.microsecond: %M:%S.%f
## The reference date for Matplotlib's internal date representation
## See https://matplotlib.org/stable/gallery/ticks/date_precision_and_epochs.
↪html
#date.epoch: 1970-01-01T00:00:00
## 'auto', 'concise':
#date.converter:                  auto
## For auto converter whether to use interval_multiples:
#date.interval_multiples:         True

## ***************************************************************************
## * TICKS                                                                   *
## ***************************************************************************
## See https://matplotlib.org/stable/api/axis_api.html#matplotlib.axis.Tick
#xtick.top:           False   # draw ticks on the top side
#xtick.bottom:        True    # draw ticks on the bottom side
#xtick.labeltop:      False   # draw label on the top
#xtick.labelbottom:   True    # draw label on the bottom
#xtick.major.size:    3.5     # major tick size in points
#xtick.minor.size:    2       # minor tick size in points
#xtick.major.width:   0.8     # major tick width in points
#xtick.minor.width:   0.6     # minor tick width in points
#xtick.major.pad:     3.5     # distance to major tick label in points
#xtick.minor.pad:     3.4     # distance to the minor tick label in points
#xtick.color:         black   # color of the ticks
#xtick.labelcolor:    inherit # color of the tick labels or inherit from␣
↪xtick.color
#xtick.labelsize:     medium  # font size of the tick labels
#xtick.direction:     out     # direction: {in, out, inout}
#xtick.minor.visible: False   # visibility of minor ticks on x-axis
#xtick.major.top:     True    # draw x axis top major ticks
#xtick.major.bottom:  True    # draw x axis bottom major ticks
#xtick.minor.top:     True    # draw x axis top minor ticks
#xtick.minor.bottom:  True    # draw x axis bottom minor ticks
#xtick.minor.ndivs:   auto    # number of minor ticks between the major ticks␣
↪on x-axis
#xtick.alignment:     center  # alignment of xticks

#ytick.left:          True    # draw ticks on the left side
```

(continues on next page)

(continued from previous page)

```
#ytick.right:         False   # draw ticks on the right side
#ytick.labelleft:     True    # draw tick labels on the left side
#ytick.labelright:    False   # draw tick labels on the right side
#ytick.major.size:    3.5     # major tick size in points
#ytick.minor.size:    2       # minor tick size in points
#ytick.major.width:   0.8     # major tick width in points
#ytick.minor.width:   0.6     # minor tick width in points
#ytick.major.pad:     3.5     # distance to major tick label in points
#ytick.minor.pad:     3.4     # distance to the minor tick label in points
#ytick.color:         black   # color of the ticks
#ytick.labelcolor:    inherit # color of the tick labels or inherit from␣
 ↪ytick.color
#ytick.labelsize:     medium  # font size of the tick labels
#ytick.direction:     out     # direction: {in, out, inout}
#ytick.minor.visible: False   # visibility of minor ticks on y-axis
#ytick.major.left:    True    # draw y axis left major ticks
#ytick.major.right:   True    # draw y axis right major ticks
#ytick.minor.left:    True    # draw y axis left minor ticks
#ytick.minor.right:   True    # draw y axis right minor ticks
#ytick.minor.ndivs:   auto    # number of minor ticks between the major ticks␣
 ↪on y-axis
#ytick.alignment:     center_baseline  # alignment of yticks


## ***************************************************************************
## * GRIDS                                                                   *
## ***************************************************************************
#grid.color:     "#b0b0b0"  # grid color
#grid.linestyle: -          # solid
#grid.linewidth: 0.8        # in points
#grid.alpha:     1.0        # transparency, between 0.0 and 1.0


## ***************************************************************************
## * LEGEND                                                                  *
## ***************************************************************************
#legend.loc:           best
#legend.frameon:       True     # if True, draw the legend on a background␣
 ↪patch
#legend.framealpha:    0.8      # legend patch transparency
#legend.facecolor:     inherit  # inherit from axes.facecolor; or color spec
#legend.edgecolor:     0.8      # background patch boundary color
#legend.fancybox:      True     # if True, use a rounded box for the
                                # legend background, else a rectangle
#legend.shadow:        False    # if True, give background a shadow effect
#legend.numpoints:     1        # the number of marker points in the legend␣
 ↪line
#legend.scatterpoints: 1        # number of scatter points
#legend.markerscale:   1.0      # the relative size of legend markers vs.␣
 ↪original
#legend.fontsize:      medium
#legend.labelcolor:    None
```

(continues on next page)

(continued from previous page)

```
#legend.title_fontsize: None    # None sets to the same as the default axes.

## Dimensions as fraction of font size:
#legend.borderpad:     0.4  # border whitespace
#legend.labelspacing:  0.5  # the vertical space between the legend entries
#legend.handlelength:  2.0  # the length of the legend lines
#legend.handleheight:  0.7  # the height of the legend handle
#legend.handletextpad: 0.8  # the space between the legend line and legend␣
↪text
#legend.borderaxespad: 0.5  # the border between the axes and legend edge
#legend.columnspacing: 2.0  # column separation


## ***************************************************************************
## * FIGURE                                                                  *
## ***************************************************************************
## See https://matplotlib.org/stable/api/figure_api.html#matplotlib.figure.
↪Figure
#figure.titlesize:   large     # size of the figure title (``Figure.
↪suptitle()``)
#figure.titleweight: normal    # weight of the figure title
#figure.labelsize:   large     # size of the figure label (``Figure.
↪sup[x|y]label()``)
#figure.labelweight: normal    # weight of the figure label
#figure.figsize:     6.4, 4.8  # figure size in inches
#figure.dpi:         100       # figure dots per inch
#figure.facecolor:   white     # figure face color
#figure.edgecolor:   white     # figure edge color
#figure.frameon:     True      # enable figure frame
#figure.max_open_warning: 20   # The maximum number of figures to open through
                               # the pyplot interface before emitting a␣
↪warning.
                               # If less than one this feature is disabled.
#figure.raise_window : True    # Raise the GUI window to front when show() is␣
↪called.

## The figure subplot parameters.  All dimensions are a fraction of the␣
↪figure width and height.
#figure.subplot.left:   0.125  # the left side of the subplots of the figure
#figure.subplot.right:  0.9    # the right side of the subplots of the figure
#figure.subplot.bottom: 0.11   # the bottom of the subplots of the figure
#figure.subplot.top:    0.88   # the top of the subplots of the figure
#figure.subplot.wspace: 0.2    # the amount of width reserved for space␣
↪between subplots,
                               # expressed as a fraction of the average axis␣
↪width
#figure.subplot.hspace: 0.2    # the amount of height reserved for space␣
↪between subplots,
                               # expressed as a fraction of the average axis␣
↪height

## Figure layout
```

(continues on next page)

(continued from previous page)

```
#figure.autolayout: False  # When True, automatically adjust subplot
                           # parameters to make the plot fit the figure
                           # using `tight_layout`
#figure.constrained_layout.use: False  # When True, automatically make plot
                                       # elements fit on the figure. (Not
                                       # compatible with `autolayout`, above).
## Padding (in inches) around axes; defaults to 3/72 inches, i.e. 3 points.
#figure.constrained_layout.h_pad:  0.04167
#figure.constrained_layout.w_pad:  0.04167
## Spacing between subplots, relative to the subplot sizes.  Much smaller␣
 ↪than for
## tight_layout (figure.subplot.hspace, figure.subplot.wspace) as constrained_
 ↪layout
## already takes surrounding texts (titles, labels, # ticklabels) into␣
 ↪account.
#figure.constrained_layout.hspace: 0.02
#figure.constrained_layout.wspace: 0.02


## ***************************************************************************
## * IMAGES                                                                  *
## ***************************************************************************
#image.aspect:          equal        # {equal, auto} or a number
#image.interpolation:   antialiased  # see help(imshow) for options
#image.interpolation_stage: data     # see help(imshow) for options
#image.cmap:            viridis      # A colormap name (plasma, magma, etc.)
#image.lut:             256          # the size of the colormap lookup table
#image.origin:          upper        # {lower, upper}
#image.resample:        True
#image.composite_image: True  # When True, all the images on a set of axes are
                              # combined into a single composite image before
                              # saving a figure as a vector graphics file,
                              # such as a PDF.


## ***************************************************************************
## * CONTOUR PLOTS                                                           *
## ***************************************************************************
#contour.negative_linestyle: dashed  # string or on-off ink sequence
#contour.corner_mask:        True    # {True, False}
#contour.linewidth:          None    # {float, None} Size of the contour line
                                     # widths. If set to None, it falls back␣
 ↪to
                                     # `line.linewidth`.
#contour.algorithm:          mpl2014 # {mpl2005, mpl2014, serial, threaded}


## ***************************************************************************
## * ERRORBAR PLOTS                                                          *
## ***************************************************************************
#errorbar.capsize: 0  # length of end cap on error bars in pixels
```

(continues on next page)

(continued from previous page)

```
## ***************************************************************************
## * HISTOGRAM PLOTS                                                         *
## ***************************************************************************
#hist.bins: 10  # The default number of histogram bins or 'auto'.


## ***************************************************************************
## * SCATTER PLOTS                                                           *
## ***************************************************************************
#scatter.marker: o         # The default marker type for scatter plots.
#scatter.edgecolors: face  # The default edge colors for scatter plots.


## ***************************************************************************
## * AGG RENDERING                                                           *
## ***************************************************************************
## Warning: experimental, 2008/10/10
#agg.path.chunksize: 0  # 0 to disable; values in the range
                        # 10000 to 100000 can improve speed slightly
                        # and prevent an Agg rendering failure
                        # when plotting very large data sets,
                        # especially if they are very gappy.
                        # It may cause minor artifacts, though.
                        # A value of 20000 is probably a good
                        # starting point.


## ***************************************************************************
## * PATHS                                                                   *
## ***************************************************************************
#path.simplify: True  # When True, simplify paths by removing "invisible"
                      # points to reduce file size and increase rendering
                      # speed
#path.simplify_threshold: 0.111111111111  # The threshold of similarity below
                                          # which vertices will be removed in
                                          # the simplification process.
#path.snap: True  # When True, rectilinear axis-aligned paths will be snapped
                  # to the nearest pixel when certain criteria are met.
                  # When False, paths will never be snapped.
#path.sketch: None  # May be None, or a tuple of the form:
                    #  path.sketch: (scale, length, randomness)
                    # - *scale* is the amplitude of the wiggle
                    #    perpendicular to the line (in pixels).
                    # - *length* is the length of the wiggle along the
                    #    line (in pixels).
                    # - *randomness* is the factor by which the length is
                    #    randomly scaled.
#path.effects:


## ***************************************************************************
```

(continues on next page)

(continued from previous page)

```
## * SAVING FIGURES                                                            *
## ****************************************************************************
## The default savefig parameters can be different from the display parameters
## e.g., you may want a higher resolution, or to make the figure
## background white
#savefig.dpi:       figure      # figure dots per inch or 'figure'
#savefig.facecolor: auto        # figure face color when saving
#savefig.edgecolor: auto        # figure edge color when saving
#savefig.format:    png         # {png, ps, pdf, svg}
#savefig.bbox:      standard    # {tight, standard}
                               # 'tight' is incompatible with generating␣
 ↪frames
                               # for animation
#savefig.pad_inches:  0.1       # padding to be used, when bbox is set to
 ↪'tight'
#savefig.directory:   ~         # default directory in savefig dialog, gets␣
 ↪updated after
                               # interactive saves, unless set to the empty␣
 ↪string (i.e.
                               # the current directory); use '.' to start at␣
 ↪the current
                               # directory but update after interactive saves
#savefig.transparent: False     # whether figures are saved with a transparent
                               # background by default
#savefig.orientation: portrait  # orientation of saved figure, for PostScript␣
 ↪output only

### macosx backend params
#macosx.window_mode : system   # How to open new figures (system, tab, window)
                              # system uses the MacOS system preferences

### tk backend params
#tk.window_focus:   False  # Maintain shell focus for TkAgg

### ps backend params
#ps.papersize:      letter  # {figure, letter, legal, ledger, A0-A10, B0-B10}
#ps.useafm:         False   # use AFM fonts, results in small files
#ps.usedistiller:   False   # {ghostscript, xpdf, None}
                           # Experimental: may produce smaller files.
                           # xpdf intended for production of publication␣
 ↪quality files,
                           # but requires ghostscript, xpdf and ps2eps
#ps.distiller.res:  6000    # dpi
#ps.fonttype:       3       # Output Type 3 (Type3) or Type 42 (TrueType)

### PDF backend params
#pdf.compression:    6  # integer from 0 to 9
                        # 0 disables compression (good for debugging)
#pdf.fonttype:       3  # Output Type 3 (Type3) or Type 42 (TrueType)
#pdf.use14corefonts: False
#pdf.inheritcolor:   False
```

(continues on next page)

(continued from previous page)

```
### SVG backend params
#svg.image_inline: True  # Write raster image data directly into the SVG file
#svg.fonttype: path      # How to handle SVG fonts:
                         #     path: Embed characters as paths -- supported
                         #           by most SVG renderers
                         #     None: Assume fonts are installed on the
                         #           machine where the SVG will be viewed.
#svg.hashsalt: None      # If not None, use this string as hash salt instead␣
 ↪of uuid4

### pgf parameter
## See https://matplotlib.org/stable/tutorials/text/pgf.html for more␣
 ↪information.
#pgf.rcfonts: True
#pgf.preamble:  # See text.latex.preamble for documentation
#pgf.texsystem: xelatex

### docstring params
#docstring.hardcopy: False  # set this when you want to generate hardcopy␣
 ↪docstring


## ***************************************************************************
## * INTERACTIVE KEYMAPS                                                     *
## ***************************************************************************
## Event keys to interact with figures/plots via keyboard.
## See https://matplotlib.org/stable/users/explain/interactive.html for more
## details on interactive navigation.  Customize these settings according to
## your needs. Leave the field(s) empty if you don't need a key-map. (i.e.,
## fullscreen : '')
#keymap.fullscreen: f, ctrl+f   # toggling
#keymap.home: h, r, home        # home or reset mnemonic
#keymap.back: left, c, backspace, MouseButton.BACK  # forward / backward keys
#keymap.forward: right, v, MouseButton.FORWARD      # for quick navigation
#keymap.pan: p                  # pan mnemonic
#keymap.zoom: o                 # zoom mnemonic
#keymap.save: s, ctrl+s         # saving current figure
#keymap.help: f1                # display help about active tools
#keymap.quit: ctrl+w, cmd+w, q  # close the current figure
#keymap.quit_all:               # close all figures
#keymap.grid: g                 # switching on/off major grids in current axes
#keymap.grid_minor: G           # switching on/off minor grids in current axes
#keymap.yscale: l               # toggle scaling of y-axes ('log'/'linear')
#keymap.xscale: k, L            # toggle scaling of x-axes ('log'/'linear')
#keymap.copy: ctrl+c, cmd+c     # copy figure to clipboard


## ***************************************************************************
## * ANIMATION                                                               *
## ***************************************************************************
#animation.html: none  # How to display the animation as HTML in
                      # the IPython notebook:
```

(continues on next page)

(continued from previous page)

```
                        #     - 'html5' uses HTML5 video tag
                        #     - 'jshtml' creates a JavaScript animation
#animation.writer:  ffmpeg         # MovieWriter 'backend' to use
#animation.codec:   h264           # Codec to use for writing movie
#animation.bitrate: -1             # Controls size/quality trade-off for movie.
                                   # -1 implies let utility auto-determine
#animation.frame_format: png       # Controls frame format used by temp files

## Path to ffmpeg binary.  Unqualified paths are resolved by subprocess.Popen.
#animation.ffmpeg_path:  ffmpeg
## Additional arguments to pass to ffmpeg.
#animation.ffmpeg_args:

## Path to ImageMagick's convert binary.  Unqualified paths are resolved by
## subprocess.Popen, except that on Windows, we look up an install of
## ImageMagick in the registry (as convert is also the name of a system tool).
#animation.convert_path: convert
## Additional arguments to pass to convert.
#animation.convert_args: -layers, OptimizePlus
#
#animation.embed_limit:  20.0      # Limit, in MB, of size of base64 encoded
                                   # animation in HTML (i.e. IPython notebook)
```

**Total running time of the script:** (0 minutes 2.164 seconds)

## 3.6 Colors

Matplotlib has support for visualizing information with a wide array of colors and colormaps. These tutorials cover the basics of how these colormaps look, how you can create your own, and how you can customize colormaps for your use case.

For even more information see the *examples page*.

### 3.6.1 Specifying colors

#### Color formats

Matplotlib recognizes the following formats to specify a color.

| Format | Example |
| --- | --- |
| RGB or RGBA (red, green, blue, alpha) tuple of float values in a closed interval [0, 1]. | • `(0.1, 0.2, 0.5)`<br>• `(0.1, 0.2, 0.5, 0.3)` |
| Case-insensitive hex RGB or RGBA string. | • `'#0f0f0f'`<br>• `'#0f0f0f80'` |
| Case-insensitive RGB or RGBA string equivalent hex shorthand of duplicated characters. | • `'#abc'` as `'#aabbcc'`<br>• `'#fb1'` as `'#ffbb11'` |
| String representation of float value in closed interval `[0, 1]` for grayscale values. | • `'0'` as black<br>• `'1'` as white<br>• `'0.8'` as light gray |
| Single character shorthand notation for some basic colors.<br>**Note:** The colors green, cyan, magenta, and yellow do not coincide with X11/CSS4 colors. Their particular shades were chosen for better visibility of colored lines against typical backgrounds. | • `'b'` as blue<br>• `'g'` as green<br>• `'r'` as red<br>• `'c'` as cyan<br>• `'m'` as magenta<br>• `'y'` as yellow<br>• `'k'` as black<br>• `'w'` as white |
| Case-insensitive X11/CSS4 color name with no spaces. | • `'aquamarine'`<br>• `'mediumseagreen'` |
| Case-insensitive color name from xkcd color survey with `'xkcd:'` prefix. | • `'xkcd:sky blue'`<br>• `'xkcd:eggshell'` |
| Case-insensitive Tableau Colors from 'T10' categorical palette.<br>**Note:** This is the default color cycle. | • `'tab:blue'`<br>• `'tab:orange'`<br>• `'tab:green'`<br>• `'tab:red'`<br>• `'tab:purple'`<br>• `'tab:brown'`<br>• `'tab:pink'`<br>• `'tab:gray'`<br>• `'tab:olive'`<br>• `'tab:cyan'` |
| "CN" color spec where `'C'` precedes a number acting as an index into the default property cycle.<br>**Note:** Matplotlib indexes color at draw time and defaults to black if cycle does not include color. | • `'C0'`<br>• `'C1'`<br>*rcParams["axes.prop_cycle"]* (default: `cycler('color', ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728', '#9467bd', '#8c564b', '#e377c2',` |

**See also:**

**The following links provide more information on colors in Matplotlib.**

- *Color Demo* Example
- `matplotlib.colors` API
- *List of named colors* Example

"Red", "Green", and "Blue" are the intensities of those colors. In combination, they represent the colorspace.

### Transparency

The *alpha* value of a color specifies its transparency, where 0 is fully transparent and 1 is fully opaque. When a color is semi-transparent, the background color will show through.

The *alpha* value determines the resulting color by blending the foreground color with the background color according to the formula

$$RGB_{result} = RGB_{background} * (1 - \alpha) + RGB_{foreground} * \alpha$$

The following plot illustrates the effect of transparency.

```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.patches import Rectangle

fig, ax = plt.subplots(figsize=(6.5, 1.65), layout='constrained')
ax.add_patch(Rectangle((-0.2, -0.35), 11.2, 0.7, color='C1', alpha=0.8))
for i, alpha in enumerate(np.linspace(0, 1, 11)):
    ax.add_patch(Rectangle((i, 0.05), 0.8, 0.6, alpha=alpha, zorder=0))
    ax.text(i+0.4, 0.85, f"{alpha:.1f}", ha='center')
    ax.add_patch(Rectangle((i, -0.05), 0.8, -0.6, alpha=alpha, zorder=2))
ax.set_xlim(-0.2, 13)
ax.set_ylim(-1, 1)
ax.set_title('alpha values')
ax.text(11.3, 0.6, 'zorder=1', va='center', color='C0')
ax.text(11.3, 0, 'zorder=2\nalpha=0.8', va='center', color='C1')
ax.text(11.3, -0.6, 'zorder=3', va='center', color='C0')
ax.axis('off')
```

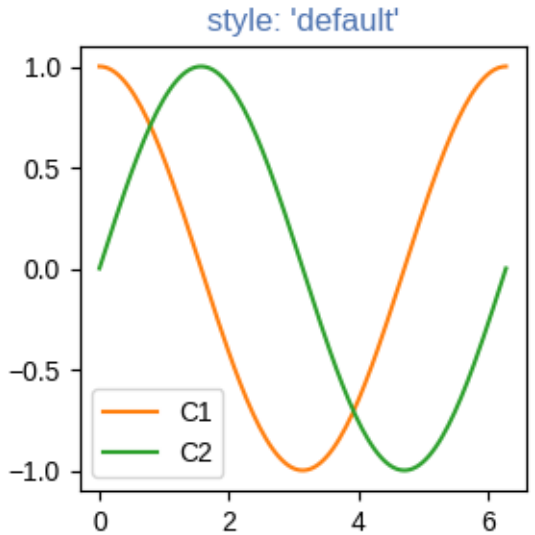
The orange rectangle is semi-transparent with *alpha* = 0.8. The top row of blue squares is drawn below and the bottom row of blue squares is drawn on top of the orange rectangle.

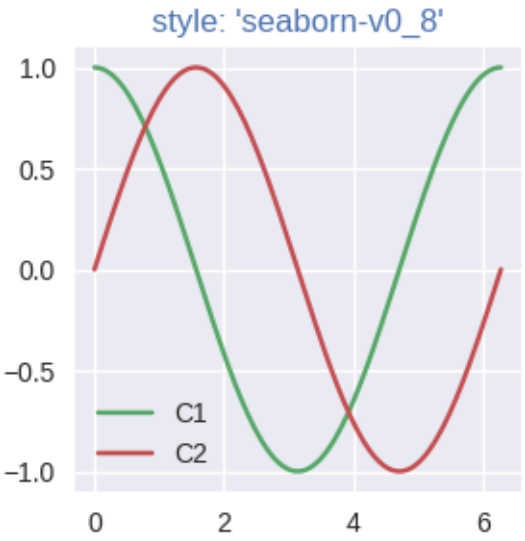
See also *Zorder Demo* to learn more on the drawing order.

### "CN" color selection

Matplotlib converts "CN" colors to RGBA when drawing Artists. The *Styling with cycler* section contains additional information about controlling colors and style properties.

```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib as mpl

th = np.linspace(0, 2*np.pi, 128)


def demo(sty):
    mpl.style.use(sty)
    fig, ax = plt.subplots(figsize=(3, 3))

    ax.set_title(f'style: {sty!r}', color='C0')

    ax.plot(th, np.cos(th), 'C1', label='C1')
    ax.plot(th, np.sin(th), 'C2', label='C2')
    ax.legend()


demo('default')
demo('seaborn-v0_8')
```






The first color `'C0'` is the title. Each plot uses the second and third colors of each style's *rcParams["axes.prop_cycle"]* (default: `cycler('color', ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728', '#9467bd', '#8c564b', '#e377c2', '#7f7f7f', '#bcbd22', '#17becf'])`). They are `'C1'` and `'C2'`, respectively.

### Comparison between X11/CSS4 and xkcd colors

The xkcd colors come from a user survey conducted by the webcomic xkcd.

95 out of the 148 X11/CSS4 color names also appear in the xkcd color survey. Almost all of them map to different color values in the X11/CSS4 and in the xkcd palette. Only 'black', 'white' and 'cyan' are identical.

For example, `'blue'` maps to `'#0000FF'` whereas `'xkcd:blue'` maps to `'#0343DF'`. Due to these name collisions, all xkcd colors have the `'xkcd:'` prefix.

The visual below shows name collisions. Color names where color values agree are in bold.

```
import matplotlib.colors as mcolors
import matplotlib.patches as mpatch

overlap = {name for name in mcolors.CSS4_COLORS
           if f'xkcd:{name}' in mcolors.XKCD_COLORS}

fig = plt.figure(figsize=[9, 5])
ax = fig.add_axes([0, 0, 1, 1])

n_groups = 3
n_rows = len(overlap) // n_groups + 1

for j, color_name in enumerate(sorted(overlap)):
    css4 = mcolors.CSS4_COLORS[color_name]
    xkcd = mcolors.XKCD_COLORS[f'xkcd:{color_name}'].upper()
```

(continues on next page)

(continued from previous page)

```
    # Pick text colour based on perceived luminance.
    rgba = mcolors.to_rgba_array([css4, xkcd])
    luma = 0.299 * rgba[:, 0] + 0.587 * rgba[:, 1] + 0.114 * rgba[:, 2]
    css4_text_color = 'k' if luma[0] > 0.5 else 'w'
    xkcd_text_color = 'k' if luma[1] > 0.5 else 'w'

    col_shift = (j // n_rows) * 3
    y_pos = j % n_rows
    text_args = dict(fontsize=10, weight='bold' if css4 == xkcd else None)
    ax.add_patch(mpatch.Rectangle((0 + col_shift, y_pos), 1, 1, color=css4))
    ax.add_patch(mpatch.Rectangle((1 + col_shift, y_pos), 1, 1, color=xkcd))
    ax.text(0.5 + col_shift, y_pos + .7, css4,
            color=css4_text_color, ha='center', **text_args)
    ax.text(1.5 + col_shift, y_pos + .7, xkcd,
            color=xkcd_text_color, ha='center', **text_args)
    ax.text(2 + col_shift, y_pos + .7, f'  {color_name}', **text_args)

for g in range(n_groups):
    ax.hlines(range(n_rows), 3*g, 3*g + 2.8, color='0.7', linewidth=1)
    ax.text(0.5 + 3*g, -0.3, 'X11/CSS4', ha='center')
    ax.text(1.5 + 3*g, -0.3, 'xkcd', ha='center')

ax.set_xlim(0, 3 * n_groups)
ax.set_ylim(n_rows, -1)
ax.axis('off')

plt.show()
```

| X11/CSS4 | xkcd | |
|---|---|---|
| #00FFFF | #13EAC9 | aqua |
| #7FFFD4 | #04D8B2 | aquamarine |
| #F0FFFF | #069AF3 | azure |
| #F5F5DC | #E6DAA6 | beige |
| **#000000** | **#000000** | **black** |
| #0000FF | #0343DF | blue |
| #A52A2A | #653700 | brown |
| #7FFF00 | #C1F80A | chartreuse |
| #D2691E | #3D1C02 | chocolate |
| #FF7F50 | #FC5A50 | coral |
| #DC143C | #8C000F | crimson |
| **#00FFFF** | **#00FFFF** | **cyan** |
| #00008B | #030764 | darkblue |
| #006400 | #054907 | darkgreen |
| #FF00FF | #ED0DD9 | fuchsia |
| #FFD700 | #DBB40C | gold |
| #DAA520 | #FAC205 | goldenrod |

| X11/CSS4 | xkcd | |
|---|---|---|
| #008000 | #15B01A | green |
| #808080 | #929591 | grey |
| #4B0082 | #380282 | indigo |
| #FFFFF0 | #FFFFCB | ivory |
| #F0E68C | #AAA662 | khaki |
| #E6E6FA | #C79FEF | lavender |
| #ADD8E6 | #7BC8F6 | lightblue |
| #90EE90 | #76FF7B | lightgreen |
| #00FF00 | #AAFF32 | lime |
| #FF00FF | #C20078 | magenta |
| #800000 | #650021 | maroon |
| #000080 | #01153E | navy |
| #808000 | #6E750E | olive |
| #FFA500 | #F97306 | orange |
| #FF4500 | #FE420F | orangered |
| #DA70D6 | #C875C4 | orchid |
| #FFC0CB | #FF81C0 | pink |

| X11/CSS4 | xkcd | |
|---|---|---|
| #DDA0DD | #580F41 | plum |
| #800080 | #7E1E9C | purple |
| #FF0000 | #E50000 | red |
| #FA8072 | #FF796C | salmon |
| #A0522D | #A9561E | sienna |
| #C0C0C0 | #C5C9C7 | silver |
| #D2B48C | #D1B26F | tan |
| #008080 | #029386 | teal |
| #FF6347 | #EF4026 | tomato |
| #40E0D0 | #06C2AC | turquoise |
| #EE82EE | #9A0EEA | violet |
| #F5DEB3 | #FBDD7E | wheat |
| **#FFFFFF** | **#FFFFFF** | **white** |
| #FFFF00 | #FFFF14 | yellow |
| #9ACD32 | #BBF90F | yellowgreen |

**Total running time of the script:** (0 minutes 1.316 seconds)

### 3.6.2 Customized Colorbars Tutorial

This tutorial shows how to build and customize standalone colorbars, i.e. without an attached plot.

A *colorbar* needs a "mappable" (*matplotlib.cm.ScalarMappable*) object (typically, an image) which indicates the colormap and the norm to be used. In order to create a colorbar without an attached image, one can instead use a *ScalarMappable* with no associated data.

```
import matplotlib.pyplot as plt
import matplotlib as mpl
```

#### Basic continuous colorbar

Here, we create a basic continuous colorbar with ticks and labels.

The arguments to the *colorbar* call are the *ScalarMappable* (constructed using the *norm* and *cmap* arguments), the axes where the colorbar should be drawn, and the colorbar's orientation.

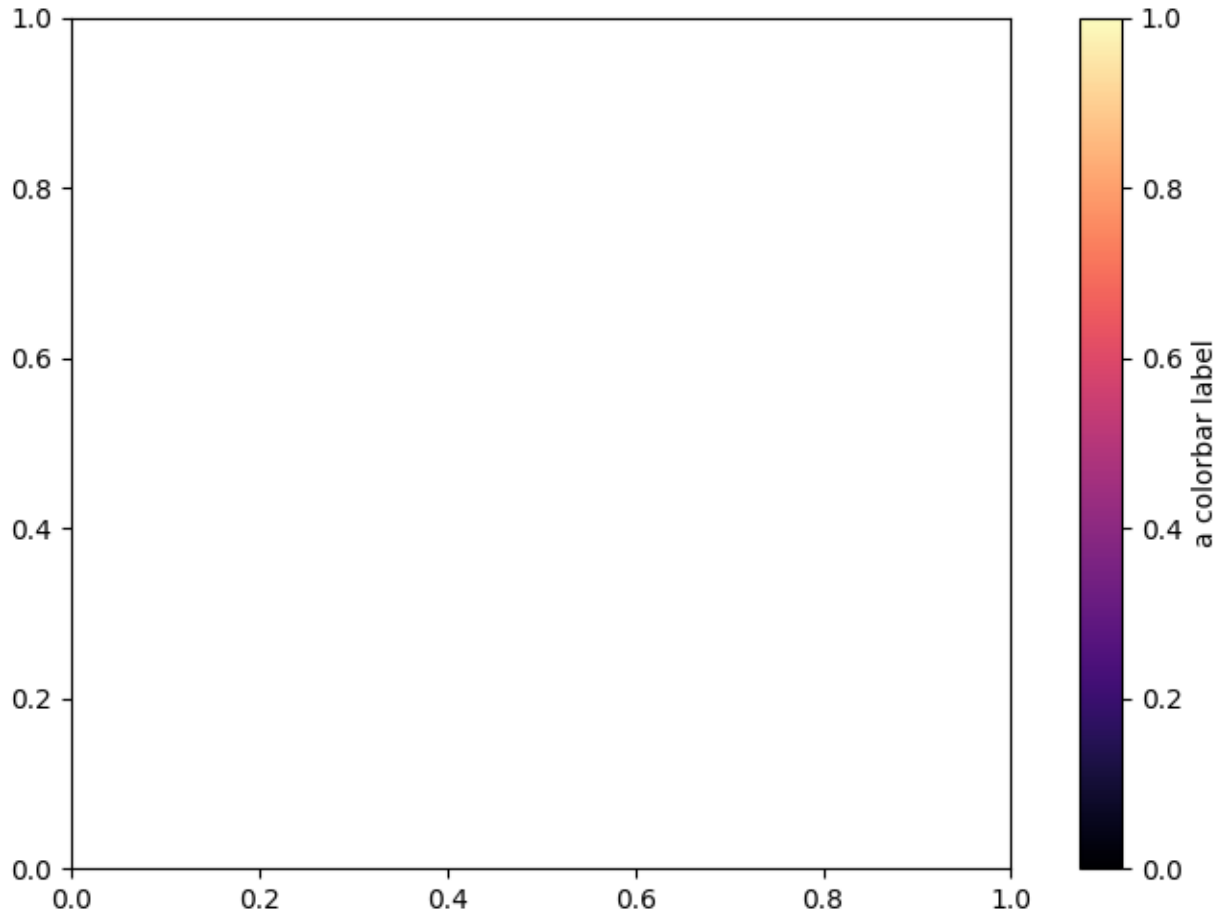
For more information see the *colorbar* API.

```
fig, ax = plt.subplots(figsize=(6, 1), layout='constrained')

cmap = mpl.cm.cool
norm = mpl.colors.Normalize(vmin=5, vmax=10)

fig.colorbar(mpl.cm.ScalarMappable(norm=norm, cmap=cmap),
             cax=ax, orientation='horizontal', label='Some Units')
```

#### Colorbar attached next to a pre-existing axes

All examples in this tutorial (except this one) show a standalone colorbar on its own figure, but it is possible to display the colorbar *next* to a pre-existing Axes *ax* by passing `ax=ax` to the colorbar() call (meaning "draw the colorbar next to *ax*") rather than `cax=ax` (meaning "draw the colorbar on *ax*").

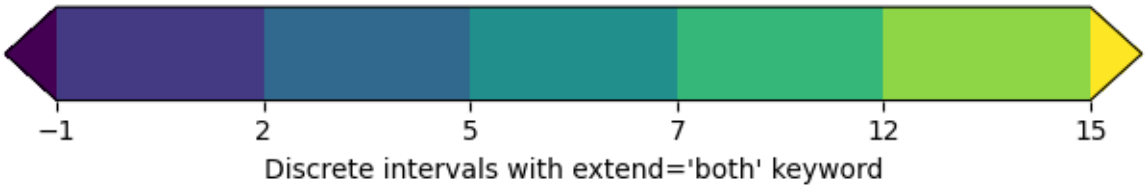
```
fig, ax = plt.subplots(layout='constrained')

fig.colorbar(mpl.cm.ScalarMappable(norm=mpl.colors.Normalize(0, 1), cmap=
 ↪'magma'),
             ax=ax, orientation='vertical', label='a colorbar label')
```




### Discrete and extended colorbar with continuous colorscale

The following example shows how to make a discrete colorbar based on a continuous cmap. We use *matplotlib.colors.BoundaryNorm* to describe the interval boundaries (which must be in increasing order), and further pass the *extend* argument to it to further display "over" and "under" colors (which are used for data outside of the norm range).

```
fig, ax = plt.subplots(figsize=(6, 1), layout='constrained')

cmap = mpl.cm.viridis
bounds = [-1, 2, 5, 7, 12, 15]
norm = mpl.colors.BoundaryNorm(bounds, cmap.N, extend='both')

fig.colorbar(mpl.cm.ScalarMappable(norm=norm, cmap=cmap),
             cax=ax, orientation='horizontal',
             label="Discrete intervals with extend='both' keyword")
```



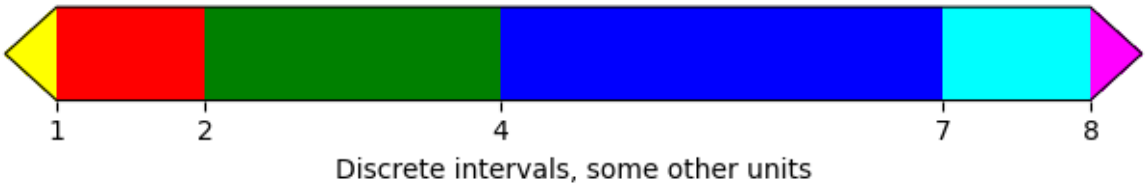

### Colorbar with arbitrary colors

The following example still uses a *BoundaryNorm* to describe discrete interval boundaries, but now uses a *matplotlib.colors.ListedColormap* to associate each interval with an arbitrary color (there must be as many intervals than there are colors). The "over" and "under" colors are set on the colormap using *Colormap.with_extremes*.

We also pass additional arguments to *colorbar*:

- To display the out-of-range values on the colorbar, we use the *extend* argument in the colorbar() call. (This is equivalent to passing the *extend* argument in the *BoundaryNorm* constructor as done in the previous example.)
- To make the length of each colorbar segment proportional to its corresponding interval, we use the *spacing* argument in the colorbar() call.

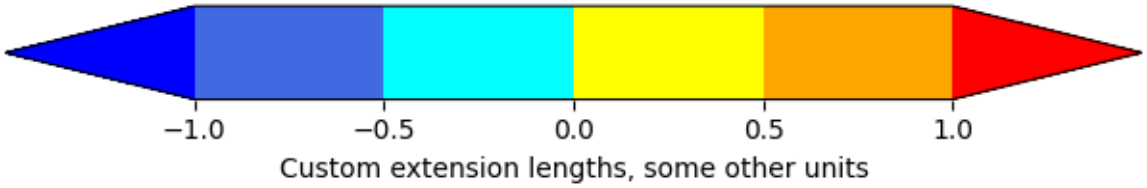
```
fig, ax = plt.subplots(figsize=(6, 1), layout='constrained')

cmap = (mpl.colors.ListedColormap(['red', 'green', 'blue', 'cyan'])
        .with_extremes(under='yellow', over='magenta'))
bounds = [1, 2, 4, 7, 8]
norm = mpl.colors.BoundaryNorm(bounds, cmap.N)

fig.colorbar(
    mpl.cm.ScalarMappable(cmap=cmap, norm=norm),
    cax=ax, orientation='horizontal',
    extend='both',
    spacing='proportional',
    label='Discrete intervals, some other units',
)
```



### Colorbar with custom extension lengths

We can customize the length colorbar extensions, on a colorbar with discrete intervals. To make the length of each extension the same as the length of the interior colors, use `extendfrac='auto'`.

```
fig, ax = plt.subplots(figsize=(6, 1), layout='constrained')

cmap = (mpl.colors.ListedColormap(['royalblue', 'cyan', 'yellow', 'orange'])
        .with_extremes(over='red', under='blue'))
bounds = [-1.0, -0.5, 0.0, 0.5, 1.0]
norm = mpl.colors.BoundaryNorm(bounds, cmap.N)

fig.colorbar(
    mpl.cm.ScalarMappable(cmap=cmap, norm=norm),
    cax=ax, orientation='horizontal',
    extend='both', extendfrac='auto',
    spacing='uniform',
    label='Custom extension lengths, some other units',
)

plt.show()
```




## 3.6.3 Creating Colormaps in Matplotlib

Matplotlib has a number of built-in colormaps accessible via *matplotlib.colormaps*. There are also external libraries like palettable that have many extra colormaps.

However, we may also want to create or manipulate our own colormaps. This can be done using the class *ListedColormap* or *LinearSegmentedColormap*. Both colormap classes map values between 0 and 1 to colors. There are however differences, as explained below.

Before manually creating or manipulating colormaps, let us first see how we can obtain colormaps and their colors from existing colormap classes.

### Getting colormaps and accessing their values

First, getting a named colormap, most of which are listed in *Choosing Colormaps in Matplotlib*, may be done using *matplotlib.colormaps*, which returns a colormap object. The length of the list of colors used internally to define the colormap can be adjusted via *Colormap.resampled*. Below we use a modest value of 8 so there are not a lot of values to look at.

```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib as mpl
from matplotlib.colors import LinearSegmentedColormap, ListedColormap

viridis = mpl.colormaps['viridis'].resampled(8)
```

The object `viridis` is a callable, that when passed a float between 0 and 1 returns an RGBA value from the colormap:

```
print(viridis(0.56))
```

```
(0.122312, 0.633153, 0.530398, 1.0)
```

### ListedColormap

*ListedColormap*s store their color values in a `.colors` attribute. The list of colors that comprise the colormap can be directly accessed using the `colors` property, or it can be accessed indirectly by calling `viridis` with an array of values matching the length of the colormap. Note that the returned list is in the form of an RGBA (N, 4) array, where N is the length of the colormap.

```
print('viridis.colors', viridis.colors)
print('viridis(range(8))', viridis(range(8)))
print('viridis(np.linspace(0, 1, 8))', viridis(np.linspace(0, 1, 8)))
```

```
viridis.colors [[0.267004 0.004874 0.329415 1.      ]
 [0.275191 0.194905 0.496005 1.      ]
 [0.212395 0.359683 0.55171  1.      ]
 [0.153364 0.497    0.557724 1.      ]
 [0.122312 0.633153 0.530398 1.      ]
 [0.288921 0.758394 0.428426 1.      ]
 [0.626579 0.854645 0.223353 1.      ]
 [0.993248 0.906157 0.143936 1.      ]]
viridis(range(8)) [[0.267004 0.004874 0.329415 1.      ]
 [0.275191 0.194905 0.496005 1.      ]
 [0.212395 0.359683 0.55171  1.      ]
 [0.153364 0.497    0.557724 1.      ]
 [0.122312 0.633153 0.530398 1.      ]
 [0.288921 0.758394 0.428426 1.      ]
 [0.626579 0.854645 0.223353 1.      ]
 [0.993248 0.906157 0.143936 1.      ]]
viridis(np.linspace(0, 1, 8)) [[0.267004 0.004874 0.329415 1.      ]
 [0.275191 0.194905 0.496005 1.      ]
 [0.212395 0.359683 0.55171  1.      ]
 [0.153364 0.497    0.557724 1.      ]
 [0.122312 0.633153 0.530398 1.      ]
 [0.288921 0.758394 0.428426 1.      ]
```

(continues on next page)

(continued from previous page)

```
 [0.626579 0.854645 0.223353 1.      ]
 [0.993248 0.906157 0.143936 1.      ]]
```

The colormap is a lookup table, so "oversampling" the colormap returns nearest-neighbor interpolation (note the repeated colors in the list below)

```
print('viridis(np.linspace(0, 1, 12))', viridis(np.linspace(0, 1, 12)))
```

```
viridis(np.linspace(0, 1, 12)) [[0.267004 0.004874 0.329415 1.      ]
 [0.267004 0.004874 0.329415 1.      ]
 [0.275191 0.194905 0.496005 1.      ]
 [0.212395 0.359683 0.55171  1.      ]
 [0.212395 0.359683 0.55171  1.      ]
 [0.153364 0.497    0.557724 1.      ]
 [0.122312 0.633153 0.530398 1.      ]
 [0.288921 0.758394 0.428426 1.      ]
 [0.288921 0.758394 0.428426 1.      ]
 [0.626579 0.854645 0.223353 1.      ]
 [0.993248 0.906157 0.143936 1.      ]
 [0.993248 0.906157 0.143936 1.      ]]
```

### LinearSegmentedColormap

*LinearSegmentedColormap*s do not have a `.colors` attribute. However, one may still call the colormap with an integer array, or with a float array between 0 and 1.

```
copper = mpl.colormaps['copper'].resampled(8)

print('copper(range(8))', copper(range(8)))
print('copper(np.linspace(0, 1, 8))', copper(np.linspace(0, 1, 8)))
```

```
copper(range(8)) [[0.         0.         0.         1.        ]
 [0.17647055 0.1116     0.07107143 1.        ]
 [0.35294109 0.2232     0.14214286 1.        ]
 [0.52941164 0.3348     0.21321429 1.        ]
 [0.70588219 0.4464     0.28428571 1.        ]
 [0.88235273 0.558      0.35535714 1.        ]
 [1.         0.6696     0.42642857 1.        ]
 [1.         0.7812     0.4975     1.        ]]
copper(np.linspace(0, 1, 8)) [[0.         0.         0.         1.        ]
 [0.17647055 0.1116     0.07107143 1.        ]
 [0.35294109 0.2232     0.14214286 1.        ]
 [0.52941164 0.3348     0.21321429 1.        ]
 [0.70588219 0.4464     0.28428571 1.        ]
 [0.88235273 0.558      0.35535714 1.        ]
 [1.         0.6696     0.42642857 1.        ]
 [1.         0.7812     0.4975     1.        ]]
```

### Creating listed colormaps

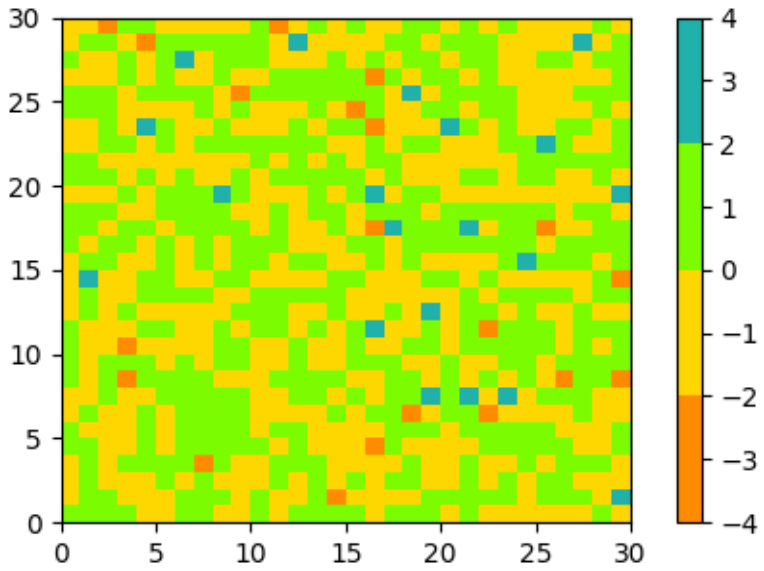
Creating a colormap is essentially the inverse operation of the above where we supply a list or array of color specifications to *ListedColormap* to make a new colormap.

Before continuing with the tutorial, let us define a helper function that takes one of more colormaps as input, creates some random data and applies the colormap(s) to an image plot of that dataset.

```
def plot_examples(colormaps):
    """
    Helper function to plot data with associated colormap.
    """
    np.random.seed(19680801)
    data = np.random.randn(30, 30)
    n = len(colormaps)
    fig, axs = plt.subplots(1, n, figsize=(n * 2 + 2, 3),
                            layout='constrained', squeeze=False)
    for [ax, cmap] in zip(axs.flat, colormaps):
        psm = ax.pcolormesh(data, cmap=cmap, rasterized=True, vmin=-4, vmax=4)
        fig.colorbar(psm, ax=ax)
    plt.show()
```

In the simplest case we might type in a list of color names to create a colormap from those.

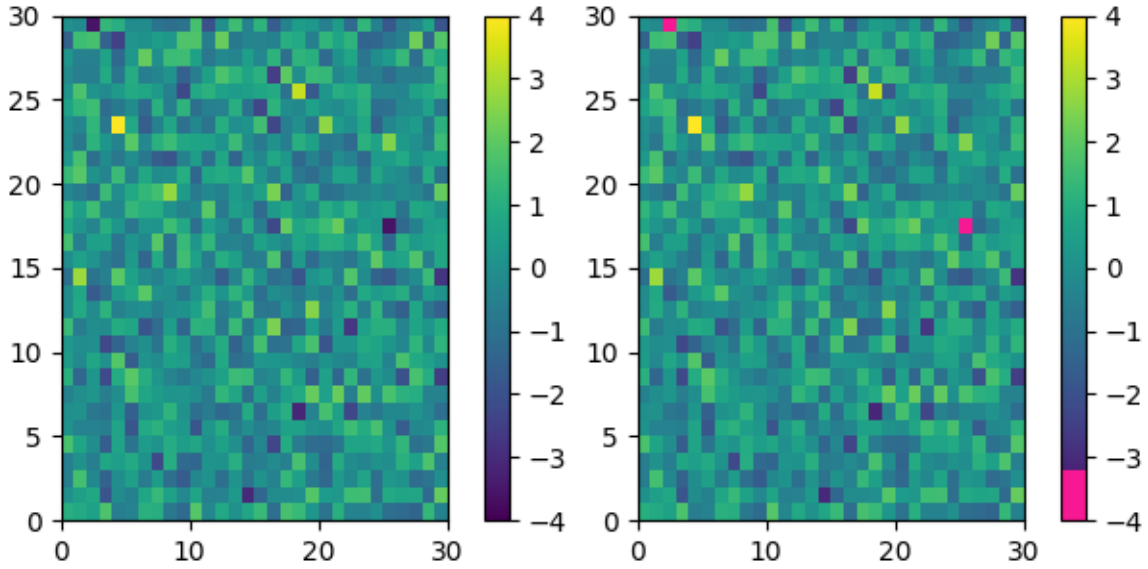
```
cmap = ListedColormap(["darkorange", "gold", "lawngreen", "lightseagreen"])
plot_examples([cmap])
```




In fact, that list may contain any valid *Matplotlib color specification*. Particularly useful for creating custom colormaps are (N, 4)-shaped arrays. Because with the variety of numpy operations that we can do on a such an array, carpentry of new colormaps from existing colormaps become quite straight forward.

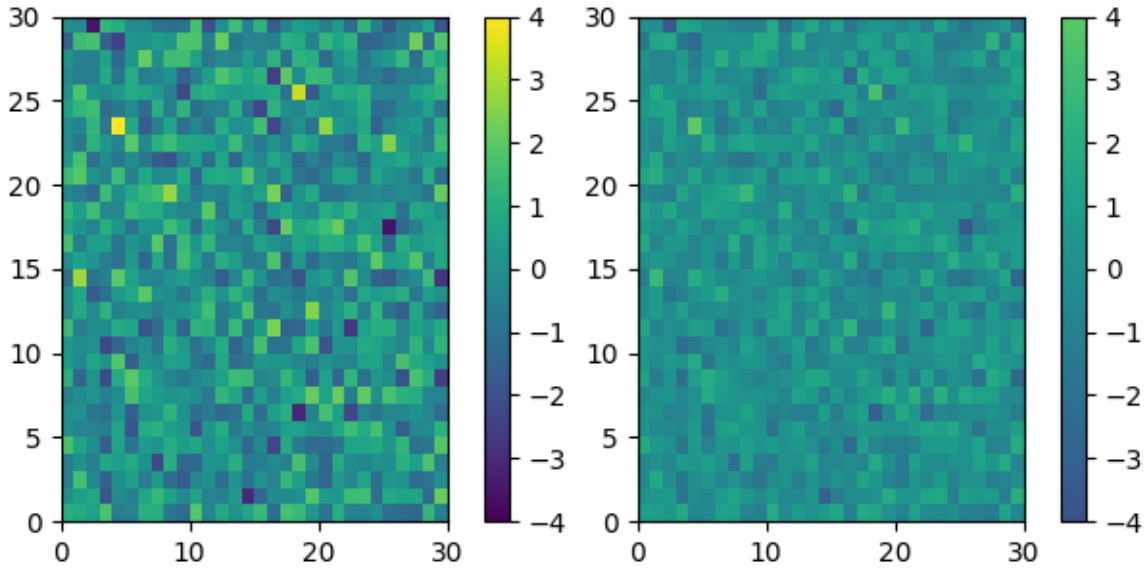
For example, suppose we want to make the first 25 entries of a 256-length "viridis" colormap pink for some reason:

```
viridis = mpl.colormaps['viridis'].resampled(256)
newcolors = viridis(np.linspace(0, 1, 256))
pink = np.array([248/256, 24/256, 148/256, 1])
newcolors[:25, :] = pink
newcmp = ListedColormap(newcolors)

plot_examples([viridis, newcmp])
```



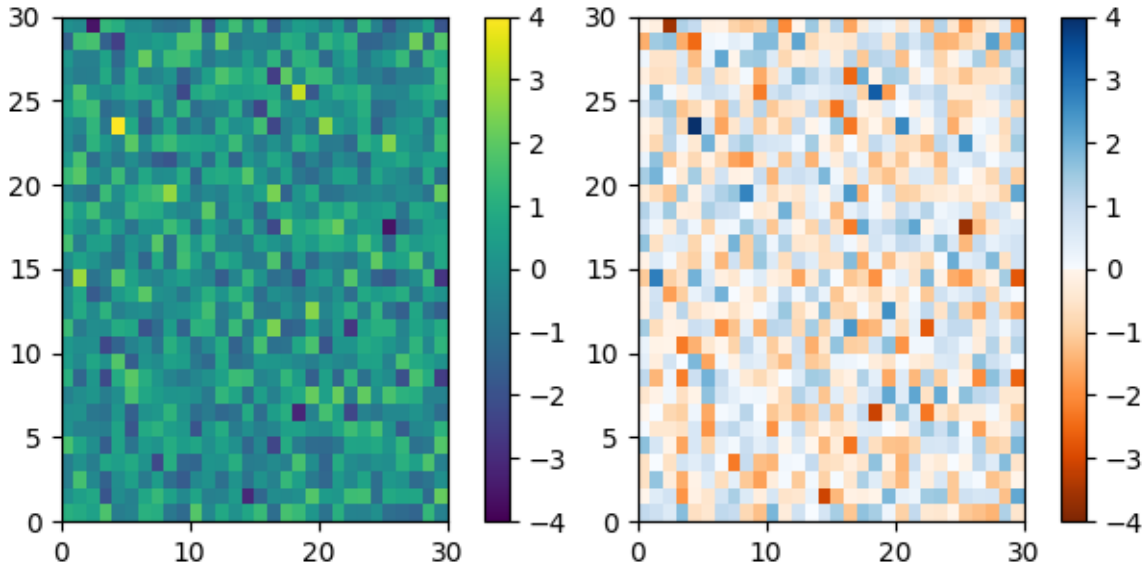

We can reduce the dynamic range of a colormap; here we choose the middle half of the colormap. Note, however, that because viridis is a listed colormap, we will end up with 128 discrete values instead of the 256 values that were in the original colormap. This method does not interpolate in color-space to add new colors.

```
viridis_big = mpl.colormaps['viridis']
newcmp = ListedColormap(viridis_big(np.linspace(0.25, 0.75, 128)))
plot_examples([viridis, newcmp])
```



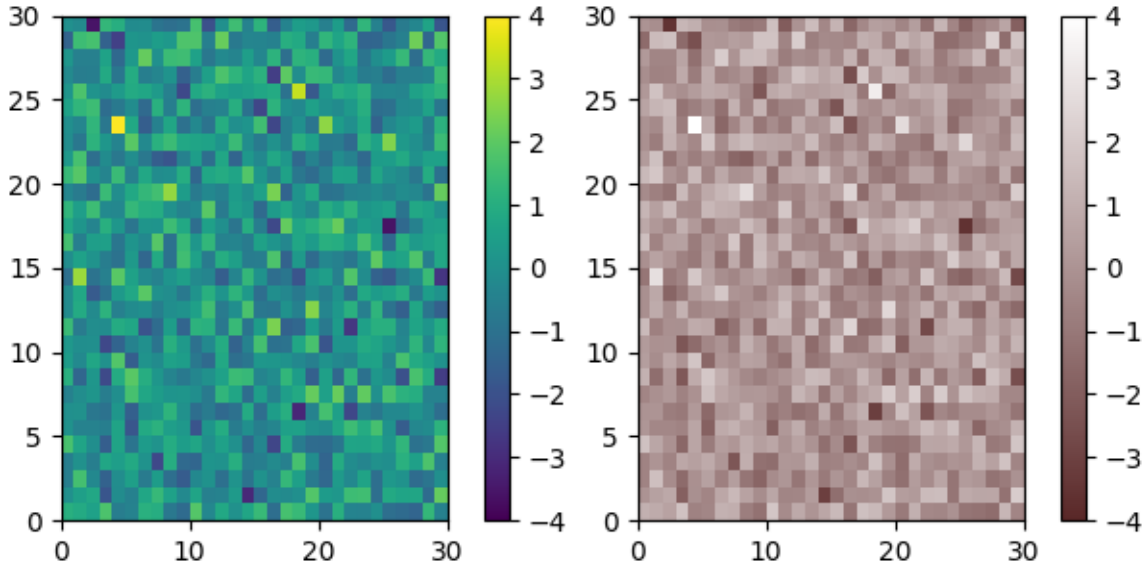
and we can easily concatenate two colormaps:

```
top = mpl.colormaps['Oranges_r'].resampled(128)
bottom = mpl.colormaps['Blues'].resampled(128)

newcolors = np.vstack((top(np.linspace(0, 1, 128)),
                       bottom(np.linspace(0, 1, 128))))
newcmp = ListedColormap(newcolors, name='OrangeBlue')
plot_examples([viridis, newcmp])
```




Of course we need not start from a named colormap, we just need to create the (N, 4) array to pass to *ListedColormap*. Here we create a colormap that goes from brown (RGB: 90, 40, 40) to white (RGB: 255, 255, 255).

```
N = 256
vals = np.ones((N, 4))
vals[:, 0] = np.linspace(90/256, 1, N)
vals[:, 1] = np.linspace(40/256, 1, N)
vals[:, 2] = np.linspace(40/256, 1, N)
newcmp = ListedColormap(vals)
plot_examples([viridis, newcmp])
```




### Creating linear segmented colormaps

The *LinearSegmentedColormap* class specifies colormaps using anchor points between which RGB(A) values are interpolated.

The format to specify these colormaps allows discontinuities at the anchor points. Each anchor point is specified as a row in a matrix of the form `[x[i] yleft[i] yright[i]]`, where `x[i]` is the anchor, and `yleft[i]` and `yright[i]` are the values of the color on either side of the anchor point.

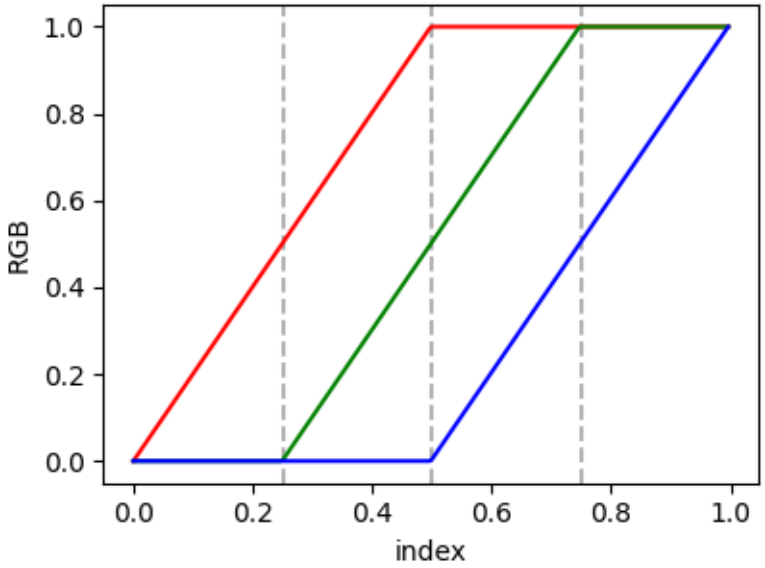
If there are no discontinuities, then `yleft[i] == yright[i]`:

```
cdict = {'red':   [[0.0,  0.0, 0.0],
                   [0.5,  1.0, 1.0],
                   [1.0,  1.0, 1.0]],
         'green': [[0.0,  0.0, 0.0],
                   [0.25, 0.0, 0.0],
                   [0.75, 1.0, 1.0],
                   [1.0,  1.0, 1.0]],
         'blue':  [[0.0,  0.0, 0.0],
                   [0.5,  0.0, 0.0],
                   [1.0,  1.0, 1.0]]}


def plot_linearmap(cdict):
    newcmp = LinearSegmentedColormap('testCmap', segmentdata=cdict, N=256)
    rgba = newcmp(np.linspace(0, 1, 256))
    fig, ax = plt.subplots(figsize=(4, 3), layout='constrained')
    col = ['r', 'g', 'b']
    for xx in [0.25, 0.5, 0.75]:
        ax.axvline(xx, color='0.7', linestyle='--')
    for i in range(3):
        ax.plot(np.arange(256)/256, rgba[:, i], color=col[i])
    ax.set_xlabel('index')
```

(continues on next page)

(continued from previous page)

```
    ax.set_ylabel('RGB')
    plt.show()

plot_linearmap(cdict)
```



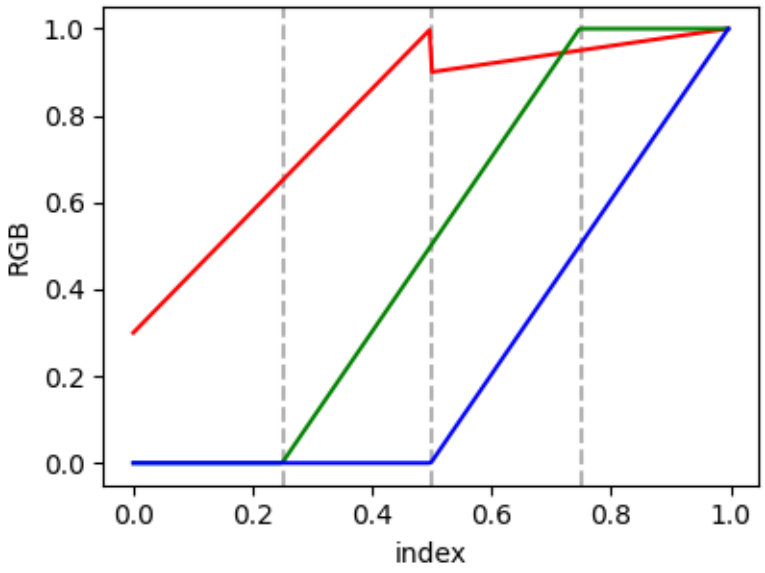

In order to make a discontinuity at an anchor point, the third column is different than the second. The matrix for each of "red", "green", "blue", and optionally "alpha" is set up as:

```
cdict['red'] = [...
                [x[i]       yleft[i]     yright[i]],
                [x[i+1]     yleft[i+1]   yright[i+1]],
               ...]
```

and for values passed to the colormap between `x[i]` and `x[i+1]`, the interpolation is between `yright[i]` and `yleft[i+1]`.

In the example below there is a discontinuity in red at 0.5. The interpolation between 0 and 0.5 goes from 0.3 to 1, and between 0.5 and 1 it goes from 0.9 to 1. Note that `red[0, 1]`, and `red[2, 2]` are both superfluous to the interpolation because `red[0, 1]` (i.e., `yleft[0]`) is the value to the left of 0, and `red[2, 2]` (i.e., `yright[2]`) is the value to the right of 1, which are outside the color mapping domain.

```
cdict['red'] = [[0.0,  0.0, 0.3],
                [0.5,  1.0, 0.9],
                [1.0,  1.0, 1.0]]
plot_linearmap(cdict)
```



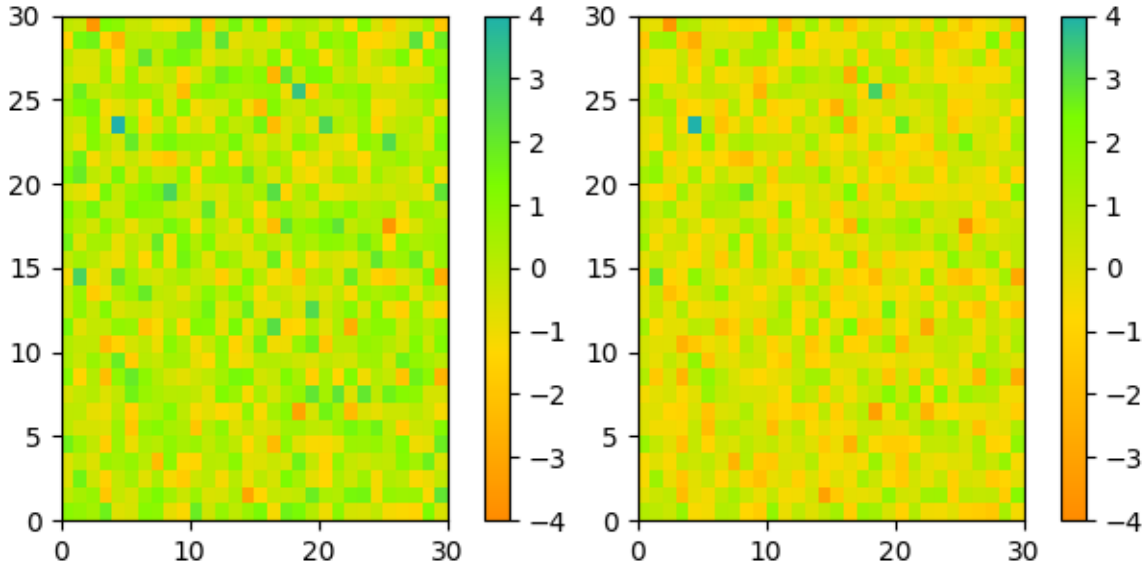

### Directly creating a segmented colormap from a list

The approach described above is very versatile, but admittedly a bit cumbersome to implement. For some basic cases, the use of *LinearSegmentedColormap.from_list* may be easier. This creates a segmented colormap with equal spacings from a supplied list of colors.

```
colors = ["darkorange", "gold", "lawngreen", "lightseagreen"]
cmap1 = LinearSegmentedColormap.from_list("mycmap", colors)
```

If desired, the nodes of the colormap can be given as numbers between 0 and 1. For example, one could have the reddish part take more space in the colormap.

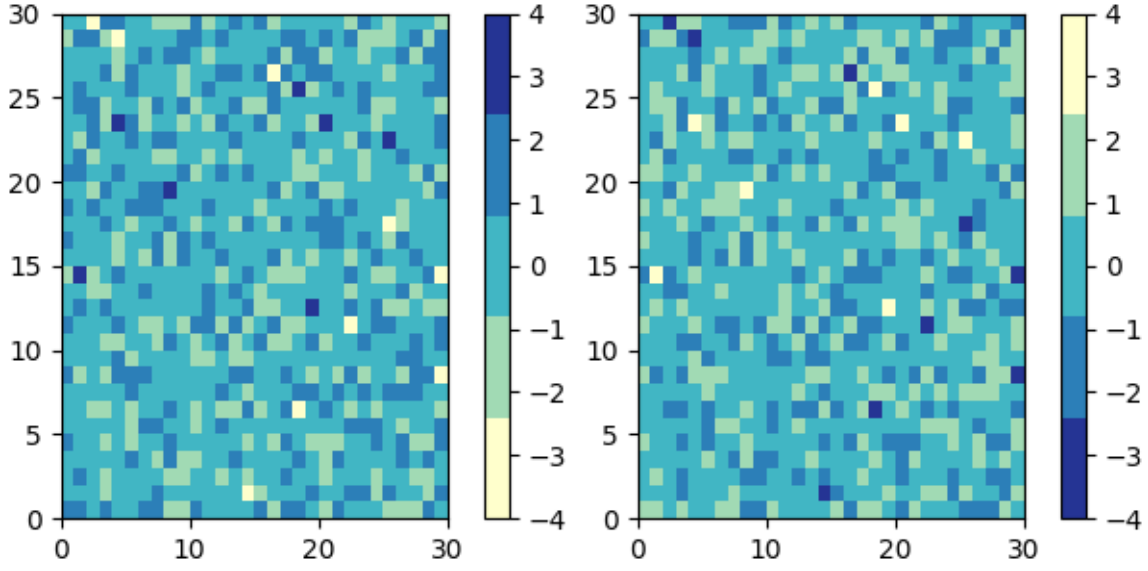
```
nodes = [0.0, 0.4, 0.8, 1.0]
cmap2 = LinearSegmentedColormap.from_list("mycmap", list(zip(nodes, colors)))

plot_examples([cmap1, cmap2])
```




### Reversing a colormap

*Colormap.reversed* creates a new colormap that is a reversed version of the original colormap.

```
colors = ["#ffffcc", "#a1dab4", "#41b6c4", "#2c7fb8", "#253494"]
my_cmap = ListedColormap(colors, name="my_cmap")

my_cmap_r = my_cmap.reversed()

plot_examples([my_cmap, my_cmap_r])
```

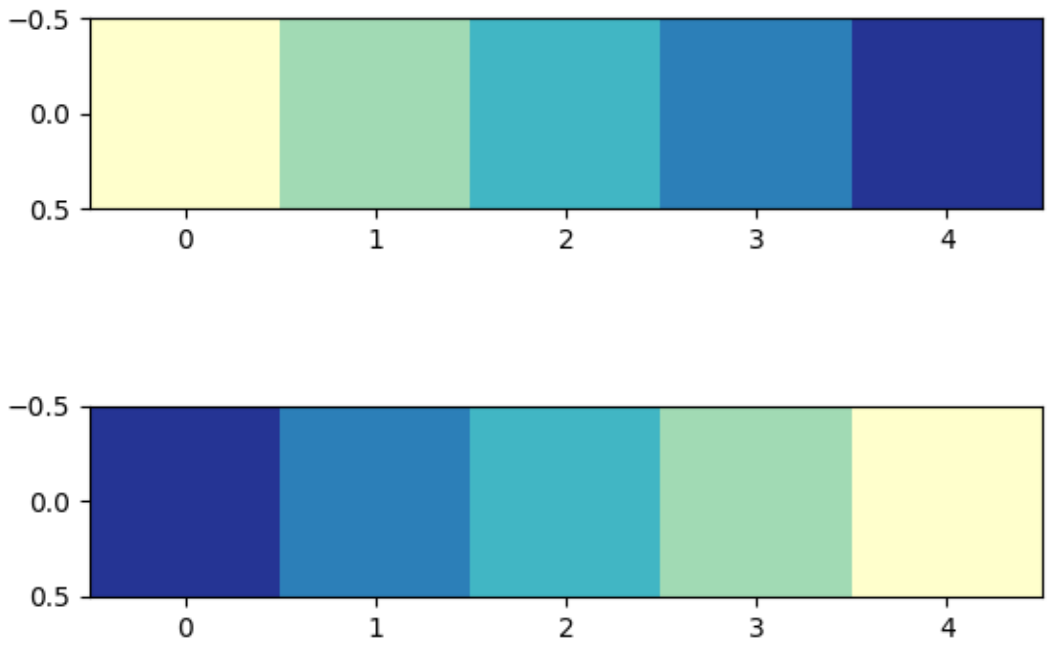



If no name is passed in, `.reversed` also names the copy by *appending '_r'* to the original colormap's name.

### Registering a colormap

Colormaps can be added to the *matplotlib.colormaps* list of named colormaps. This allows the colormaps to be accessed by name in plotting functions:

```
# my_cmap, my_cmap_r from reversing a colormap
mpl.colormaps.register(cmap=my_cmap)
mpl.colormaps.register(cmap=my_cmap_r)

data = [[1, 2, 3, 4, 5]]

fig, (ax1, ax2) = plt.subplots(nrows=2)

ax1.imshow(data, cmap='my_cmap')
ax2.imshow(data, cmap='my_cmap_r')

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.pcolormesh*

- *matplotlib.figure.Figure.colorbar*
- *matplotlib.colors*
- *matplotlib.colors.LinearSegmentedColormap*
- *matplotlib.colors.ListedColormap*
- *matplotlib.cm*
- *matplotlib.colormaps*

**Total running time of the script:** (0 minutes 3.565 seconds)

### 3.6.4 Colormap normalization

Objects that use colormaps by default linearly map the colors in the colormap from data values *vmin* to *vmax*. For example:

```
pcm = ax.pcolormesh(x, y, Z, vmin=-1., vmax=1., cmap='RdBu_r')
```

will map the data in *Z* linearly from -1 to +1, so *Z=0* will give a color at the center of the colormap *RdBu_r* (white in this case).

Matplotlib does this mapping in two steps, with a normalization from the input data to [0, 1] occurring first, and then mapping onto the indices in the colormap. Normalizations are classes defined in the *matplotlib.colors()* module. The default, linear normalization is *matplotlib.colors.Normalize()*.

Artists that map data to color pass the arguments *vmin* and *vmax* to construct a *matplotlib.colors.Normalize()* instance, then call it:

```
>>> import matplotlib as mpl
>>> norm = mpl.colors.Normalize(vmin=-1, vmax=1)
>>> norm(0)
0.5
```

However, there are sometimes cases where it is useful to map data to colormaps in a non-linear fashion.

#### Logarithmic

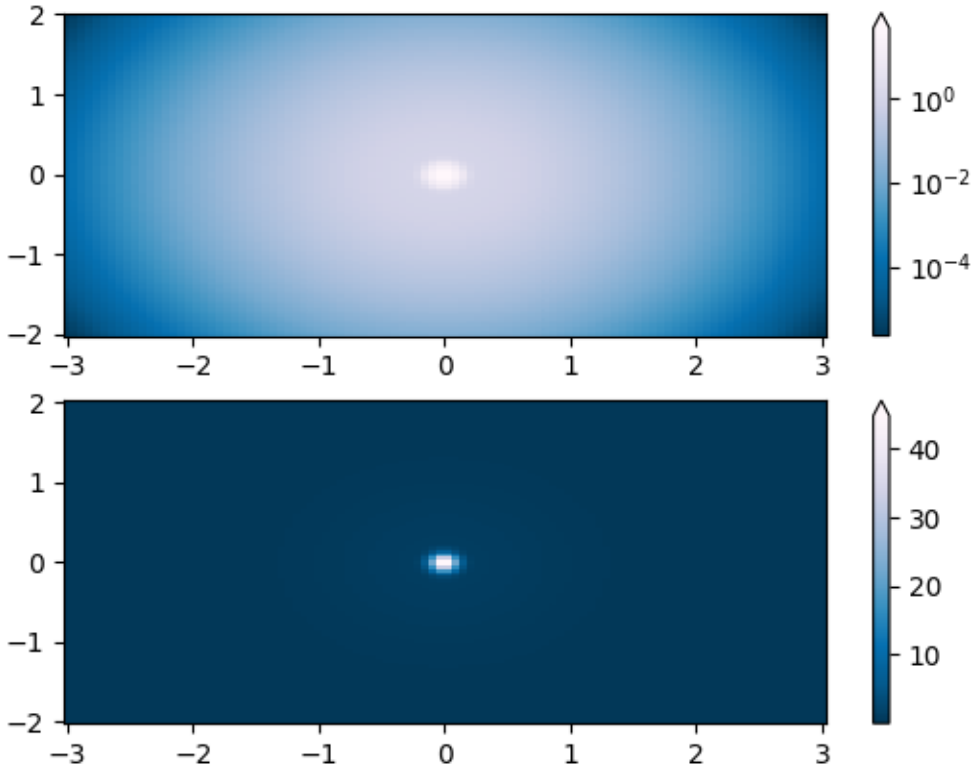
One of the most common transformations is to plot data by taking its logarithm (to the base-10). This transformation is useful to display changes across disparate scales. Using *colors.LogNorm* normalizes the data via $log_{10}$. In the example below, there are two bumps, one much smaller than the other. Using *colors.LogNorm*, the shape and location of each bump can clearly be seen:

```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib import cm
```

(continues on next page)

(continued from previous page)

```
import matplotlib.cbook as cbook
import matplotlib.colors as colors

N = 100
X, Y = np.mgrid[-3:3:complex(0, N), -2:2:complex(0, N)]

# A low hump with a spike coming out of the top right.  Needs to have
# z/colour axis on a log scale, so we see both hump and spike. A linear
# scale only shows the spike.
Z1 = np.exp(-X**2 - Y**2)
Z2 = np.exp(-(X * 10)**2 - (Y * 10)**2)
Z = Z1 + 50 * Z2

fig, ax = plt.subplots(2, 1)

pcm = ax[0].pcolor(X, Y, Z,
                   norm=colors.LogNorm(vmin=Z.min(), vmax=Z.max()),
                   cmap='PuBu_r', shading='auto')
fig.colorbar(pcm, ax=ax[0], extend='max')

pcm = ax[1].pcolor(X, Y, Z, cmap='PuBu_r', shading='auto')
fig.colorbar(pcm, ax=ax[1], extend='max')
plt.show()
```




## Centered

In many cases, data is symmetrical around a center, for example, positive and negative anomalies around a center 0. In this case, we would like the center to be mapped to 0.5 and the datapoint with the largest deviation from the center to be mapped to 1.0, if its value is greater than the center, or 0.0 otherwise. The norm *colors.CenteredNorm* creates such a mapping automatically. It is well suited to be combined with a divergent colormap which uses different colors edges that meet in the center at an unsaturated color.

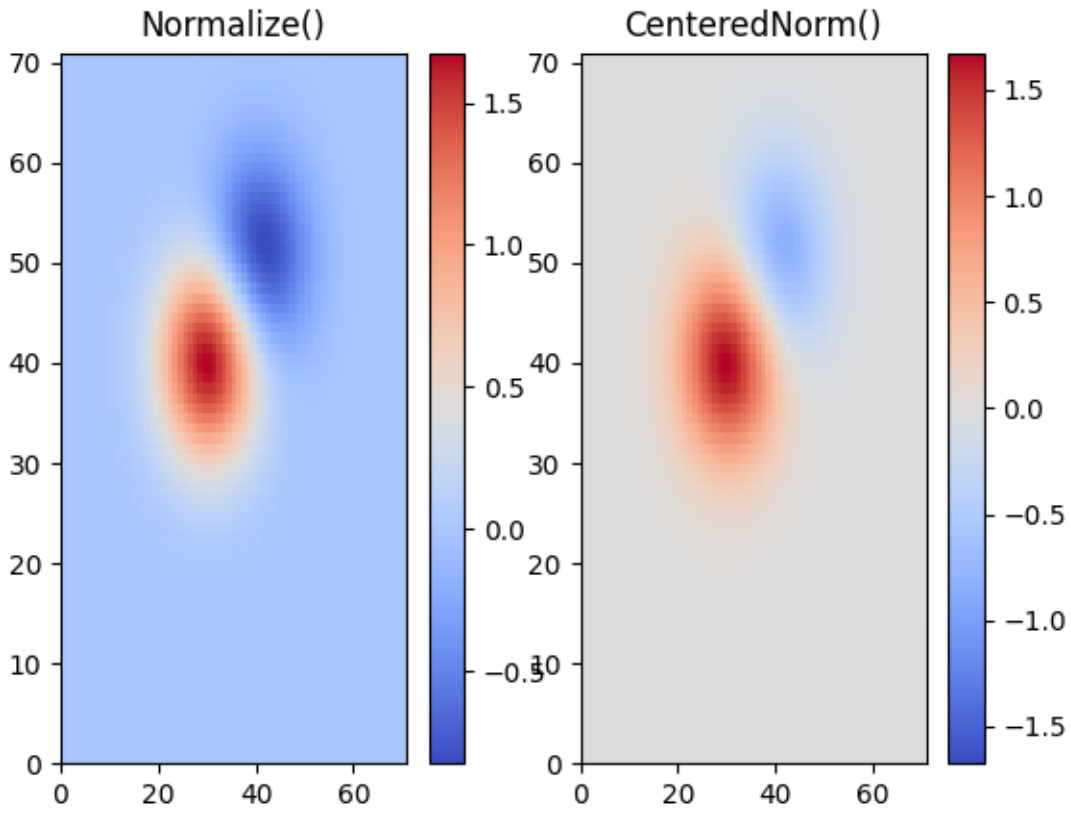
If the center of symmetry is different from 0, it can be set with the *vcenter* argument. For logarithmic scaling on both sides of the center, see *colors.SymLogNorm* below; to apply a different mapping above and below the center, use *colors.TwoSlopeNorm* below.

```
delta = 0.1
x = np.arange(-3.0, 4.001, delta)
y = np.arange(-4.0, 3.001, delta)
X, Y = np.meshgrid(x, y)
Z1 = np.exp(-X**2 - Y**2)
Z2 = np.exp(-(X - 1)**2 - (Y - 1)**2)
Z = (0.9*Z1 - 0.5*Z2) * 2

# select a divergent colormap
```

(continues on next page)

(continued from previous page)

```
cmap = cm.coolwarm

fig, (ax1, ax2) = plt.subplots(ncols=2)
pc = ax1.pcolormesh(Z, cmap=cmap)
fig.colorbar(pc, ax=ax1)
ax1.set_title('Normalize()')

pc = ax2.pcolormesh(Z, norm=colors.CenteredNorm(), cmap=cmap)
fig.colorbar(pc, ax=ax2)
ax2.set_title('CenteredNorm()')

plt.show()
```

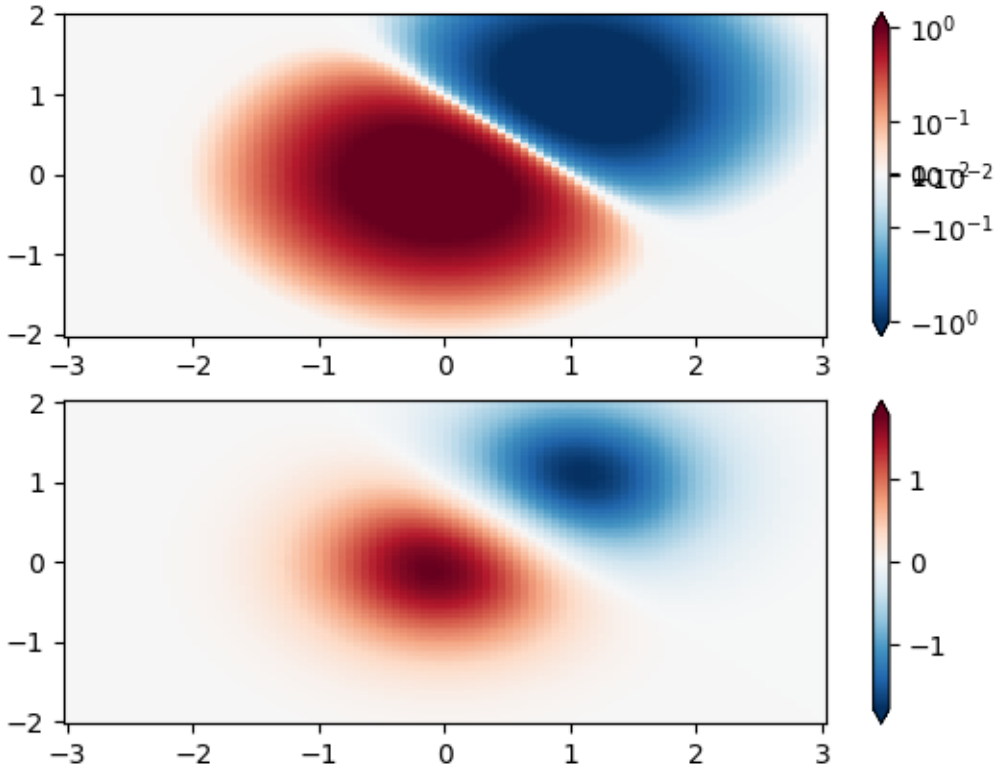


### Symmetric logarithmic

Similarly, it sometimes happens that there is data that is positive and negative, but we would still like a logarithmic scaling applied to both. In this case, the negative numbers are also scaled logarithmically, and mapped to smaller numbers; e.g., if `vmin=-vmax`, then the negative numbers are mapped from 0 to 0.5 and the positive from 0.5 to 1.

Since the logarithm of values close to zero tends toward infinity, a small range around zero needs to be mapped linearly. The parameter *linthresh* allows the user to specify the size of this range (*-linthresh*, *linthresh*). The size of this range in the colormap is set by *linscale*. When *linscale* == 1.0 (the default), the space used for the positive and negative halves of the linear range will be equal to one decade in the logarithmic range.

```
N = 100
X, Y = np.mgrid[-3:3:complex(0, N), -2:2:complex(0, N)]
Z1 = np.exp(-X**2 - Y**2)
Z2 = np.exp(-(X - 1)**2 - (Y - 1)**2)
Z = (Z1 - Z2) * 2

fig, ax = plt.subplots(2, 1)

pcm = ax[0].pcolormesh(X, Y, Z,
                       norm=colors.SymLogNorm(linthresh=0.03, linscale=0.03,
                                              vmin=-1.0, vmax=1.0, base=10),
                       cmap='RdBu_r', shading='auto')
fig.colorbar(pcm, ax=ax[0], extend='both')

pcm = ax[1].pcolormesh(X, Y, Z, cmap='RdBu_r', vmin=-np.max(Z), shading='auto
 ↪')
fig.colorbar(pcm, ax=ax[1], extend='both')
plt.show()
```




### Power-law

Sometimes it is useful to remap the colors onto a power-law relationship (i.e. $y = x^\gamma$, where $\gamma$ is the power). For this we use the *colors.PowerNorm*. It takes as an argument *gamma* (*gamma* == 1.0 will just yield the default linear normalization):

---

**Note:** There should probably be a good reason for plotting the data using this type of transformation. Technical viewers are used to linear and logarithmic axes and data transformations. Power laws are less common, and viewers should explicitly be made aware that they have been used.

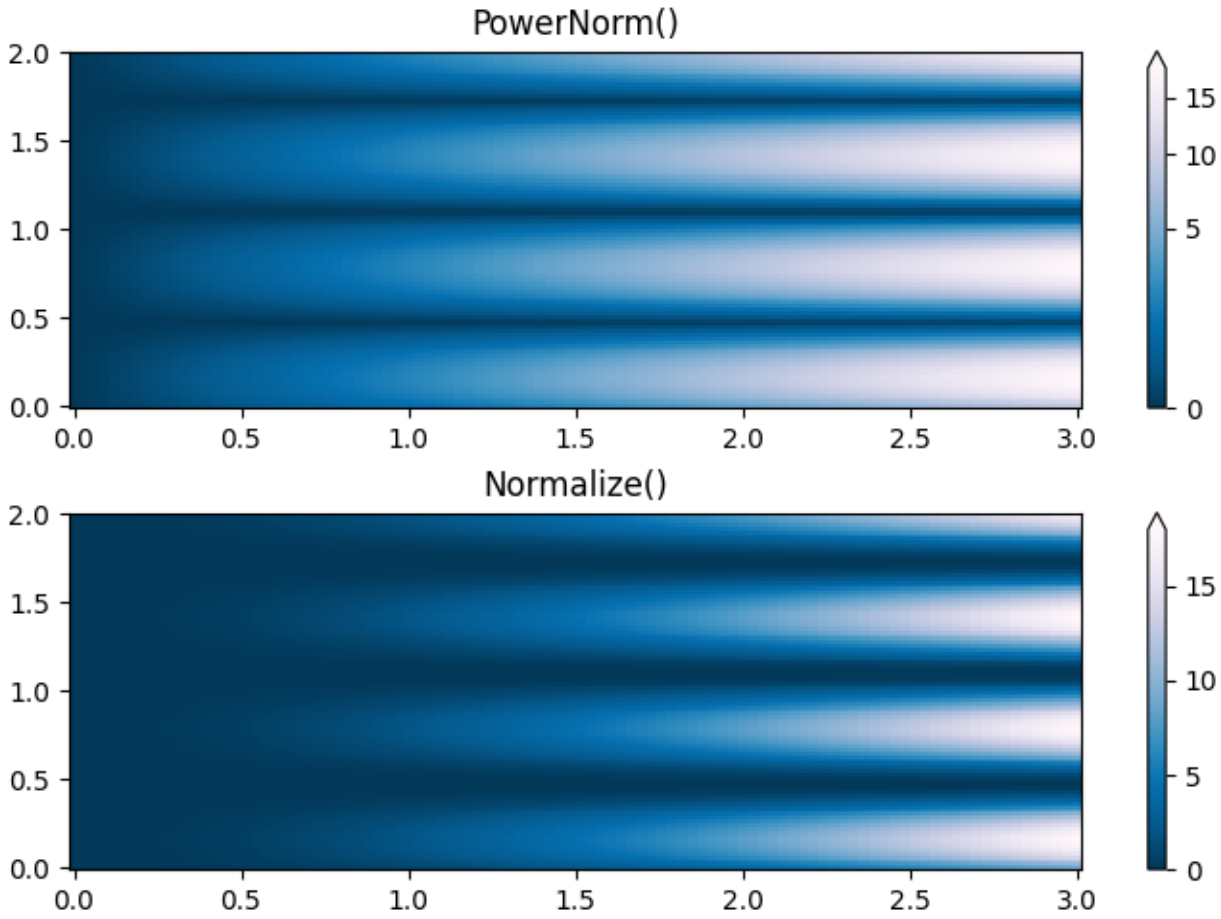
---

```
N = 100
X, Y = np.mgrid[0:3:complex(0, N), 0:2:complex(0, N)]
Z1 = (1 + np.sin(Y * 10.)) * X**2

fig, ax = plt.subplots(2, 1, layout='constrained')

pcm = ax[0].pcolormesh(X, Y, Z1, norm=colors.PowerNorm(gamma=0.5),
                       cmap='PuBu_r', shading='auto')
fig.colorbar(pcm, ax=ax[0], extend='max')
```

(continues on next page)

(continued from previous page)

```
ax[0].set_title('PowerNorm()')

pcm = ax[1].pcolormesh(X, Y, Z1, cmap='PuBu_r', shading='auto')
fig.colorbar(pcm, ax=ax[1], extend='max')
ax[1].set_title('Normalize()')
plt.show()
```



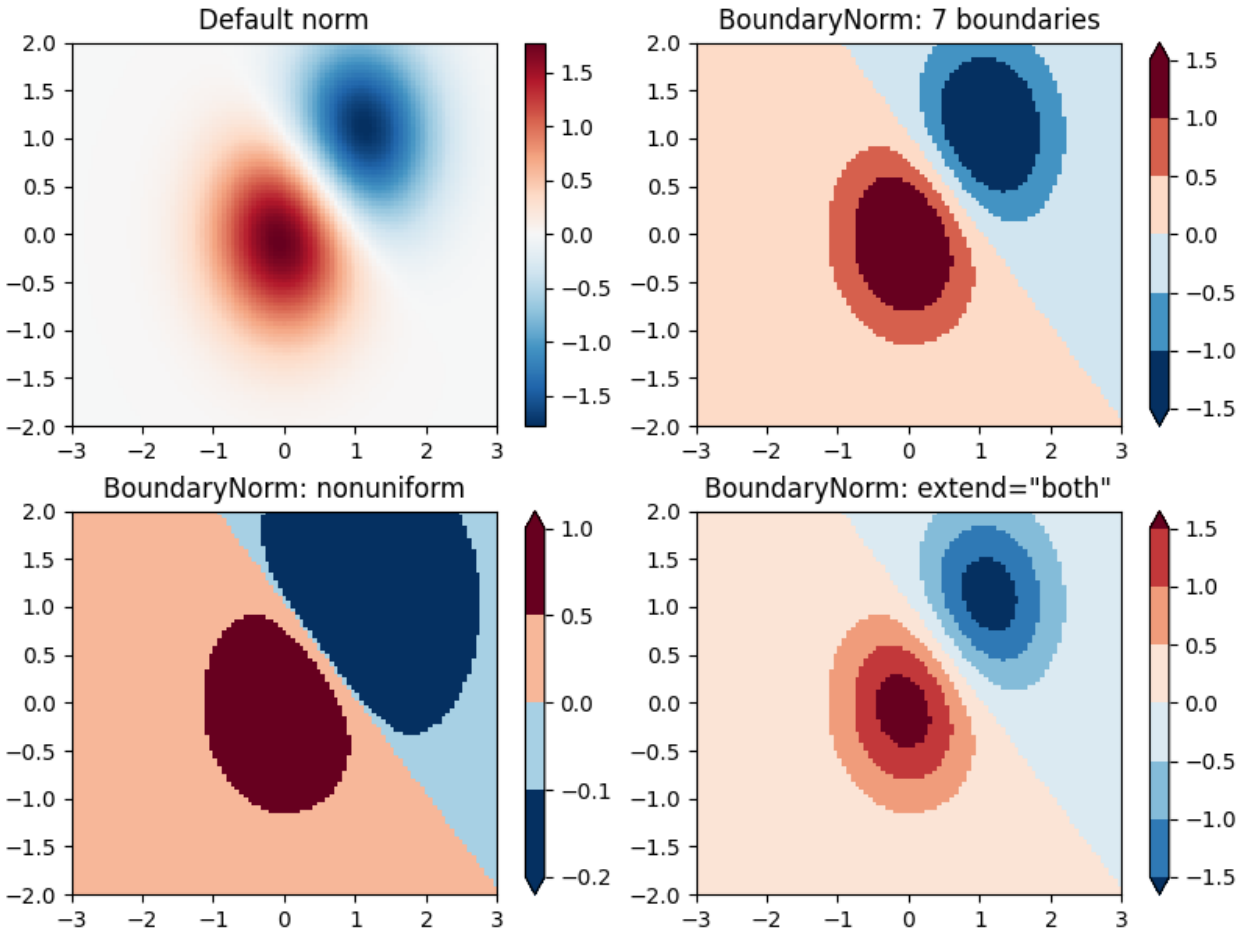

### Discrete bounds

Another normalization that comes with Matplotlib is *colors.BoundaryNorm*. In addition to *vmin* and *vmax*, this takes as arguments boundaries between which data is to be mapped. The colors are then linearly distributed between these "bounds". It can also take an *extend* argument to add upper and/or lower out-of-bounds values to the range over which the colors are distributed. For instance:

```
>>> import matplotlib.colors as colors
>>> bounds = np.array([-0.25, -0.125, 0, 0.5, 1])
>>> norm = colors.BoundaryNorm(boundaries=bounds, ncolors=4)
>>> print(norm([-0.2, -0.15, -0.02, 0.3, 0.8, 0.99]))
[0 0 1 2 3 3]
```

Note: Unlike the other norms, this norm returns values from 0 to *ncolors*-1.

```
N = 100
X, Y = np.meshgrid(np.linspace(-3, 3, N), np.linspace(-2, 2, N))
Z1 = np.exp(-X**2 - Y**2)
Z2 = np.exp(-(X - 1)**2 - (Y - 1)**2)
Z = ((Z1 - Z2) * 2)[:-1, :-1]

fig, ax = plt.subplots(2, 2, figsize=(8, 6), layout='constrained')
ax = ax.flatten()

# Default norm:
pcm = ax[0].pcolormesh(X, Y, Z, cmap='RdBu_r')
fig.colorbar(pcm, ax=ax[0], orientation='vertical')
ax[0].set_title('Default norm')

# Even bounds give a contour-like effect:
bounds = np.linspace(-1.5, 1.5, 7)
norm = colors.BoundaryNorm(boundaries=bounds, ncolors=256)
pcm = ax[1].pcolormesh(X, Y, Z, norm=norm, cmap='RdBu_r')
fig.colorbar(pcm, ax=ax[1], extend='both', orientation='vertical')
ax[1].set_title('BoundaryNorm: 7 boundaries')

# Bounds may be unevenly spaced:
bounds = np.array([-0.2, -0.1, 0, 0.5, 1])
norm = colors.BoundaryNorm(boundaries=bounds, ncolors=256)
pcm = ax[2].pcolormesh(X, Y, Z, norm=norm, cmap='RdBu_r')
fig.colorbar(pcm, ax=ax[2], extend='both', orientation='vertical')
ax[2].set_title('BoundaryNorm: nonuniform')

# With out-of-bounds colors:
bounds = np.linspace(-1.5, 1.5, 7)
norm = colors.BoundaryNorm(boundaries=bounds, ncolors=256, extend='both')
pcm = ax[3].pcolormesh(X, Y, Z, norm=norm, cmap='RdBu_r')
# The colorbar inherits the "extend" argument from BoundaryNorm.
fig.colorbar(pcm, ax=ax[3], orientation='vertical')
ax[3].set_title('BoundaryNorm: extend="both"')
plt.show()
```




### TwoSlopeNorm: Different mapping on either side of a center

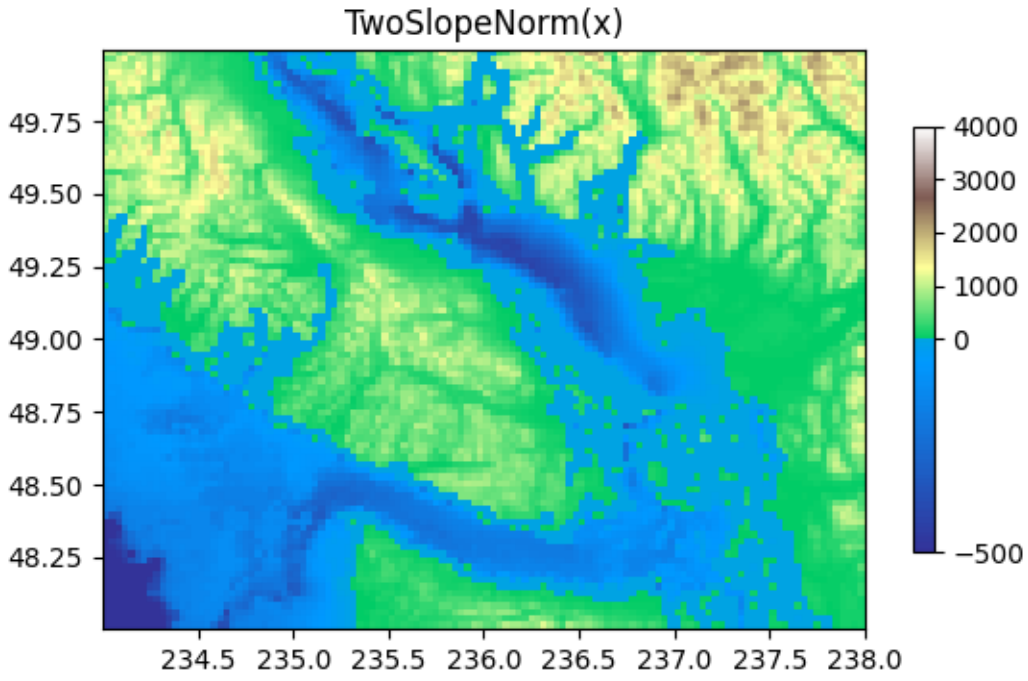
Sometimes we want to have a different colormap on either side of a conceptual center point, and we want those two colormaps to have different linear scales. An example is a topographic map where the land and ocean have a center at zero, but land typically has a greater elevation range than the water has depth range, and they are often represented by a different colormap.

```
dem = cbook.get_sample_data('topobathy.npz')
topo = dem['topo']
longitude = dem['longitude']
latitude = dem['latitude']

fig, ax = plt.subplots()
# make a colormap that has land and ocean clearly delineated and of the
# same length (256 + 256)
colors_undersea = plt.cm.terrain(np.linspace(0, 0.17, 256))
colors_land = plt.cm.terrain(np.linspace(0.25, 1, 256))
all_colors = np.vstack((colors_undersea, colors_land))
terrain_map = colors.LinearSegmentedColormap.from_list(
    'terrain_map', all_colors)
```

(continues on next page)

(continued from previous page)

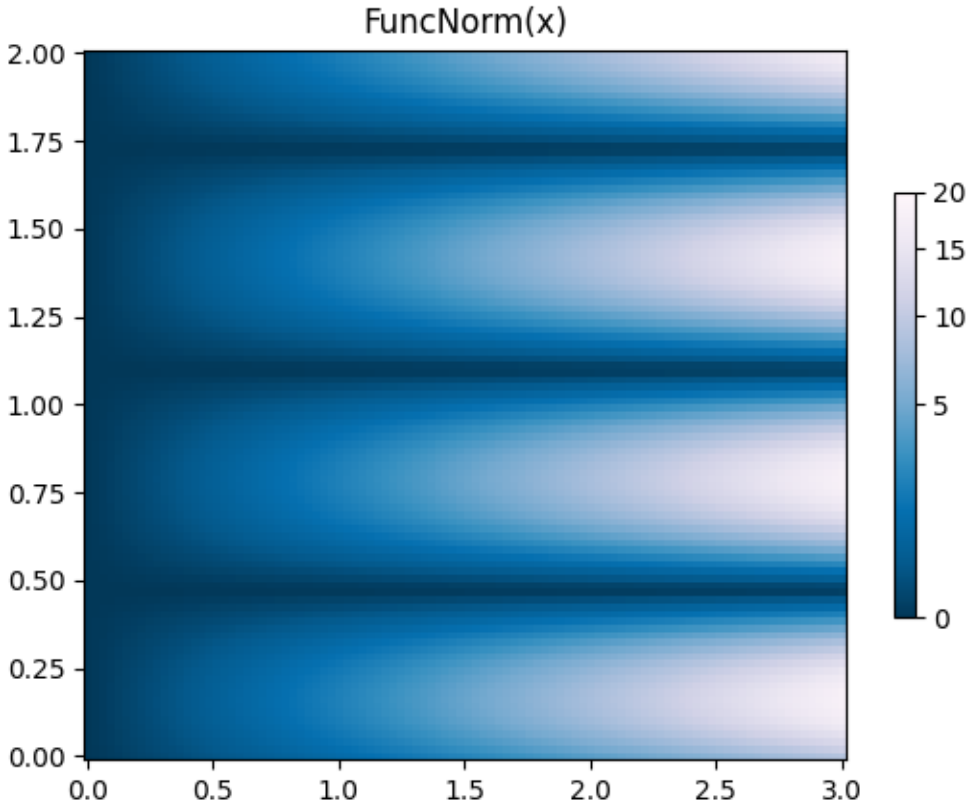
```
# make the norm:  Note the center is offset so that the land has more
# dynamic range:
divnorm = colors.TwoSlopeNorm(vmin=-500., vcenter=0, vmax=4000)

pcm = ax.pcolormesh(longitude, latitude, topo, rasterized=True, norm=divnorm,
                    cmap=terrain_map, shading='auto')
# Simple geographic plot, set aspect ratio because distance between lines of
# longitude depends on latitude.
ax.set_aspect(1 / np.cos(np.deg2rad(49)))
ax.set_title('TwoSlopeNorm(x)')
cb = fig.colorbar(pcm, shrink=0.6)
cb.set_ticks([-500, 0, 1000, 2000, 3000, 4000])
plt.show()
```



### FuncNorm: Arbitrary function normalization

If the above norms do not provide the normalization you want, you can use *FuncNorm* to define your own. Note that this example is the same as *PowerNorm* with a power of 0.5:

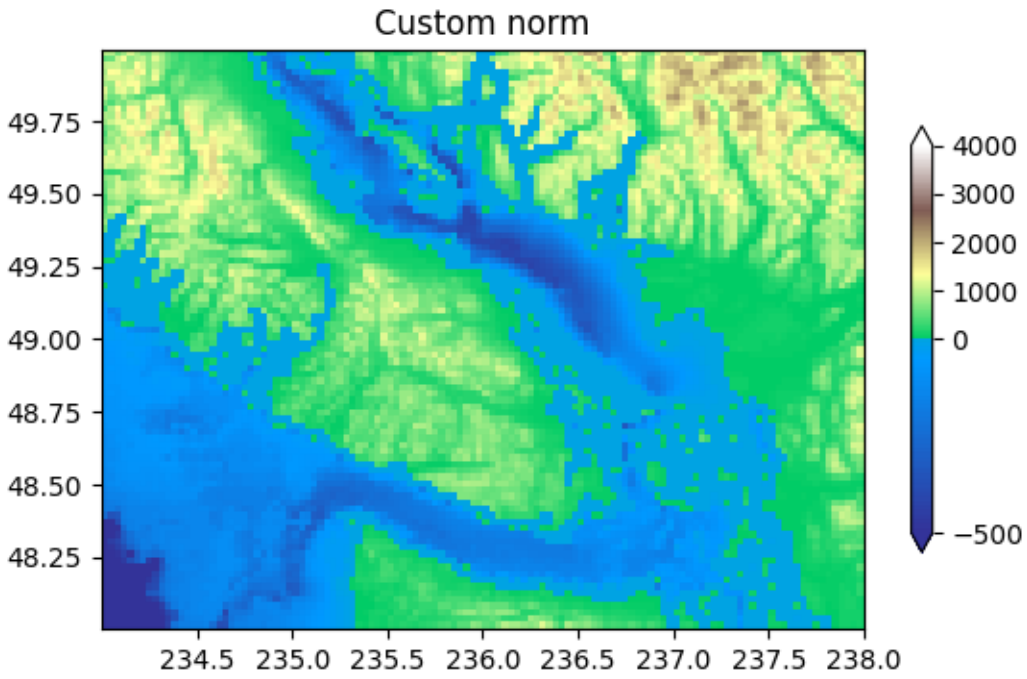
```
def _forward(x):
    return np.sqrt(x)


def _inverse(x):
    return x**2

N = 100
X, Y = np.mgrid[0:3:complex(0, N), 0:2:complex(0, N)]
Z1 = (1 + np.sin(Y * 10.)) * X**2
fig, ax = plt.subplots()

norm = colors.FuncNorm((_forward, _inverse), vmin=0, vmax=20)
pcm = ax.pcolormesh(X, Y, Z1, norm=norm, cmap='PuBu_r', shading='auto')
ax.set_title('FuncNorm(x)')
fig.colorbar(pcm, shrink=0.6)
plt.show()
```



### Custom normalization: Manually implement two linear ranges

The *TwoSlopeNorm* described above makes a useful example for defining your own norm. Note for the colorbar to work, you must define an inverse for your norm:

```
class MidpointNormalize(colors.Normalize):
    def __init__(self, vmin=None, vmax=None, vcenter=None, clip=False):
        self.vcenter = vcenter
        super().__init__(vmin, vmax, clip)

    def __call__(self, value, clip=None):
        # I'm ignoring masked values and all kinds of edge cases to make a
        # simple example...
        # Note also that we must extrapolate beyond vmin/vmax
        x, y = [self.vmin, self.vcenter, self.vmax], [0, 0.5, 1.]
        return np.ma.masked_array(np.interp(value, x, y,
                                            left=-np.inf, right=np.inf))

    def inverse(self, value):
        y, x = [self.vmin, self.vcenter, self.vmax], [0, 0.5, 1]
        return np.interp(value, x, y, left=-np.inf, right=np.inf)


fig, ax = plt.subplots()
midnorm = MidpointNormalize(vmin=-500., vcenter=0, vmax=4000)

pcm = ax.pcolormesh(longitude, latitude, topo, rasterized=True, norm=midnorm,
                    cmap=terrain_map, shading='auto')
ax.set_aspect(1 / np.cos(np.deg2rad(49)))
ax.set_title('Custom norm')
cb = fig.colorbar(pcm, shrink=0.6, extend='both')
cb.set_ticks([-500, 0, 1000, 2000, 3000, 4000])

plt.show()
```




**Total running time of the script:** (0 minutes 5.004 seconds)

### 3.6.5 Choosing Colormaps in Matplotlib

Matplotlib has a number of built-in colormaps accessible via *matplotlib.colormaps*. There are also external libraries that have many extra colormaps, which can be viewed in the Third-party colormaps section of the Matplotlib documentation. Here we briefly discuss how to choose between the many options. For help on creating your own colormaps, see *Creating Colormaps in Matplotlib*.

To get a list of all registered colormaps, you can do:

```
from matplotlib import colormaps
list(colormaps)
```

### Overview

The idea behind choosing a good colormap is to find a good representation in 3D colorspace for your data set. The best colormap for any given data set depends on many things including:

- Whether representing form or metric data ([Ware])
- Your knowledge of the data set (*e.g.*, is there a critical value from which the other values deviate?)
- If there is an intuitive color scheme for the parameter you are plotting
- If there is a standard in the field the audience may be expecting

For many applications, a perceptually uniform colormap is the best choice; i.e. a colormap in which equal steps in data are perceived as equal steps in the color space. Researchers have found that the human brain perceives changes in the lightness parameter as changes in the data much better than, for example, changes in hue. Therefore, colormaps which have monotonically increasing lightness through the colormap will be better interpreted by the viewer. Wonderful examples of perceptually uniform colormaps can be found in the Third-party colormaps section as well.

Color can be represented in 3D space in various ways. One way to represent color is using CIELAB. In CIELAB, color space is represented by lightness, $L^*$; red-green, $a^*$; and yellow-blue, $b^*$. The lightness parameter $L^*$ can then be used to learn more about how the matplotlib colormaps will be perceived by viewers.

An excellent starting resource for learning about human perception of colormaps is from [IBM].

### Classes of colormaps

Colormaps are often split into several categories based on their function (see, *e.g.*, [Moreland]):

1. Sequential: change in lightness and often saturation of color incrementally, often using a single hue; should be used for representing information that has ordering.
2. Diverging: change in lightness and possibly saturation of two different colors that meet in the middle at an unsaturated color; should be used when the information being plotted has a critical middle value, such as topography or when the data deviates around zero.
3. Cyclic: change in lightness of two different colors that meet in the middle and beginning/end at an unsaturated color; should be used for values that wrap around at the endpoints, such as phase angle, wind direction, or time of day.
4. Qualitative: often are miscellaneous colors; should be used to represent information which does not have ordering or relationships.

```
from colorspacious import cspace_converter

import matplotlib.pyplot as plt
import numpy as np

import matplotlib as mpl
```

First, we'll show the range of each colormap. Note that some seem to change more "quickly" than others.

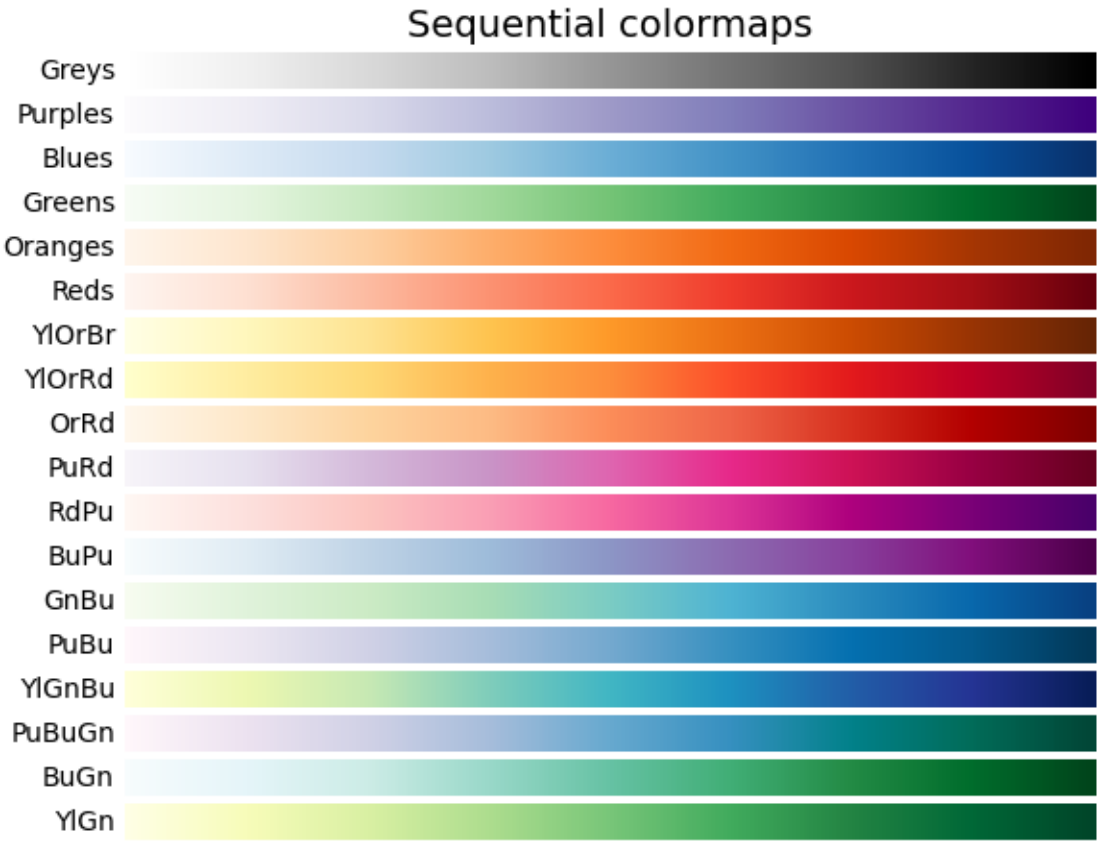
```
cmaps = {}

gradient = np.linspace(0, 1, 256)
gradient = np.vstack((gradient, gradient))


def plot_color_gradients(category, cmap_list):
    # Create figure and adjust figure height to number of colormaps
    nrows = len(cmap_list)
    figh = 0.35 + 0.15 + (nrows + (nrows - 1) * 0.1) * 0.22
    fig, axs = plt.subplots(nrows=nrows + 1, figsize=(6.4, figh))
    fig.subplots_adjust(top=1 - 0.35 / figh, bottom=0.15 / figh,
                        left=0.2, right=0.99)
    axs[0].set_title(f'{category} colormaps', fontsize=14)

    for ax, name in zip(axs, cmap_list):
        ax.imshow(gradient, aspect='auto', cmap=mpl.colormaps[name])
        ax.text(-0.01, 0.5, name, va='center', ha='right', fontsize=10,
                transform=ax.transAxes)

    # Turn off *all* ticks & spines, not just the ones with colormaps.
    for ax in axs:
        ax.set_axis_off()

    # Save colormap list for later.
    cmaps[category] = cmap_list
```

### Sequential

For the Sequential plots, the lightness value increases monotonically through the colormaps. This is good. Some of the $L^*$ values in the colormaps span from 0 to 100 (binary and the other grayscale), and others start around $L^* = 20$. Those that have a smaller range of $L^*$ will accordingly have a smaller perceptual range. Note also that the $L^*$ function varies amongst the colormaps: some are approximately linear in $L^*$ and others are more curved.

```
plot_color_gradients('Perceptually Uniform Sequential',
                     ['viridis', 'plasma', 'inferno', 'magma', 'cividis'])
```

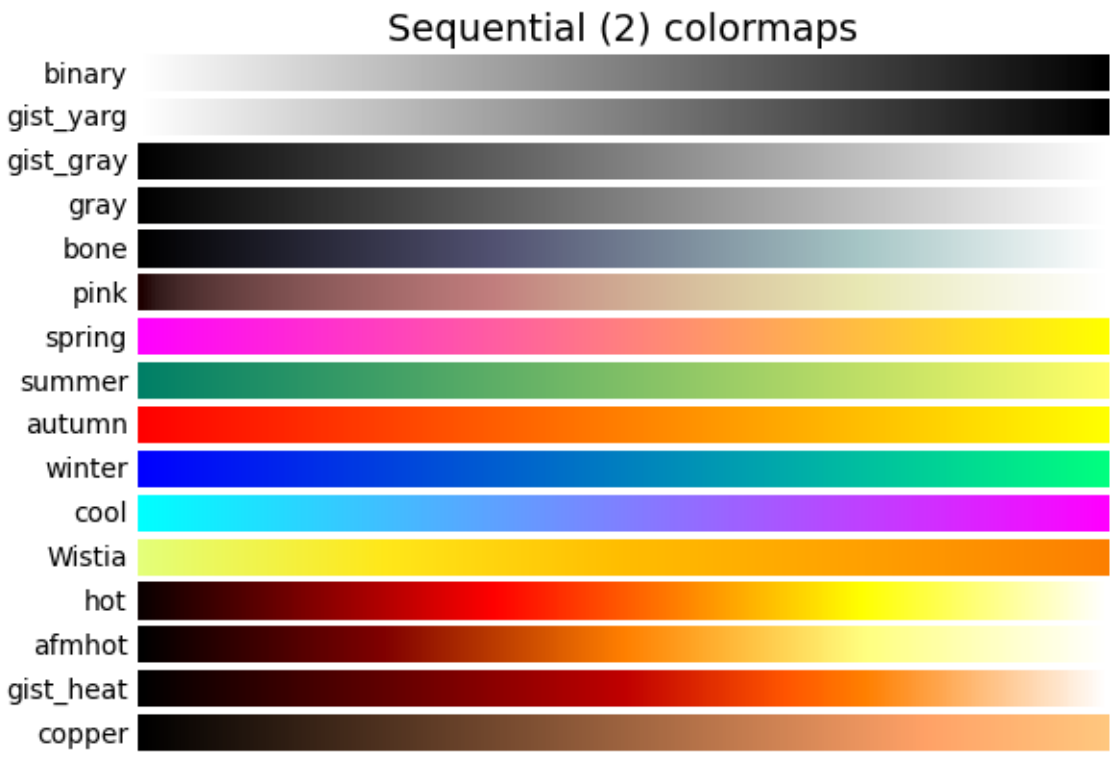
```
plot_color_gradients('Sequential',
                     ['Greys', 'Purples', 'Blues', 'Greens', 'Oranges', 'Reds
 ↪',
                      'YlOrBr', 'YlOrRd', 'OrRd', 'PuRd', 'RdPu', 'BuPu',
                      'GnBu', 'PuBu', 'YlGnBu', 'PuBuGn', 'BuGn', 'YlGn'])
```




### Sequential2

Many of the $L^*$ values from the Sequential2 plots are monotonically increasing, but some (autumn, cool, spring, and winter) plateau or even go both up and down in $L^*$ space. Others (afmhot, copper, gist_heat, and hot) have kinks in the $L^*$ functions. Data that is being represented in a region of the colormap that is at a plateau or kink will lead to a perception of banding of the data in those values in the colormap (see [mycarta-banding] for an excellent example of this).

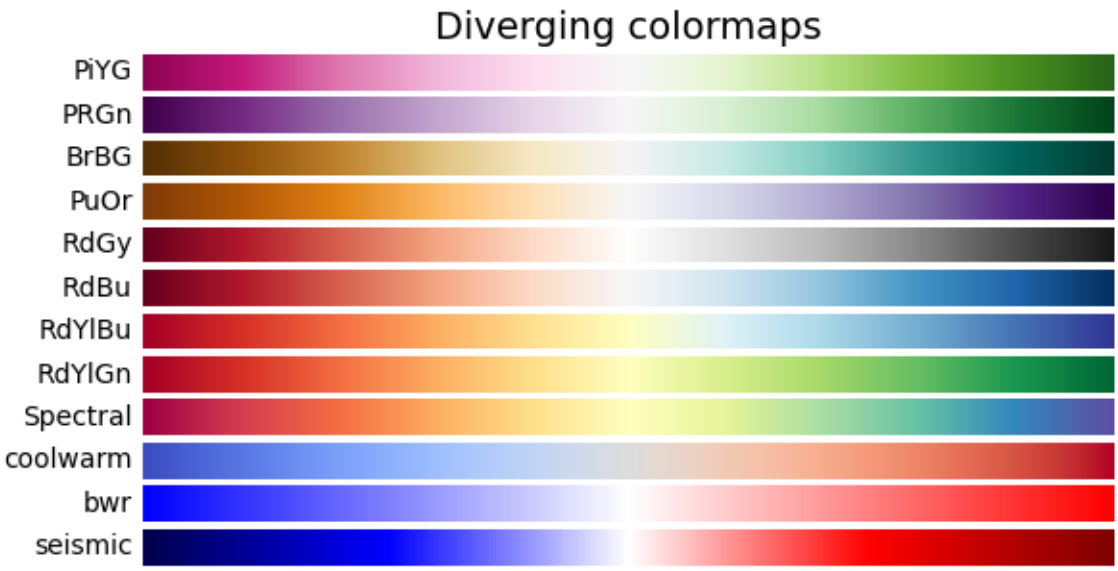
```
plot_color_gradients('Sequential (2)',
                     ['binary', 'gist_yarg', 'gist_gray', 'gray', 'bone',
                      'pink', 'spring', 'summer', 'autumn', 'winter', 'cool',
                      'Wistia', 'hot', 'afmhot', 'gist_heat', 'copper'])
```




### Diverging

For the Diverging maps, we want to have monotonically increasing $L^*$ values up to a maximum, which should be close to $L^* = 100$, followed by monotonically decreasing $L^*$ values. We are looking for approximately equal minimum $L^*$ values at opposite ends of the colormap. By these measures, BrBG and RdBu are good options. coolwarm is a good option, but it doesn't span a wide range of $L^*$ values (see grayscale section below).

```
plot_color_gradients('Diverging',
                     ['PiYG', 'PRGn', 'BrBG', 'PuOr', 'RdGy', 'RdBu', 'RdYlBu
 ↪',
                      'RdYlGn', 'Spectral', 'coolwarm', 'bwr', 'seismic'])
```

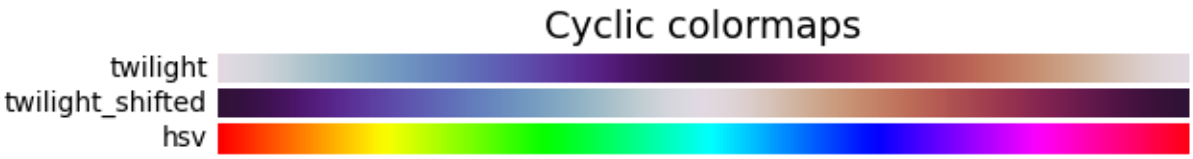



### Cyclic

For Cyclic maps, we want to start and end on the same color, and meet a symmetric center point in the middle. $L^*$ should change monotonically from start to middle, and inversely from middle to end. It should be symmetric on the increasing and decreasing side, and only differ in hue. At the ends and middle, $L^*$ will reverse direction, which should be smoothed in $L^*$ space to reduce artifacts. See [kovesi-colormaps] for more information on the design of cyclic maps.

The often-used HSV colormap is included in this set of colormaps, although it is not symmetric to a center point. Additionally, the $L^*$ values vary widely throughout the colormap, making it a poor choice for representing data for viewers to see perceptually. See an extension on this idea at [mycarta-jet].

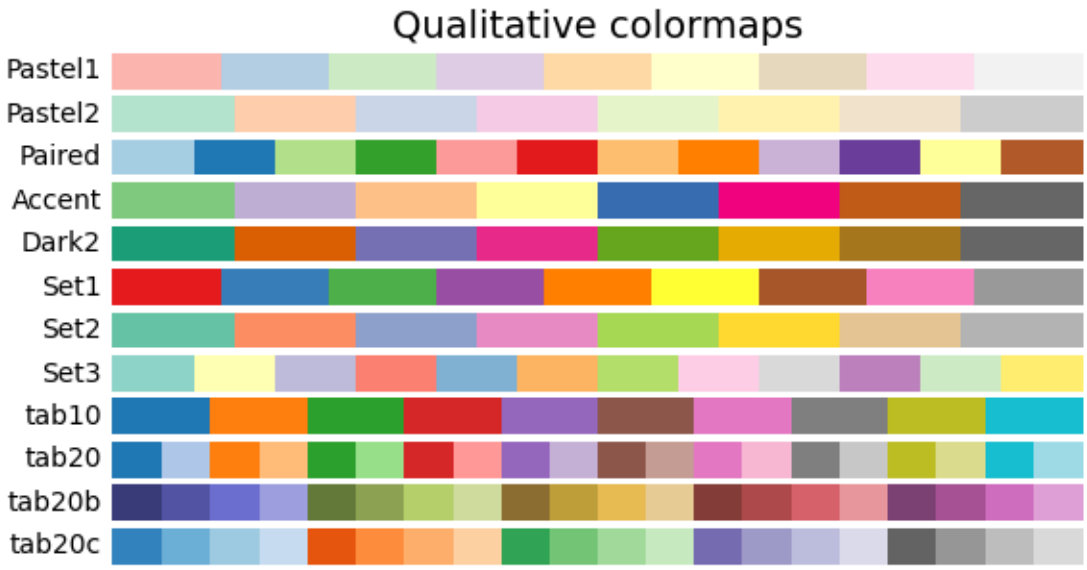
```
plot_color_gradients('Cyclic', ['twilight', 'twilight_shifted', 'hsv'])
```



### Qualitative

Qualitative colormaps are not aimed at being perceptual maps, but looking at the lightness parameter can verify that for us. The $L^*$ values move all over the place throughout the colormap, and are clearly not monotonically increasing. These would not be good options for use as perceptual colormaps.

```
plot_color_gradients('Qualitative',
                     ['Pastel1', 'Pastel2', 'Paired', 'Accent', 'Dark2',
                      'Set1', 'Set2', 'Set3', 'tab10', 'tab20', 'tab20b',
                      'tab20c'])
```




### Miscellaneous

Some of the miscellaneous colormaps have particular uses for which they have been created. For example, gist_earth, ocean, and terrain all seem to be created for plotting topography (green/brown) and water depths (blue) together. We would expect to see a divergence in these colormaps, then, but multiple kinks may not be ideal, such as in gist_earth and terrain. CMRmap was created to convert well to grayscale, though it does appear to have some small kinks in $L^*$. cubehelix was created to vary smoothly in both lightness and hue, but appears to have a small hump in the green hue area. turbo was created to display depth and disparity data.

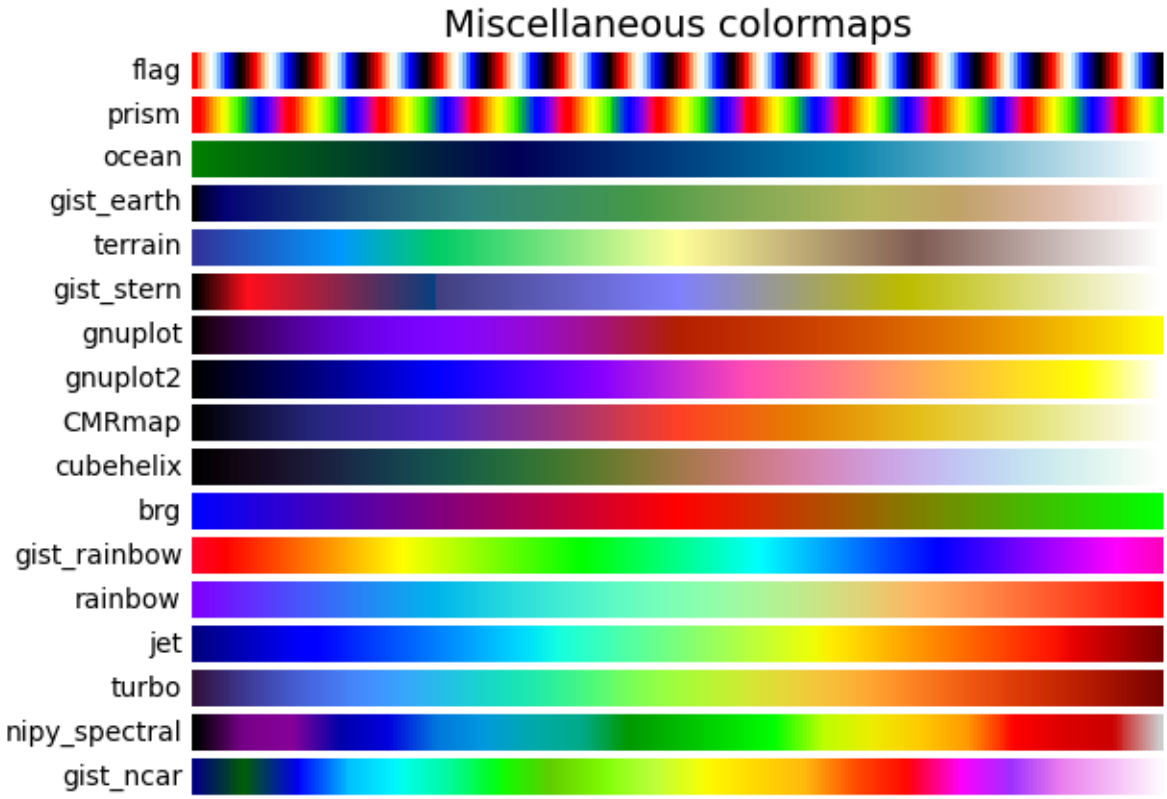
The often-used jet colormap is included in this set of colormaps. We can see that the $L^*$ values vary widely throughout the colormap, making it a poor choice for representing data for viewers to see perceptually. See an extension on this idea at [mycarta-jet] and [turbo].

```
plot_color_gradients('Miscellaneous',
                     ['flag', 'prism', 'ocean', 'gist_earth', 'terrain',
                      'gist_stern', 'gnuplot', 'gnuplot2', 'CMRmap',
```

(continues on next page)

(continued from previous page)

```
                      'cubehelix', 'brg', 'gist_rainbow', 'rainbow', 'jet',
                      'turbo', 'nipy_spectral', 'gist_ncar'])

plt.show()
```




## Lightness of Matplotlib colormaps

Here we examine the lightness values of the matplotlib colormaps. Note that some documentation on the colormaps is available ([list-colormaps]).

```
mpl.rcParams.update({'font.size': 12})

# Number of colormap per subplot for particular cmap categories
_DSUBS = {'Perceptually Uniform Sequential': 5, 'Sequential': 6,
          'Sequential (2)': 6, 'Diverging': 6, 'Cyclic': 3,
          'Qualitative': 4, 'Miscellaneous': 6}

# Spacing between the colormaps of a subplot
_DC = {'Perceptually Uniform Sequential': 1.4, 'Sequential': 0.7,
       'Sequential (2)': 1.4, 'Diverging': 1.4, 'Cyclic': 1.4,
       'Qualitative': 1.4, 'Miscellaneous': 1.4}
```

(continues on next page)

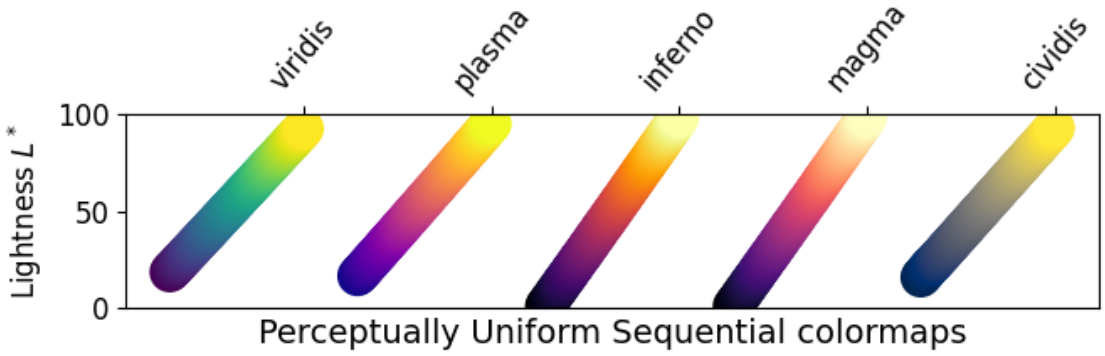
(continued from previous page)

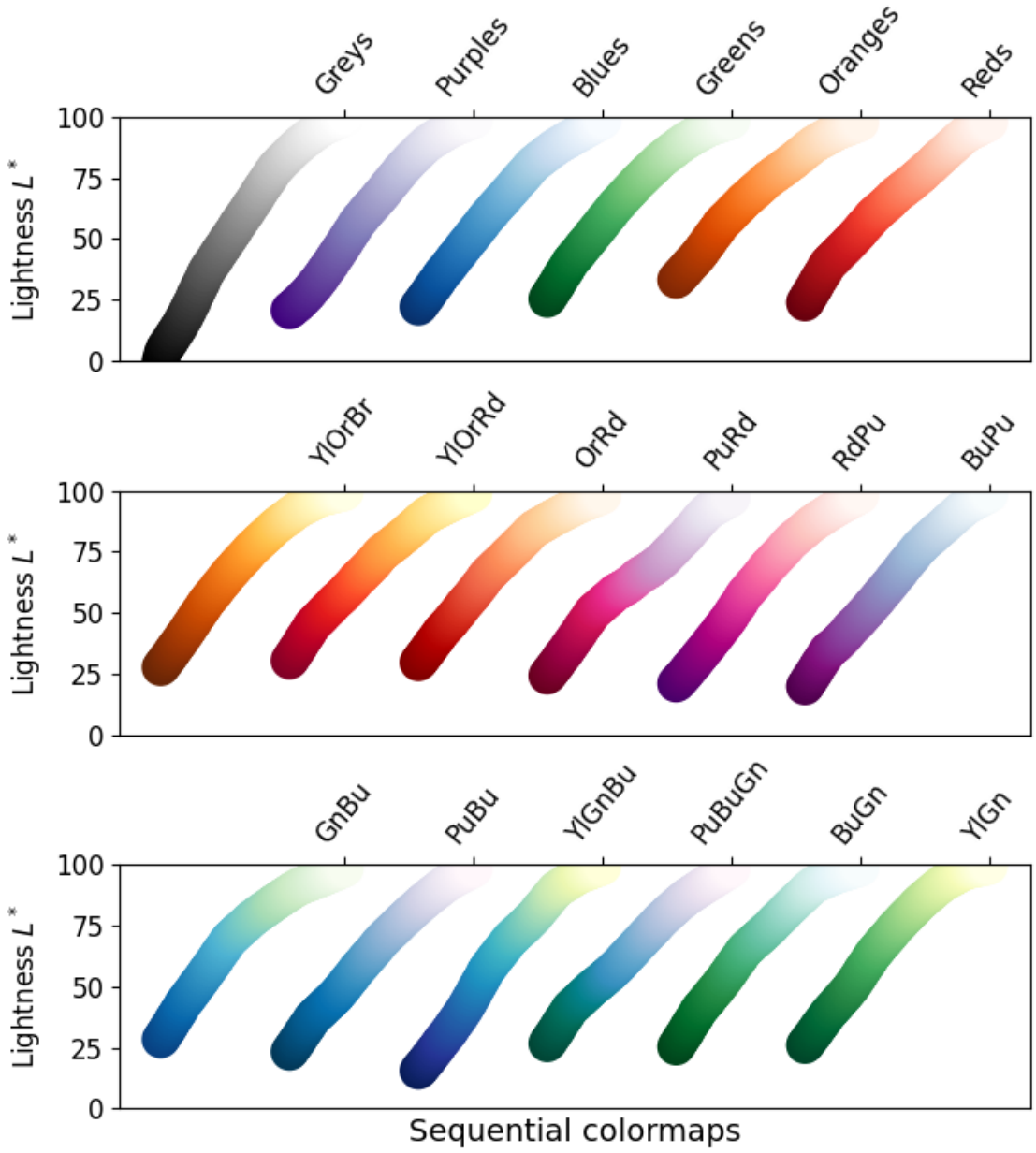
```
# Indices to step through colormap
x = np.linspace(0.0, 1.0, 100)

# Do plot
for cmap_category, cmap_list in cmaps.items():

    # Do subplots so that colormaps have enough space.
    # Default is 6 colormaps per subplot.
    dsub = _DSUBS.get(cmap_category, 6)
    nsubplots = int(np.ceil(len(cmap_list) / dsub))

    # squeeze=False to handle similarly the case of a single subplot
    fig, axs = plt.subplots(nrows=nsubplots, squeeze=False,
                            figsize=(7, 2.6*nsubplots))

    for i, ax in enumerate(axs.flat):

        locs = []  # locations for text labels

        for j, cmap in enumerate(cmap_list[i*dsub:(i+1)*dsub]):

            # Get RGB values for colormap and convert the colormap in
            # CAM02-UCS colorspace.  lab[0, :, 0] is the lightness.
            rgb = mpl.colormaps[cmap](x)[np.newaxis, :, :3]
            lab = cspace_converter("sRGB1", "CAM02-UCS")(rgb)

            # Plot colormap L values.  Do separately for each category
            # so each plot can be pretty.  To make scatter markers change
            # color along plot:
            # https://stackoverflow.com/q/8202605/

            if cmap_category == 'Sequential':
                # These colormaps all start at high lightness, but we want␣
 ↪them
                # reversed to look nice in the plot, so reverse the order.
                y_ = lab[0, ::-1, 0]
                c_ = x[::-1]
            else:
                y_ = lab[0, :, 0]
                c_ = x

            dc = _DC.get(cmap_category, 1.4)  # cmaps horizontal spacing
            ax.scatter(x + j*dc, y_, c=c_, cmap=cmap, s=300, linewidths=0.0)

            # Store locations for colormap labels
            if cmap_category in ('Perceptually Uniform Sequential',
                                 'Sequential'):
                locs.append(x[-1] + j*dc)
            elif cmap_category in ('Diverging', 'Qualitative', 'Cyclic',
                                   'Miscellaneous', 'Sequential (2)'):
                locs.append(x[int(x.size/2.)] + j*dc)
```

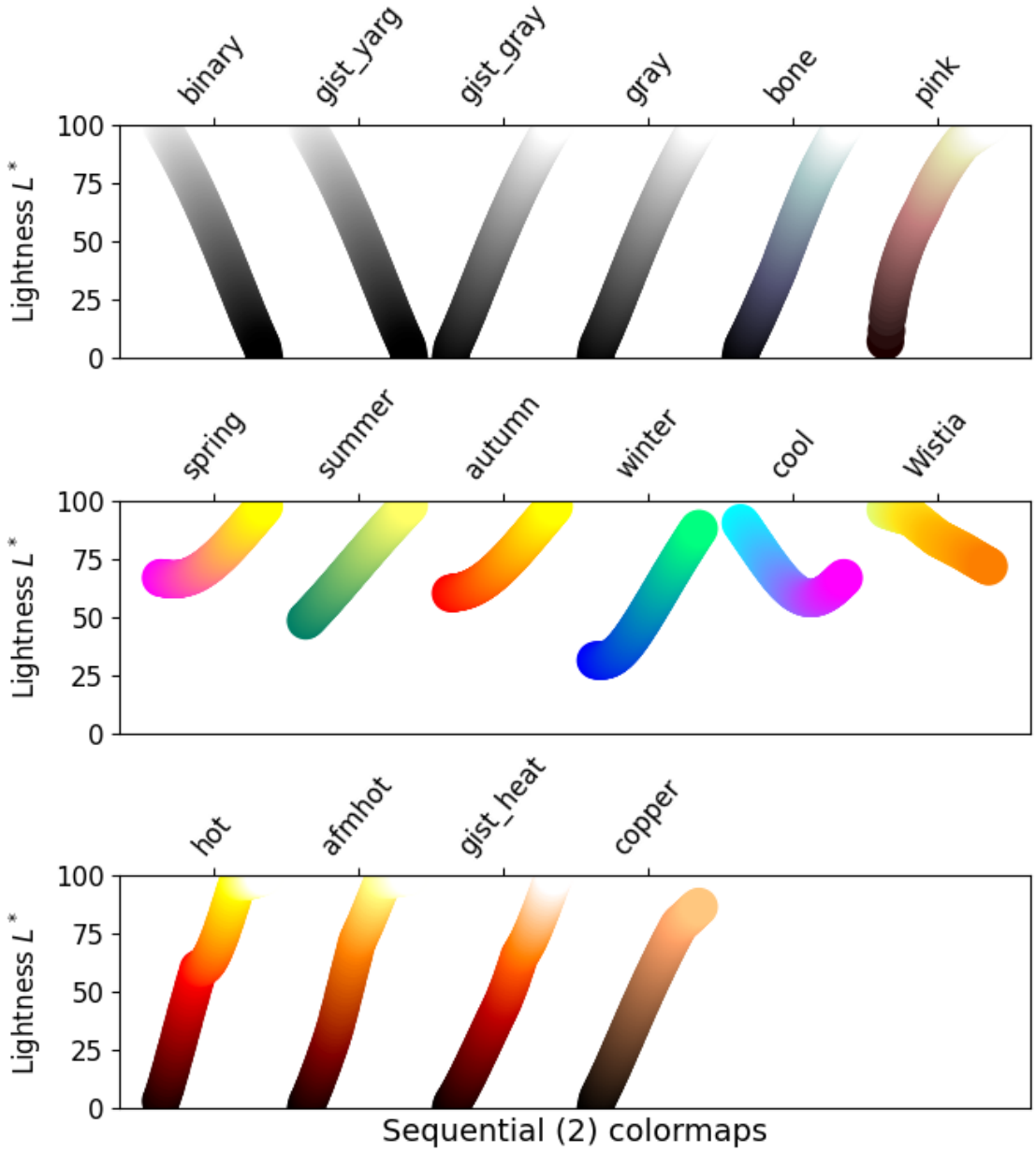
(continues on next page)

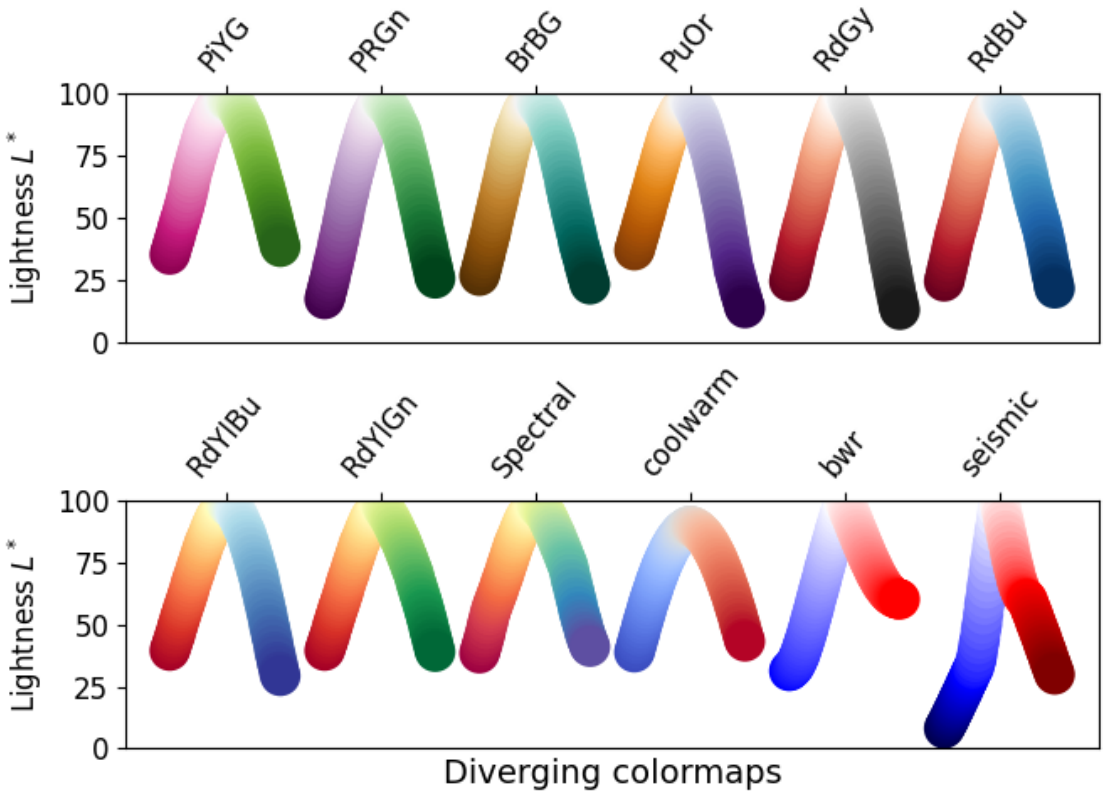
(continued from previous page)

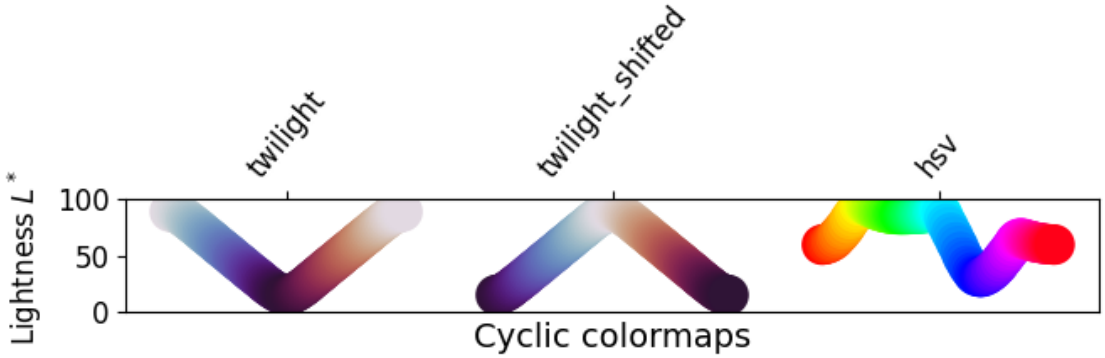
```
        # Set up the axis limits:
        #   * the 1st subplot is used as a reference for the x-axis limits
        #   * lightness values goes from 0 to 100 (y-axis limits)
        ax.set_xlim(axs[0, 0].get_xlim())
        ax.set_ylim(0.0, 100.0)

        # Set up labels for colormaps
        ax.xaxis.set_ticks_position('top')
        ticker = mpl.ticker.FixedLocator(locs)
        ax.xaxis.set_major_locator(ticker)
        formatter = mpl.ticker.FixedFormatter(cmap_list[i*dsub:(i+1)*dsub])
        ax.xaxis.set_major_formatter(formatter)
        ax.xaxis.set_tick_params(rotation=50)
        ax.set_ylabel('Lightness $L^*$', fontsize=12)

    ax.set_xlabel(cmap_category + ' colormaps', fontsize=14)

    fig.tight_layout(h_pad=0.0, pad=1.5)
    plt.show()
```

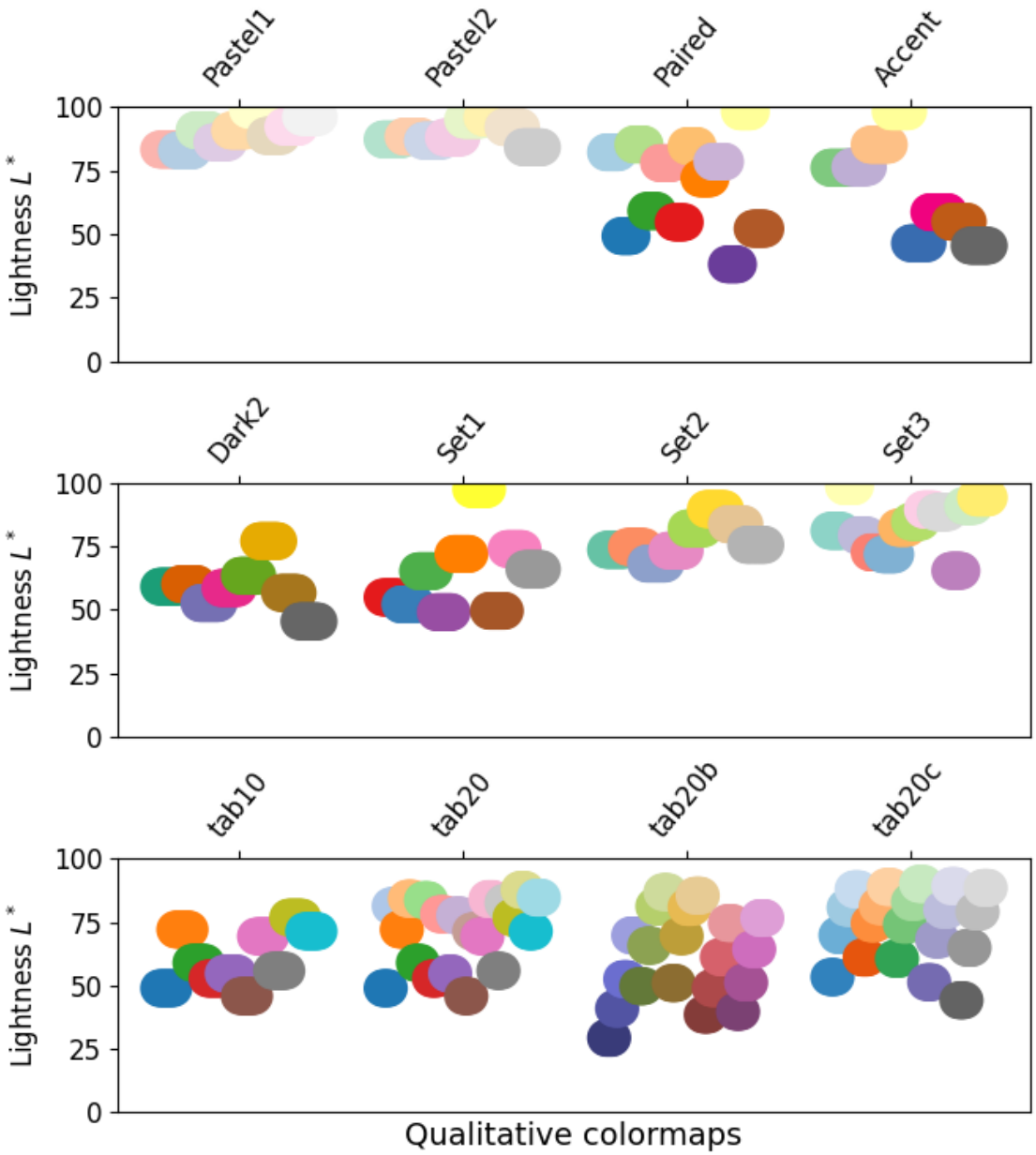


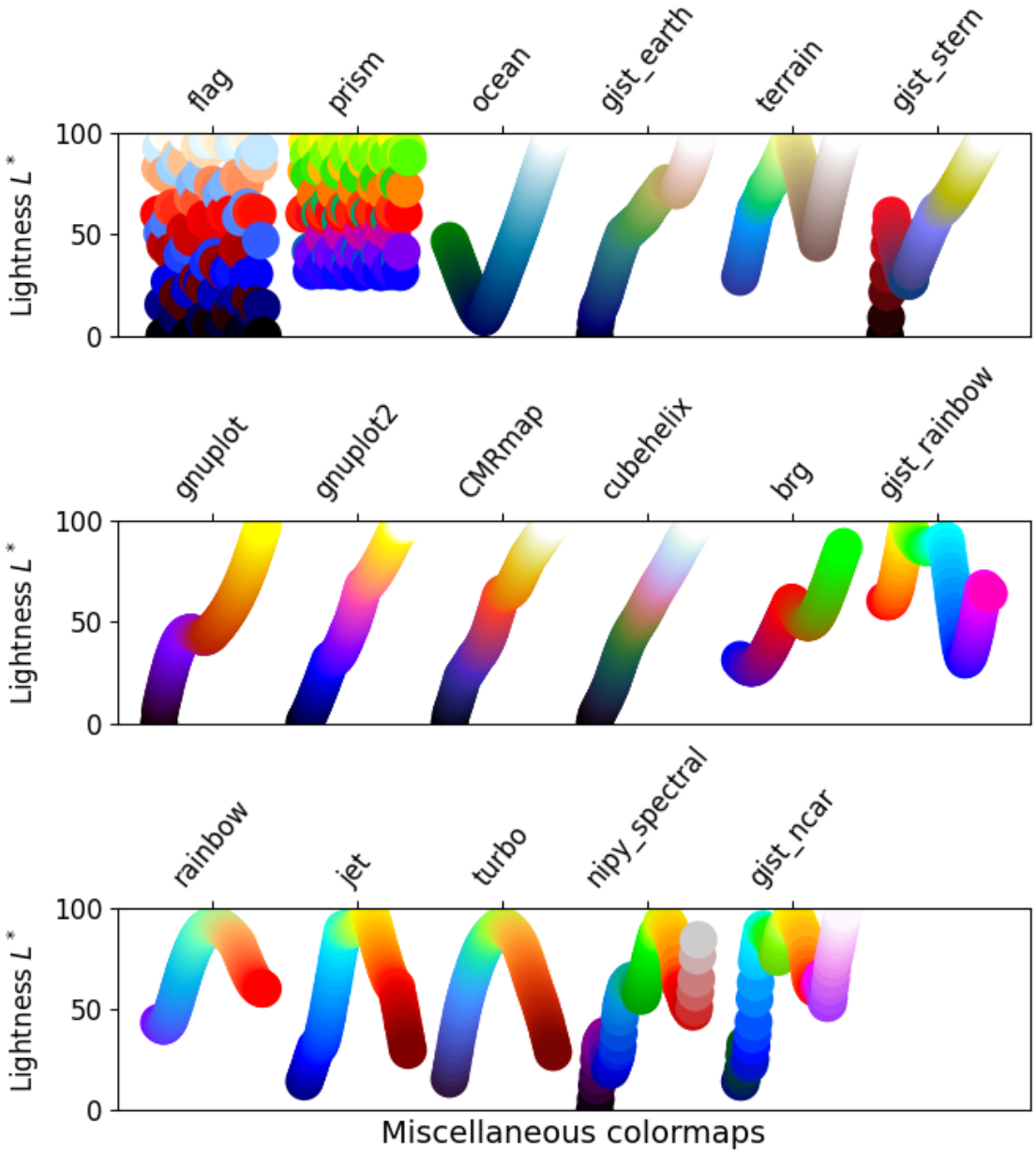

Greys
Purples
Blues
Greens
Oranges
Reds
YlOrBr
YlOrRd
OrRd
PuRd
RdPu
BuPu
GnBu
PuBu
YlGnBu
PuBuGn
BuGn
YlGn
Lightness $L^*$
100
75
50
25
0
Sequential colormaps


binary
gist_yarg
gist_gray
gray
bone
pink
spring
summer
autumn
winter
cool
Wistia
hot
afmhot
gist_heat
copper
Lightness $L^*$
100
75
50
25
0
Sequential (2) colormaps

- 
PiYG
PRGn
BrBG
PuOr
RdGy
RdBu
RdYlBu
RdYlGn
Spectral
coolwarm
bwr
seismic
Lightness $L^*$
100
75
50
25
0
Diverging colormaps

- 
twilight
twilight_shifted
hsv
Lightness $L^*$
100
50
0
Cyclic colormaps


Pastel1
Pastel2
Paired
Accent
Dark2
Set1
Set2
Set3
tab10
tab20
tab20b
tab20c
Lightness $L^*$
100
75
50
25
0
Qualitative colormaps




- 

### Grayscale conversion

It is important to pay attention to conversion to grayscale for color plots, since they may be printed on black and white printers. If not carefully considered, your readers may end up with indecipherable plots because the grayscale changes unpredictably through the colormap.

Conversion to grayscale is done in many different ways [bw]. Some of the better ones use a linear combination of the rgb values of a pixel, but weighted according to how we perceive color intensity. A nonlinear method of conversion to grayscale is to use the $L^*$ values of the pixels. In general, similar principles apply for this question as they do for presenting one's information perceptually; that is, if a colormap is chosen that is

monotonically increasing in $L^*$ values, it will print in a reasonable manner to grayscale.

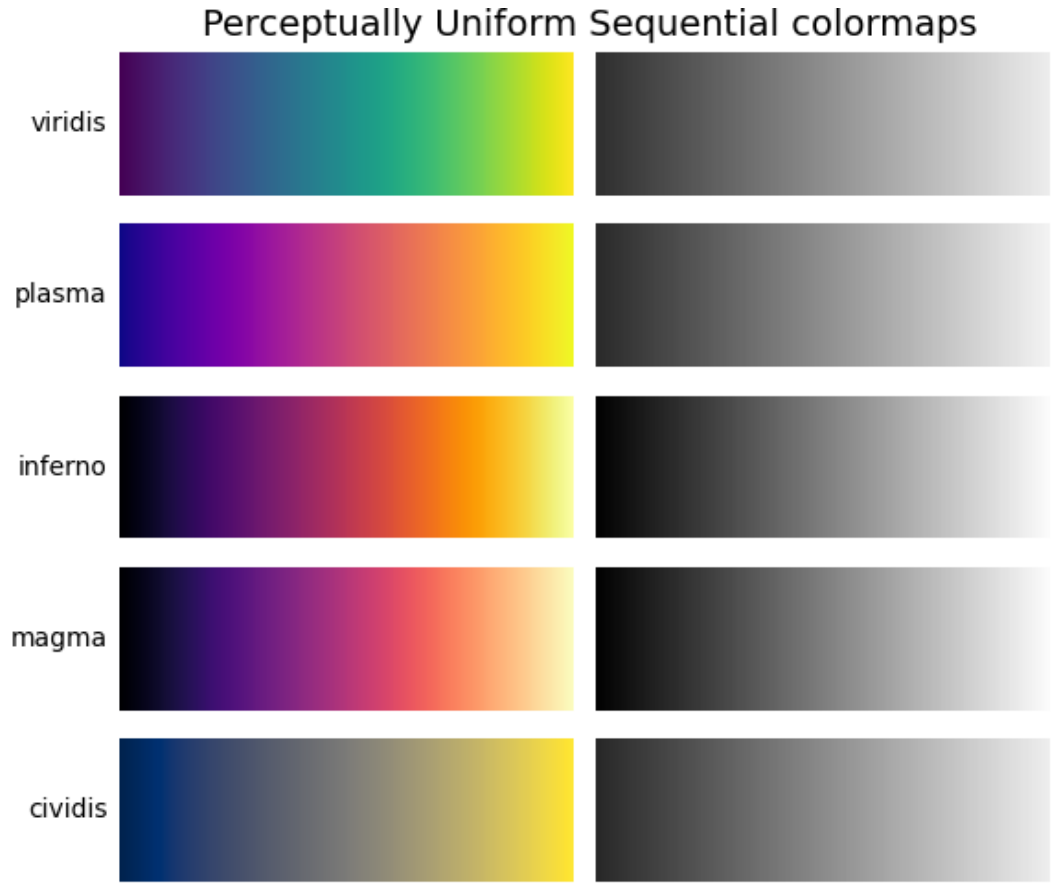
With this in mind, we see that the Sequential colormaps have reasonable representations in grayscale. Some of the Sequential2 colormaps have decent enough grayscale representations, though some (autumn, spring, summer, winter) have very little grayscale change. If a colormap like this was used in a plot and then the plot was printed to grayscale, a lot of the information may map to the same gray values. The Diverging colormaps mostly vary from darker gray on the outer edges to white in the middle. Some (PuOr and seismic) have noticeably darker gray on one side than the other and therefore are not very symmetric. coolwarm has little range of gray scale and would print to a more uniform plot, losing a lot of detail. Note that overlaid, labeled contours could help differentiate between one side of the colormap vs. the other since color cannot be used once a plot is printed to grayscale. Many of the Qualitative and Miscellaneous colormaps, such as Accent, hsv, jet and turbo, change from darker to lighter and back to darker grey throughout the colormap. This would make it impossible for a viewer to interpret the information in a plot once it is printed in grayscale.

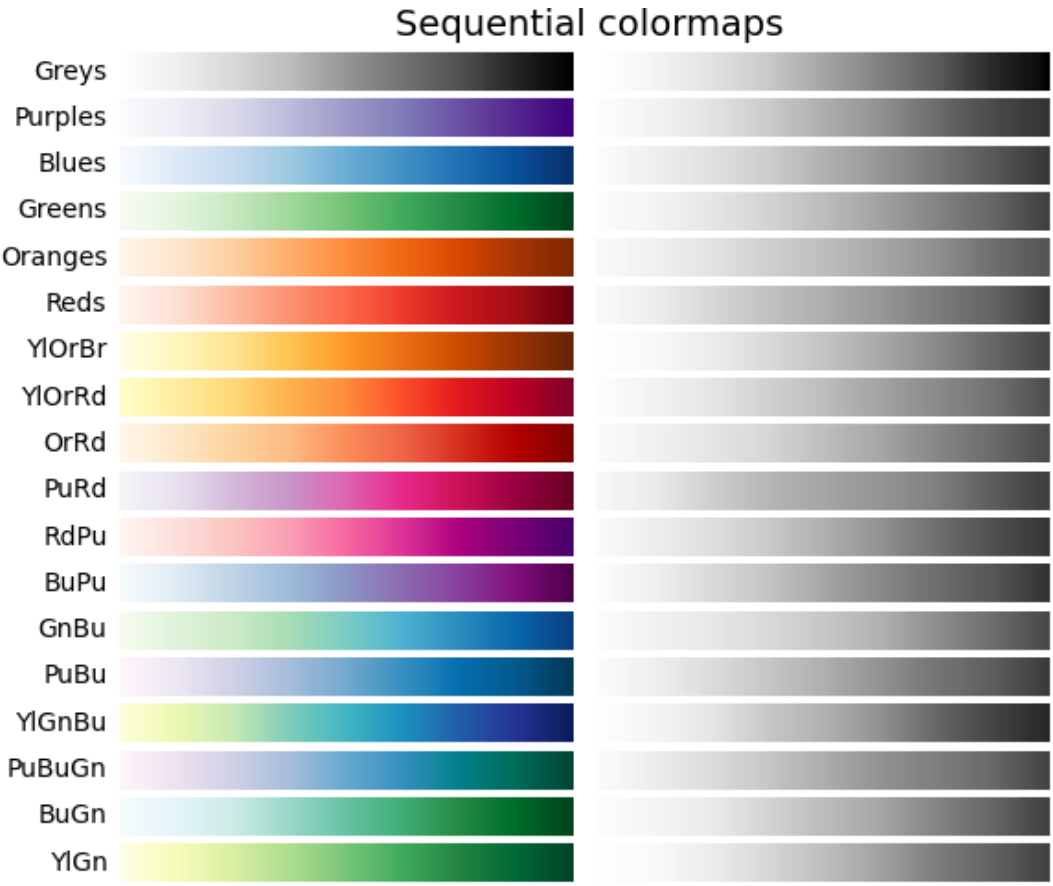
```
mpl.rcParams.update({'font.size': 14})

# Indices to step through colormap.
x = np.linspace(0.0, 1.0, 100)

gradient = np.linspace(0, 1, 256)
gradient = np.vstack((gradient, gradient))


def plot_color_gradients(cmap_category, cmap_list):
    fig, axs = plt.subplots(nrows=len(cmap_list), ncols=2)
    fig.subplots_adjust(top=0.95, bottom=0.01, left=0.2, right=0.99,
                        wspace=0.05)
    fig.suptitle(cmap_category + ' colormaps', fontsize=14, y=1.0, x=0.6)

    for ax, name in zip(axs, cmap_list):

        # Get RGB values for colormap.
        rgb = mpl.colormaps[name](x)[np.newaxis, :, :3]

        # Get colormap in CAM02-UCS colorspace. We want the lightness.
        lab = cspace_converter("sRGB1", "CAM02-UCS")(rgb)
        L = lab[0, :, 0]
        L = np.float32(np.vstack((L, L, L)))

        ax[0].imshow(gradient, aspect='auto', cmap=mpl.colormaps[name])
        ax[1].imshow(L, aspect='auto', cmap='binary_r', vmin=0., vmax=100.)
        pos = list(ax[0].get_position().bounds)
        x_text = pos[0] - 0.01
        y_text = pos[1] + pos[3]/2.
        fig.text(x_text, y_text, name, va='center', ha='right', fontsize=10)

    # Turn off *all* ticks & spines, not just the ones with colormaps.
    for ax in axs.flat:
        ax.set_axis_off()

    plt.show()
```

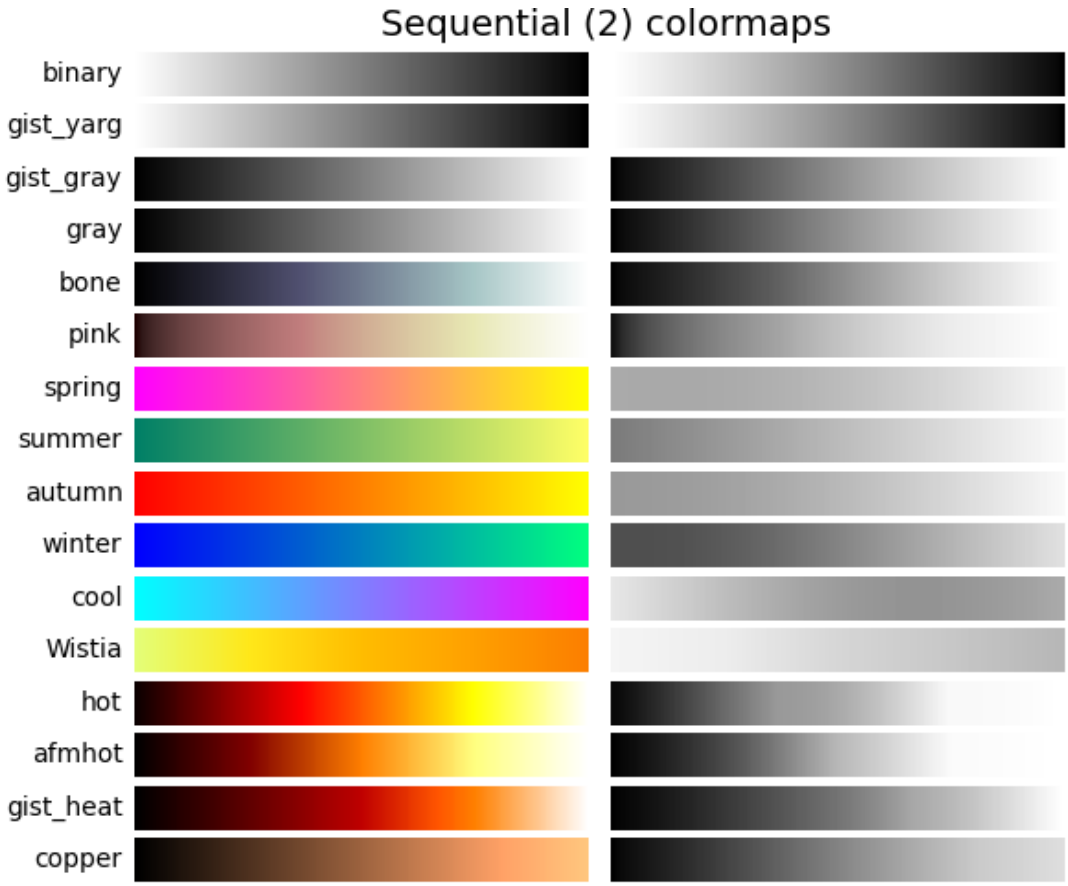
(continues on next page)

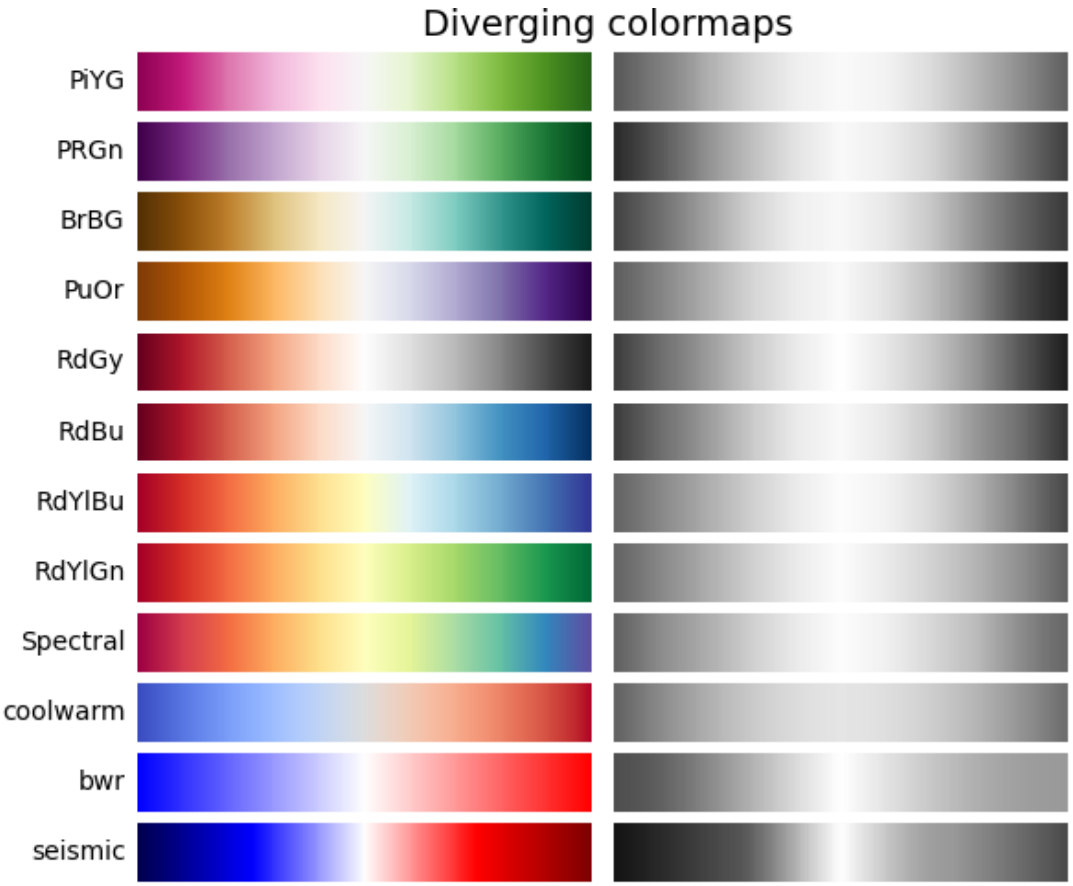
(continued from previous page)

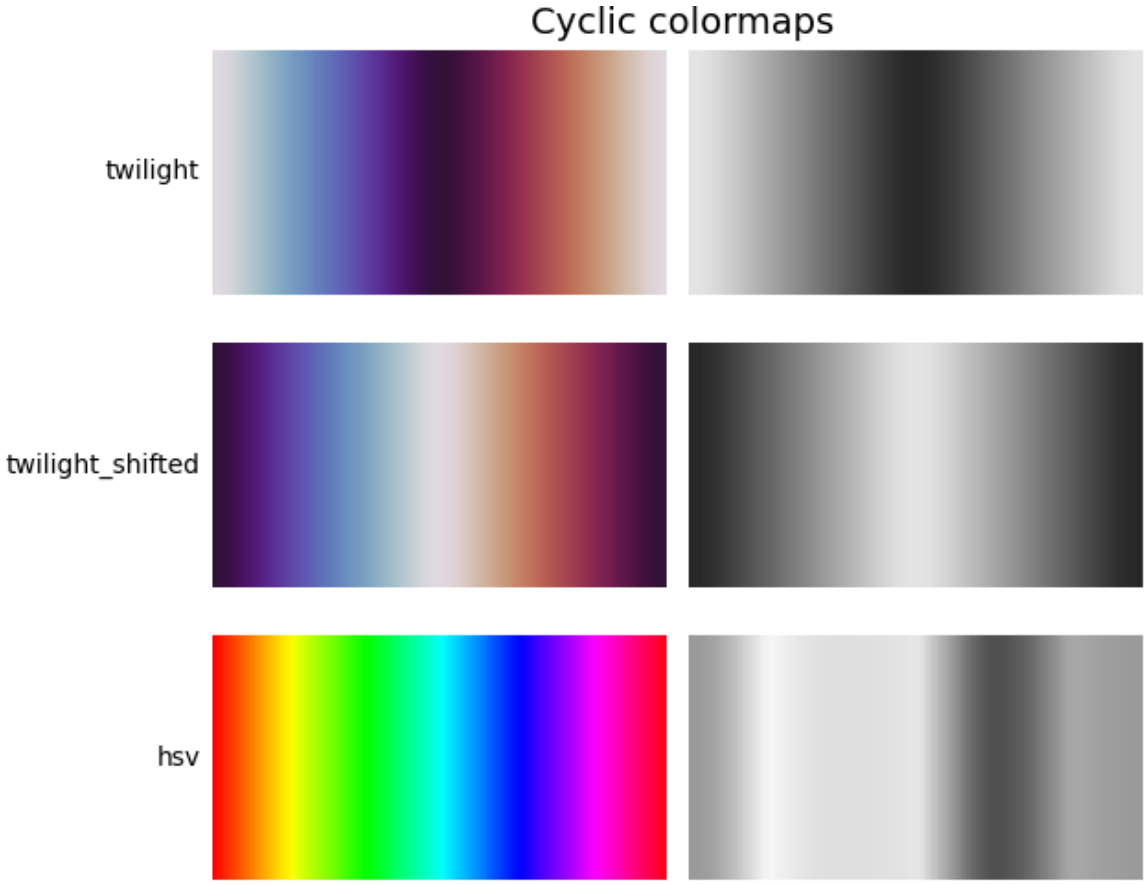
```
for cmap_category, cmap_list in cmaps.items():

    plot_color_gradients(cmap_category, cmap_list)
```

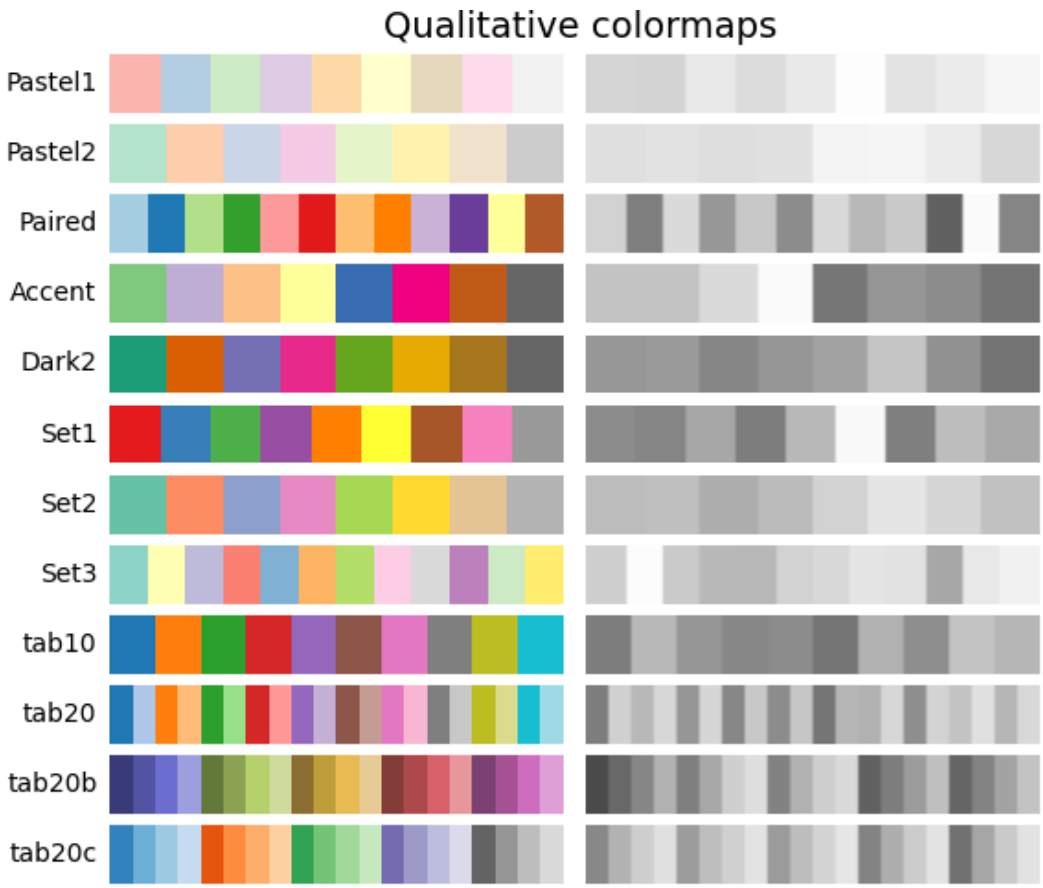


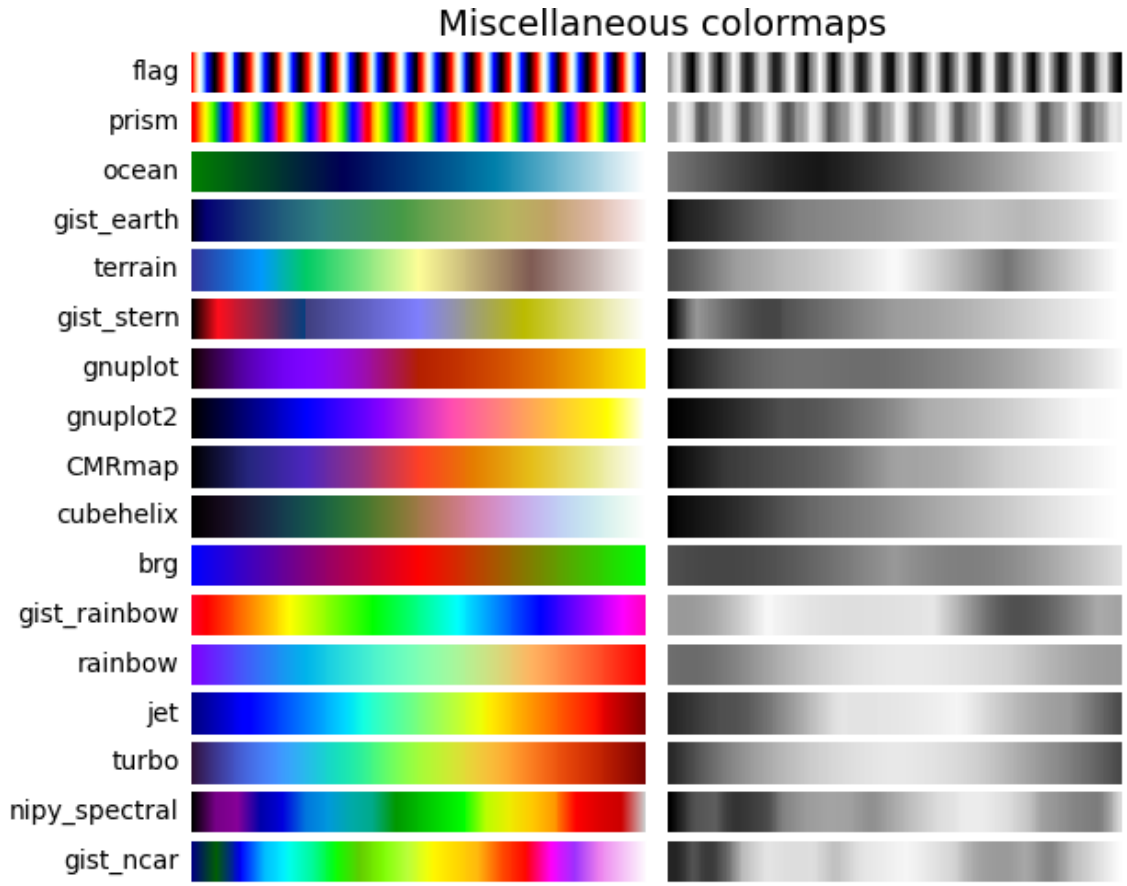

Sequential colormaps
Greys
Purples
Blues
Greens
Oranges
Reds
YlOrBr
YlOrRd
OrRd
PuRd
RdPu
BuPu
GnBu
PuBu
YlGnBu
PuBuGn
BuGn
YlGn


Sequential (2) colormaps
binary
gist_yarg
gist_gray
gray
bone
pink
spring
summer
autumn
winter
cool
Wistia
hot
afmhot
gist_heat
copper


Diverging colormaps
PiYG
PRGn
BrBG
PuOr
RdGy
RdBu
RdYlBu
RdYlGn
Spectral
coolwarm
bwr
seismic


Cyclic colormaps
twilight
twilight_shifted
hsv


Qualitative colormaps
Pastel1
Pastel2
Paired
Accent
Dark2
Set1
Set2
Set3
tab10
tab20
tab20b
tab20c




### Color vision deficiencies

There is a lot of information available about color blindness (*e.g.*, [colorblindness]). Additionally, there are tools available to convert images to how they look for different types of color vision deficiencies.

The most common form of color vision deficiency involves differentiating between red and green. Thus, avoiding colormaps with both red and green will avoid many problems in general.

### References

**Total running time of the script:** (0 minutes 11.750 seconds)

## 3.7 Text

Matplotlib has extensive text support, including support for mathematical expressions, TrueType support for raster and vector outputs, newline separated text with arbitrary rotations, and Unicode support. These tutorials cover the basics of working with text in Matplotlib.

For even more information see the *examples page*.

### 3.7.1 Text in Matplotlib

Introduction to plotting and working with text in Matplotlib.

Matplotlib has extensive text support, including support for mathematical expressions, truetype support for raster and vector outputs, newline separated text with arbitrary rotations, and Unicode support.

Because it embeds fonts directly in output documents, e.g., for postscript or PDF, what you see on the screen is what you get in the hardcopy. FreeType support produces very nice, antialiased fonts, that look good even at small raster sizes. Matplotlib includes its own `matplotlib.font_manager` (thanks to Paul Barrett), which implements a cross platform, W3C compliant font finding algorithm.

The user has a great deal of control over text properties (font size, font weight, text location and color, etc.) with sensible defaults set in the *rc file*. And significantly, for those interested in mathematical or scientific figures, Matplotlib implements a large number of TeX math symbols and commands, supporting *mathematical expressions* anywhere in your figure.

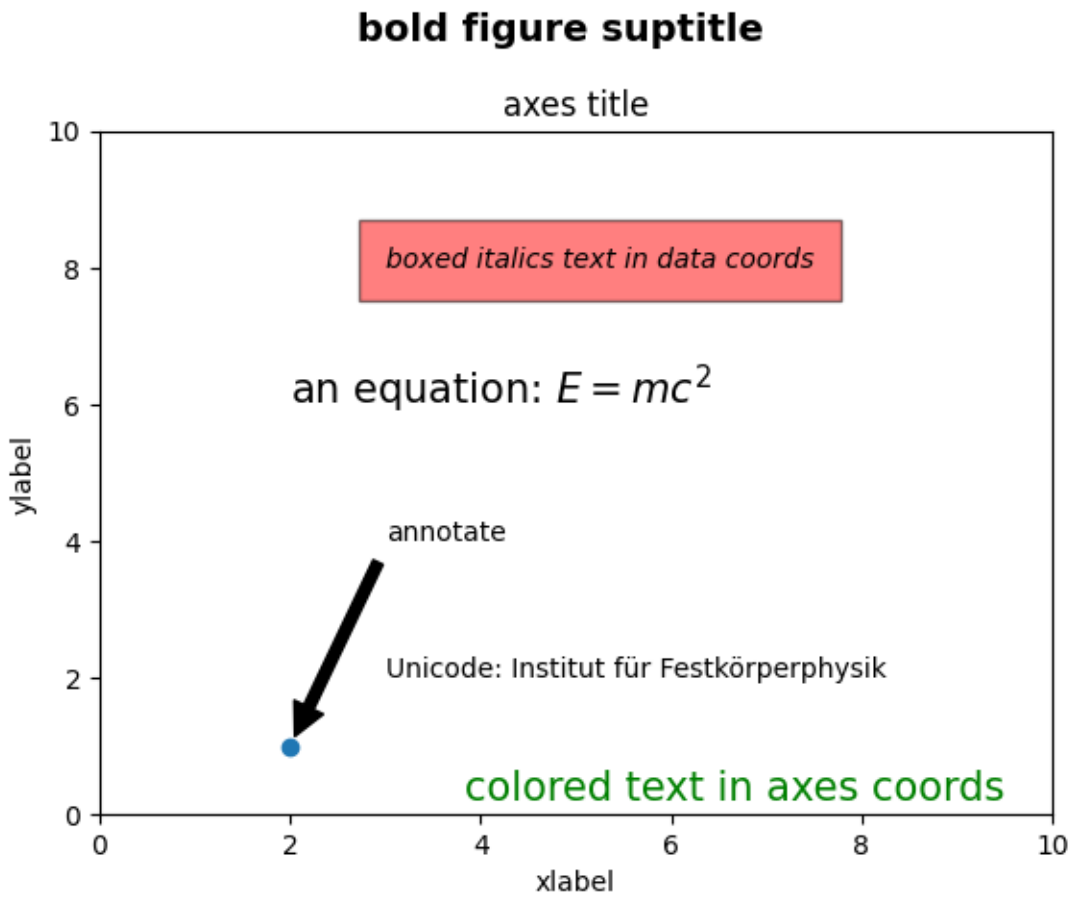
#### Basic text commands

The following commands are used to create text in the implicit and explicit interfaces (see *Matplotlib Application Interfaces (APIs)* for an explanation of the tradeoffs):

| implicit API | explicit API | description |
|---|---|---|
| `text` | `text` | Add text at an arbitrary location of the `Axes`. |
| `annotate` | `annotate` | Add an annotation, with an optional arrow, at an arbitrary location of the `Axes`. |
| `xlabel` | `set_xlabel` | Add a label to the `Axes`'s x-axis. |
| `ylabel` | `set_ylabel` | Add a label to the `Axes`'s y-axis. |
| `title` | `set_title` | Add a title to the `Axes`. |
| `figtext` | `text` | Add text at an arbitrary location of the `Figure`. |
| `suptitle` | `suptitle` | Add a title to the `Figure`. |

All of these functions create and return a *Text* instance, which can be configured with a variety of font and other properties. The example below shows all of these commands in action, and more detail is provided in the sections that follow.

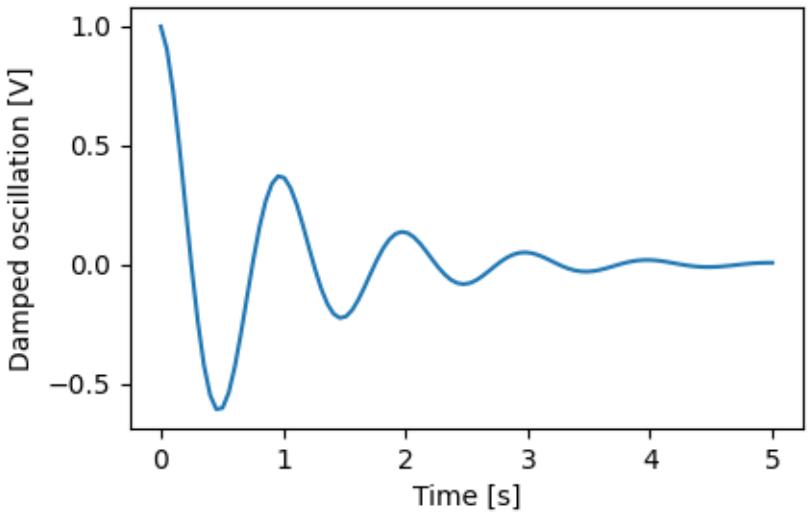
```
import matplotlib.pyplot as plt

import matplotlib

fig = plt.figure()
ax = fig.add_subplot()
fig.subplots_adjust(top=0.85)

# Set titles for the figure and the subplot respectively
fig.suptitle('bold figure suptitle', fontsize=14, fontweight='bold')
ax.set_title('axes title')

ax.set_xlabel('xlabel')
ax.set_ylabel('ylabel')

# Set both x- and y-axis limits to [0, 10] instead of default [0, 1]
ax.axis([0, 10, 0, 10])

ax.text(3, 8, 'boxed italics text in data coords', style='italic',
        bbox={'facecolor': 'red', 'alpha': 0.5, 'pad': 10})

ax.text(2, 6, r'an equation: $E=mc^2$', fontsize=15)

ax.text(3, 2, 'Unicode: Institut für Festkörperphysik')

ax.text(0.95, 0.01, 'colored text in axes coords',
        verticalalignment='bottom', horizontalalignment='right',
        transform=ax.transAxes,
        color='green', fontsize=15)

ax.plot([2], [1], 'o')
ax.annotate('annotate', xy=(2, 1), xytext=(3, 4),
            arrowprops=dict(facecolor='black', shrink=0.05))

plt.show()
```




### Labels for x- and y-axis

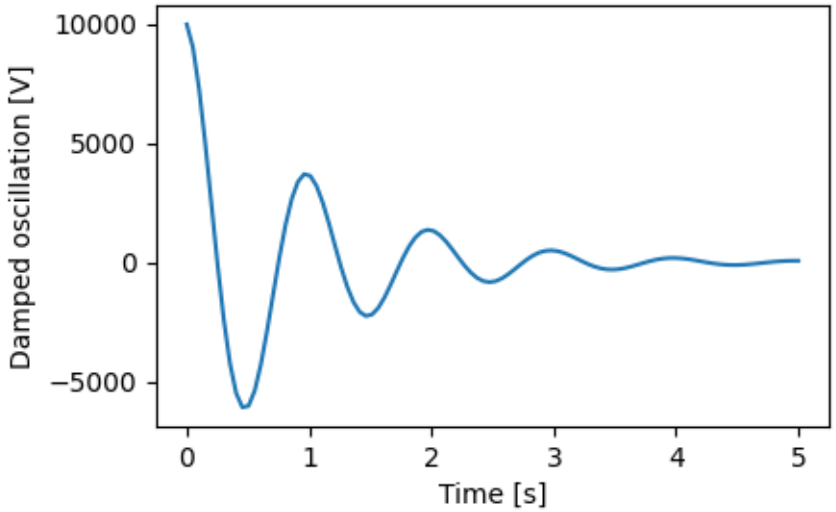
Specifying the labels for the x- and y-axis is straightforward, via the *set_xlabel* and *set_ylabel* methods.

```
import matplotlib.pyplot as plt
import numpy as np

x1 = np.linspace(0.0, 5.0, 100)
y1 = np.cos(2 * np.pi * x1) * np.exp(-x1)

fig, ax = plt.subplots(figsize=(5, 3))
fig.subplots_adjust(bottom=0.15, left=0.2)
ax.plot(x1, y1)
ax.set_xlabel('Time [s]')
ax.set_ylabel('Damped oscillation [V]')

plt.show()
```




The x- and y-labels are automatically placed so that they clear the x- and y-ticklabels. Compare the plot below with that above, and note the y-label is to the left of the one above.

```
fig, ax = plt.subplots(figsize=(5, 3))
fig.subplots_adjust(bottom=0.15, left=0.2)
ax.plot(x1, y1*10000)
ax.set_xlabel('Time [s]')
ax.set_ylabel('Damped oscillation [V]')

plt.show()
```



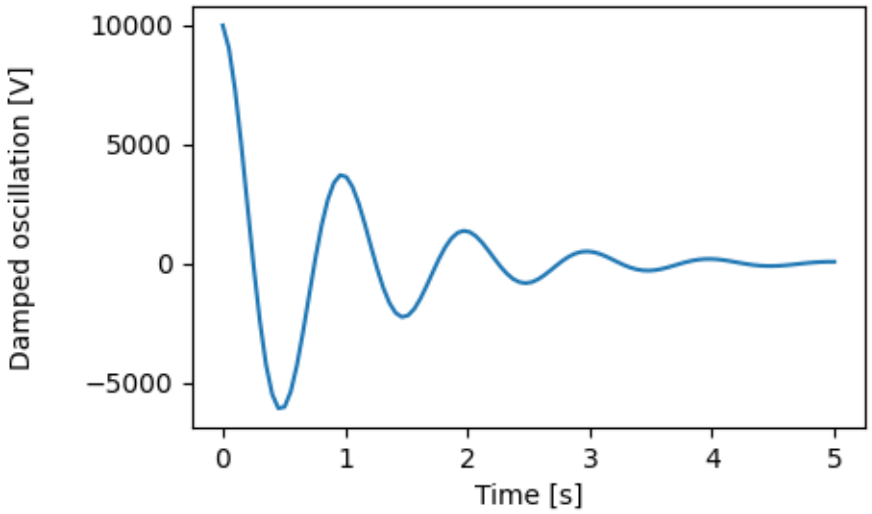

If you want to move the labels, you can specify the *labelpad* keyword argument, where the value is points (1/72", the same unit used to specify fontsizes).

```
fig, ax = plt.subplots(figsize=(5, 3))
```

(continues on next page)

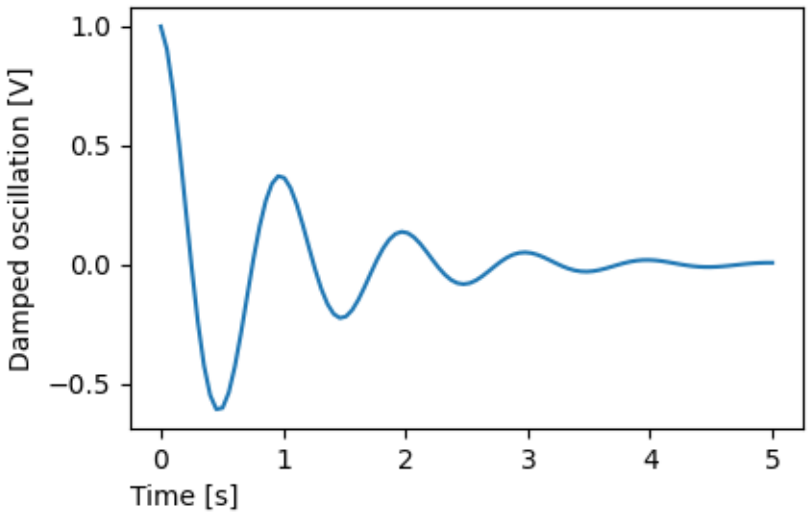
(continued from previous page)

```
fig.subplots_adjust(bottom=0.15, left=0.2)
ax.plot(x1, y1*10000)
ax.set_xlabel('Time [s]')
ax.set_ylabel('Damped oscillation [V]', labelpad=18)

plt.show()
```



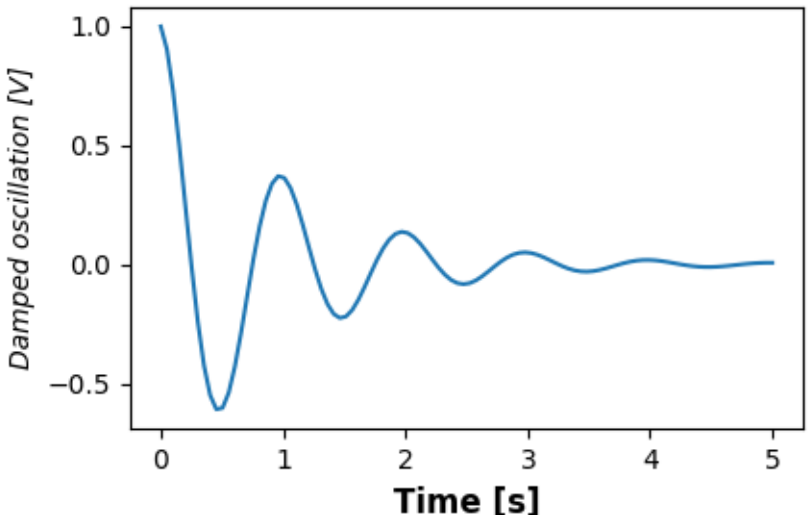

Or, the labels accept all the *Text* keyword arguments, including *position*, via which we can manually specify the label positions. Here we put the xlabel to the far left of the axis. Note, that the y-coordinate of this position has no effect - to adjust the y-position we need to use the *labelpad* keyword argument.

```
fig, ax = plt.subplots(figsize=(5, 3))
fig.subplots_adjust(bottom=0.15, left=0.2)
ax.plot(x1, y1)
ax.set_xlabel('Time [s]', position=(0., 1e6), horizontalalignment='left')
ax.set_ylabel('Damped oscillation [V]')

plt.show()
```



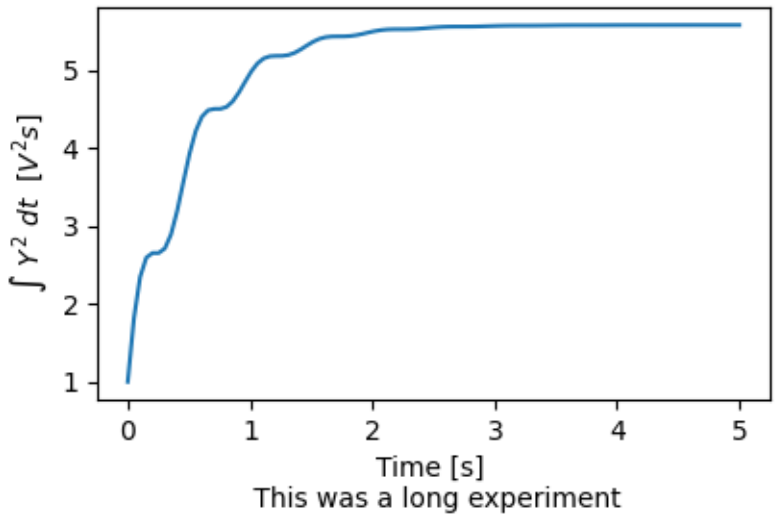

All the labelling in this tutorial can be changed by manipulating the *matplotlib.font_manager.FontProperties* method, or by named keyword arguments to *set_xlabel*

```
from matplotlib.font_manager import FontProperties

font = FontProperties()
font.set_family('serif')
font.set_name('Times New Roman')
font.set_style('italic')

fig, ax = plt.subplots(figsize=(5, 3))
fig.subplots_adjust(bottom=0.15, left=0.2)
ax.plot(x1, y1)
ax.set_xlabel('Time [s]', fontsize='large', fontweight='bold')
ax.set_ylabel('Damped oscillation [V]', fontproperties=font)

plt.show()
```




Finally, we can use native TeX rendering in all text objects and have multiple lines:

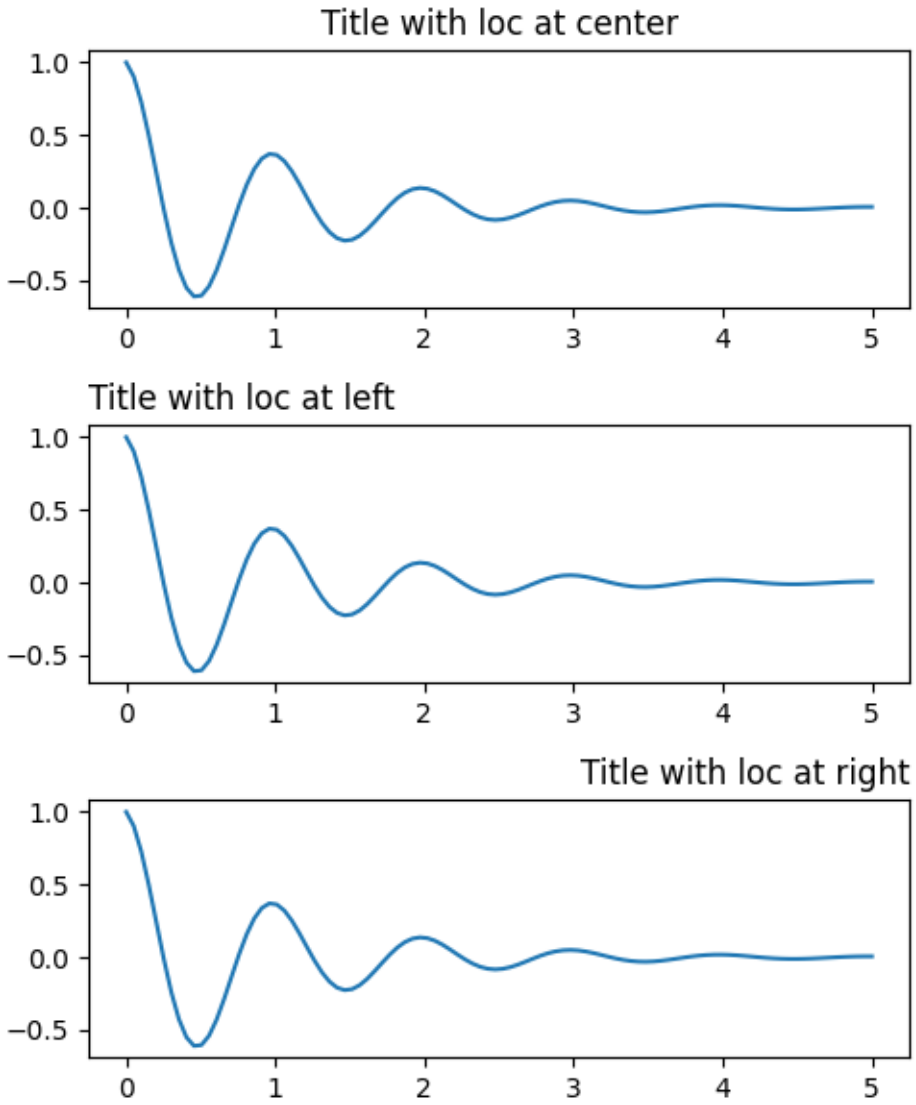
```
fig, ax = plt.subplots(figsize=(5, 3))
fig.subplots_adjust(bottom=0.2, left=0.2)
ax.plot(x1, np.cumsum(y1**2))
ax.set_xlabel('Time [s] \n This was a long experiment')
ax.set_ylabel(r'$\int\ Y^2\ dt\ \ [V^2 s]$')
plt.show()
```



### Titles

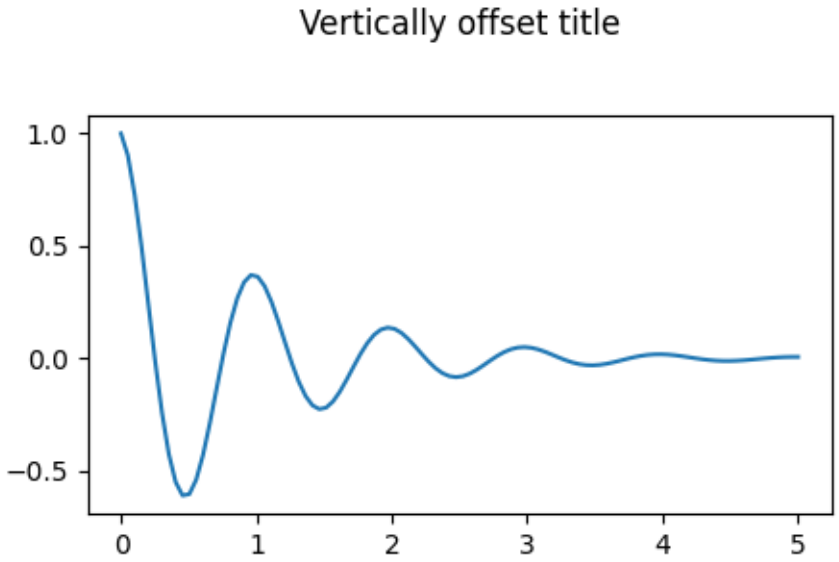
Subplot titles are set in much the same way as labels, but there is the *loc* keyword arguments that can change the position and justification from the default value of `loc=center`.

```
fig, axs = plt.subplots(3, 1, figsize=(5, 6), tight_layout=True)
locs = ['center', 'left', 'right']
for ax, loc in zip(axs, locs):
    ax.plot(x1, y1)
    ax.set_title('Title with loc at '+loc, loc=loc)
plt.show()
```




Vertical spacing for titles is controlled via *rcParams["axes.titlepad"]* (default: `6.0`). Setting to a different value moves the title.

```
fig, ax = plt.subplots(figsize=(5, 3))
fig.subplots_adjust(top=0.8)
ax.plot(x1, y1)
ax.set_title('Vertically offset title', pad=30)
plt.show()
```




### Ticks and ticklabels

Placing ticks and ticklabels is a very tricky aspect of making a figure. Matplotlib does its best to accomplish the task automatically, but it also offers a very flexible framework for determining the choices for tick locations, and how they are labelled.

### Terminology

*Axes* have an *matplotlib.axis.Axis* object for the `ax.xaxis` and `ax.yaxis` that contain the information about how the labels in the axis are laid out.

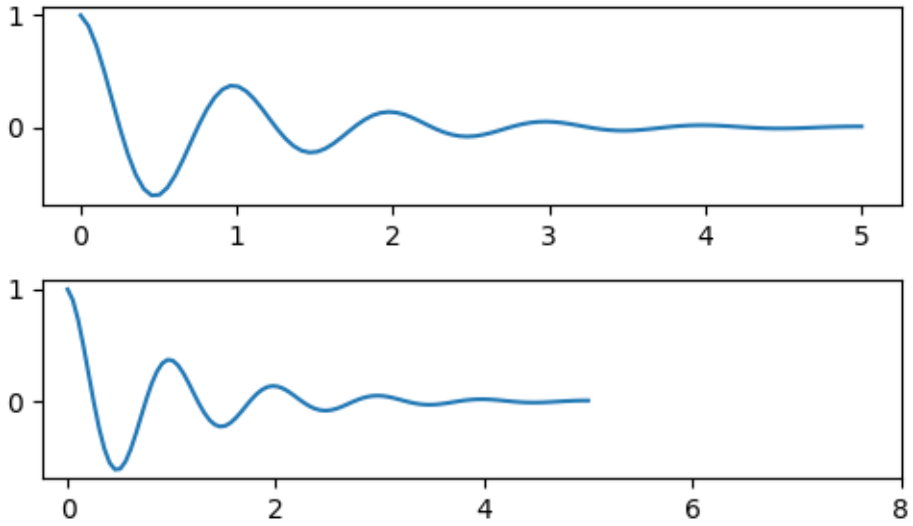
The axis API is explained in detail in the documentation to *axis*.

An Axis object has major and minor ticks. The Axis has *Axis.set_major_locator* and *Axis.set_minor_locator* methods that use the data being plotted to determine the location of major and minor ticks. There are also *Axis.set_major_formatter* and *Axis.set_minor_formatter* methods that format the tick labels.

### Simple ticks

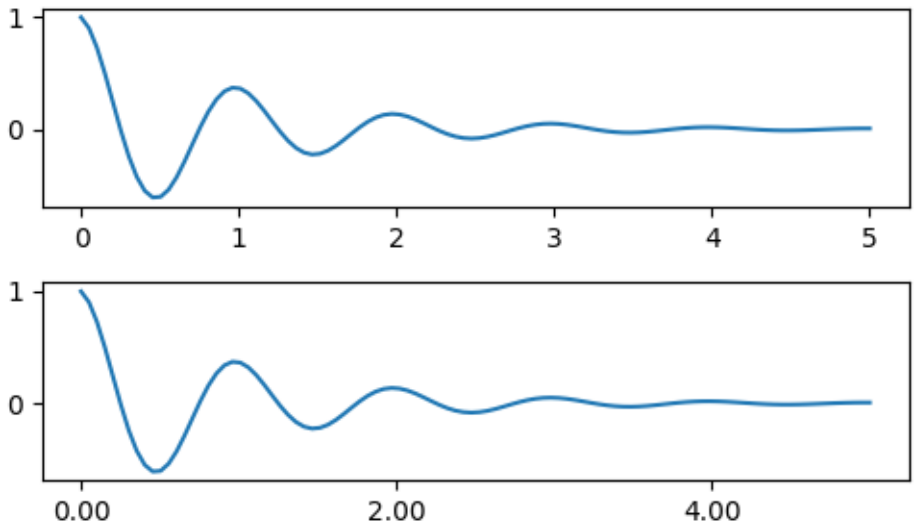
It is often convenient to simply define the tick values, and sometimes the tick labels, overriding the default locators and formatters. This is discouraged because it breaks interactive navigation of the plot. It also can reset the axis limits: note that the second plot has the ticks we asked for, including ones that are well outside the automatic view limits.

```
fig, axs = plt.subplots(2, 1, figsize=(5, 3), tight_layout=True)
axs[0].plot(x1, y1)
axs[1].plot(x1, y1)
axs[1].xaxis.set_ticks(np.arange(0., 8.1, 2.))
plt.show()
```




We can of course fix this after the fact, but it does highlight a weakness of hard-coding the ticks. This example also changes the format of the ticks:

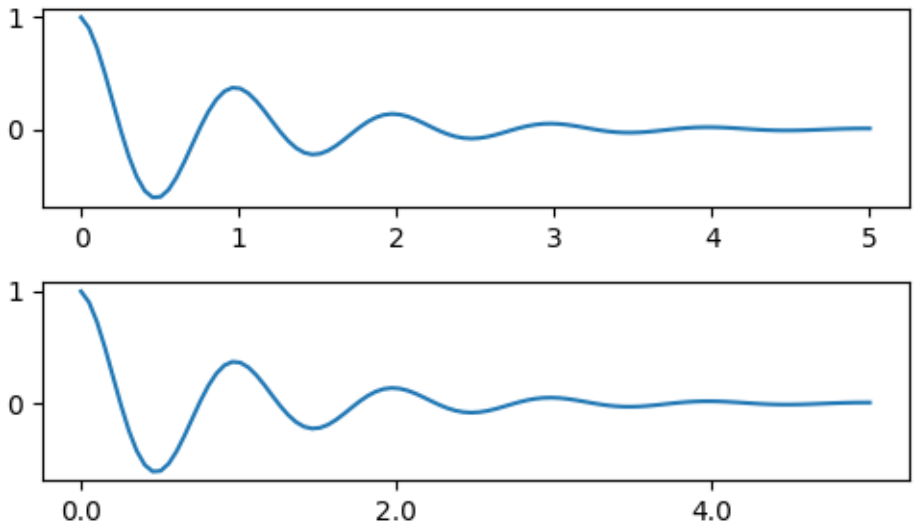
```
fig, axs = plt.subplots(2, 1, figsize=(5, 3), tight_layout=True)
axs[0].plot(x1, y1)
axs[1].plot(x1, y1)
ticks = np.arange(0., 8.1, 2.)
# list comprehension to get all tick labels...
tickla = [f'{tick:1.2f}' for tick in ticks]
axs[1].xaxis.set_ticks(ticks)
axs[1].xaxis.set_ticklabels(tickla)
axs[1].set_xlim(axs[0].get_xlim())
plt.show()
```




### Tick Locators and Formatters

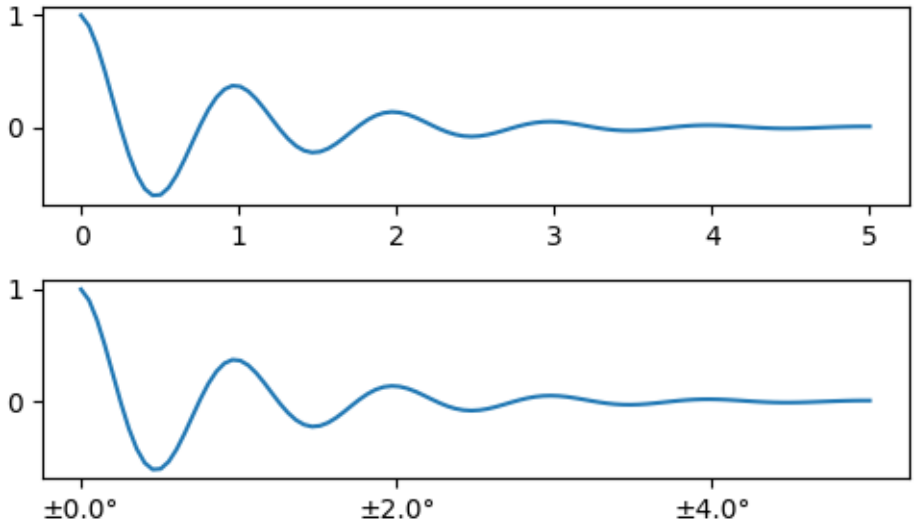
Instead of making a list of all the ticklabels, we could have used *matplotlib.ticker.StrMethodFormatter* (new-style `str.format()` format string) or *matplotlib.ticker.FormatStrFormatter* (old-style '%' format string) and passed it to the `ax.xaxis`. A *matplotlib.ticker.StrMethodFormatter* can also be created by passing a `str` without having to explicitly create the formatter.

```
fig, axs = plt.subplots(2, 1, figsize=(5, 3), tight_layout=True)
axs[0].plot(x1, y1)
axs[1].plot(x1, y1)
ticks = np.arange(0., 8.1, 2.)
axs[1].xaxis.set_ticks(ticks)
axs[1].xaxis.set_major_formatter('{x:1.1f}')
axs[1].set_xlim(axs[0].get_xlim())
plt.show()
```




And of course we could have used a non-default locator to set the tick locations. Note we still pass in the tick values, but the x-limit fix used above is *not* needed.

```
fig, axs = plt.subplots(2, 1, figsize=(5, 3), tight_layout=True)
axs[0].plot(x1, y1)
axs[1].plot(x1, y1)
locator = matplotlib.ticker.FixedLocator(ticks)
axs[1].xaxis.set_major_locator(locator)
axs[1].xaxis.set_major_formatter('±{x}°')
plt.show()
```

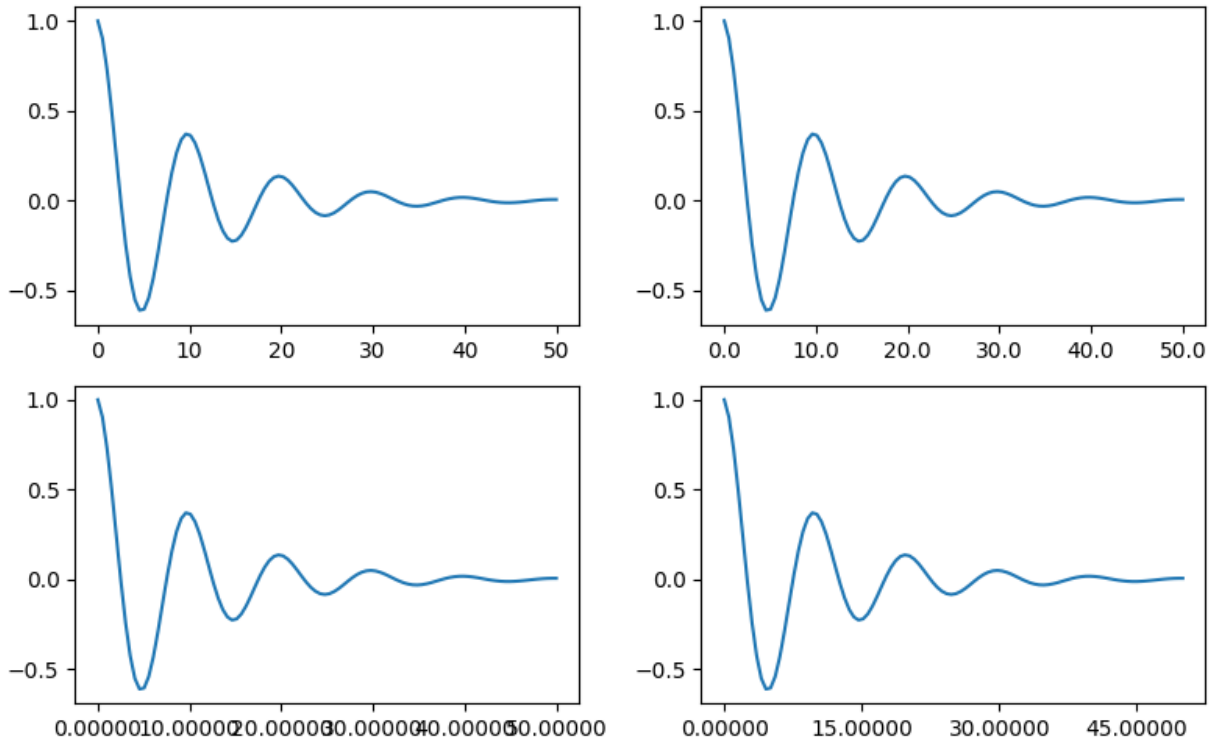



The default formatter is the *matplotlib.ticker.MaxNLocator* called as `ticker.MaxNLocator(self, nbins='auto', steps=[1, 2, 2.5, 5, 10])` The *steps* keyword contains a list of multiples that can be used for tick values. i.e. in this case, 2, 4, 6 would be acceptable ticks, as would 20, 40, 60 or 0.2, 0.4, 0.6. However, 3, 6, 9 would not be acceptable because 3

doesn't appear in the list of steps.

`nbins=auto` uses an algorithm to determine how many ticks will be acceptable based on how long the axis is. The fontsize of the ticklabel is taken into account, but the length of the tick string is not (because it's not yet known.) In the bottom row, the ticklabels are quite large, so we set `nbins=4` to make the labels fit in the right-hand plot.

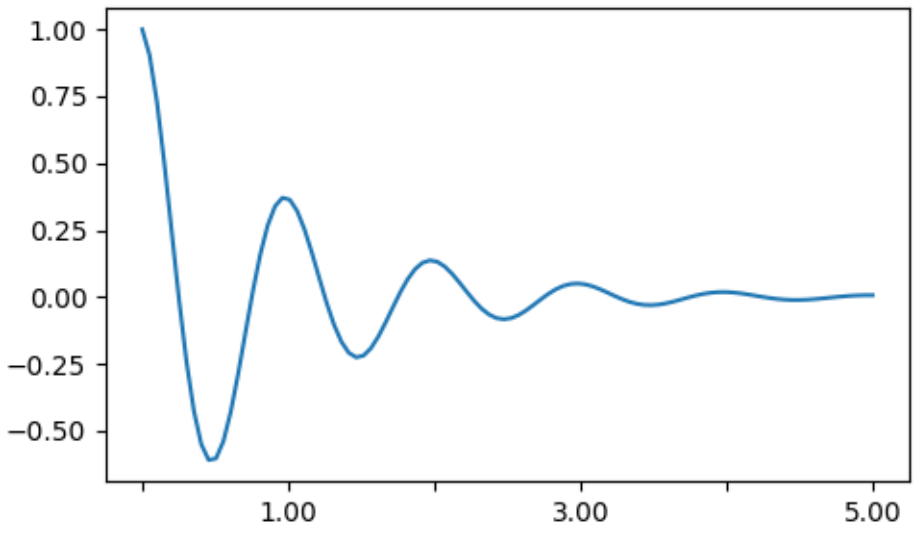
```
fig, axs = plt.subplots(2, 2, figsize=(8, 5), tight_layout=True)
for n, ax in enumerate(axs.flat):
    ax.plot(x1*10., y1)

formatter = matplotlib.ticker.FormatStrFormatter('%1.1f')
locator = matplotlib.ticker.MaxNLocator(nbins='auto', steps=[1, 4, 10])
axs[0, 1].xaxis.set_major_locator(locator)
axs[0, 1].xaxis.set_major_formatter(formatter)

formatter = matplotlib.ticker.FormatStrFormatter('%1.5f')
locator = matplotlib.ticker.AutoLocator()
axs[1, 0].xaxis.set_major_formatter(formatter)
axs[1, 0].xaxis.set_major_locator(locator)

formatter = matplotlib.ticker.FormatStrFormatter('%1.5f')
locator = matplotlib.ticker.MaxNLocator(nbins=4)
axs[1, 1].xaxis.set_major_formatter(formatter)
axs[1, 1].xaxis.set_major_locator(locator)

plt.show()
```




Finally, we can specify functions for the formatter using *matplotlib.ticker.FuncFormatter*. Fur-

ther, like *matplotlib.ticker.StrMethodFormatter*, passing a function will automatically create a *matplotlib.ticker.FuncFormatter*.

```
def formatoddticks(x, pos):
    """Format odd tick positions."""
    if x % 2:
        return f'{x:1.2f}'
    else:
        return ''


fig, ax = plt.subplots(figsize=(5, 3), tight_layout=True)
ax.plot(x1, y1)
locator = matplotlib.ticker.MaxNLocator(nbins=6)
ax.xaxis.set_major_formatter(formatoddticks)
ax.xaxis.set_major_locator(locator)

plt.show()
```




### Dateticks

Matplotlib can accept `datetime.datetime` and `numpy.datetime64` objects as plotting arguments. Dates and times require special formatting, which can often benefit from manual intervention. In order to help, dates have special Locators and Formatters, defined in the *matplotlib.dates* module.

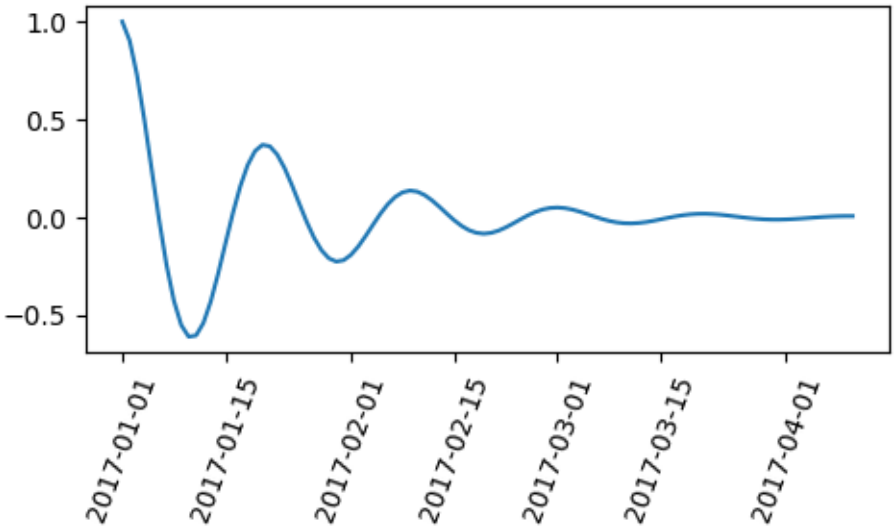
A simple example is as follows. Note how we have to rotate the tick labels so that they don't over-run each other.

```
import datetime

fig, ax = plt.subplots(figsize=(5, 3), tight_layout=True)
base = datetime.datetime(2017, 1, 1, 0, 0, 1)
```

(continues on next page)

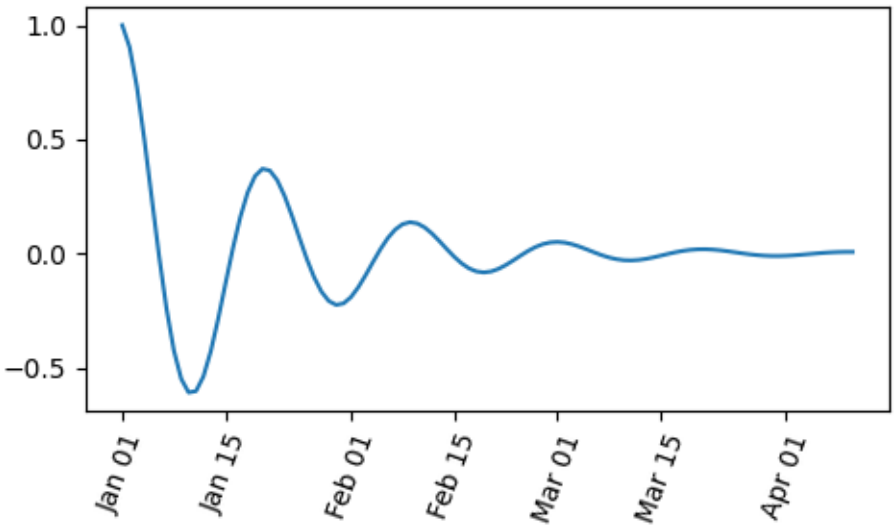
(continued from previous page)

```
time = [base + datetime.timedelta(days=x) for x in range(len(x1))]

ax.plot(time, y1)
ax.tick_params(axis='x', rotation=70)
plt.show()
```




We can pass a format to *matplotlib.dates.DateFormatter*. Also note that the 29th and the next month are very close together. We can fix this by using the *dates.DayLocator* class, which allows us to specify a list of days of the month to use. Similar formatters are listed in the *matplotlib.dates* module.

```
import matplotlib.dates as mdates

locator = mdates.DayLocator(bymonthday=[1, 15])
formatter = mdates.DateFormatter('%b %d')

fig, ax = plt.subplots(figsize=(5, 3), tight_layout=True)
ax.xaxis.set_major_locator(locator)
ax.xaxis.set_major_formatter(formatter)
ax.plot(time, y1)
ax.tick_params(axis='x', rotation=70)
plt.show()
```




### Legends and Annotations

- Legends: *Legend guide*
- Annotations: *Annotations*

**Total running time of the script:** (0 minutes 4.706 seconds)

## 3.7.2 Text properties and layout

Controlling properties of text and its layout with Matplotlib.

`matplotlib.text.Text` instances have a variety of properties which can be configured via keyword arguments to `set_title`, `set_xlabel`, `text`, etc.

| Property | Value Type |
|---|---|
| alpha | float |
| backgroundcolor | any matplotlib *color* |
| bbox | *Rectangle* prop dict plus key 'pad' which is a pad in points |
| clip_box | a matplotlib.transform.Bbox instance |
| clip_on | bool |
| clip_path | a *Path* instance and a *Transform* instance, a *Patch* |
| color | any matplotlib *color* |
| family | [ 'serif' \| 'sans-serif' \| 'cursive' \| 'fantasy' \| 'monospace' ] |
| fontproperties | *FontProperties* |
| horizontalalignment or ha | [ 'center' \| 'right' \| 'left' ] |
| label | any string |
| linespacing | float |
| multialignment | ['left' \| 'right' \| 'center' ] |
| name or fontname | string e.g., ['Sans' \| 'Courier' \| 'Helvetica' ...] |
| picker | [None\|float\|bool\|callable] |
| position | (x, y) |
| rotation | [ angle in degrees \| 'vertical' \| 'horizontal' ] |
| size or fontsize | [ size in points \| relative size, e.g., 'smaller', 'x-large' ] |
| style or fontstyle | [ 'normal' \| 'italic' \| 'oblique' ] |
| text | string or anything printable with '%s' conversion |
| transform | *Transform* subclass |
| variant | [ 'normal' \| 'small-caps' ] |
| verticalalignment or va | [ 'center' \| 'top' \| 'bottom' \| 'baseline' ] |
| visible | bool |
| weight or fontweight | [ 'normal' \| 'bold' \| 'heavy' \| 'light' \| 'ultrabold' \| 'ultra-light'] |
| x | float |
| y | float |
| zorder | any number |

You can lay out text with the alignment arguments `horizontalalignment`, `verticalalignment`, and `multialignment`. `horizontalalignment` controls whether the x positional argument for the text indicates the left, center or right side of the text bounding box. `verticalalignment` controls whether the y positional argument for the text indicates the bottom, center or top side of the text bounding box. `multialignment`, for newline separated strings only, controls whether the different lines are left, center or right justified. Here is an example which uses the *text()* command to show the various alignment possibilities. The use of `transform=ax.transAxes` throughout the code indicates that the coordinates are given relative to the Axes bounding box, with (0, 0) being the lower left of the Axes and (1, 1) the upper right.

```
import matplotlib.pyplot as plt

import matplotlib.patches as patches
```

(continues on next page)

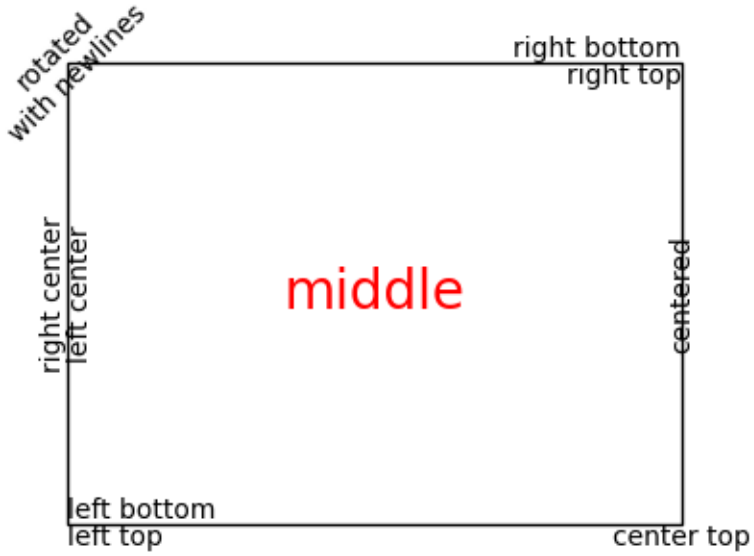
(continued from previous page)

```
# build a rectangle in axes coords
left, width = .25, .5
bottom, height = .25, .5
right = left + width
top = bottom + height

fig = plt.figure()
ax = fig.add_axes([0, 0, 1, 1])

# axes coordinates: (0, 0) is bottom left and (1, 1) is upper right
p = patches.Rectangle(
    (left, bottom), width, height,
    fill=False, transform=ax.transAxes, clip_on=False
    )

ax.add_patch(p)

ax.text(left, bottom, 'left top',
        horizontalalignment='left',
        verticalalignment='top',
        transform=ax.transAxes)

ax.text(left, bottom, 'left bottom',
        horizontalalignment='left',
        verticalalignment='bottom',
        transform=ax.transAxes)

ax.text(right, top, 'right bottom',
        horizontalalignment='right',
        verticalalignment='bottom',
        transform=ax.transAxes)

ax.text(right, top, 'right top',
        horizontalalignment='right',
        verticalalignment='top',
        transform=ax.transAxes)

ax.text(right, bottom, 'center top',
        horizontalalignment='center',
        verticalalignment='top',
        transform=ax.transAxes)

ax.text(left, 0.5*(bottom+top), 'right center',
        horizontalalignment='right',
        verticalalignment='center',
        rotation='vertical',
        transform=ax.transAxes)

ax.text(left, 0.5*(bottom+top), 'left center',
        horizontalalignment='left',
        verticalalignment='center',
```

(continues on next page)

(continued from previous page)

```
        rotation='vertical',
        transform=ax.transAxes)

ax.text(0.5*(left+right), 0.5*(bottom+top), 'middle',
        horizontalalignment='center',
        verticalalignment='center',
        fontsize=20, color='red',
        transform=ax.transAxes)

ax.text(right, 0.5*(bottom+top), 'centered',
        horizontalalignment='center',
        verticalalignment='center',
        rotation='vertical',
        transform=ax.transAxes)

ax.text(left, top, 'rotated\nwith newlines',
        horizontalalignment='center',
        verticalalignment='center',
        rotation=45,
        transform=ax.transAxes)

ax.set_axis_off()
plt.show()
```




### 3.7.3 Default Font

The base default font is controlled by a set of rcParams. To set the font for mathematical expressions, use the rcParams beginning with `mathtext` (see *mathtext*).

| rcParam | usage |
|---|---|
| `'font.family'` | List of font families (installed on user's machine) and/or `{'cursive', 'fantasy', 'monospace', 'sans', 'sans serif', 'sans-serif', 'serif'}`. |
| `'font.style'` | The default style, ex `'normal'`, `'italic'`. |
| `'font.variant'` | Default variant, ex `'normal'`, `'small-caps'` (untested) |
| `'font.stretch'` | Default stretch, ex `'normal'`, `'condensed'` (incomplete) |
| `'font.weight'` | Default weight. Either string or integer |
| `'font.size'` | Default font size in points. Relative font sizes (`'large'`, `'x-small'`) are computed against this size. |

Matplotlib can use font families installed on the user's computer, i.e. Helvetica, Times, etc. Font families can also be specified with generic-family aliases like (`{'cursive', 'fantasy', 'monospace', 'sans', 'sans serif', 'sans-serif', 'serif'}`).

---

**Note:** To access the full list of available fonts:

```
matplotlib.font_manager.get_font_names()
```

---

The mapping between the generic family aliases and actual font families (mentioned at *default rcParams*) is controlled by the following rcParams:

| CSS-based generic-family alias | rcParam with mappings |
|---|---|
| `'serif'` | `'font.serif'` |
| `'monospace'` | `'font.monospace'` |
| `'fantasy'` | `'font.fantasy'` |
| `'cursive'` | `'font.cursive'` |
| `{'sans', 'sans serif', 'sans-serif'}` | `'font.sans-serif'` |

If any of generic family names appear in `'font.family'`, we replace that entry by all the entries in the corresponding rcParam mapping. For example:

```
matplotlib.rcParams['font.family'] = ['Family1', 'serif', 'Family2']
matplotlib.rcParams['font.serif'] = ['SerifFamily1', 'SerifFamily2']

# This is effectively translated to:
matplotlib.rcParams['font.family'] = ['Family1', 'SerifFamily1', 'SerifFamily2
 ↪', 'Family2']
```

### Text with non-latin glyphs

As of v2.0 the *default font*, DejaVu, contains glyphs for many western alphabets, but not other scripts, such as Chinese, Korean, or Japanese.

To set the default font to be one that supports the code points you need, prepend the font name to `'font.family'` (recommended), or to the desired alias lists.

```
# first method
matplotlib.rcParams['font.family'] = ['Source Han Sans TW', 'sans-serif']

# second method
matplotlib.rcParams['font.family'] = ['sans-serif']
matplotlib.rcParams['sans-serif'] = ['Source Han Sans TW', ...]
```

The generic family alias lists contain fonts that are either shipped alongside Matplotlib (so they have 100% chance of being found), or fonts which have a very high probability of being present in most systems.

A good practice when setting custom font families is to append a generic-family to the font-family list as a last resort.

You can also set it in your `.matplotlibrc` file:

```
font.family: Source Han Sans TW, Arial, sans-serif
```

To control the font used on per-artist basis use the *name*, *fontname* or *fontproperties* keyword arguments documented in *Text properties and layout*.

On linux, fc-list can be a useful tool to discover the font name; for example

```
$ fc-list :lang=zh family
Noto to Sans Mono CJK TC,Noto Sans Mono CJK TC Bold
Noto Sans CJK TC,Noto Sans CJK TC Medium
Noto Sans CJK TC,Noto Sans CJK TC DemiLight
Noto Sans CJK KR,Noto Sans CJK KR Black
Noto Sans CJK TC,Noto Sans CJK TC Black
Noto Sans Mono CJK TC,Noto Sans Mono CJK TC Regular
Noto Sans CJK SC,Noto Sans CJK SC Light
```

lists all of the fonts that support Chinese.

### 3.7.4 Annotations

Annotations are graphical elements, often pieces of text, that explain, add context to, or otherwise highlight some portion of the visualized data. `annotate` supports a number of coordinate systems for flexibly positioning data and annotations relative to each other and a variety of options of for styling the text. Axes.annotate also provides an optional arrow from the text to the data and this arrow can be styled in various ways. `text` can also be used for simple text annotation, but does not provide as much flexibility in positioning and styling as `annotate`.

**Table of Contents**

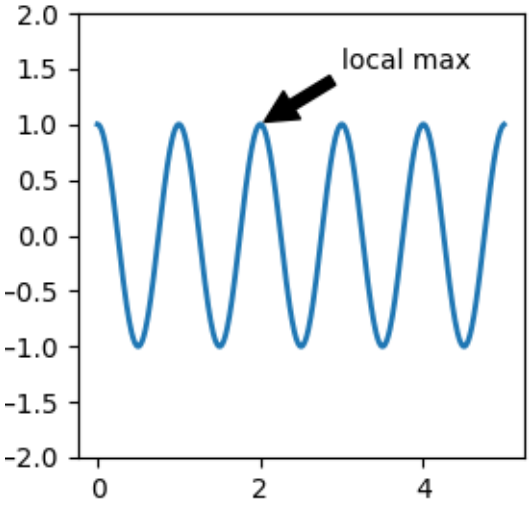

- *Annotations*
  - *Basic annotation*
    - *Annotating data*
    - *Annotating an Artist*
    - *Annotating with arrows*
    - *Placing text annotations relative to data*
  - *Advanced annotation*
    - *Annotating with boxed text*
    - *Defining custom box styles*
    - *Customizing annotation arrows*

        * *Placing Artist at anchored Axes locations*
    - *Coordinate systems for annotations*
        * `Transform` *instance*
        * `Artist` *instance*
        * *Callable that returns* `Transform` *of* `BboxBase`
        * *Blended coordinate specification*
        * `text.OffsetFrom`
    - *Non-text annotations*
        * *Using ConnectionPatch*
        * *Zoom effect between Axes*


### Basic annotation

In an annotation, there are two points to consider: the location of the data being annotated *xy* and the location of the annotation text *xytext*. Both of these arguments are `(x, y)` tuples:

```
import matplotlib.pyplot as plt
import numpy as np

fig, ax = plt.subplots(figsize=(3, 3))

t = np.arange(0.0, 5.0, 0.01)
s = np.cos(2*np.pi*t)
line, = ax.plot(t, s, lw=2)

ax.annotate('local max', xy=(2, 1), xytext=(3, 1.5),
            arrowprops=dict(facecolor='black', shrink=0.05))
ax.set_ylim(-2, 2)
```




In this example, both the *xy* (arrow tip) and *xytext* locations (text location) are in data coordinates. There are a variety of other coordinate systems one can choose -- you can specify the coordinate system of *xy* and *xytext* with one of the following strings for *xycoords* and *textcoords* (default is 'data')

| argument | coordinate system |
|---|---|
| 'figure points' | points from the lower left corner of the figure |
| 'figure pixels' | pixels from the lower left corner of the figure |
| 'figure fraction' | (0, 0) is lower left of figure and (1, 1) is upper right |
| 'axes points' | points from lower left corner of the Axes |
| 'axes pixels' | pixels from lower left corner of the Axes |
| 'axes fraction' | (0, 0) is lower left of Axes and (1, 1) is upper right |
| 'data' | use the axes data coordinate system |

The following strings are also valid arguments for *textcoords*

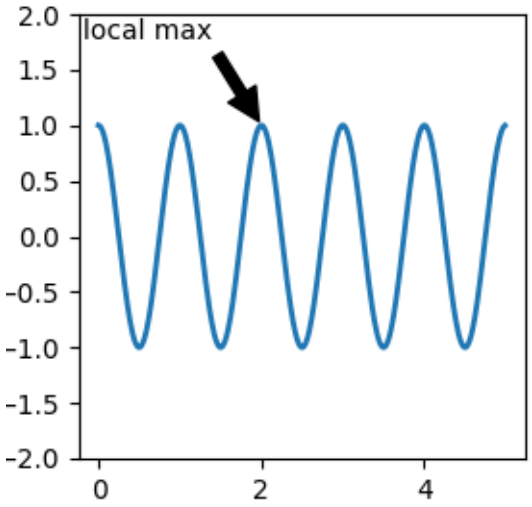
| argument | coordinate system |
|---|---|
| 'offset points' | offset (in points) from the xy value |
| 'offset pixels' | offset (in pixels) from the xy value |

For physical coordinate systems (points or pixels) the origin is the bottom-left of the figure or Axes. Points are typographic points meaning that they are a physical unit measuring 1/72 of an inch. Points and pixels are discussed in further detail in *Plotting in physical coordinates*.

### Annotating data

This example places the text coordinates in fractional axes coordinates:

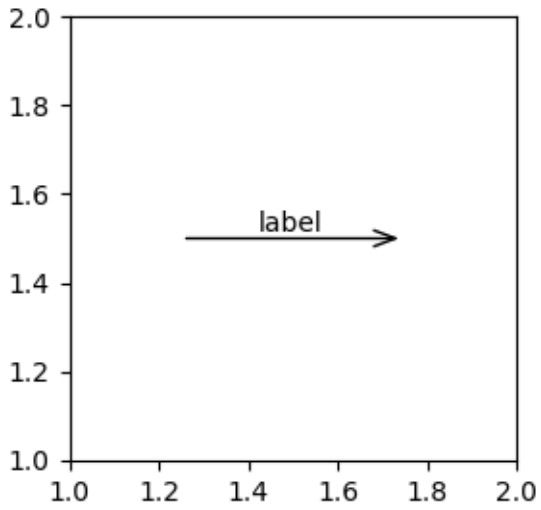
```
fig, ax = plt.subplots(figsize=(3, 3))

t = np.arange(0.0, 5.0, 0.01)
s = np.cos(2*np.pi*t)
line, = ax.plot(t, s, lw=2)

ax.annotate('local max', xy=(2, 1), xycoords='data',
            xytext=(0.01, .99), textcoords='axes fraction',
            va='top', ha='left',
            arrowprops=dict(facecolor='black', shrink=0.05))
ax.set_ylim(-2, 2)
```




### Annotating an Artist

Annotations can be positioned relative to an *Artist* instance by passing that Artist in as *xycoords*. Then *xy* is interpreted as a fraction of the Artist's bounding box.

```
import matplotlib.patches as mpatches

fig, ax = plt.subplots(figsize=(3, 3))
arr = mpatches.FancyArrowPatch((1.25, 1.5), (1.75, 1.5),
                               arrowstyle='->,head_width=.15', mutation_
 ↪scale=20)
ax.add_patch(arr)
ax.annotate("label", (.5, .5), xycoords=arr, ha='center', va='bottom')
ax.set(xlim=(1, 2), ylim=(1, 2))
```




Here the annotation is placed at position (.5,.5) relative to the arrow's lower left corner and is vertically and horizontally at that position. Vertically, the bottom aligns to that reference point so that the label is above the line. For an example of chaining annotation Artists, see the *Artist section* of *Coordinate systems for annotations*.

### Annotating with arrows

You can enable drawing of an arrow from the text to the annotated point by giving a dictionary of arrow properties in the optional keyword argument *arrowprops*.

| *arrowprops* key | description |
|---|---|
| width | the width of the arrow in points |
| frac | the fraction of the arrow length occupied by the head |
| headwidth | the width of the base of the arrow head in points |
| shrink | move the tip and base some percent away from the annotated point and text |
| **kwargs | any key for `matplotlib.patches.Polygon`, e.g., `facecolor` |

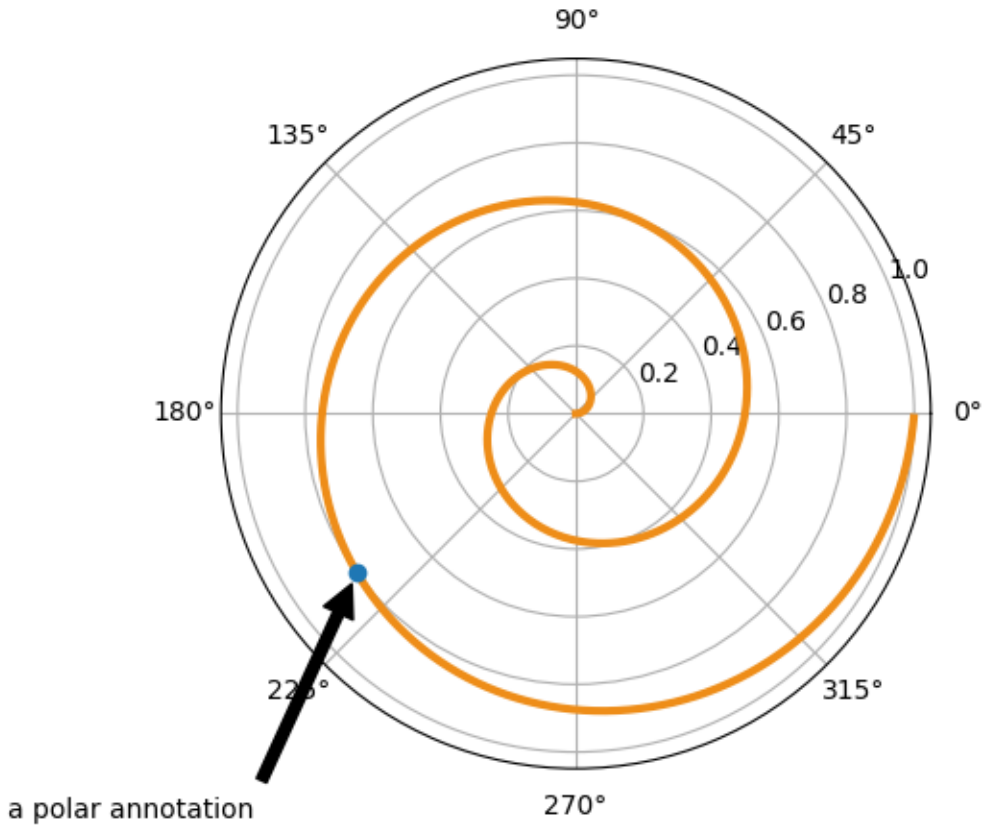
In the example below, the *xy* point is in the data coordinate system since *xycoords* defaults to 'data'. For a polar Axes, this is in (theta, radius) space. The text in this example is placed in the fractional figure coordinate system. `matplotlib.text.Text` keyword arguments like *horizontalalignment*, *verticalalignment* and *fontsize* are passed from `annotate` to the `Text` instance.

```
fig = plt.figure()
ax = fig.add_subplot(projection='polar')
r = np.arange(0, 1, 0.001)
theta = 2 * 2*np.pi * r
line, = ax.plot(theta, r, color='#ee8d18', lw=3)

ind = 800
```

(continues on next page)

(continued from previous page)

```
thisr, thistheta = r[ind], theta[ind]
ax.plot([thistheta], [thisr], 'o')
ax.annotate('a polar annotation',
            xy=(thistheta, thisr),  # theta, radius
            xytext=(0.05, 0.05),    # fraction, fraction
            textcoords='figure fraction',
            arrowprops=dict(facecolor='black', shrink=0.05),
            horizontalalignment='left',
            verticalalignment='bottom')
```

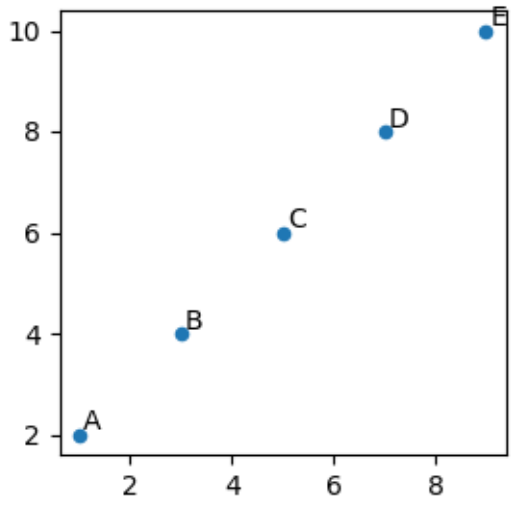



For more on plotting with arrows, see *Customizing annotation arrows*

### Placing text annotations relative to data

Annotations can be positioned at a relative offset to the *xy* input to annotation by setting the *textcoords* keyword argument to `'offset points'` or `'offset pixels'`.

```
fig, ax = plt.subplots(figsize=(3, 3))
x = [1, 3, 5, 7, 9]
y = [2, 4, 6, 8, 10]
annotations = ["A", "B", "C", "D", "E"]
ax.scatter(x, y, s=20)

for xi, yi, text in zip(x, y, annotations):
    ax.annotate(text,
                xy=(xi, yi), xycoords='data',
                xytext=(1.5, 1.5), textcoords='offset points')
```




The annotations are offset 1.5 points (1.5*1/72 inches) from the *xy* values.

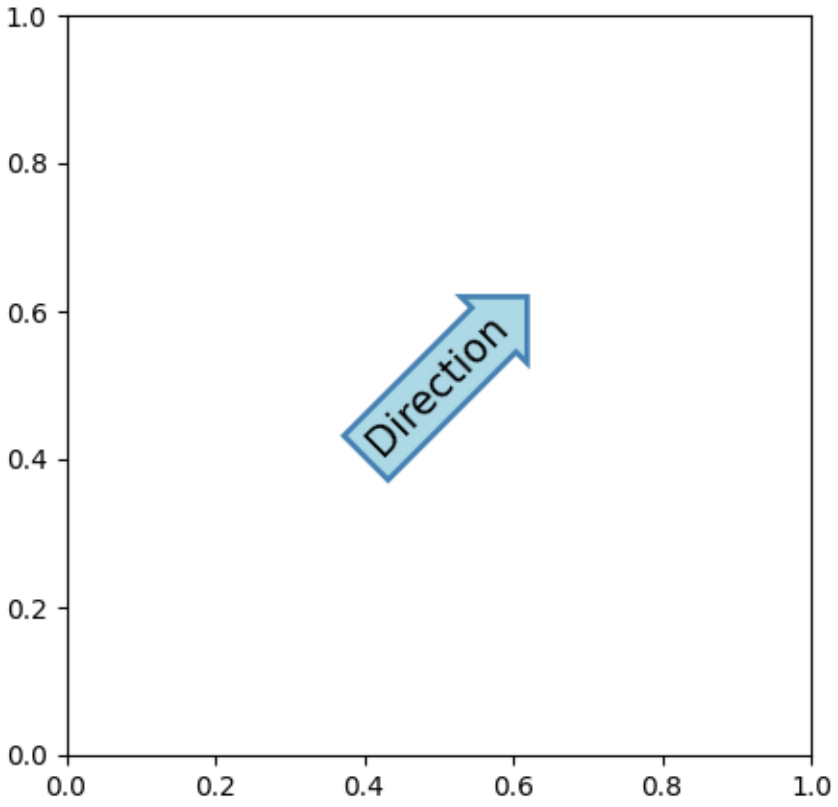
### Advanced annotation

We recommend reading *Basic annotation*, `text()` and `annotate()` before reading this section.

### Annotating with boxed text

`text` takes a *bbox* keyword argument, which draws a box around the text:

```
fig, ax = plt.subplots(figsize=(5, 5))
t = ax.text(0.5, 0.5, "Direction",
            ha="center", va="center", rotation=45, size=15,
            bbox=dict(boxstyle="rarrow,pad=0.3",
                      fc="lightblue", ec="steelblue", lw=2))
```

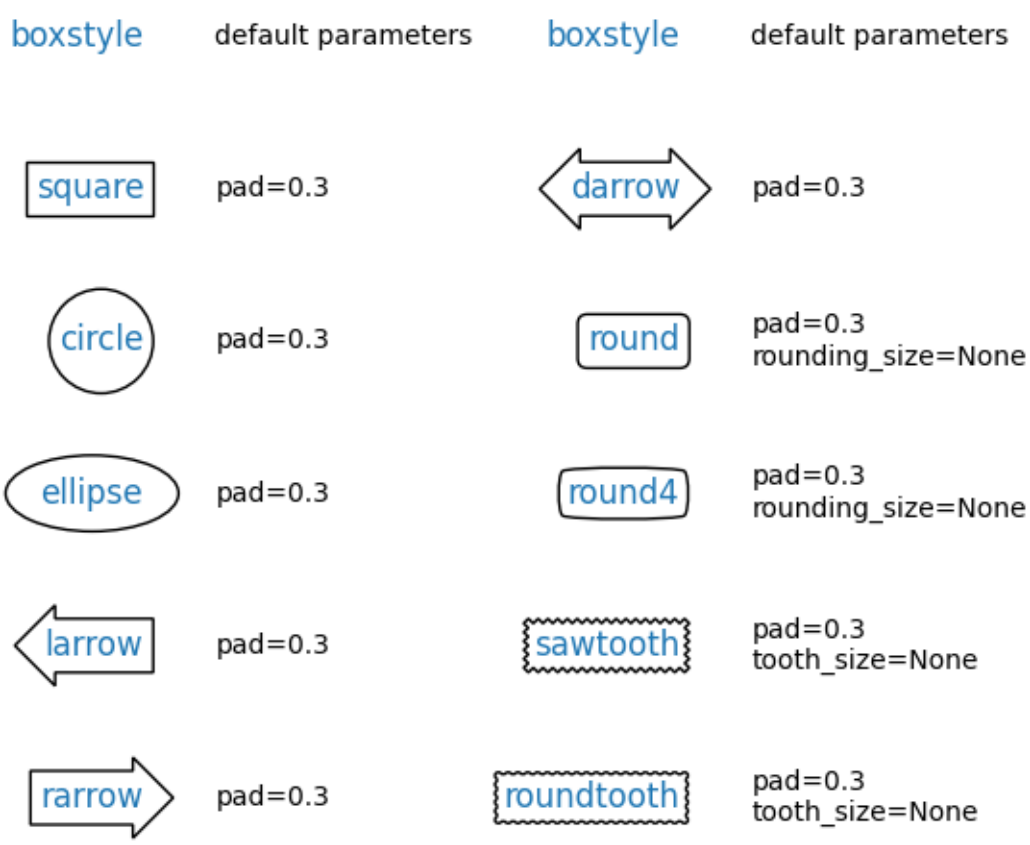



The arguments are the name of the box style with its attributes as keyword arguments. Currently, following box styles are implemented:

| Class | Name | Attrs |
|---|---|---|
| Circle | `circle` | pad=0.3 |
| DArrow | `darrow` | pad=0.3 |
| Ellipse | `ellipse` | pad=0.3 |
| LArrow | `larrow` | pad=0.3 |
| RArrow | `rarrow` | pad=0.3 |
| Round | `round` | pad=0.3,rounding_size=None |
| Round4 | `round4` | pad=0.3,rounding_size=None |
| Roundtooth | `roundtooth` | pad=0.3,tooth_size=None |
| Sawtooth | `sawtooth` | pad=0.3,tooth_size=None |
| Square | `square` | pad=0.3 |

The patch object (box) associated with the text can be accessed using:

```
bb = t.get_bbox_patch()
```


boxstyle
default parameters
boxstyle
default parameters
square
pad=0.3
darrow
pad=0.3
circle
pad=0.3
round
pad=0.3
rounding_size=None
ellipse
pad=0.3
round4
pad=0.3
rounding_size=None
larrow
pad=0.3
sawtooth
pad=0.3
tooth_size=None
rarrow
pad=0.3
roundtooth
pad=0.3
tooth_size=None

The return value is a *FancyBboxPatch*; patch properties (facecolor, edgewidth, etc.) can be accessed and modified as usual. *FancyBboxPatch.set_boxstyle* sets the box shape:

```
bb.set_boxstyle("rarrow", pad=0.6)
```

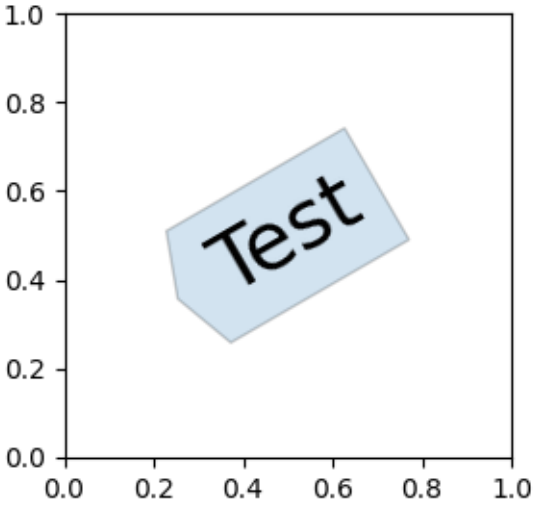
The attribute arguments can also be specified within the style name with separating comma:

```
bb.set_boxstyle("rarrow, pad=0.6")
```

### Defining custom box styles

You can use a custom box style. The value for the `boxstyle` can be a callable object in the following forms:

```
from matplotlib.path import Path


def custom_box_style(x0, y0, width, height, mutation_size):
    """
    Given the location and size of the box, return the path of the box around
    it. Rotation is automatically taken care of.

    Parameters
    ----------
    x0, y0, width, height : float
       Box location and size.
    mutation_size : float
    Mutation reference scale, typically the text font size.
    """
    # padding
    mypad = 0.3
    pad = mutation_size * mypad
    # width and height with padding added.
    width = width + 2 * pad
    height = height + 2 * pad
    # boundary of the padded box
    x0, y0 = x0 - pad, y0 - pad
    x1, y1 = x0 + width, y0 + height
    # return the new path
    return Path([(x0, y0), (x1, y0), (x1, y1), (x0, y1),
                 (x0-pad, (y0+y1)/2), (x0, y0), (x0, y0)],
                closed=True)

fig, ax = plt.subplots(figsize=(3, 3))
ax.text(0.5, 0.5, "Test", size=30, va="center", ha="center", rotation=30,
        bbox=dict(boxstyle=custom_box_style, alpha=0.2))
```

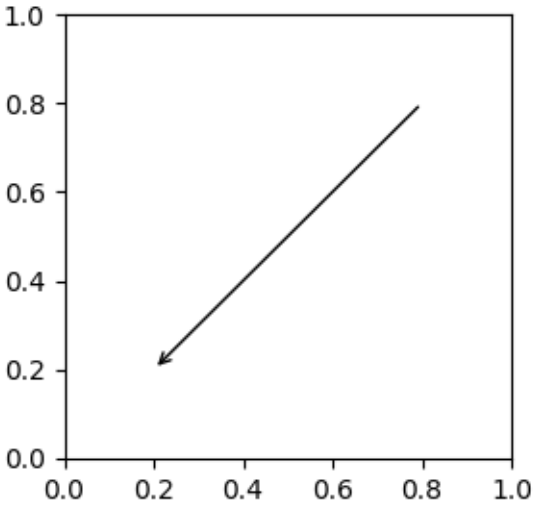



See also *Custom box styles*. Similarly, you can define a custom `ConnectionStyle` and a custom `ArrowStyle`. View the source code at `patches` to learn how each class is defined.

### Customizing annotation arrows

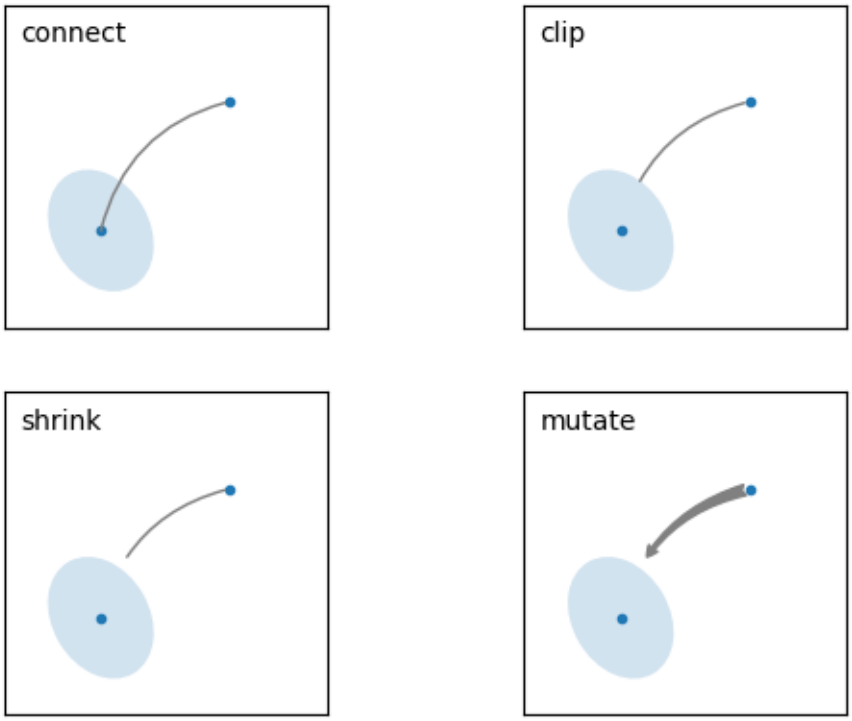
An arrow connecting *xy* to *xytext* can be optionally drawn by specifying the *arrowprops* argument. To draw only an arrow, use empty string as the first argument:

```
fig, ax = plt.subplots(figsize=(3, 3))
ax.annotate("",
            xy=(0.2, 0.2), xycoords='data',
            xytext=(0.8, 0.8), textcoords='data',
            arrowprops=dict(arrowstyle="->", connectionstyle="arc3"))
```



The arrow is drawn as follows:

1. A path connecting the two points is created, as specified by the *connectionstyle* parameter.
2. The path is clipped to avoid patches *patchA* and *patchB*, if these are set.
3. The path is further shrunk by *shrinkA* and *shrinkB* (in pixels).
4. The path is transmuted to an arrow patch, as specified by the *arrowstyle* parameter.



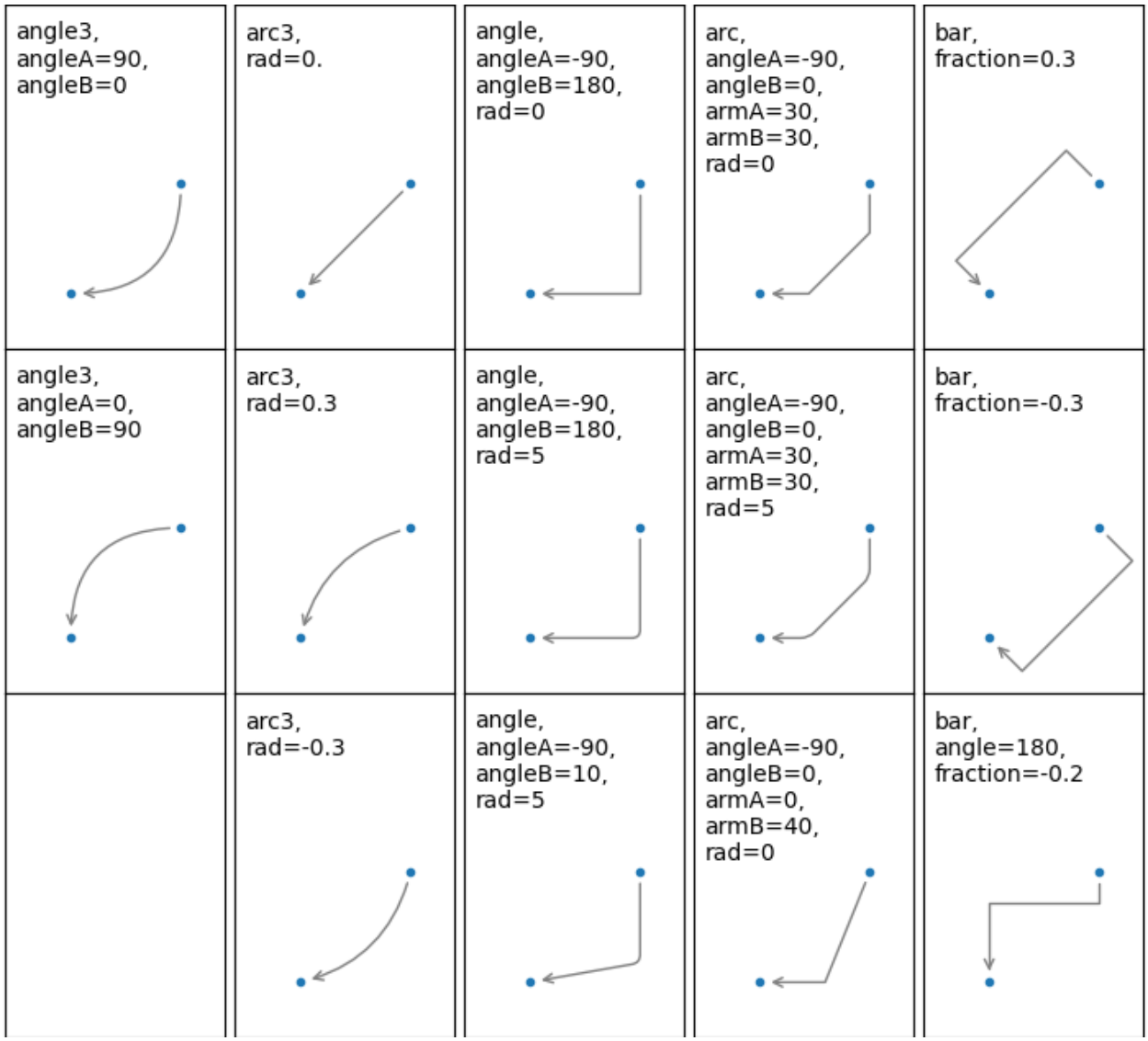

The creation of the connecting path between two points is controlled by `connectionstyle` key and the following styles are available:

| Name | Attrs |
|---|---|
| `angle` | angleA=90,angleB=0,rad=0.0 |
| `angle3` | angleA=90,angleB=0 |
| `arc` | angleA=0,angleB=0,armA=None,armB=None,rad=0.0 |
| `arc3` | rad=0.0 |
| `bar` | armA=0.0,armB=0.0,fraction=0.3,angle=None |

Note that "3" in `angle3` and `arc3` is meant to indicate that the resulting path is a quadratic spline segment (three control points). As will be discussed below, some arrow style options can only be used when the

connecting path is a quadratic spline.

The behavior of each connection style is (limitedly) demonstrated in the example below. (Warning: The behavior of the `bar` style is currently not well-defined and may be changed in the future).



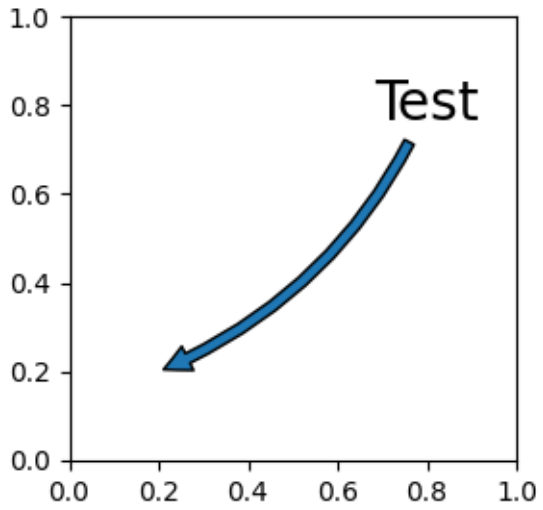

The connecting path (after clipping and shrinking) is then mutated to an arrow patch, according to the given `arrowstyle`:

| Name | Attrs |
|---|---|
| - | None |
| -> | head_length=0.4,head_width=0.2 |
| -[ | widthB=1.0,lengthB=0.2,angleB=None |
| \|-\| | widthA=1.0,widthB=1.0 |
| -\|> | head_length=0.4,head_width=0.2 |
| <- | head_length=0.4,head_width=0.2 |
| <-> | head_length=0.4,head_width=0.2 |
| <\|- | head_length=0.4,head_width=0.2 |
| <\|-\|> | head_length=0.4,head_width=0.2 |
| fancy | head_length=0.4,head_width=0.4,tail_width=0.4 |
| simple | head_length=0.5,head_width=0.5,tail_width=0.2 |
| wedge | tail_width=0.3,shrink_factor=0.5 |

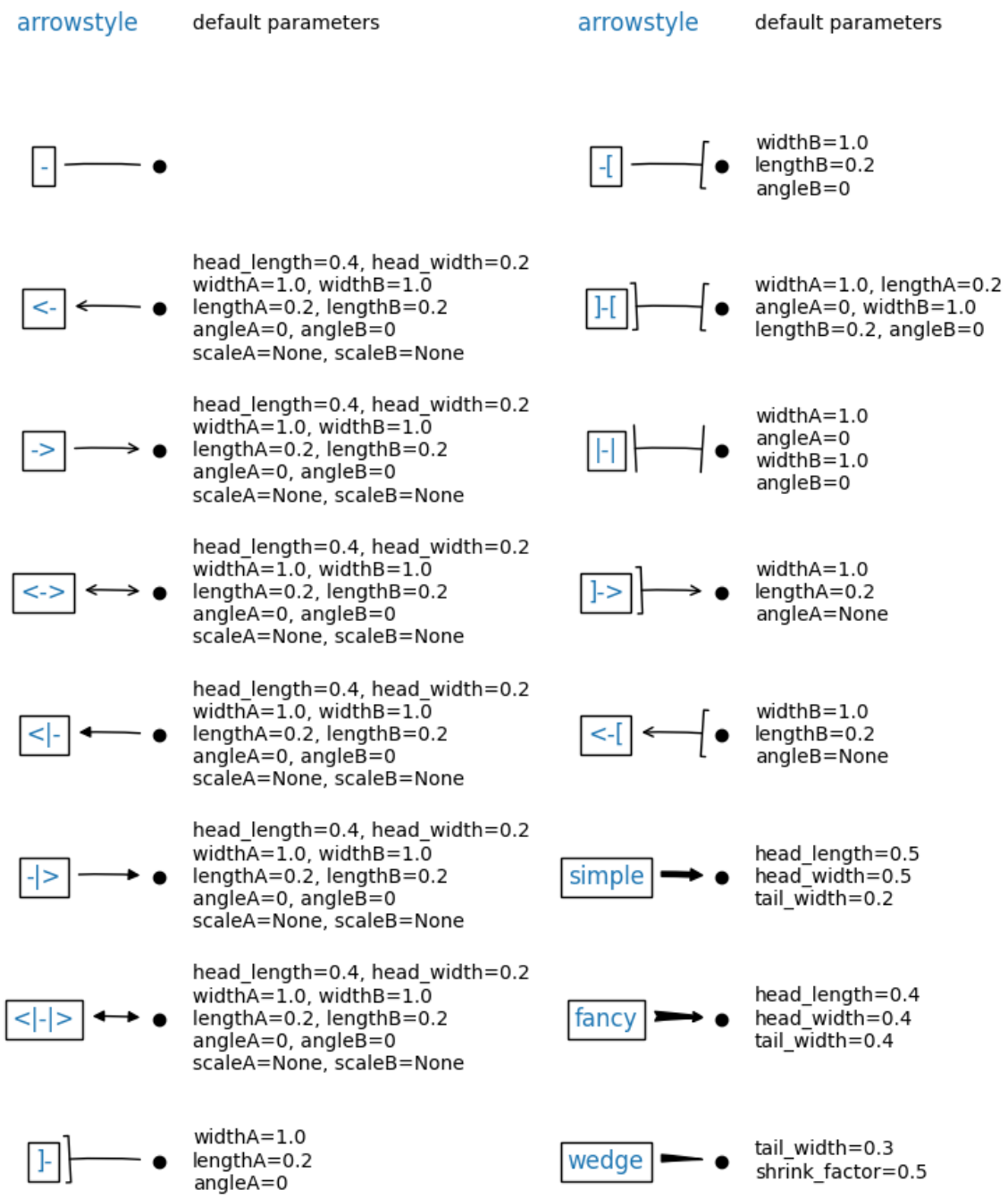
Some arrowstyles only work with connection styles that generate a quadratic-spline segment. They are `fancy`, `simple`, and `wedge`. For these arrow styles, you must use the "angle3" or "arc3" connection style.

If the annotation string is given, the patch is set to the bbox patch of the text by default.

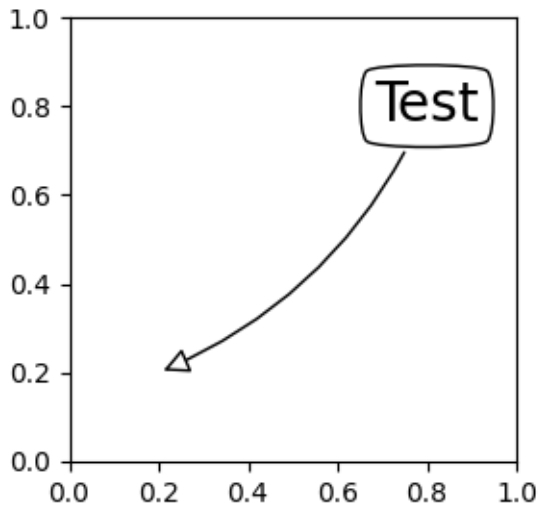
```
fig, ax = plt.subplots(figsize=(3, 3))

ax.annotate("Test",
            xy=(0.2, 0.2), xycoords='data',
            xytext=(0.8, 0.8), textcoords='data',
            size=20, va="center", ha="center",
            arrowprops=dict(arrowstyle="simple",
                            connectionstyle="arc3,rad=-0.2"))
```




As with `text`, a box around the text can be drawn using the *bbox* argument.


arrowstyle
default parameters
-
<-
head_length=0.4, head_width=0.2
widthA=1.0, widthB=1.0
lengthA=0.2, lengthB=0.2
angleA=0, angleB=0
scaleA=None, scaleB=None
->
head_length=0.4, head_width=0.2
widthA=1.0, widthB=1.0
lengthA=0.2, lengthB=0.2
angleA=0, angleB=0
scaleA=None, scaleB=None
<->
head_length=0.4, head_width=0.2
widthA=1.0, widthB=1.0
lengthA=0.2, lengthB=0.2
angleA=0, angleB=0
scaleA=None, scaleB=None
<|-
head_length=0.4, head_width=0.2
widthA=1.0, widthB=1.0
lengthA=0.2, lengthB=0.2
angleA=0, angleB=0
scaleA=None, scaleB=None
-|>
head_length=0.4, head_width=0.2
widthA=1.0, widthB=1.0
lengthA=0.2, lengthB=0.2
angleA=0, angleB=0
scaleA=None, scaleB=None
<|-|>
head_length=0.4, head_width=0.2
widthA=1.0, widthB=1.0
lengthA=0.2, lengthB=0.2
angleA=0, angleB=0
scaleA=None, scaleB=None
]-
widthA=1.0
lengthA=0.2
angleA=0
arrowstyle
default parameters
-[
widthB=1.0
lengthB=0.2
angleB=0
]-[
widthA=1.0, lengthA=0.2
angleA=0, widthB=1.0
lengthB=0.2, angleB=0
|-|
widthA=1.0
angleA=0
widthB=1.0
angleB=0
]->
widthA=1.0
lengthA=0.2
angleA=None
<-[
widthB=1.0
lengthB=0.2
angleB=None
simple
head_length=0.5
head_width=0.5
tail_width=0.2
fancy
head_length=0.4
head_width=0.4
tail_width=0.4
wedge
tail_width=0.3
shrink_factor=0.5

```
fig, ax = plt.subplots(figsize=(3, 3))

ann = ax.annotate("Test",
                  xy=(0.2, 0.2), xycoords='data',
                  xytext=(0.8, 0.8), textcoords='data',
                  size=20, va="center", ha="center",
                  bbox=dict(boxstyle="round4", fc="w"),
                  arrowprops=dict(arrowstyle="-|>",
                                  connectionstyle="arc3,rad=-0.2",
                                  fc="w"))
```



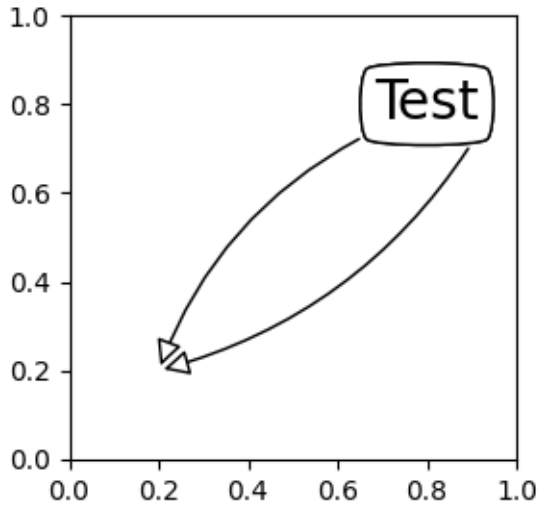

By default, the starting point is set to the center of the text extent. This can be adjusted with `relpos` key value. The values are normalized to the extent of the text. For example, (0, 0) means lower-left corner and (1, 1) means top-right.

```
fig, ax = plt.subplots(figsize=(3, 3))

ann = ax.annotate("Test",
                  xy=(0.2, 0.2), xycoords='data',
                  xytext=(0.8, 0.8), textcoords='data',
                  size=20, va="center", ha="center",
                  bbox=dict(boxstyle="round4", fc="w"),
                  arrowprops=dict(arrowstyle="-|>",
                                  connectionstyle="arc3,rad=0.2",
                                  relpos=(0., 0.),
                                  fc="w"))

ann = ax.annotate("Test",
                  xy=(0.2, 0.2), xycoords='data',
                  xytext=(0.8, 0.8), textcoords='data',
                  size=20, va="center", ha="center",
                  bbox=dict(boxstyle="round4", fc="w"),
                  arrowprops=dict(arrowstyle="-|>",
```

(continues on next page)

(continued from previous page)

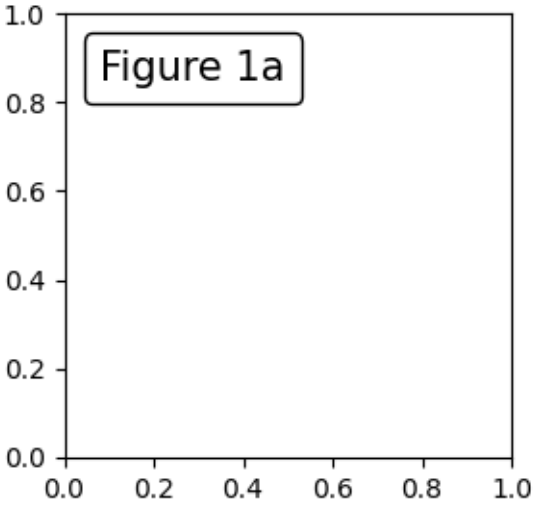
```
                                    connectionstyle="arc3,rad=-0.2",
                                    relpos=(1., 0.),
                                    fc="w"))
```




### Placing Artist at anchored Axes locations

There are classes of artists that can be placed at an anchored location in the Axes. A common example is the legend. This type of artist can be created by using the *OffsetBox* class. A few predefined classes are available in *matplotlib.offsetbox* and in *mpl_toolkits.axes_grid1.anchored_artists*.

```
from matplotlib.offsetbox import AnchoredText

fig, ax = plt.subplots(figsize=(3, 3))
at = AnchoredText("Figure 1a",
                  prop=dict(size=15), frameon=True, loc='upper left')
at.patch.set_boxstyle("round,pad=0.,rounding_size=0.2")
ax.add_artist(at)
```

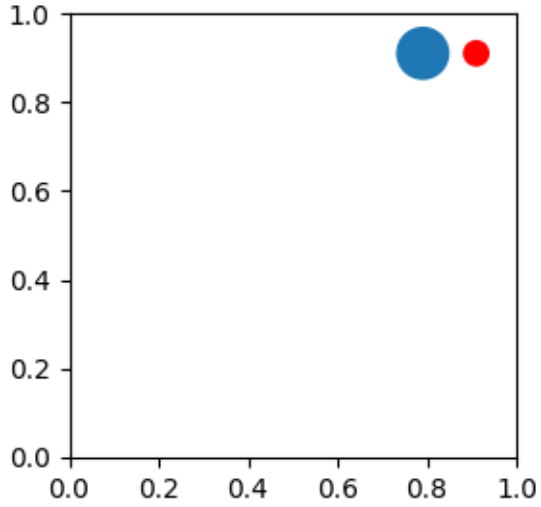



The *loc* keyword has same meaning as in the legend command.

A simple application is when the size of the artist (or collection of artists) is known in pixel size during the time of creation. For example, If you want to draw a circle with fixed size of 20 pixel x 20 pixel (radius = 10 pixel), you can utilize *AnchoredDrawingArea*. The instance is created with a size of the drawing area (in pixels), and arbitrary artists can be added to the drawing area. Note that the extents of the artists that are added to the drawing area are not related to the placement of the drawing area itself. Only the initial size matters.

The artists that are added to the drawing area should not have a transform set (it will be overridden) and the dimensions of those artists are interpreted as a pixel coordinate, i.e., the radius of the circles in above example are 10 pixels and 5 pixels, respectively.

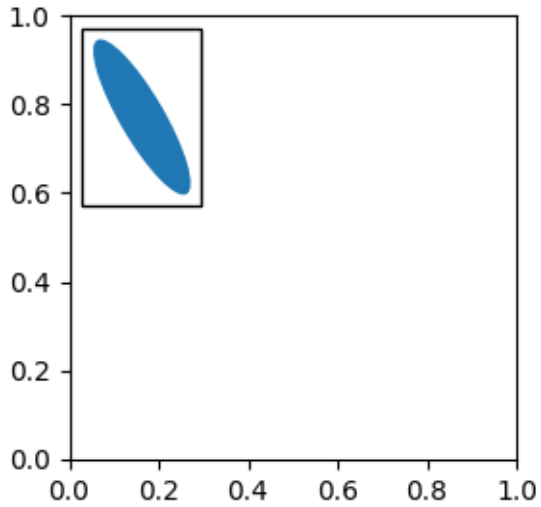
```
from matplotlib.patches import Circle
from mpl_toolkits.axes_grid1.anchored_artists import AnchoredDrawingArea

fig, ax = plt.subplots(figsize=(3, 3))
ada = AnchoredDrawingArea(40, 20, 0, 0,
                          loc='upper right', pad=0., frameon=False)
p1 = Circle((10, 10), 10)
ada.drawing_area.add_artist(p1)
p2 = Circle((30, 10), 5, fc="r")
ada.drawing_area.add_artist(p2)
ax.add_artist(ada)
```




Sometimes, you want your artists to scale with the data coordinate (or coordinates other than canvas pixels). You can use *AnchoredAuxTransformBox* class. This is similar to *AnchoredDrawingArea* except that the extent of the artist is determined during the drawing time respecting the specified transform.

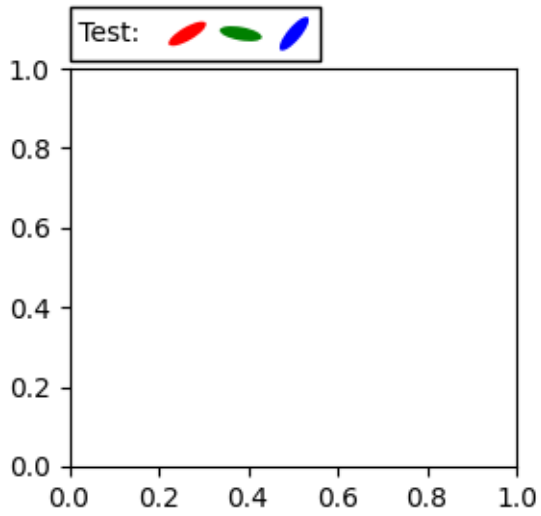
The ellipse in the example below will have width and height corresponding to 0.1 and 0.4 in data coordinates and will be automatically scaled when the view limits of the Axes change.

```
from matplotlib.patches import Ellipse
from mpl_toolkits.axes_grid1.anchored_artists import AnchoredAuxTransformBox

fig, ax = plt.subplots(figsize=(3, 3))
box = AnchoredAuxTransformBox(ax.transData, loc='upper left')
el = Ellipse((0, 0), width=0.1, height=0.4, angle=30)  # in data coordinates!
box.drawing_area.add_artist(el)
ax.add_artist(box)
```



Another method of anchoring an artist relative to a parent Axes or anchor point is via the *bbox_to_anchor* argument of *AnchoredOffsetbox*. This artist can then be automatically positioned relative to another artist using *HPacker* and *VPacker*:

```
from matplotlib.offsetbox import (AnchoredOffsetbox, DrawingArea, HPacker,
                                  TextArea)

fig, ax = plt.subplots(figsize=(3, 3))

box1 = TextArea(" Test: ", textprops=dict(color="k"))
box2 = DrawingArea(60, 20, 0, 0)

el1 = Ellipse((10, 10), width=16, height=5, angle=30, fc="r")
el2 = Ellipse((30, 10), width=16, height=5, angle=170, fc="g")
el3 = Ellipse((50, 10), width=16, height=5, angle=230, fc="b")
box2.add_artist(el1)
box2.add_artist(el2)
box2.add_artist(el3)

box = HPacker(children=[box1, box2],
              align="center",
              pad=0, sep=5)

anchored_box = AnchoredOffsetbox(loc='lower left',
                                 child=box, pad=0.,
                                 frameon=True,
                                 bbox_to_anchor=(0., 1.02),
                                 bbox_transform=ax.transAxes,
                                 borderpad=0.,)

ax.add_artist(anchored_box)
fig.subplots_adjust(top=0.8)
```




Note that, unlike in *Legend*, the `bbox_transform` is set to *IdentityTransform* by default

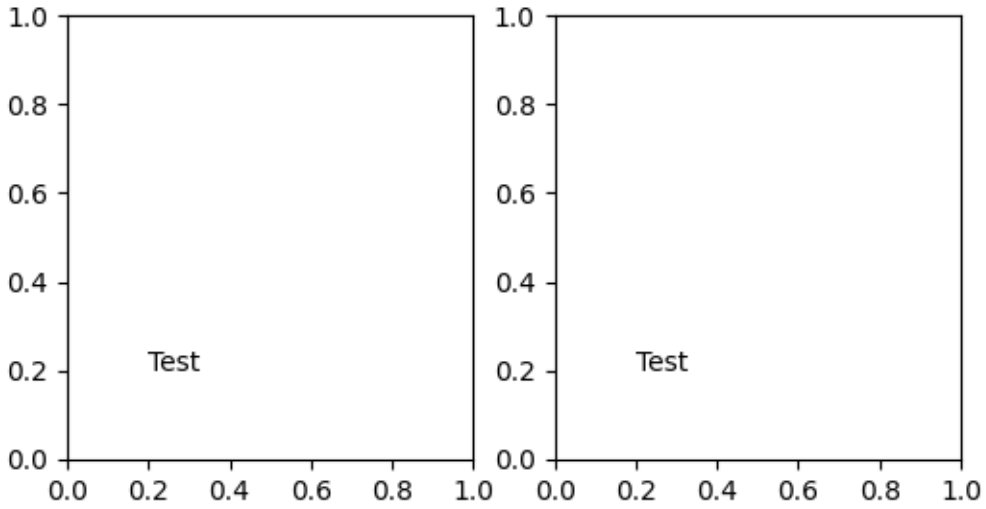
### Coordinate systems for annotations

Matplotlib Annotations support several types of coordinate systems. The examples in *Basic annotation* used the `data` coordinate system; Some others more advanced options are:

#### `Transform` instance

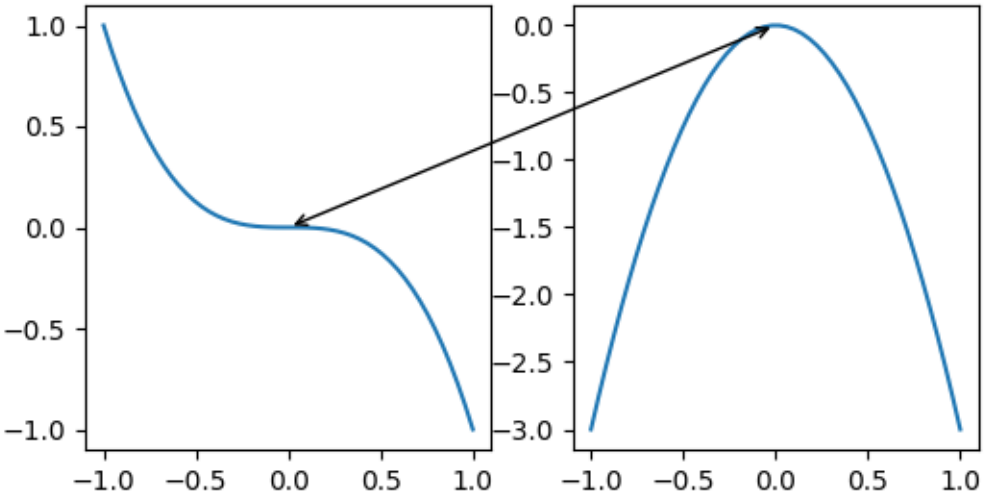
Transforms map coordinates into different coordinate systems, usually the display coordinate system. See *Transformations Tutorial* for a detailed explanation. Here Transform objects are used to identify the coordinate system of the corresponding points. For example, the `Axes.transAxes` transform positions the annotation relative to the Axes coordinates; therefore using it is identical to setting the coordinate system to "axes fraction":

```
fig, (ax1, ax2) = plt.subplots(nrows=1, ncols=2, figsize=(6, 3))
ax1.annotate("Test", xy=(0.2, 0.2), xycoords=ax1.transAxes)
ax2.annotate("Test", xy=(0.2, 0.2), xycoords="axes fraction")
```




Another commonly used *`Transform`* instance is `Axes.transData`. This transform is the coordinate system of the data plotted in the Axes. In this example, it is used to draw an arrow between related data points in two Axes. We have passed an empty text because in this case, the annotation connects data points.

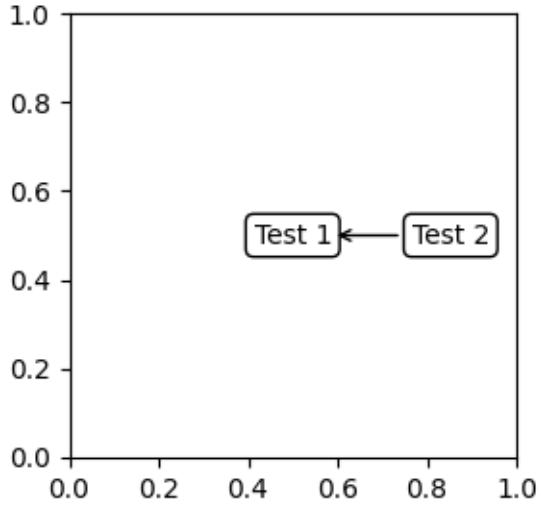
```
x = np.linspace(-1, 1)

fig, (ax1, ax2) = plt.subplots(nrows=1, ncols=2, figsize=(6, 3))
ax1.plot(x, -x**3)
ax2.plot(x, -3*x**2)
ax2.annotate("",
             xy=(0, 0), xycoords=ax1.transData,
             xytext=(0, 0), textcoords=ax2.transData,
             arrowprops=dict(arrowstyle="<->"))
```




### `Artist` instance

The *xy* value (or *xytext*) is interpreted as a fractional coordinate of the bounding box (bbox) of the artist:

```
fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(3, 3))
an1 = ax.annotate("Test 1",
                  xy=(0.5, 0.5), xycoords="data",
                  va="center", ha="center",
                  bbox=dict(boxstyle="round", fc="w"))

an2 = ax.annotate("Test 2",
                  xy=(1, 0.5), xycoords=an1,  # (1, 0.5) of an1's bbox
                  xytext=(30, 0), textcoords="offset points",
                  va="center", ha="left",
                  bbox=dict(boxstyle="round", fc="w"),
                  arrowprops=dict(arrowstyle="->"))
```

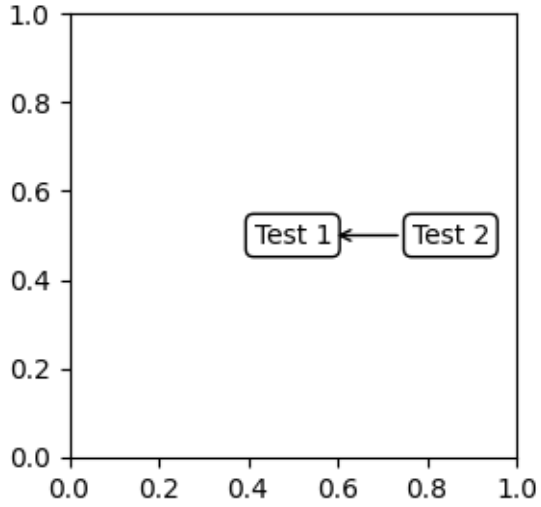



Note that you must ensure that the extent of the coordinate artist (*an1* in this example) is determined before *an2* gets drawn. Usually, this means that *an2* needs to be drawn after *an1*. The base class for all bounding boxes is *BboxBase*

### Callable that returns `Transform` of `BboxBase`

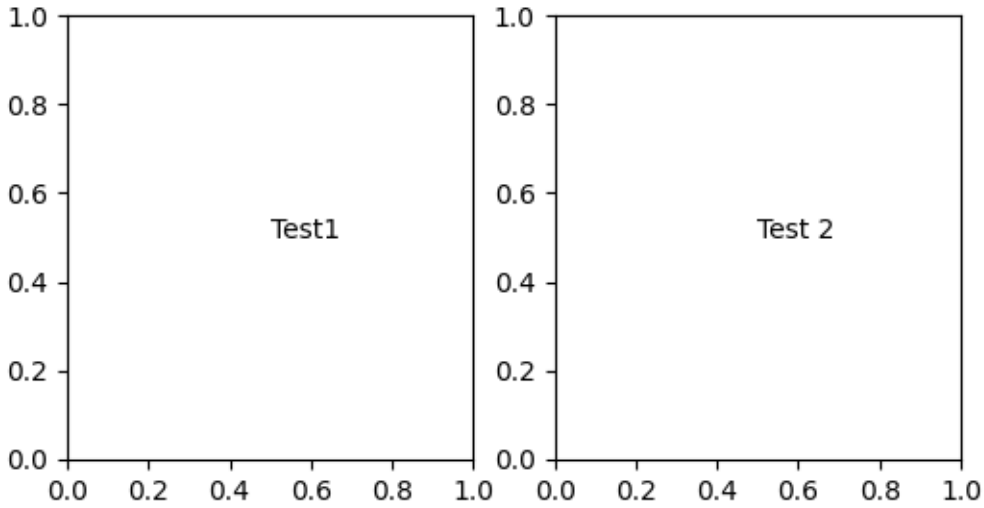
A callable object that takes the renderer instance as single argument, and returns either a *Transform* or a *BboxBase*. For example, the return value of *Artist.get_window_extent* is a bbox, so this method is identical to (2) passing in the artist:

```
fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(3, 3))
an1 = ax.annotate("Test 1",
                  xy=(0.5, 0.5), xycoords="data",
                  va="center", ha="center",
                  bbox=dict(boxstyle="round", fc="w"))

an2 = ax.annotate("Test 2",
                  xy=(1, 0.5), xycoords=an1.get_window_extent,
                  xytext=(30, 0), textcoords="offset points",
                  va="center", ha="left",
                  bbox=dict(boxstyle="round", fc="w"),
                  arrowprops=dict(arrowstyle="->"))
```




*Artist.get_window_extent* is the bounding box of the Axes object and is therefore identical to setting the coordinate system to axes fraction:

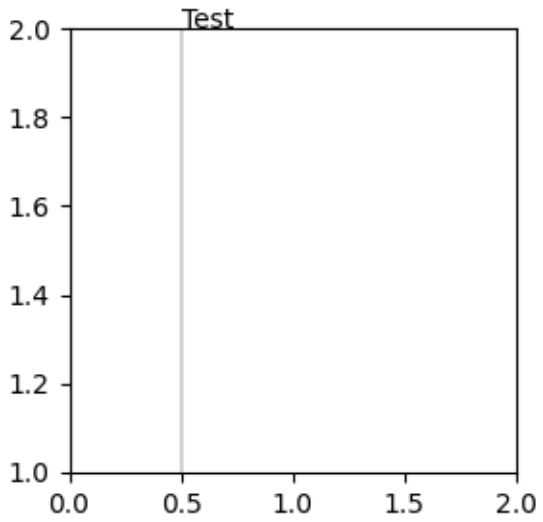
```
fig, (ax1, ax2) = plt.subplots(nrows=1, ncols=2, figsize=(6, 3))

an1 = ax1.annotate("Test1", xy=(0.5, 0.5), xycoords="axes fraction")
an2 = ax2.annotate("Test 2", xy=(0.5, 0.5), xycoords=ax2.get_window_extent)
```



### Blended coordinate specification

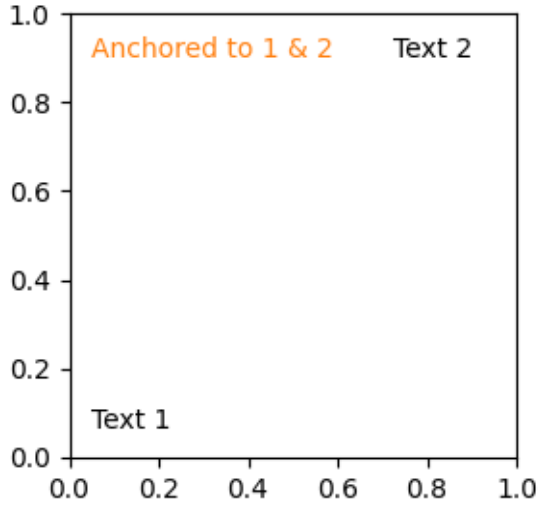
A blended pair of coordinate specifications -- the first for the x-coordinate, and the second is for the y-coordinate. For example, x=0.5 is in data coordinates, and y=1 is in normalized axes coordinates:

```
fig, ax = plt.subplots(figsize=(3, 3))
ax.annotate("Test", xy=(0.5, 1), xycoords=("data", "axes fraction"))
ax.axvline(x=.5, color='lightgray')
ax.set(xlim=(0, 2), ylim=(1, 2))
```




Any of the supported coordinate systems can be used in a blended specification. For example, the text "Anchored to 1 & 2" is positioned relative to the two *Text* Artists:

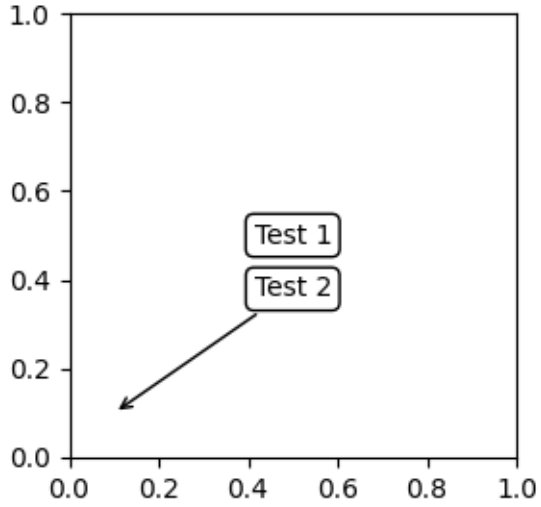
```
fig, ax = plt.subplots(figsize=(3, 3))

t1 = ax.text(0.05, .05, "Text 1", va='bottom', ha='left')
t2 = ax.text(0.90, .90, "Text 2", ha='right')
t3 = ax.annotate("Anchored to 1 & 2", xy=(0, 0), xycoords=(t1, t2),
                 va='bottom', color='tab:orange',)
```




### text.OffsetFrom

Sometimes, you want your annotation with some "offset points", not from the annotated point but from some other point or artist. *text.OffsetFrom* is a helper for such cases.

```
from matplotlib.text import OffsetFrom

fig, ax = plt.subplots(figsize=(3, 3))
an1 = ax.annotate("Test 1", xy=(0.5, 0.5), xycoords="data",
                  va="center", ha="center",
                  bbox=dict(boxstyle="round", fc="w"))

offset_from = OffsetFrom(an1, (0.5, 0))
an2 = ax.annotate("Test 2", xy=(0.1, 0.1), xycoords="data",
                  xytext=(0, -10), textcoords=offset_from,
                  # xytext is offset points from "xy=(0.5, 0), xycoords=an1"
                  va="top", ha="center",
                  bbox=dict(boxstyle="round", fc="w"),
                  arrowprops=dict(arrowstyle="->"))
```

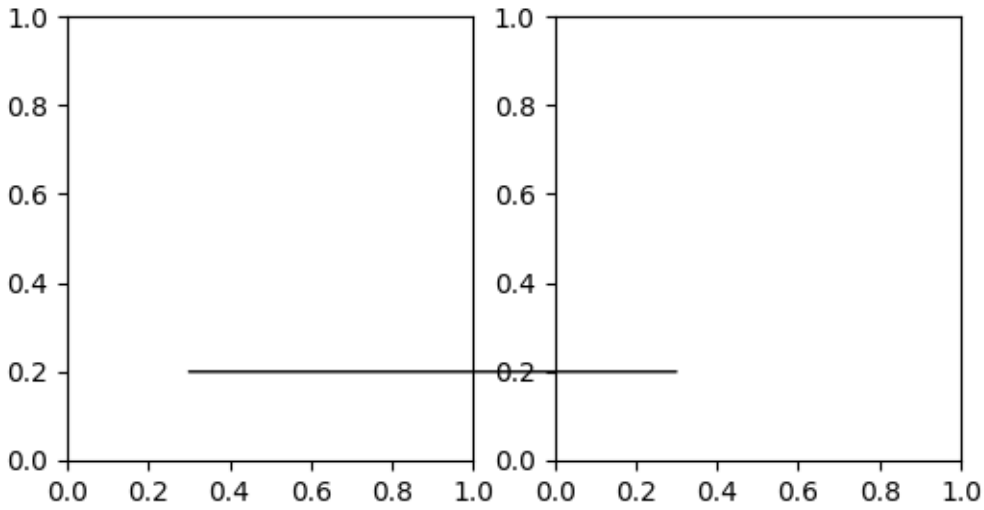



### Non-text annotations

### Using ConnectionPatch

*ConnectionPatch* is like an annotation without text. While *annotate* is sufficient in most situations, *ConnectionPatch* is useful when you want to connect points in different Axes. For example, here we connect the point *xy* in the data coordinates of `ax1` to point *xy* in the data coordinates of `ax2`:

```
from matplotlib.patches import ConnectionPatch

fig, (ax1, ax2) = plt.subplots(nrows=1, ncols=2, figsize=(6, 3))
xy = (0.3, 0.2)
con = ConnectionPatch(xyA=xy, coordsA=ax1.transData,
                      xyB=xy, coordsB=ax2.transData)

fig.add_artist(con)
```




Here, we added the `ConnectionPatch` to the *figure* (with `add_artist`) rather than to either Axes. This ensures that the ConnectionPatch artist is drawn on top of both Axes, and is also necessary when using *constrained_layout* for positioning the Axes.

### Zoom effect between Axes

`mpl_toolkits.axes_grid1.inset_locator` defines some patch classes useful for interconnecting two Axes.

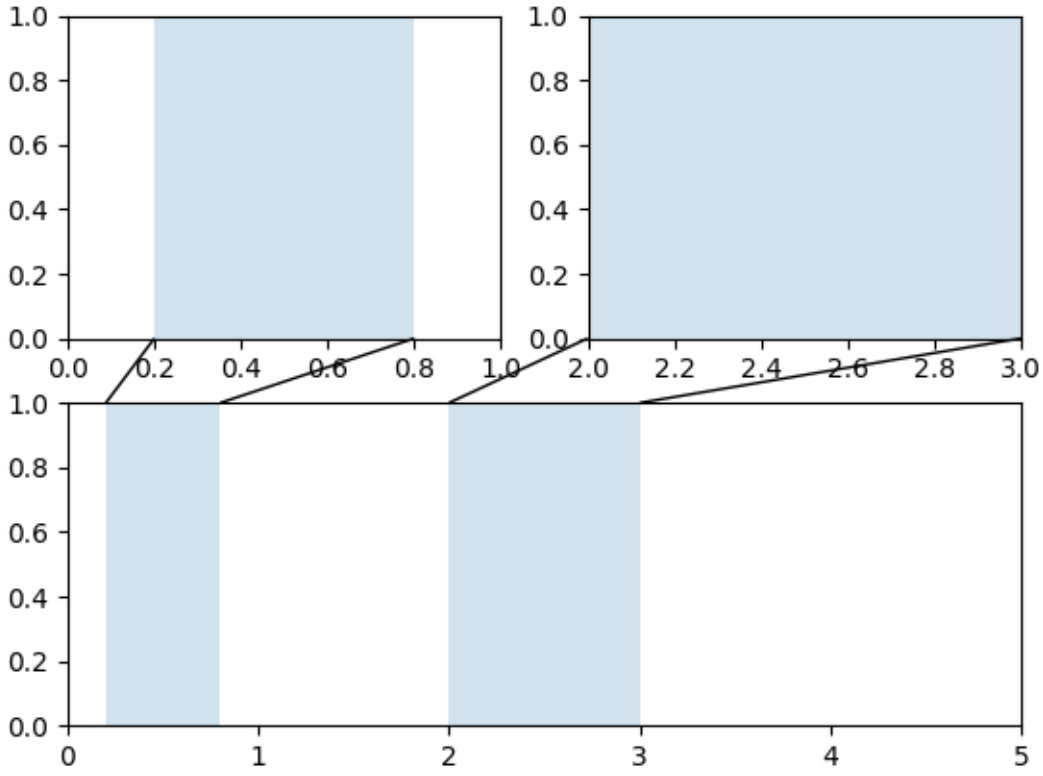
The code for this figure is at *Axes Zoom Effect* and familiarity with *Transformations Tutorial* is recommended.

**Total running time of the script:** (0 minutes 3.327 seconds)

## 3.7.5 Fonts in Matplotlib

Matplotlib needs fonts to work with its text engine, some of which are shipped alongside the installation. The default font is DejaVu Sans which covers most European writing systems. However, users can configure the default fonts, and provide their own custom fonts. See *Customizing text properties* for details and *Text with non-latin glyphs* in particular for glyphs not supported by DejaVu Sans.

Matplotlib also provides an option to offload text rendering to a TeX engine (`usetex=True`), see *Text rendering with LaTeX*.


1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
2.0
2.2
2.4
2.6
2.8
3.0
0
1
2
3
4
5

### Fonts in PDF and PostScript

Fonts have a long (and sometimes incompatible) history in computing, leading to different platforms supporting different types of fonts. In practice, Matplotlib supports three font specifications (in addition to pdf 'core fonts', which are explained later in the guide):

Table 1: Type of Fonts

| Type 1 (PDF) | Type 3 (PDF/PS) | TrueType (PDF) |
|---|---|---|
| One of the oldest types, introduced by Adobe | Similar to Type 1 in terms of introduction | Newer than previous types, used commonly today, introduced by Apple |
| Restricted subset of PostScript, charstrings are in bytecode | Full PostScript language, allows embedding arbitrary code (in theory, even render fractals when rasterizing!) | Include a virtual machine that can execute code! |
| These fonts support font hinting | Do not support font hinting | Hinting supported (virtual machine processes the "hints") |
| Non-subsetted through Matplotlib | Subsetted via external module ttconv | Subsetted via external module fontTools |

**Note:** Adobe disabled support for authoring with Type 1 fonts in January 2023.

Other font specifications which Matplotlib supports:

- Type 42 fonts (PS):
  - PostScript wrapper around TrueType fonts
  - 42 is the Answer to Life, the Universe, and Everything!
  - Matplotlib uses the external library fontTools to subset these types of fonts
- OpenType fonts:
  - OpenType is a new standard for digital type fonts, developed jointly by Adobe and Microsoft
  - Generally contain a much larger character set!
  - Limited support with Matplotlib

### Font subsetting

The PDF and PostScript formats support embedding fonts in files, allowing the display program to correctly render the text, independent of what fonts are installed on the viewer's computer and without the need to pre-rasterize the text. This ensures that if the output is zoomed or resized the text does not become pixelated. However, embedding full fonts in the file can lead to large output files, particularly with fonts with many glyphs such as those that support CJK (Chinese/Japanese/Korean).

The solution to this problem is to subset the fonts used in the document and only embed the glyphs actually used. This gets both vector text and small files sizes. Computing the subset of the font required and writing the new (reduced) font are both complex problem and thus Matplotlib relies on fontTools and a vendored fork of ttconv.

Currently Type 3, Type 42, and TrueType fonts are subsetted. Type 1 fonts are not.

### Core Fonts

In addition to the ability to embed fonts, as part of the PostScript and PDF specification there are 14 Core Fonts that compliant viewers must ensure are available. If you restrict your document to only these fonts you do not have to embed any font information in the document but still get vector text.

This is especially helpful to generate *really lightweight* documents:

```
# trigger core fonts for PDF backend
plt.rcParams["pdf.use14corefonts"] = True
# trigger core fonts for PS backend
plt.rcParams["ps.useafm"] = True

chars = "AFM ftw!"
fig, ax = plt.subplots()
ax.text(0.5, 0.5, chars)

fig.savefig("AFM_PDF.pdf", format="pdf")
fig.savefig("AFM_PS.ps", format="ps")
```

### Fonts in SVG

Text can output to SVG in two ways controlled by *rcParams["svg.fonttype"]* (default: `'path'`):

- as a path (`'path'`) in the SVG
- as string in the SVG with font styling on the element (`'none'`)

When saving via `'path'` Matplotlib will compute the path of the glyphs used as vector paths and write those to the output. The advantage of doing so is that the SVG will look the same on all computers independent of what fonts are installed. However the text will not be editable after the fact. In contrast, saving with `'none'` will result in smaller files and the text will appear directly in the markup. However, the appearance may vary based on the SVG viewer and what fonts are available.

### Fonts in Agg

To output text to raster formats via Agg, Matplotlib relies on FreeType. Because the exact rendering of the glyphs changes between FreeType versions we pin to a specific version for our image comparison tests.

### How Matplotlib selects fonts

Internally, using a font in Matplotlib is a three step process:

1. a *FontProperties* object is created (explicitly or implicitly)
2. based on the *FontProperties* object the methods on *FontManager* are used to select the closest "best" font Matplotlib is aware of (except for `'none'` mode of SVG).
3. the Python proxy for the font object is used by the backend code to render the text -- the exact details depend on the backend via *font_manager.get_font*.

The algorithm to select the "best" font is a modified version of the algorithm specified by the CSS1 Specifications which is used by web browsers. This algorithm takes into account the font family name (e.g. "Arial", "Noto Sans CJK", "Hack", ...), the size, style, and weight. In addition to family names that map directly to fonts there are five "generic font family names" (serif, monospace, fantasy, cursive, and sans-serif) that will internally be mapped to any one of a set of fonts.

Currently the public API for doing step 2 is *FontManager.findfont* (and that method on the global *FontManager* instance is aliased at the module level as *font_manager.findfont*), which will only find a single font and return the absolute path to the font on the filesystem.

### Font fallback

There is no font that covers the entire Unicode space thus it is possible for the users to require a mix of glyphs that cannot be satisfied from a single font. While it has been possible to use multiple fonts within a Figure, on distinct *Text* instances, it was not previous possible to use multiple fonts in the same *Text* instance (as a web browser does). As of Matplotlib 3.6 the Agg, SVG, PDF, and PS backends will "fallback" through multiple fonts in a single *Text* instance:

```
fig, ax = plt.subplots()
ax.text(
    .5, .5, "There are 𓀀𓀁𓀂𓀃 in between!",
    family=['DejaVu Sans', 'Noto Sans CJK JP', 'Noto Sans TC'],
    ha='center'
)
```

Internally this is implemented by setting The "font family" on *FontProperties* objects to a list of font families. A (currently) private API extracts a list of paths to all of the fonts found and then constructs a single *ft2font.FT2Font* object that is aware of all of the fonts. Each glyph of the string is rendered using the first font in the list that contains that glyph.

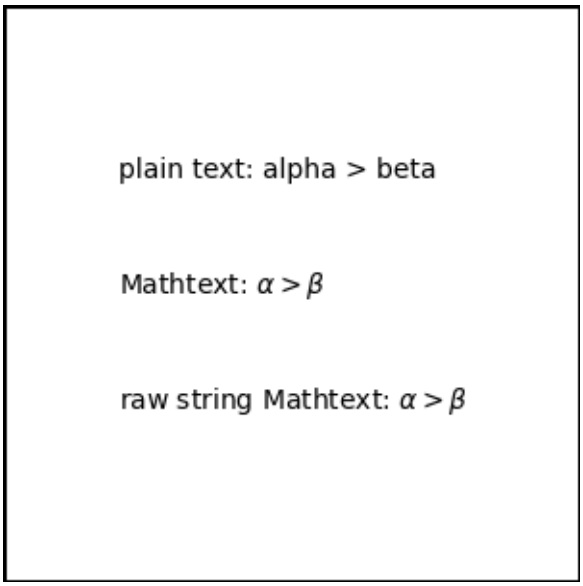
A majority of this work was done by Aitik Gupta supported by Google Summer of Code 2021.

### 3.7.6 Writing mathematical expressions

Matplotlib implements a lightweight TeX expression parser and layout engine and *Mathtext* is the subset of Tex markup that this engine supports. Note that Matplotlib can also render all text directly using TeX if *rcParams["text.usetex"]* (default: `False`) is *True*; see *Text rendering with LaTeX* for more details. Mathtext support is available if *rcParams["text.usetex"]* (default: `False`) is *False*.

Any string can be processed as Mathtext by placing the string inside a pair of dollar signs `'$'`. Mathtext often contains many backslashes `'\'`; so that the backslashes do not need to be escaped, Mathtext is often written using raw strings. For example:

```
import matplotlib.pyplot as plt

fig = plt.figure(figsize=(3, 3), linewidth=1, edgecolor='black')
fig.text(.2, .7, "plain text: alpha > beta")
fig.text(.2, .5, "Mathtext: $\\alpha > \\beta$")
fig.text(.2, .3, r"raw string Mathtext: $\alpha > \beta$")
```




**See also:**

*Mathtext example*

TeX does *not* need to be installed to use Mathtext because Matplotlib ships with the Mathtext parser and engine. The Mathtext layout engine is a fairly direct adaptation of the layout algorithms in Donald Knuth's TeX. To render mathematical text using a different TeX engine, see *Text rendering with LaTeX*.

---

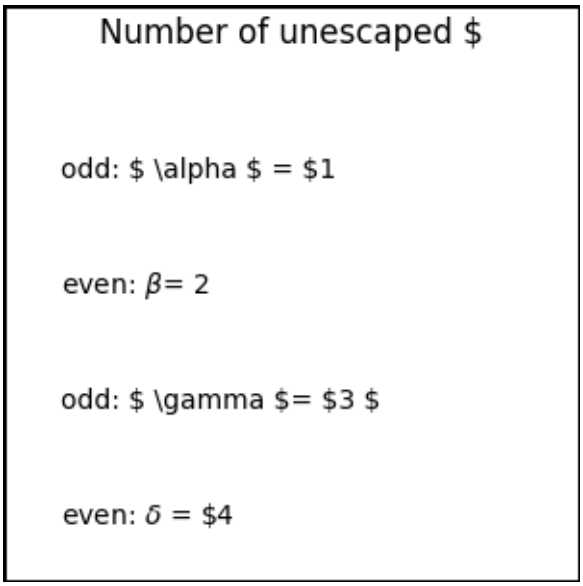
**Note:** To generate html output in documentation that will exactly match the output generated by `mathtext`, use the *matplotlib.sphinxext.mathmpl* Sphinx extension.

---

### Special characters

Mathtext must be placed between a pair of (US) dollar signs `'$'`. A literal dollar symbol `'$'` in a string containing Mathtext must be escaped using a backslash: `'\$'`. A string may contain multiple pairs of dollar signs, resulting in multiple Mathtext expressions. Strings with an odd number of dollar signs are rendered solely as plain text.

```
fig = plt.figure(figsize=(3, 3), linewidth=1, edgecolor='black')
fig.suptitle("Number of unescaped $")
fig.text(.1, .7, r"odd: $ \alpha $ = $1")
fig.text(.1, .5, r"even: $ \beta $= $ 2 $")
fig.text(.1, .3, r'odd: $ \gamma $= \$3 $')
fig.text(.1, .1, r'even: $ \delta $ = $ \$4 $')
```




While Mathtext aims for compatibility with regular TeX, it diverges on when special characters need to be escaped. In TeX the dollar sign must be escaped `'\$'` in non-math text, while in Matplotlib the dollar sign must be escaped when writing Mathtext.

These other special characters are also escaped in non-math TeX, while in Matplotlib their behavior is dependent on how *rcParams["text.usetex"]* (default: `False`) is set:

```
# $ % & ~ _ ^ \ { } \( \) \[ \]
```

See the *usetex tutorial* for more information.

### Subscripts and superscripts

To make subscripts and superscripts, use the `'_'` and `'^'` symbols:

```
r'$\alpha_i > \beta_i$'
```

$$\alpha_i > \beta_i$$

To display multi-letter subscripts or superscripts correctly, you should put them in curly braces `{...}`:

```
r'$\alpha^{ic} > \beta_{ic}$'
```

$$\alpha^{ic} > \beta_{ic}$$

Some symbols automatically put their sub/superscripts under and over the operator. For example, to write the sum of $x_i$ from 0 to $\infty$, you could do:

```
r'$\sum_{i=0}^\infty x_i$'
```

$$\sum_{i=0}^{\infty} x_i$$

### Fractions, binomials, and stacked numbers

Fractions, binomials, and stacked numbers can be created with the `\frac{}{}`, `\binom{}{}` and `\genfrac{}{}{}{}{}{}` commands, respectively:

```
r'$\frac{3}{4} \binom{3}{4} \genfrac{}{}{0}{}{3}{4}$'
```

produces

$$\frac{3}{4}\binom{3}{4}\genfrac{}{}{0}{}{3}{4}$$

Fractions can be arbitrarily nested:

```
r'$\frac{5 - \frac{1}{x}}{4}$'
```

produces

$$\frac{5 - \frac{1}{x}}{4}$$

Note that special care needs to be taken to place parentheses and brackets around fractions. Doing things the obvious way produces brackets that are too small:

```
r'$(\frac{5 - \frac{1}{x}}{4})$'
```

$$(\frac{5 - \frac{1}{x}}{4})$$

The solution is to precede the bracket with `\left` and `\right` to inform the parser that those brackets encompass the entire object.:

```
r'$\left(\frac{5 - \frac{1}{x}}{4}\right)$'
```

$$\left(\frac{5 - \frac{1}{x}}{4}\right)$$

### Radicals

Radicals can be produced with the `\sqrt[]{}` command. For example:

```
r'$\sqrt{2}$'
```

$$\sqrt{2}$$

Any base can (optionally) be provided inside square brackets. Note that the base must be a simple expression, and cannot contain layout commands such as fractions or sub/superscripts:

```
r'$\sqrt[3]{x}$'
```

$$\sqrt[3]{x}$$

### Fonts

The default font is *italics* for mathematical symbols.

This default can be changed using `rcParams["mathtext.default"]` (default: `'it'`). For setting rcParams, see *Customizing Matplotlib with style sheets and rcParams*. For example, setting the default to `regular` allows you to use the same font for math text and regular non-math text.

To change fonts, e.g., to write "sin" in a Roman font, enclose the text in a font command:

```
r'$s(t) = \mathcal{A}\mathrm{sin}(2 \omega t)$'
```

$$s(t) = \mathcal{A}\mathrm{sin}(2\omega t)$$

More conveniently, many commonly used function names that are typeset in a Roman font have shortcuts. So the expression above could be written as follows:

```
r'$s(t) = \mathcal{A}\sin(2 \omega t)$'
```

$$s(t) = \mathcal{A}\,\sin(2\omega t)$$

Here "s" and "t" are variable in italics font (default), "sin" is in Roman font, and the amplitude "A" is in calligraphy font. Note in the example above the calligraphy `A` is squished into the `sin`. You can use a spacing command to add a little whitespace between them:

```
r's(t) = \mathcal{A}\/\sin(2 \omega t)'
```

$$s(t) = \mathcal{A}\ \sin(2\omega t)$$

Mathtext can use DejaVu Sans (default), DejaVu Serif, Computer Modern fonts from (La)TeX, STIX fonts which are designed to blend well with Times, or a Unicode font that you provide. The Mathtext font can be selected via `rcParams["mathtext.fontset"]` (default: `'dejavusans'`).

The choices available with all fonts are:

| Command | Result |
|---|---|
| `\mathrm{Roman}` | Roman |
| `\mathit{Italic}` | *Italic* |
| `\mathtt{Typewriter}` | `Typewriter` |
| `\mathcal{CALLIGRAPHY}` | $\mathcal{CALLIGRAPHY}$ |

When using the STIX fonts, you also have the choice of:

| Command | Result |
|---|---|
| `\mathbb{blackboard}` | $\mathbb{blackboard}$ |
| `\mathrm{\mathbb{blackboard}}` | $\mathrm{\mathbb{blackboard}}$ |
| `\mathfrak{Fraktur}` | $\mathfrak{Fraktur}$ |
| `\mathsf{sansserif}` | sansserif |
| `\mathrm{\mathsf{sansserif}}` | sansserif |
| `\mathbfit{bolditalic}` | $\mathbfit{bolditalic}$ |

There are also five global "font sets" to choose from, which are selected using the `mathtext.fontset` parameter in *matplotlibrc*.

**`dejavusans`: DejaVu Sans**

$$\mathcal{R} \prod_{i=\alpha}^{\infty} a_i \sin\left(2\pi f x_i\right) \tag{3.1}$$

**`dejavuserif`: DejaVu Serif**

$$\mathcal{R} \prod_{i=\alpha}^{\infty} a_i \sin\left(2\pi f x_i\right) \tag{3.2}$$

**`cm`: Computer Modern (TeX)**

$$\mathcal{R} \prod_{i=\alpha}^{\infty} a_i \sin\left(2\pi f x_i\right) \tag{3.3}$$

**`stix`: STIX (designed to blend well with Times)**

$$\mathscr{R} \prod_{i=\alpha}^{\infty} a_i \sin\left(2\pi f x_i\right) \tag{3.4}$$

**`stixsans`: STIX sans-serif**

$$\mathscr{R} \prod_{i=\alpha}^{\infty} a_i \sin\left(2\pi f x_i\right) \tag{3.5}$$

Additionally, you can use `\mathdefault{...}` or its alias `\mathregular{...}` to use the font used for regular text outside of Mathtext. There are a number of limitations to this approach, most notably that far fewer symbols will be available, but it can be useful to make math expressions blend well with other text in the plot.

For compatibility with popular packages, `\text{...}` is available and uses the `\mathrm{...}` font, but otherwise retains spaces and renders - as a dash (not minus).

### Custom fonts

Mathtext also provides a way to use custom fonts for math. This method is fairly tricky to use, and should be considered an experimental feature for patient users only. By setting *`rcParams["mathtext.fontset"]`* (default: `'dejavusans'`) to `custom`, you can then set the following parameters, which control which font file to use for a particular set of math characters.

| Parameter | Corresponds to |
|---|---|
| `mathtext.it` | `\mathit{}` or default italic |
| `mathtext.rm` | `\mathrm{}` Roman (upright) |
| `mathtext.tt` | `\mathtt{}` Typewriter (monospace) |
| `mathtext.bf` | `\mathbf{}` bold |
| `mathtext.bfit` | `\mathbfit{}` bold italic |
| `mathtext.cal` | `\mathcal{}` calligraphic |
| `mathtext.sf` | `\mathsf{}` sans-serif |

Each parameter should be set to a fontconfig font descriptor, as defined in *Fonts in Matplotlib*. The fonts used should have a Unicode mapping in order to find any non-Latin characters, such as Greek. If you want to use a math symbol that is not contained in your custom fonts, you can set *`rcParams["mathtext.fallback"]`* (default: `'cm'`) to either `'cm'`, `'stix'` or `'stixsans'` which will cause the Mathtext system to use characters from an alternative font whenever a particular character cannot be found in the custom font.

Note that the math glyphs specified in Unicode have evolved over time, and many fonts may not have glyphs in the correct place for Mathtext.

### Accents

An accent command may precede any symbol to add an accent above it. There are long and short forms for some of them.

| Command | Result |
|---|---|
| `\acute a` or `\'a` | $\acute{a}$ |
| `\bar a` | $\bar{a}$ |
| `\breve a` | $\breve{a}$ |
| `\dot a` or `\.a` | $\dot{a}$ |
| `\ddot a` or `\''a` | $\ddot{a}$ |
| `\dddot a` | $\dddot{a}$ |
| `\ddddot a` | $\ddddot{a}$ |
| `\grave a` or `\`a` | $\grave{a}$ |
| `\hat a` or `\^a` | $\hat{a}$ |
| `\tilde a` or `\~a` | $\tilde{a}$ |
| `\vec a` | $\vec{a}$ |
| `\overline{abc}` | $\overline{abc}$ |

In addition, there are two special accents that automatically adjust to the width of the symbols below:

| Command | Result |
|---|---|
| `\widehat{xyz}` | $\widehat{xyz}$ |
| `\widetilde{xyz}` | $\widetilde{xyz}$ |

Care should be taken when putting accents on lower-case i's and j's. Note that in the following `\imath` is used to avoid the extra dot over the i:

```
r"$\hat i\ \ \hat \imath$"
```

$\hat{i}\ \hat{\imath}$

### Symbols

You can also use a large number of the TeX symbols, as in `\infty`, `\leftarrow`, `\sum`, `\int`.

**Lower-case Greek**

| | | | |
|---|---|---|---|
| α \alpha | β \beta | γ \gamma | δ \delta |
| ε \epsilon | ε \varepsilon | ζ \zeta | η \eta |
| θ \theta | ϑ \vartheta | ι \iota | κ \kappa |
| ϰ \varkappa | λ \lambda | μ \mu | ν \nu |
| ξ \xi | π \pi | ϖ \varpi | ρ \rho |
| ϱ \varrho | σ \sigma | ς \varsigma | τ \tau |
| υ \upsilon | χ \chi | ψ \psi | ω \omega |
| ϕ \phi | φ \varphi | ϝ \digamma | |

**Upper-case Greek**

| | | | |
|---|---|---|---|
| Γ \Gamma | Δ \Delta | Θ \Theta | Λ \Lambda |
| Ξ \Xi | Π \Pi | Σ \Sigma | Υ \Upsilon |
| Φ \Phi | Ψ \Psi | Ω \Omega | |

**Hebrew**

| | | | |
|---|---|---|---|
| ד \daleth | ג \gimel | ב \beth | א \aleph |

**Latin named characters**

| | | | | | |
|---|---|---|---|---|---|
| Å \AA | Æ \AE | Ð \DH | Ø \O | Þ \Thorn | ß \ss |
| å \aa | æ \ae | ð \eth | ð \dh | ø \o | þ \thorn |
| Œ \OE | œ \oe | | | | |

**Delimiters**

| | | | | |
|---|---|---|---|---|
| ( ( | ( \leftparen | ) \rightparen | ) ) | . . |
| / / | < < | > > | [ [ | [ \lbrack |
| \ \backslash | \| \vert | \| \| | ] ] | ] \rbrack |
| { \leftbrace | { \{ | { \lbrace | } \} | } \rightbrace |
| } \rbrace | ‖ \Vert | ‖ \\| | ↑ \uparrow | ↓ \downarrow |
| ↕ \updownarrow | ⇑ \Uparrow | ⇓ \Downarrow | ⇕ \Updownarrow | ⌈ \lceil |
| ⌉ \rceil | ⌊ \lfloor | ⌋ \rfloor | ⟨ \langle | ⟩ \rangle |
| ( \lgroup | ) \rgroup | | | |

**Big symbols**

| | | | | |
|---|---|---|---|---|
| ∏ \prod | ∐ \coprod | ∑ \sum | ∫ \int | ∬ \iint |
| ∭ \iiint | ∮ \oint | ∯ \oiint | ∰ \oiiint | ⋀ \bigwedge |
| ⋁ \bigvee | ⋂ \bigcap | ⋃ \bigcup | ⨀ \bigodot | ⨁ \bigoplus |
| ⨂ \bigotimes | ⨄ \biguplus | ⨆ \bigsqcup | ⨌ \iiiint | |

### Standard function names

| | | | | |
|---|---|---|---|---|
| Pr \Pr | arccos \arccos | arcsin \arcsin | arctan \arctan | arg \arg |
| cos \cos | cosh \cosh | cot \cot | coth \coth | csc \csc |
| deg \deg | det \det | dim \dim | exp \exp | gcd \gcd |
| hom \hom | inf \inf | ker \ker | lg \lg | lim \lim |
| liminf \liminf | limsup \limsup | ln \ln | log \log | max \max |
| min \min | sec \sec | sin \sin | sinh \sinh | sup \sup |
| tan \tan | tanh \tanh | | | |

### Binary operation symbols

| | | | |
|---|---|---|---|
| * * | + + | - − | ± \pm |
| × \times | ÷ \div | † \dagger | ‡ \ddagger |
| − − | ∓ \mp | ∔ \dotplus | / \slash |
| ∖ \setminus | ∗ \ast | ∘ \circ | • \bullet |
| ∧ \wedge | ∨ \vee | ∩ \cap | ∪ \cup |
| ∸ \dotminus | ∹ \minuscolon | ∺ \dotsminusdots | ≀ \wr |
| ⊍ \cupdot | ⊎ \uplus | ⊓ \sqcap | ⊔ \sqcup |
| ⊕ \oplus | ⊖ \ominus | ⊗ \otimes | ⊘ \oslash |
| ⊙ \odot | ⊚ \circledcirc | ⊛ \circledast | ⊝ \circleddash |
| ⊞ \boxplus | ⊟ \boxminus | ⊠ \boxtimes | ⊡ \boxdot |
| ⊴ \unlhd | ⊵ \unrhd | ⊺ \intercal | ⊻ \veebar |
| ⊼ \barwedge | ⊽ \barvee | ⋄ \diamond | ⋅ \cdot |
| ⋆ \star | ⋇ \divideontimes | ⋋ \leftthreetimes | ⋌ \rightthreetimes |
| ⋎ \curlyvee | ⋏ \curlywedge | ⋒ \Cap | ⋓ \Cup |
| ⊼ \doublebarwedge | ⦶ \obar | △ \bigtriangleup | ▷ \triangleright |
| ▷ \rhd | ▽ \bigtriangledown | ◁ \triangleleft | ◁ \lhd |
| ○ \bigcirc | ◫ \boxbar | ⨿ \amalg | ⩛ \merge |

### Relation symbols

| | | | |
|---|---|---|---|
| : : | < < | = = | = \equal |
| > > | ϶ \backepsilon | … \dots | ∈ \in |
| ∉ \notin | ∈ \smallin | ∋ \ni | ∌ \notsmallowns |
| ∋ \smallowns | ∝ \propto | ∝ \varpropto | ∟ \rightangle |
| ∣ \mid | ∤ \nmid | ∥ \parallel | ∦ \nparallel |
| ∴ \therefore | ∵ \because | ∶ \ratio | ∼ \sim |
| ∽ \backsim | ≁ \nsim | ≂ \eqsim | ≃ \simeq |
| ≄ \nsimeq | ≅ \cong | ≆ \simneqq | ≇ \ncong |
| ≈ \approx | ≉ \napprox | ≊ \approxeq | ≋ \approxident |

continues on next

Table 2 – continued from previous page

| | | | |
|---|---|---|---|
| ≦ \backcong | ≍ \asymp | ≎ \Bumpeq | ≏ \bumpeq |
| ≐ \doteq | ≑ \doteqdot | ≑ \Doteq | ≒ \fallingdotse |
| ≓ \risingdotseq | ≔ \coloneq | ≕ \eqcolon | ≖ \eqcirc |
| ≗ \circeq | ≘ \arceq | ≙ \wedgeq | ≚ \veeeq |
| ≛ \stareq | ≜ \triangleq | ≜ \triangleeq | ≝ \eqdef |
| ≞ \measeq | ≟ \questeq | ≠ \neq | ≠ \ne |
| ≡ \equiv | ≢ \nequiv | ≣ \Equiv | ≤ \leq |
| ≥ \geq | ≦ \leqq | ≧ \geqq | ≨ \lneqq |
| ≩ \gneqq | ≪ \ll | ≫ \gg | ≬ \between |
| ≮ \nless | ≯ \ngtr | ≰ \nleq | ≱ \ngeq |
| ≲ \lesssim | ≳ \gtrsim | ≴ \nlesssim | ≵ \ngtrsim |
| ≶ \lessgtr | ≷ \gtrless | ≸ \nlessgtr | ≹ \ngtrless |
| ≺ \prec | ≻ \succ | ≼ \preccurlyeq | ≼ \preceq |
| ≽ \succeq | ≽ \succcurlyeq | ≾ \precsim | ≿ \succsim |
| ⊀ \nprec | ⊁ \nsucc | ⊂ \subset | ⊃ \supset |
| ⊄ \nsubset | ⊅ \nsupset | ⊆ \subseteq | ⊇ \supseteq |
| ⊈ \nsubseteq | ⊉ \nsupseteq | ⊊ \subsetneq | ⊋ \supsetneq |
| ⊏ \sqsubset | ⊐ \sqsupset | ⊑ \sqsubseteq | ⊒ \sqsupseteq |
| ⊜ \oequal | ⊢ \vdash | ⊣ \dashv | ⊤ \top |
| ⊥ \bot | ⊦ \rightassert | ⊨ \models | ⊨ \vDash |
| ⊩ \Vdash | ⊪ \Vvdash | ⊫ \rightModels | ⊬ \nvdash |
| ⊭ \nvDash | ⊮ \nVdash | ⊯ \nVDash | ⊱ \scurel |
| ⊴ \trianglelefteq | ⊵ \trianglerighteq | ⊾ \measuredrightangle | ⊿ \varlrtriangl |
| ⋈ \bowtie | ⋉ \ltimes | ⋊ \rtimes | ⋍ \backsimeq |
| ⋐ \Subset | ⋑ \Supset | ⋔ \pitchfork | ⋕ \equalparalle |
| ⋖ \lessdot | ⋗ \gtrdot | ⋘ \lll | ⋙ \ggg |
| ⋚ \lesseqgtr | ⋛ \gtreqless | ⋜ \eqless | ⋝ \eqgtr |
| ⋞ \curlyeqprec | ⋟ \curlyeqsucc | ⋠ \npreccurlyeq | ⋡ \nsucccurlyeq |
| ⋢ \nsqsubseteq | ⋣ \nsqsupseteq | ⋤ \sqsubsetneq | ⋥ \sqsupsetneq |
| ⋦ \lnsim | ⋧ \gnsim | ⋨ \precnsim | ⋩ \succnsim |
| ⋪ \ntriangleleft | ⋫ \ntriangleright | ⋬ \ntrianglelefteq | ⋭ \ntrianglerig |
| ⋲ \disin | ⋹ \isins | ⋺ \varisins | ⋾ \isindot |
| ⋿ \isinobar | ⋽ \varisinobar | ⋼ \isinvb | ⋻ \isinE |
| ⋺ \nisd | ⋾ \nis | ⋽ \varnis | ⋿ \niobar |
| ⋻ \varniobar | ⌅ \bagmember | ⌢ \frown | ⌣ \smile |
| △ \triangle | ▶ \blacktriangleright | ▷ \triangleright | ▷ \vartriangler |
| ◀ \blacktriangleleft | ◁ \triangleleft | ◁ \vartriangleleft | ⊥ \perp |
| ⋈ \Join | ⩽ \leqslant | ⩾ \geqslant | ⪅ \lessapprox |
| ⪆ \gtrapprox | ⪉ \lnapprox | ⪊ \gnapprox | ⪋ \lesseqqgtr |
| ⪌ \gtreqqless | ⪕ \eqslantless | ⪖ \eqslantgtr | ⪷ \precapprox |
| ⪸ \succapprox | ⪹ \precnapprox | ⪺ \succnapprox | ⫅ \subseteqq |
| ⫆ \supseteqq | ⫋ \subsetneqq | ⫌ \supsetneqq | |

**Arrow symbols**

| | | | |
|---|---|---|---|
| ← \overleftarrow | \overleftrightarrow | ← \leftarrow | ↑ \uparrow |
| → \rightarrow | → \to | ↓ \downarrow | ↔ \leftrightarrow |
| ↕ \updownarrow | ↖ \nwarrow | ↗ \nearrow | ↘ \searrow |
| ↙ \swarrow | ↚ \nleftarrow | ↛ \nrightarrow | ↜ \leftsquigarrow |
| ⇝ \rightsquigarrow | ↞ \twoheadleftarrow | ↑ \twoheaduparrow | ↠ \twoheadrightarrow |
| ↡ \twoheaddownarrow | ↢ \leftarrowtail | ↣ \rightarrowtail | ↤ \mapsfrom |
| ↥ \mapsup | ↦ \mapsto | ↧ \mapsdown | ↕ \updownarrowbar |
| ↩ \hookleftarrow | ↪ \hookrightarrow | ↫ \looparrowleft | ↬ \looparrowright |
| ↭ \leftrightsquigarrow | ↮ \nleftrightarrow | ↯ \downzigzagarrow | ↰ \Lsh |
| ↱ \Rsh | ↲ \Ldsh | ↳ \Rdsh | ↶ \curvearrowleft |
| ↷ \curvearrowright | ↺ \circlearrowleft | ↻ \cwopencirclearrow | ↻ \circlearrowright |
| ↼ \leftharpoonup | ↽ \leftharpoondown | ↾ \upharpoonright | ↿ \upharpoonleft |
| ⇀ \rightharpoonup | ⇁ \rightharpoondown | ⇂ \downharpoonright | ⇃ \downharpoonleft |
| ⇄ \rightleftarrows | ⇅ \updownarrows | ⇆ \leftrightarrows | ⇇ \leftleftarrows |
| ⇈ \upuparrows | ⇉ \rightrightarrows | ⇊ \downdownarrows | ⇋ \leftrightharpoons |
| ⇌ \rightleftharpoons | ⇍ \nLeftarrow | ⇎ \nLeftrightarrow | ⇏ \nRightarrow |
| ⇐ \Leftarrow | ⇑ \Uparrow | ⇒ \Rightarrow | ⇓ \Downarrow |
| ⇔ \Leftrightarrow | ⇕ \Updownarrow | ⇖ \Nwarrow | ⇗ \Nearrow |
| ⇘ \Searrow | ⇙ \Swarrow | ⇚ \Lleftarrow | ⇛ \Rrightarrow |
| ⇝ \rightzigzagarrow | ⇝ \leadsto | ⇤ \barleftarrow | ⇥ \rightarrowbar |
| ⊌ \cupleftarrow | ⊸ \multimap | ⟵ \longleftarrow | ⟶ \longrightarrow |
| ⟷ \longleftrightarrow | ⟸ \Longleftarrow | ⟹ \Longrightarrow | ⟺ \Longleftrightarrow |
| ⟼ \longmapsto | ⇠ \dashleftarrow | ⇢ \dashrightarrow | |

**Dot symbols**

| | | | |
|---|---|---|---|
| … \ldots | ∴ \therefore | ∵ \because | ∷ \Colon |
| ⋮ \vdots | ⋯ \cdots | ⋰ \adots | ⋱ \ddots |

**Black-board characters**

| | | | | | |
|---|---|---|---|---|---|
| $\mathbb{C}$ \BbbC | $\mathbb{N}$ \BbbN | $\mathbb{P}$ \BbbP | $\mathbb{Q}$ \BbbQ | $\mathbb{R}$ \BbbR | $\mathbb{Z}$ \BbbZ |

**Script characters**

| | | | | | |
|---|---|---|---|---|---|
| $\mathscr{g}$ \scrg | $\mathscr{H}$ \scrH | $\mathscr{I}$ \scrI | $\mathscr{L}$ \scrL | $\mathscr{R}$ \scrR | $\mathscr{B}$ \scrB |
| e \scre | $\mathscr{E}$ \scrE | $\mathscr{F}$ \scrF | $\mathscr{M}$ \scrM | $\mathscr{o}$ \scro | |

**Fraktur characters**

| | |
|---|---|
| $\mathfrak{Z}$ \frakZ | $\mathfrak{C}$ \frakC |

**Miscellaneous symbols**

| | | | |
|---|---|---|---|
| $ \$ | ¢ \cent | £ \sterling | ¥ \yen |
| § \S | © \copyright | ¬ \neg | ® \circledR |
| ° \degree | ¶ \P | ħ \hbar | ı \imath |
| ı \i | Ł \L | ł \l | ƛ \lambdabar |
| ȷ \jmath | † \dag | ‡ \ddag | ‰ \perthousand |
| ′ \prime | ‵ \backprime | $\hbar$ \hslash | $\Im$ \Im |
| $\ell$ \ell | $\wp$ \wp | $\Re$ \Re | $\mho$ \mho |
| $\exists$ \Finv | ⅁ \Game | $\forall$ \forall | $\complement$ \complement |
| $\partial$ \partial | $\exists$ \exists | $\nexists$ \nexists | $\varnothing$ \emptyset |
| ∅ \varnothing | $\Delta$ \increment | $\nabla$ \nabla | ■ \QED |
| $\infty$ \infty | $\angle$ \angle | $\measuredangle$ \measuredangle | $\sphericalangle$ \sphericalangle |
| ∾ \ac | ∿ \sinewave | ⊹ \hermitmatrix | Ⓢ \circledS |
| ■ \blacksquare | △ \triangle | $\vartriangle$ \vartriangle | ▲ \blacktriangle |
| ▼ \blacktriangledown | ▽ \triangledown | ★ \bigstar | ☡ \danger |
| ♠ \spadesuit | ♡ \heartsuit | ♢ \diamondsuit | ♣ \clubsuit |
| ♧ \clubsuitopen | ♭ \flat | ♮ \natural | ♯ \sharp |
| ✓ \checkmark | ✠ \maltese | | |

If a particular symbol does not have a name (as is true of many of the more obscure symbols in the STIX fonts), Unicode characters can also be used:

```
r'$\u23ce$'
```

### 3.7.7 Text rendering with XeLaTeX/LuaLaTeX via the `pgf` backend

Using the `pgf` backend, Matplotlib can export figures as pgf drawing commands that can be processed with pdflatex, xelatex or lualatex. XeLaTeX and LuaLaTeX have full Unicode support and can use any font that is installed in the operating system, making use of advanced typographic features of OpenType, AAT and Graphite. Pgf pictures created by `plt.savefig('figure.pgf')` can be embedded as raw commands in LaTeX documents. Figures can also be directly compiled and saved to PDF with `plt.savefig('figure.pdf')` by switching the backend

```
matplotlib.use('pgf')
```

or by explicitly requesting the use of the `pgf` backend

```
plt.savefig('figure.pdf', backend='pgf')
```

or by registering it for handling pdf output

```
from matplotlib.backends.backend_pgf import FigureCanvasPgf
matplotlib.backend_bases.register_backend('pdf', FigureCanvasPgf)
```

The last method allows you to keep using regular interactive backends and to save xelatex, lualatex or pdflatex compiled PDF files from the graphical user interface. Note that, in that case, the interactive display will still use the standard interactive backends (e.g., QtAgg), and in particular use latex to compile relevant text snippets.

Matplotlib's pgf support requires a recent LaTeX installation that includes the TikZ/PGF packages (such as TeXLive), preferably with XeLaTeX or LuaLaTeX installed. If either pdftocairo or ghostscript is present on your system, figures can optionally be saved to PNG images as well. The executables for all applications must be located on your *PATH*.

*rcParams* that control the behavior of the pgf backend:

| Parameter | Documentation |
|---|---|
| pgf.preamble | Lines to be included in the LaTeX preamble |
| pgf.rcfonts | Setup fonts from rc params using the fontspec package |
| pgf.texsystem | Either "xelatex" (default), "lualatex" or "pdflatex" |

**Note:** TeX defines a set of special characters, such as:

```
# $ % & ~ _ ^ \ { }
```

Generally, these characters must be escaped correctly. For convenience, some characters (_, ^, %) are automatically escaped outside of math environments. Other characters are not escaped as they are commonly needed in actual TeX expressions. However, one can configure TeX to treat them as "normal" characters (known as "catcode 12" to TeX) via a custom preamble, such as:

```
plt.rcParams["pgf.preamble"] = (
    r"\AtBeginDocument{\catcode`\&=12\catcode`\#=12}")
```

### Multi-Page PDF Files

The pgf backend also supports multipage pdf files using *PdfPages*

```
from matplotlib.backends.backend_pgf import PdfPages
import matplotlib.pyplot as plt

with PdfPages('multipage.pdf', metadata={'author': 'Me'}) as pdf:

    fig1, ax1 = plt.subplots()
    ax1.plot([1, 5, 3])
    pdf.savefig(fig1)

    fig2, ax2 = plt.subplots()
    ax2.plot([1, 5, 3])
    pdf.savefig(fig2)
```

### Font specification

The fonts used for obtaining the size of text elements or when compiling figures to PDF are usually defined in the *rcParams*. You can also use the LaTeX default Computer Modern fonts by clearing the lists for *rcParams["font.serif"]* (default: `['DejaVu Serif', 'Bitstream Vera Serif', 'Computer Modern Roman', 'New Century Schoolbook', 'Century Schoolbook L', 'Utopia', 'ITC Bookman', 'Bookman', 'Nimbus Roman No9 L', 'Times New Roman', 'Times', 'Palatino', 'Charter', 'serif']`), *rcParams["font.sans-serif"]* (default: `['DejaVu Sans', 'Bitstream Vera Sans', 'Computer Modern Sans Serif', 'Lucida Grande', 'Verdana', 'Geneva', 'Lucid', 'Arial', 'Helvetica', 'Avant Garde', 'sans-serif']`) or *rcParams["font.monospace"]* (default: `['DejaVu Sans Mono', 'Bitstream Vera Sans Mono', 'Computer Modern Typewriter', 'Andale Mono', 'Nimbus Mono L', 'Courier New', 'Courier', 'Fixed', 'Terminal', 'monospace']`). Please note that the glyph coverage of these fonts is very limited. If you want to keep the Computer Modern font face but require extended Unicode support, consider installing the Computer Modern Unicode fonts *CMU Serif*, *CMU Sans Serif*, etc.

When saving to `.pgf`, the font configuration Matplotlib used for the layout of the figure is included in the header of the text file.

```
"""
=========
PGF fonts
=========
"""

import matplotlib.pyplot as plt

plt.rcParams.update({
```

(continues on next page)

(continued from previous page)

```
    "font.family": "serif",
    # Use LaTeX default serif font.
    "font.serif": [],
    # Use specific cursive fonts.
    "font.cursive": ["Comic Neue", "Comic Sans MS"],
})

fig, ax = plt.subplots(figsize=(4.5, 2.5))

ax.plot(range(5))

ax.text(0.5, 3., "serif")
ax.text(0.5, 2., "monospace", family="monospace")
ax.text(2.5, 2., "sans-serif", family="DejaVu Sans")  # Use specific sans␣
↪font.
ax.text(2.5, 1., "comic", family="cursive")
ax.set_xlabel("µ is not $\\mu$")

fig.tight_layout(pad=.5)
```

### Custom preamble

Full customization is possible by adding your own commands to the preamble. Use *rcParams["pgf.preamble"]* (default: `''`) if you want to configure the math fonts, using `unicode-math` for example, or for loading additional packages. Also, if you want to do the font configuration yourself instead of using the fonts specified in the rc parameters, make sure to disable *rcParams["pgf.rcfonts"]* (default: `True`).

```
"""
============
PGF preamble
============
"""

import matplotlib as mpl

mpl.use("pgf")
```

### Choosing the TeX system

The TeX system to be used by Matplotlib is chosen by *rcParams["pgf.texsystem"]* (default: `'xelatex'`). Possible values are `'xelatex'` (default), `'lualatex'` and `'pdflatex'`. Please note that when selecting pdflatex, the fonts and Unicode handling must be configured in the preamble.

```
"""
=============
```

(continues on next page)

(continued from previous page)

```
PGF texsystem
=============
"""

import matplotlib.pyplot as plt

plt.rcParams.update({
    "pgf.texsystem": "pdflatex",
    "pgf.preamble": "\n".join([
         r"\usepackage[utf8x]{inputenc}",
         r"\usepackage[T1]{fontenc}",
         r"\usepackage{cmbright}",
    ]),
})

fig, ax = plt.subplots(figsize=(4.5, 2.5))

ax.plot(range(5))

ax.text(0.5, 3., "serif", family="serif")
ax.text(0.5, 2., "monospace", family="monospace")
ax.text(2.5, 2., "sans-serif", family="sans-serif")
ax.set_xlabel(r"µ is not $\mu$")

fig.tight_layout(pad=.5)
```

### Troubleshooting

- Please note that the TeX packages found in some Linux distributions and MiKTeX installations are dramatically outdated. Make sure to update your package catalog and upgrade or install a recent TeX distribution.
- On Windows, the *PATH* environment variable may need to be modified to include the directories containing the latex, dvipng and ghostscript executables. See *Environment variables* and *Setting environment variables in Windows* for details.
- Sometimes the font rendering in figures that are saved to png images is very bad. This happens when the pdftocairo tool is not available and ghostscript is used for the pdf to png conversion.
- Make sure what you are trying to do is possible in a LaTeX document, that your LaTeX syntax is valid and that you are using raw strings if necessary to avoid unintended escape sequences.
- *rcParams["pgf.preamble"]* (default: `''`) provides lots of flexibility, and lots of ways to cause problems. When experiencing problems, try to minimalize or disable the custom preamble.
- Configuring an `unicode-math` environment can be a bit tricky. The TeXLive distribution for example provides a set of math fonts which are usually not installed system-wide. XeTeX, unlike LuaLatex, cannot find these fonts by their name, which is why you might have to specify `\setmathfont{xits-math.otf}` instead of `\setmathfont{XITS Math}` or alternatively make the fonts available to your OS. See this tex.stackexchange.com question for more details.

- If the font configuration used by Matplotlib differs from the font setting in yout LaTeX document, the alignment of text elements in imported figures may be off. Check the header of your `.pgf` file if you are unsure about the fonts Matplotlib used for the layout.
- Vector images and hence `.pgf` files can become bloated if there are a lot of objects in the graph. This can be the case for image processing or very big scatter graphs. In an extreme case this can cause TeX to run out of memory: "TeX capacity exceeded, sorry" You can configure latex to increase the amount of memory available to generate the `.pdf` image as discussed on tex.stackexchange.com. Another way would be to "rasterize" parts of the graph causing problems using either the `rasterized=True` keyword, or `.set_rasterized(True)` as per *this example*.
- Various math fonts are compiled and rendered only if corresponding font packages are loaded. Specifically, when using `\mathbf{}` on Greek letters, the default computer modern font may not contain them, in which case the letter is not rendered. In such scenarios, the `lmodern` package should be loaded.
- If you still need help, please see *Get help*

### 3.7.8 Text rendering with LaTeX

Matplotlib can use LaTeX to render text. This is activated by setting `text.usetex : True` in your rcParams, or by setting the `usetex` property to True on individual *`Text`* objects. Text handling through LaTeX is slower than Matplotlib's very capable *mathtext*, but is more flexible, since different LaTeX packages (font packages, math packages, etc.) can be used. The results can be striking, especially when you take care to use the same fonts in your figures as in the main document.

Matplotlib's LaTeX support requires a working LaTeX installation. For the *Agg backends, dvipng is additionally required; for the PS backend, PSfrag, dvips and Ghostscript are additionally required. For the PDF and SVG backends, if LuaTeX is present, it will be used to speed up some post-processing steps, but note that it is not used to parse the TeX string itself (only LaTeX is supported). The executables for these external dependencies must all be located on your *`PATH`*.

Only a small number of font families (defined by the PSNFSS scheme) are supported. They are listed here, with the corresponding LaTeX font selection commands and LaTeX packages, which are automatically used.

| generic family | fonts |
|---|---|
| serif (`\rmfamily`) | Computer Modern Roman, Palatino (`mathpazo`), Times (`mathptmx`), Bookman (`bookman`), New Century Schoolbook (`newcent`), Charter (`charter`) |
| sans-serif (`\sffamily`) | Computer Modern Serif, Helvetica (`helvet`), Avant Garde (`avant`) |
| cursive (`\rmfamily`) | Zapf Chancery (`chancery`) |
| monospace (`\ttfamily`) | Computer Modern Typewriter, Courier (`courier`) |

The default font family (which does not require loading any LaTeX package) is Computer Modern. All other families are Adobe fonts. Times and Palatino each have their own accompanying math fonts, while the other Adobe serif fonts make use of the Computer Modern math fonts.

To enable LaTeX and select a font, use e.g.:

```
plt.rcParams.update({
    "text.usetex": True,
    "font.family": "Helvetica"
})
```

or equivalently, set your *matplotlibrc* to:

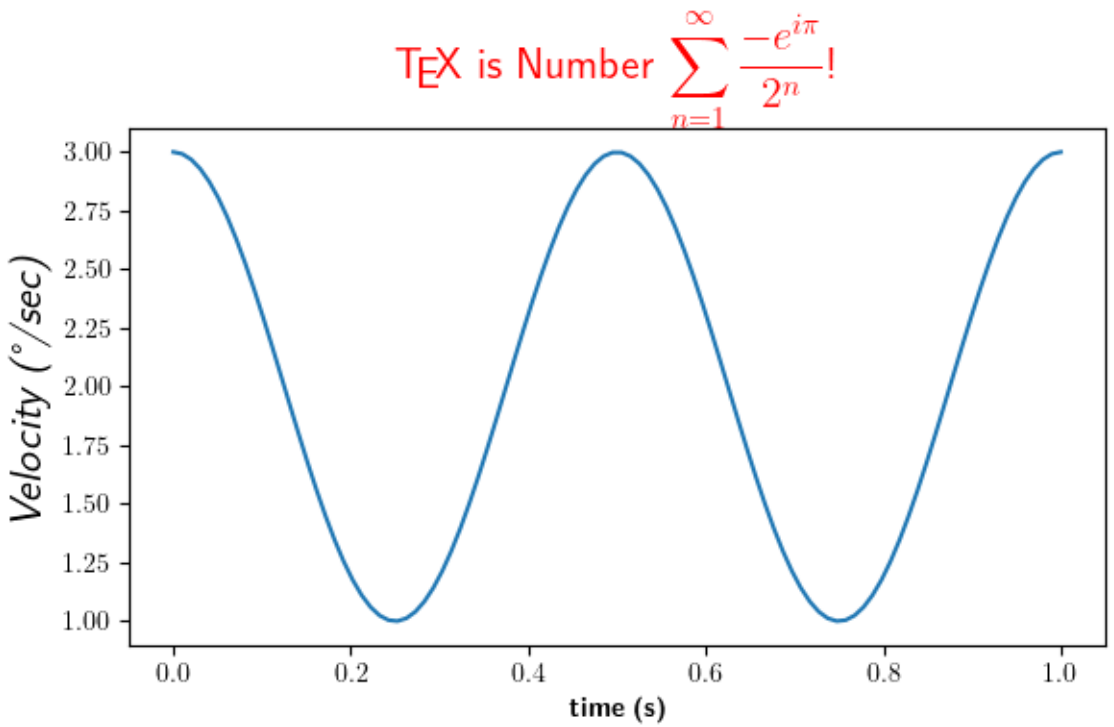
```
text.usetex : true
font.family : Helvetica
```

It is also possible to instead set `font.family` to one of the generic family names and then configure the corresponding generic family; e.g.:

```
plt.rcParams.update({
    "text.usetex": True,
    "font.family": "sans-serif",
    "font.sans-serif": "Helvetica",
})
```

(this was the required approach until Matplotlib 3.5).

Here is the standard example, *Rendering math equations using TeX*:




Note that display math mode (`$$ e=mc^2 $$`) is not supported, but adding the command `\displaystyle`, as in the above demo, will produce the same results.

Non-ASCII characters (e.g. the degree sign in the y-label above) are supported to the extent that they are supported by inputenc.

---

**Note:** For consistency with the non-usetex case, Matplotlib special-cases newlines, so that single-newlines yield linebreaks (rather than being interpreted as whitespace in standard LaTeX).

Matplotlib uses the underscore package so that underscores (_) are printed "as-is" in text mode (rather than causing an error as in standard LaTeX). Underscores still introduce subscripts in math mode.

---

**Note:** Certain characters require special escaping in TeX, such as:

```
# $ % & ~ ^ \ { } \( \) \[ \]
```

Therefore, these characters will behave differently depending on *rcParams["text.usetex"]* (default: `False`). As noted above, underscores (_) do not require escaping outside of math mode.

---

### PostScript options

In order to produce encapsulated PostScript (EPS) files that can be embedded in a new LaTeX document, the default behavior of Matplotlib is to distill the output, which removes some PostScript operators used by LaTeX that are illegal in an EPS file. This step produces results which may be unacceptable to some users, because the text is coarsely rasterized and converted to bitmaps, which are not scalable like standard PostScript, and the text is not searchable. One workaround is to set *rcParams["ps.distiller.res"]* (default: `6000`) to a higher value (perhaps 6000) in your rc settings, which will produce larger files but may look better and scale reasonably. A better workaround, which requires Poppler or Xpdf, can be activated by changing *rcParams["ps.usedistiller"]* (default: `None`) to `xpdf`. This alternative produces PostScript without rasterizing text, so it scales properly, can be edited in Adobe Illustrator, and searched text in pdf documents.

### Possible hangups

- On Windows, the *PATH* environment variable may need to be modified to include the directories containing the latex, dvipng and ghostscript executables. See *Environment variables* and *Setting environment variables in Windows* for details.
- Using MiKTeX with Computer Modern fonts, if you get odd *Agg and PNG results, go to MiKTeX/Options and update your format files
- On Ubuntu and Gentoo, the base texlive install does not ship with the type1cm package. You may need to install some of the extra packages to get all the goodies that come bundled with other LaTeX distributions.
- Some progress has been made so Matplotlib uses the dvi files directly for text layout. This allows LaTeX to be used for text layout with the pdf and svg backends, as well as the *Agg and PS backends. In the future, a LaTeX installation may be the only external dependency.

#### Troubleshooting

- Try deleting your `.matplotlib/tex.cache` directory. If you don't know where to find `.matplotlib`, see *matplotlib configuration and cache directory locations*.
- Make sure LaTeX, dvipng and ghostscript are each working and on your *PATH*.
- Make sure what you are trying to do is possible in a LaTeX document, that your LaTeX syntax is valid and that you are using raw strings if necessary to avoid unintended escape sequences.
- *rcParams["text.latex.preamble"]* (default: `''`) is not officially supported. This option provides lots of flexibility, and lots of ways to cause problems. Please disable this option before reporting problems to the mailing list.
- If you still need help, please see *Get help*.

## 3.8 Animations using Matplotlib

Based on its plotting functionality, Matplotlib also provides an interface to generate animations using the *animation* module. An animation is a sequence of frames where each frame corresponds to a plot on a *Figure*. This tutorial covers a general guideline on how to create such animations and the different options available.

### 3.8.1 Animations using Matplotlib

Based on its plotting functionality, Matplotlib also provides an interface to generate animations using the *animation* module. An animation is a sequence of frames where each frame corresponds to a plot on a *Figure*. This tutorial covers a general guideline on how to create such animations and the different options available.

```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib.animation as animation
```

#### Animation classes

The animation process in Matplotlib can be thought of in 2 different ways:

- *FuncAnimation*: Generate data for first frame and then modify this data for each frame to create an animated plot.
- *ArtistAnimation*: Generate a list (iterable) of artists that will draw in each frame in the animation.

*FuncAnimation* is more efficient in terms of speed and memory as it draws an artist once and then modifies it. On the other hand *ArtistAnimation* is flexible as it allows any iterable of artists to be animated in a sequence.

### FuncAnimation

The *FuncAnimation* class allows us to create an animation by passing a function that iteratively modifies the data of a plot. This is achieved by using the *setter* methods on various *Artist* (examples: *Line2D*, *PathCollection*, etc.). A usual *FuncAnimation* object takes a *Figure* that we want to animate and a function *func* that modifies the data plotted on the figure. It uses the *frames* parameter to determine the length of the animation. The *interval* parameter is used to determine time in milliseconds between drawing of two frames. Animating using *FuncAnimation* typically requires these steps:

1) Plot the initial figure as you would in a static plot. Save all the created artists, which are returned by the plot functions, in variables so that you can access and modify them later in the animation function.
2) Create an animation function that updates the artists for a given frame. Typically, this calls `set_*` methods of the artists.
3) Create a *FuncAnimation*, passing the *Figure* and the animation function.
4) Save or show the animation using one of the following methods:
   - *pyplot.show* to show the animation in a window
   - *Animation.to_html5_video* to create a HTML `<video>` tag
   - *Animation.to_jshtml* to create HTML code with interactive JavaScript animation controls
   - *Animation.save* to save the animation to a file

The following table shows a few plotting methods, the artists they return and some commonly used `set_*` methods that update the underlying data. While updating data is the most common operation in animations, you can also update other aspects such as color or text position.

| Plotting method | Artist | Data set methods |
|---|---|---|
| *Axes.plot* | *lines.Line2D* | *set_data*, *set_xdata*, *set_ydata* |
| *Axes.scatter* | *collections.PathCollection* | *set_offsets* |
| *Axes.imshow* | *image.AxesImage* | `AxesImage.set_data` |
| *Axes.annotate* | *text.Annotation* | *update_positions* |
| *Axes.barh* | *patches.Rectangle* | *set_angle*, *set_bounds*, *set_height*, *set_width*, *set_x*, *set_y*, *set_xy* |
| *Axes.fill* | *patches.Polygon* | *set_xy* |
| *Axes.add_patch*(*patches.Ellipse*) | *patches.Ellipse* | *set_angle*, *set_center*, *set_height*, *set_width* |
| *Axes.set_title*, *Axes.text* | *text.Text* | *set_text* |

Covering the set methods for all types of artists is beyond the scope of this tutorial but can be found in their

respective documentations. An example of such update methods in use for *Axes.scatter* and *Axes.plot* is as follows.

```
fig, ax = plt.subplots()
t = np.linspace(0, 3, 40)
g = -9.81
v0 = 12
z = g * t**2 / 2 + v0 * t

v02 = 5
z2 = g * t**2 / 2 + v02 * t

scat = ax.scatter(t[0], z[0], c="b", s=5, label=f'v0 = {v0} m/s')
line2 = ax.plot(t[0], z2[0], label=f'v0 = {v02} m/s')[0]
ax.set(xlim=[0, 3], ylim=[-4, 10], xlabel='Time [s]', ylabel='Z [m]')
ax.legend()


def update(frame):
    # for each frame, update the data stored on each artist.
    x = t[:frame]
    y = z[:frame]
    # update the scatter plot:
    data = np.stack([x, y]).T
    scat.set_offsets(data)
    # update the line plot:
    line2.set_xdata(t[:frame])
    line2.set_ydata(z2[:frame])
    return (scat, line2)


ani = animation.FuncAnimation(fig=fig, func=update, frames=40, interval=30)
plt.show()
```

### ArtistAnimation

*ArtistAnimation* can be used to generate animations if there is data stored on various different artists. This list of artists is then converted frame by frame into an animation. For example, when we use *Axes.barh* to plot a bar-chart, it creates a number of artists for each of the bar and error bars. To update the plot, one would need to update each of the bars from the container individually and redraw them. Instead, *animation.ArtistAnimation* can be used to plot each frame individually and then stitched together to form an animation. A barchart race is a simple example for this.

```
fig, ax = plt.subplots()
rng = np.random.default_rng(19680801)
data = np.array([20, 20, 20, 20])
x = np.array([1, 2, 3, 4])

artists = []
colors = ['tab:blue', 'tab:red', 'tab:green', 'tab:purple']
```

(continues on next page)

(continued from previous page)

```
for i in range(20):
    data += rng.integers(low=0, high=10, size=data.shape)
    container = ax.barh(x, data, color=colors)
    artists.append(container)


ani = animation.ArtistAnimation(fig=fig, artists=artists, interval=400)
plt.show()
```

### Animation writers

Animation objects can be saved to disk using various multimedia writers (ex: Pillow, *ffpmeg*, *imagemagick*). Not all video formats are supported by all writers. There are 4 major types of writers:

- *PillowWriter* - Uses the Pillow library to create the animation.
- *HTMLWriter* - Used to create JavaScript-based animations.
- Pipe-based writers - *FFMpegWriter* and *ImageMagickWriter* are pipe based writers. These writers pipe each frame to the utility (*ffmpeg* / *imagemagick*) which then stitches all of them together to create the animation.
- File-based writers - *FFMpegFileWriter* and *ImageMagickFileWriter* are examples of file-based writers. These writers are slower than their pipe-based alternatives but are more useful for debugging as they save each frame in a file before stitching them together into an animation.

### Saving Animations

| Writer | Supported Formats |
|---|---|
| *PillowWriter* | .gif, .apng, .webp |
| *HTMLWriter* | .htm, .html, .png |
| *FFMpegWriter*<br>*FFMpegFileWriter* | All formats supported by *ffmpeg*: `ffmpeg -formats` |
| *ImageMagickWriter*<br>*ImageMagickFileWriter* | All formats supported by *imagemagick*: `magick -list format` |

To save animations using any of the writers, we can use the *animation.Animation.save* method. It takes the *filename* that we want to save the animation as and the *writer*, which is either a string or a writer object. It also takes an *fps* argument. This argument is different than the *interval* argument that

*FuncAnimation* or *ArtistAnimation* uses. *fps* determines the frame rate that the **saved** animation uses, whereas *interval* determines the frame rate that the **displayed** animation uses.

Below are a few examples that show how to save an animation with different writers.

Pillow writers:

```
ani.save(filename="/tmp/pillow_example.gif", writer="pillow")
ani.save(filename="/tmp/pillow_example.apng", writer="pillow")
```

HTML writers:

```
ani.save(filename="/tmp/html_example.html", writer="html")
ani.save(filename="/tmp/html_example.htm", writer="html")
ani.save(filename="/tmp/html_example.png", writer="html")
```

FFMpegWriter:

```
ani.save(filename="/tmp/ffmpeg_example.mkv", writer="ffmpeg")
ani.save(filename="/tmp/ffmpeg_example.mp4", writer="ffmpeg")
ani.save(filename="/tmp/ffmpeg_example.mjpeg", writer="ffmpeg")
```

Imagemagick writers:

```
ani.save(filename="/tmp/imagemagick_example.gif", writer="imagemagick")
ani.save(filename="/tmp/imagemagick_example.webp", writer="imagemagick")
ani.save(filename="apng:/tmp/imagemagick_example.apng",
         writer="imagemagick", extra_args=["-quality", "100"])
```

(the `extra_args` for *apng* are needed to reduce filesize by ~10x)

**Total running time of the script:** (0 minutes 7.943 seconds)

### 3.8.2 Faster rendering by using blitting

*Blitting* is a standard technique in raster graphics that, in the context of Matplotlib, can be used to (drastically) improve performance of interactive figures. For example, the *animation* and *widgets* modules use blitting internally. Here, we demonstrate how to implement your own blitting, outside of these classes.

Blitting speeds up repetitive drawing by rendering all non-changing graphic elements into a background image once. Then, for every draw, only the changing elements need to be drawn onto this background. For example, if the limits of an Axes have not changed, we can render the empty Axes including all ticks and labels once, and only draw the changing data later.

The strategy is

- Prepare the constant background:
  - Draw the figure, but exclude all artists that you want to animate by marking them as *animated* (see *Artist.set_animated*).
  - Save a copy of the RBGA buffer.
- Render the individual images:

- Restore the copy of the RGBA buffer.
- Redraw the animated artists using *Axes.draw_artist* / *Figure.draw_artist*.
- Show the resulting image on the screen.

One consequence of this procedure is that your animated artists are always drawn on top of the static artists.

Not all backends support blitting. You can check if a given canvas does via the *FigureCanvasBase.supports_blit* property.

**Warning:** This code does not work with the macosx backend (but does work with other GUI backends on Mac).

### Minimal example

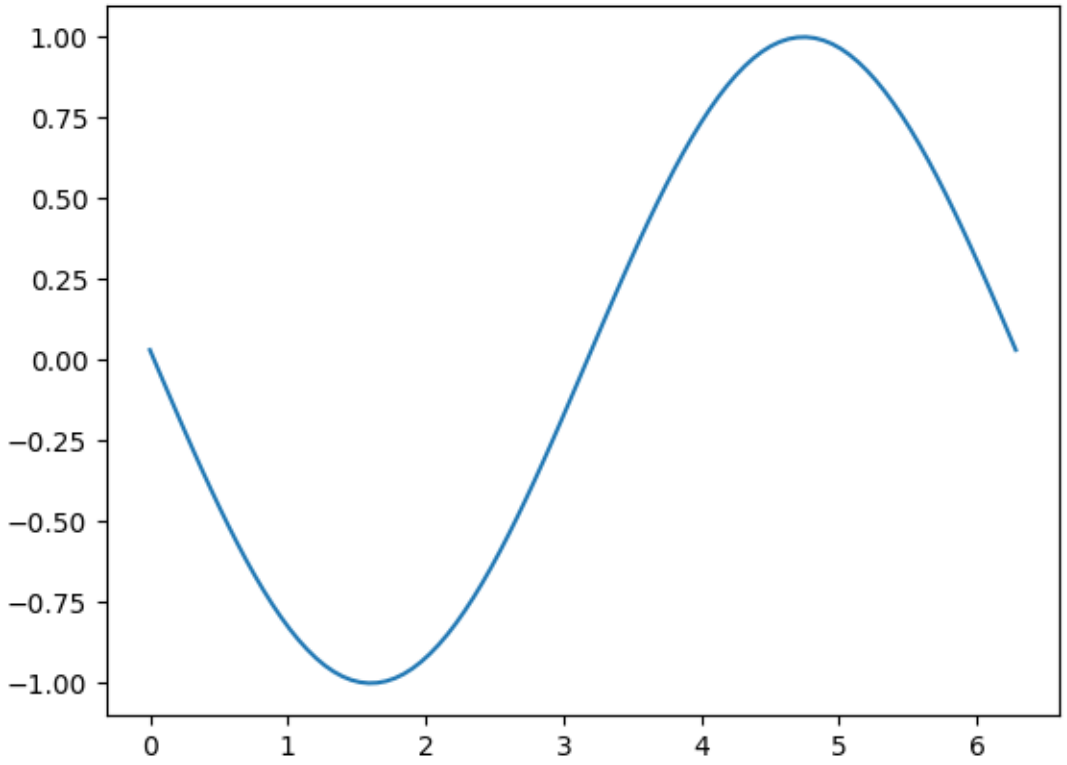
We can use the *FigureCanvasAgg* methods *copy_from_bbox* and *restore_region* in conjunction with setting `animated=True` on our artist to implement a minimal example that uses blitting to accelerate rendering

```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 2 * np.pi, 100)

fig, ax = plt.subplots()

# animated=True tells matplotlib to only draw the artist when we
# explicitly request it
(ln,) = ax.plot(x, np.sin(x), animated=True)

# make sure the window is raised, but the script keeps going
plt.show(block=False)

# stop to admire our empty window axes and ensure it is rendered at
# least once.
#
# We need to fully draw the figure at its final size on the screen
# before we continue on so that :
#  a) we have the correctly sized and drawn background to grab
#  b) we have a cached renderer so that ``ax.draw_artist`` works
# so we spin the event loop to let the backend process any pending operations
plt.pause(0.1)

# get copy of entire figure (everything inside fig.bbox) sans animated artist
bg = fig.canvas.copy_from_bbox(fig.bbox)
# draw the animated artist, this uses a cached renderer
ax.draw_artist(ln)
# show the result to the screen, this pushes the updated RGBA buffer from the
# renderer to the GUI framework so you can see it
fig.canvas.blit(fig.bbox)
```

(continues on next page)

(continued from previous page)

```
for j in range(100):
    # reset the background back in the canvas state, screen unchanged
    fig.canvas.restore_region(bg)
    # update the artist, neither the canvas state nor the screen have changed
    ln.set_ydata(np.sin(x + (j / 100) * np.pi))
    # re-render the artist, updating the canvas state, but not the screen
    ax.draw_artist(ln)
    # copy the image to the GUI state, but screen might not be changed yet
    fig.canvas.blit(fig.bbox)
    # flush any pending GUI events, re-painting the screen if needed
    fig.canvas.flush_events()
    # you can put a pause in if you want to slow things down
    # plt.pause(.1)
```




This example works and shows a simple animation, however because we are only grabbing the background once, if the size of the figure in pixels changes (due to either the size or dpi of the figure changing) , the background will be invalid and result in incorrect (but sometimes cool looking!) images. There is also a global variable and a fair amount of boilerplate which suggests we should wrap this in a class.

### Class-based example

We can use a class to encapsulate the boilerplate logic and state of restoring the background, drawing the artists, and then blitting the result to the screen. Additionally, we can use the `'draw_event'` callback to capture a new background whenever a full re-draw happens to handle resizes correctly.

```
class BlitManager:
    def __init__(self, canvas, animated_artists=()):
        """
        Parameters
        ----------
        canvas : FigureCanvasAgg
            The canvas to work with, this only works for subclasses of the Agg
            canvas which have the `~FigureCanvasAgg.copy_from_bbox` and
            `~FigureCanvasAgg.restore_region` methods.

        animated_artists : Iterable[Artist]
            List of the artists to manage
        """
        self.canvas = canvas
        self._bg = None
        self._artists = []

        for a in animated_artists:
            self.add_artist(a)
        # grab the background on every draw
        self.cid = canvas.mpl_connect("draw_event", self.on_draw)

    def on_draw(self, event):
        """Callback to register with 'draw_event'."""
        cv = self.canvas
        if event is not None:
            if event.canvas != cv:
                raise RuntimeError
        self._bg = cv.copy_from_bbox(cv.figure.bbox)
        self._draw_animated()

    def add_artist(self, art):
        """
        Add an artist to be managed.

        Parameters
        ----------
        art : Artist

            The artist to be added.  Will be set to 'animated' (just
            to be safe).  *art* must be in the figure associated with
            the canvas this class is managing.

        """
        if art.figure != self.canvas.figure:
            raise RuntimeError
```

(continues on next page)

(continued from previous page)

```
        art.set_animated(True)
        self._artists.append(art)

    def _draw_animated(self):
        """Draw all of the animated artists."""
        fig = self.canvas.figure
        for a in self._artists:
            fig.draw_artist(a)

    def update(self):
        """Update the screen with animated artists."""
        cv = self.canvas
        fig = cv.figure
        # paranoia in case we missed the draw event,
        if self._bg is None:
            self.on_draw(None)
        else:
            # restore the background
            cv.restore_region(self._bg)
            # draw all of the animated artists
            self._draw_animated()
            # update the GUI state
            cv.blit(fig.bbox)
        # let the GUI event loop process anything it has to do
        cv.flush_events()
```

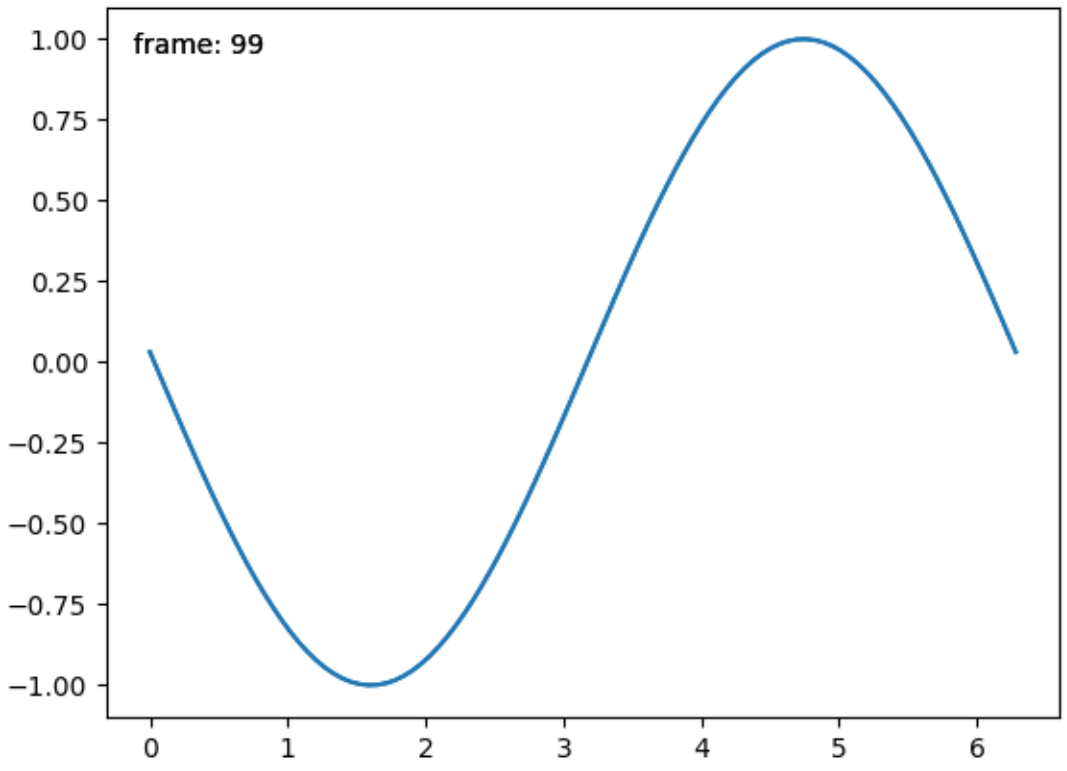
Here is how we would use our class. This is a slightly more complicated example than the first case as we add a text frame counter as well.

```
# make a new figure
fig, ax = plt.subplots()
# add a line
(ln,) = ax.plot(x, np.sin(x), animated=True)
# add a frame number
fr_number = ax.annotate(
    "0",
    (0, 1),
    xycoords="axes fraction",
    xytext=(10, -10),
    textcoords="offset points",
    ha="left",
    va="top",
    animated=True,
)
bm = BlitManager(fig.canvas, [ln, fr_number])
# make sure our window is on the screen and drawn
plt.show(block=False)
plt.pause(.1)

for j in range(100):
    # update the artists
```

(continues on next page)

(continued from previous page)

```
    ln.set_ydata(np.sin(x + (j / 100) * np.pi))
    fr_number.set_text(f"frame: {j}")
    # tell the blitting manager to do its thing
    bm.update()
```




This class does not depend on *pyplot* and is suitable to embed into larger GUI application.

**Total running time of the script:** (0 minutes 1.058 seconds)

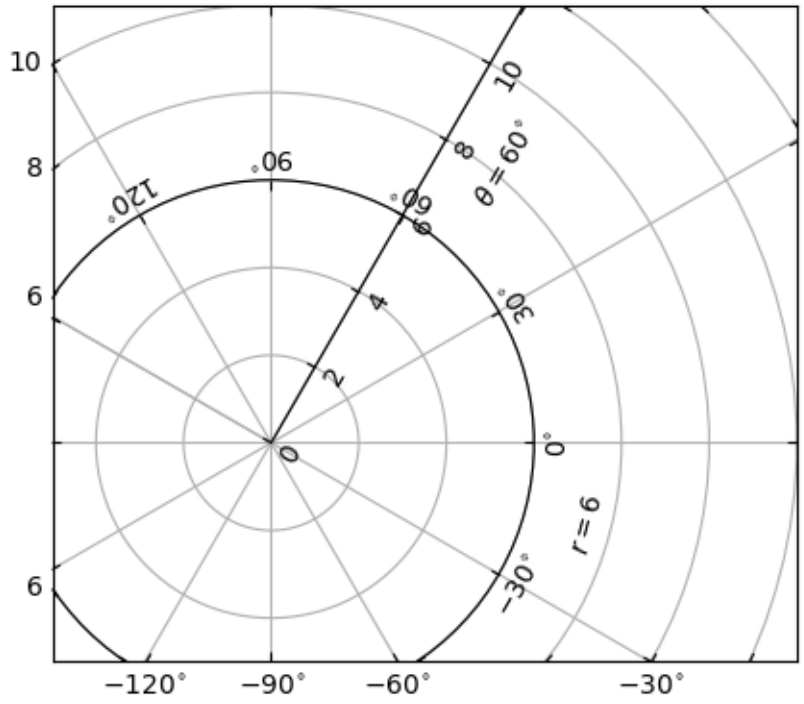
## 3.9 User Toolkits

Here you can find examples and explanations of how to use various toolkits available in Matplotlib.

### 3.9.1 The axisartist toolkit

**Warning:** *axisartist* uses a custom Axes class (derived from the Matplotlib's original Axes class). As a side effect, some commands (mostly tick-related) do not work.

The *axisartist* contains a custom Axes class that is meant to support curvilinear grids (e.g., the world coordinate system in astronomy). Unlike Matplotlib's original Axes class which uses Axes.xaxis and Axes.yaxis to draw ticks, ticklines, etc., axisartist uses a special artist (AxisArtist) that can handle ticks, ticklines, etc. for curved coordinate systems.




Since it uses special artists, some Matplotlib commands that work on Axes.xaxis and Axes.yaxis may not work.

### axisartist

The *axisartist* module provides a custom (and very experimental) Axes class, where each axis (left, right, top, and bottom) have a separate associated artist which is responsible for drawing the axis-line, ticks, ticklabels, and labels. You can also create your own axis, which can pass through a fixed position in the axes coordinate, or a fixed position in the data coordinate (i.e., the axis floats around when viewlimit changes).

The axes class, by default, has its xaxis and yaxis invisible, and has 4 additional artists which are responsible for drawing the 4 axis spines in "left", "right", "bottom", and "top". They are accessed as ax.axis["left"], ax.axis["right"], and so on, i.e., ax.axis is a dictionary that contains artists (note that ax.axis is still a callable method and it behaves as an original Axes.axis method in Matplotlib).

To create an Axes,

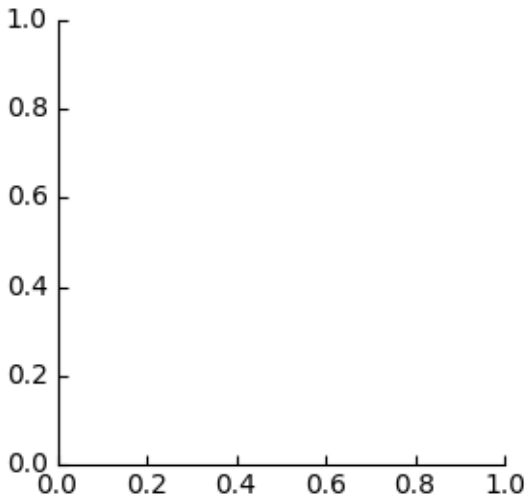
```
import mpl_toolkits.axisartist as AA
fig = plt.figure()
fig.add_axes([0.1, 0.1, 0.8, 0.8], axes_class=AA.Axes)
```

or to create a subplot

```
fig.add_subplot(111, axes_class=AA.Axes)
# Given that 111 is the default, one can also do
fig.add_subplot(axes_class=AA.Axes)
```

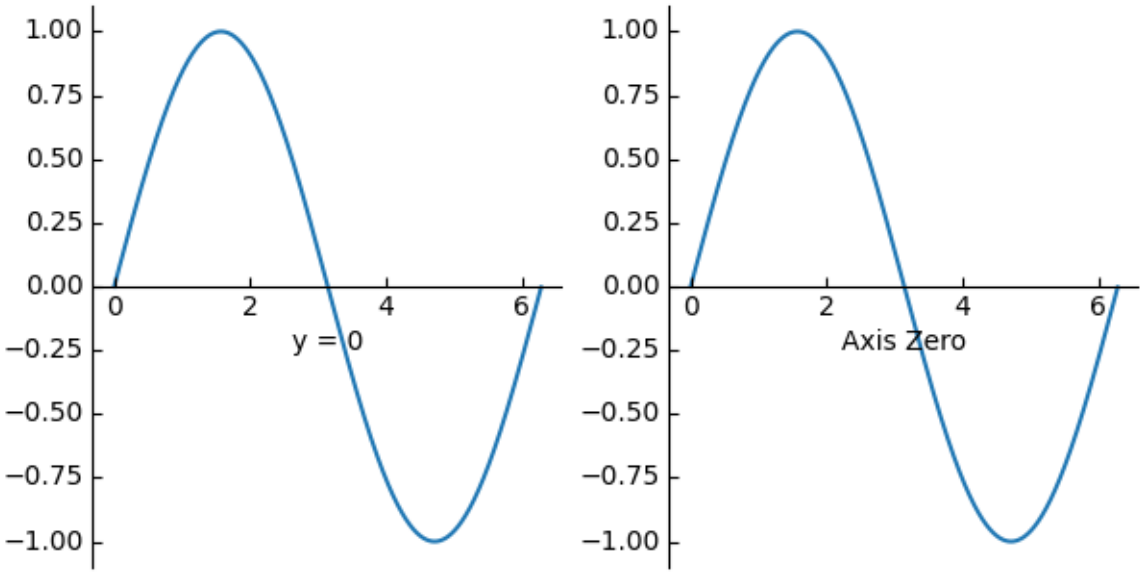
For example, you can hide the right and top spines using:

```
ax.axis["right"].set_visible(False)
ax.axis["top"].set_visible(False)
```




It is also possible to add a horizontal axis. For example, you may have an horizontal axis at y=0 (in data coordinate).

```
ax.axis["y=0"] = ax.new_floating_axis(nth_coord=0, value=0)
```




Or a fixed axis with some offset

```
# make new (right-side) yaxis, but with some offset
ax.axis["right2"] = ax.new_fixed_axis(loc="right", offset=(20, 0))
```

### axisartist with ParasiteAxes

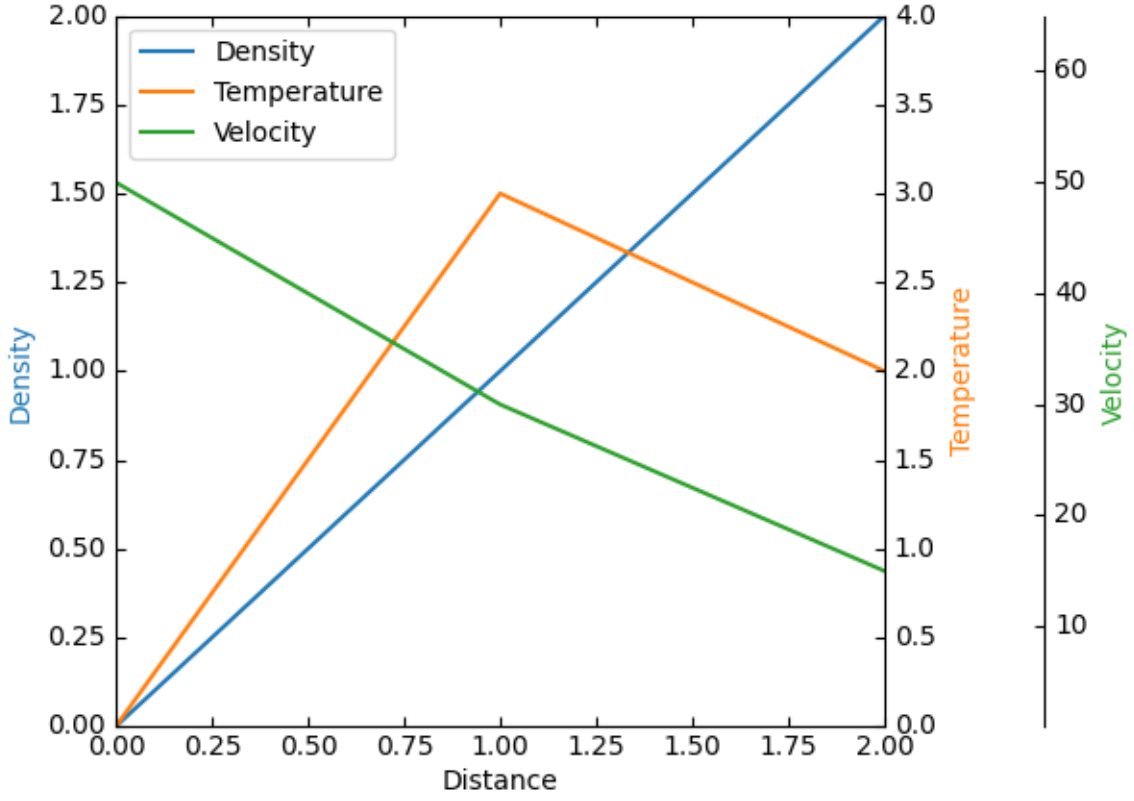
Most commands in the axes_grid1 toolkit can take an axes_class keyword argument, and the commands create an Axes of the given class. For example, to create a host subplot with axisartist.Axes,

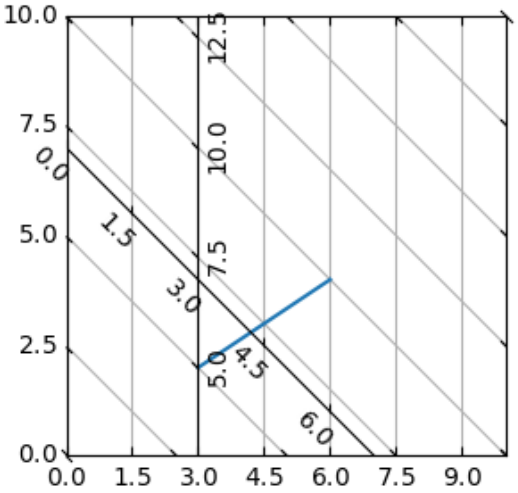
```
import mpl_toolkits.axisartist as AA
from mpl_toolkits.axes_grid1 import host_subplot

host = host_subplot(111, axes_class=AA.Axes)
```

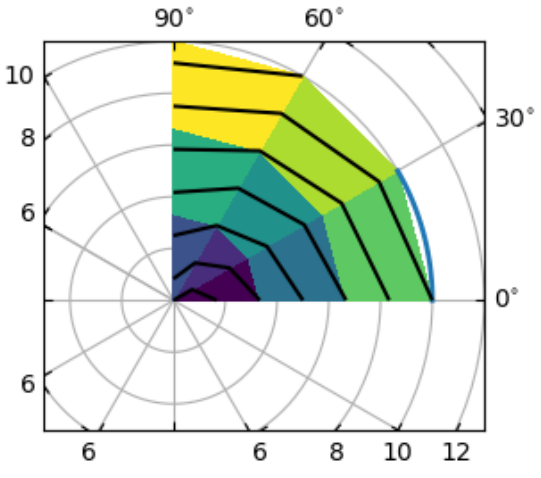
Here is an example that uses ParasiteAxes.

### Curvilinear grid

The motivation behind the AxisArtist module is to support a curvilinear grid and ticks.

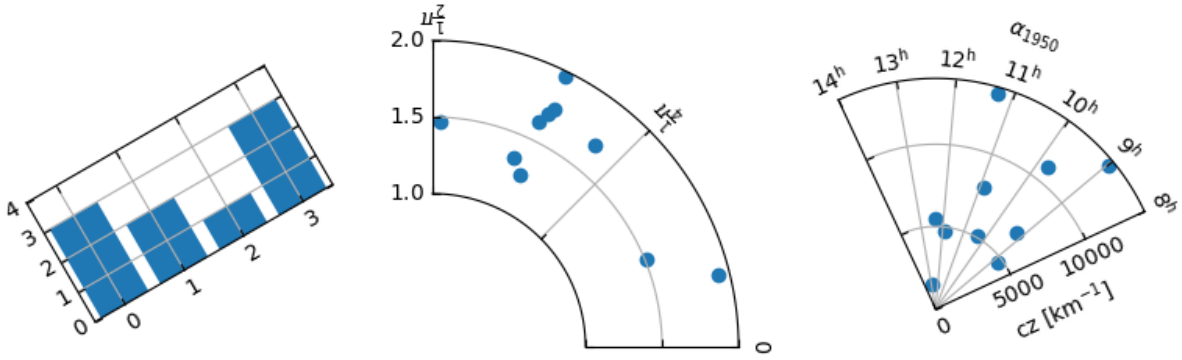

Density
Temperature
Velocity
Distance
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
60
50
40
30
20
10
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00


10.0
7.5
5.0
2.5
0.0
12.5
10.0
7.5
5.0
4.5
6.0
0.0
1.5
3.0
0.0
1.5
3.0
4.5
6.0
7.5
9.0


90°
60°
30°
0°
10
8
6
6
6
6
8
10
12

### Floating Axes

AxisArtist also supports a Floating Axes whose outer axes are defined as floating axis.




### axisartist namespace

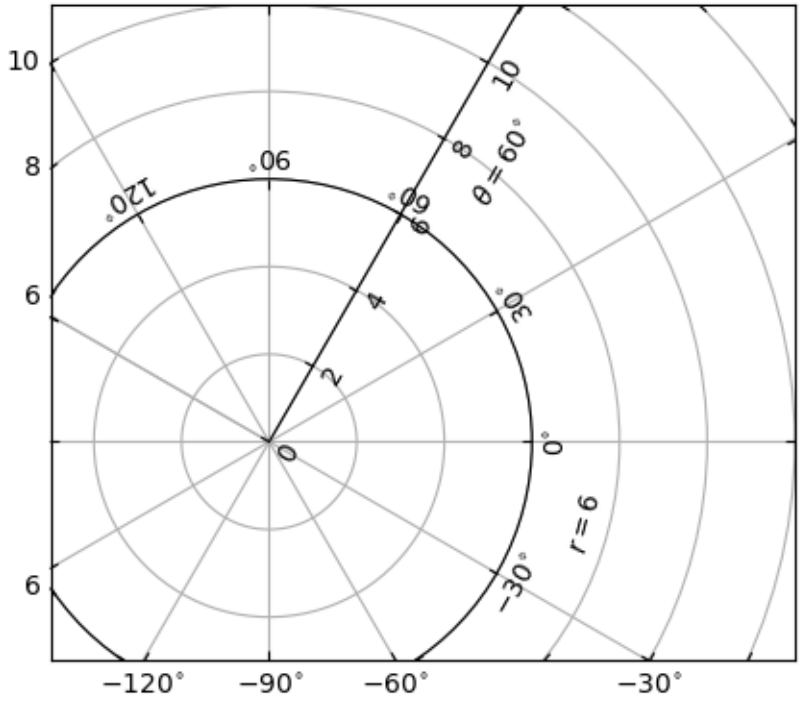
The *axisartist* namespace includes a derived Axes implementation. The biggest difference is that the artists responsible to draw axis line, ticks, ticklabel and axis labels are separated out from the Matplotlib's Axis class, which are much more than artists in the original Matplotlib. This change was strongly motivated to support curvilinear grid. Here are a few things that mpl_toolkits.axisartist.Axes is different from original Axes from Matplotlib.

- Axis elements (axis line(spine), ticks, ticklabel and axis labels) are drawn by a AxisArtist instance. Unlike Axis, left, right, top and bottom axis are drawn by separate artists. And each of them may have different tick location and different tick labels.
- gridlines are drawn by a Gridlines instance. The change was motivated that in curvilinear coordinate, a gridline may not cross axis-lines (i.e., no associated ticks). In the original Axes class, gridlines are tied to ticks.
- ticklines can be rotated if necessary (i.e, along the gridlines)

In summary, all these changes was to support

- a curvilinear grid.
- a floating axis

*mpl_toolkits.axisartist.Axes* class defines a *axis* attribute, which is a dictionary of AxisArtist instances. By default, the dictionary has 4 AxisArtist instances, responsible for drawing of left, right, bottom and top axis.


10
8
6
6
90°
120°
60°
30°
0°
−30°
−120°
−90°
−60°
−30°
θ = 60°
r = 6
0
2
4
8
10

xaxis and yaxis attributes are still available, however they are set to not visible. As separate artists are used for rendering axis, some axis-related method in Matplotlib may have no effect. In addition to AxisArtist instances, the mpl_toolkits.axisartist.Axes will have *gridlines* attribute (Gridlines), which obviously draws grid lines.

In both AxisArtist and Gridlines, the calculation of tick and grid location is delegated to an instance of GridHelper class. mpl_toolkits.axisartist.Axes class uses GridHelperRectlinear as a grid helper. The GridHelperRectlinear class is a wrapper around the *xaxis* and *yaxis* of Matplotlib's original Axes, and it was meant to work as the way how Matplotlib's original axes works. For example, tick location changes using set_ticks method and etc. should work as expected. But change in artist properties (e.g., color) will not work in general, although some effort has been made so that some often-change attributes (color, etc.) are respected.

### AxisArtist

AxisArtist can be considered as a container artist with following attributes which will draw ticks, labels, etc.

- line
- major_ticks, major_ticklabels
- minor_ticks, minor_ticklabels
- offsetText
- label

### line

Derived from Line2D class. Responsible for drawing a spinal(?) line.

### major_ticks, minor_ticks

Derived from Line2D class. Note that ticks are markers.

### major_ticklabels, minor_ticklabels

Derived from Text. Note that it is not a list of Text artist, but a single artist (similar to a collection).

### axislabel

Derived from Text.

### Default AxisArtists

By default, following for axis artists are defined.:

```
ax.axis["left"], ax.axis["bottom"], ax.axis["right"], ax.axis["top"]
```

The ticklabels and axislabel of the top and the right axis are set to not visible.

For example, if you want to change the color attributes of major_ticklabels of the bottom x-axis

```
ax.axis["bottom"].major_ticklabels.set_color("b")
```

Similarly, to make ticklabels invisible

```
ax.axis["bottom"].major_ticklabels.set_visible(False)
```

AxisArtist provides a helper method to control the visibility of ticks, ticklabels, and label. To make ticklabel invisible,

```
ax.axis["bottom"].toggle(ticklabels=False)
```

To make all of ticks, ticklabels, and (axis) label invisible

```
ax.axis["bottom"].toggle(all=False)
```

To turn all off but ticks on

```
ax.axis["bottom"].toggle(all=False, ticks=True)
```

To turn all on but (axis) label off

```
ax.axis["bottom"].toggle(all=True, label=False)
```

ax.axis's __getitem__ method can take multiple axis names. For example, to turn ticklabels of "top" and "right" axis on,

```
ax.axis["top", "right"].toggle(ticklabels=True)
```

Note that `ax.axis["top", "right"]` returns a simple proxy object that translate above code to something like below.

```
for n in ["top", "right"]:
    ax.axis[n].toggle(ticklabels=True)
```

So, any return values in the for loop are ignored. And you should not use it anything more than a simple method.

Like the list indexing ":" means all items, i.e.,

```
ax.axis[:].major_ticks.set_color("r")
```

changes tick color in all axis.

### HowTo

1. Changing tick locations and label.

   Same as the original Matplotlib's axes:

   ```
   ax.set_xticks([1, 2, 3])
   ```

2. Changing axis properties like color, etc.

   Change the properties of appropriate artists. For example, to change the color of the ticklabels:

   ```
   ax.axis["left"].major_ticklabels.set_color("r")
   ```

3. To change the attributes of multiple axis:

   ```
   ax.axis["left", "bottom"].major_ticklabels.set_color("r")
   ```

   or to change the attributes of all axis:

   ```
   ax.axis[:].major_ticklabels.set_color("r")
   ```

4. To change the tick size (length), you need to use axis.major_ticks.set_ticksize method. To change the direction of the ticks (ticks are in opposite direction of ticklabels by default), use axis.major_ticks.set_tick_out method.

   To change the pad between ticks and ticklabels, use axis.major_ticklabels.set_pad method.

   To change the pad between ticklabels and axis label, axis.label.set_pad method.

### Rotation and alignment of TickLabels

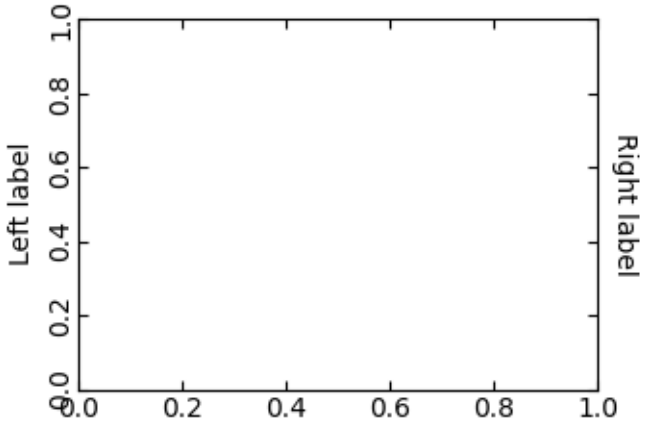
This is also quite different from standard Matplotlib and can be confusing. When you want to rotate the ticklabels, first consider using "set_axis_direction" method.

```
ax1.axis["left"].major_ticklabels.set_axis_direction("top")
ax1.axis["right"].label.set_axis_direction("left")
```

The parameter for set_axis_direction is one of ["left", "right", "bottom", "top"].

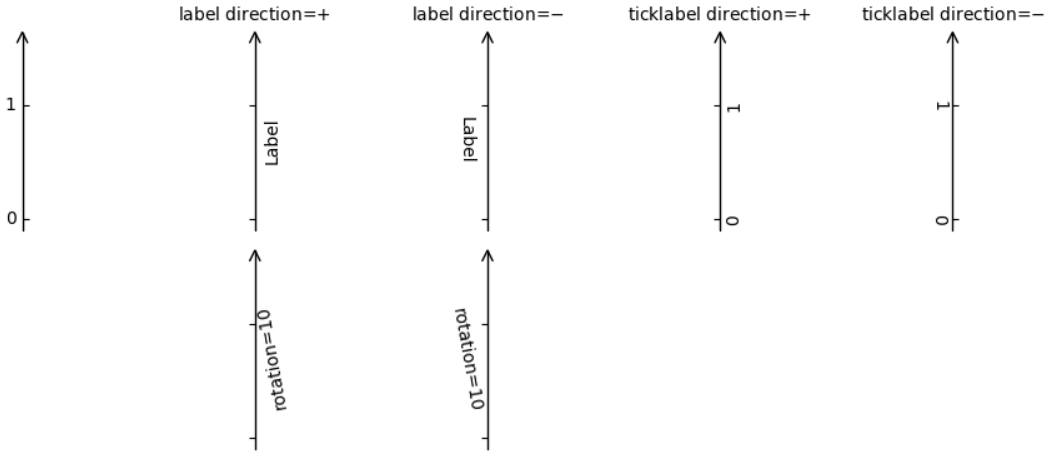
You must understand some underlying concept of directions.

- There is a reference direction which is defined as the direction of the axis line with increasing coordinate. For example, the reference direction of the left x-axis is from bottom to top.




The direction, text angle, and alignments of the ticks, ticklabels and axis-label is determined with respect to the reference direction

- *label_direction* and *ticklabel_direction* are either the right-hand side (+) of the reference direction or the left-hand side (-).
- ticks are by default drawn toward the opposite direction of the ticklabels.
- text rotation of ticklabels and label is determined in reference to the *ticklabel_direction* or *label_direction*, respectively. The rotation of ticklabels and label is anchored.




On the other hand, there is a concept of "axis_direction". This is a default setting of above properties for each, "bottom", "left", "top", and "right" axis.

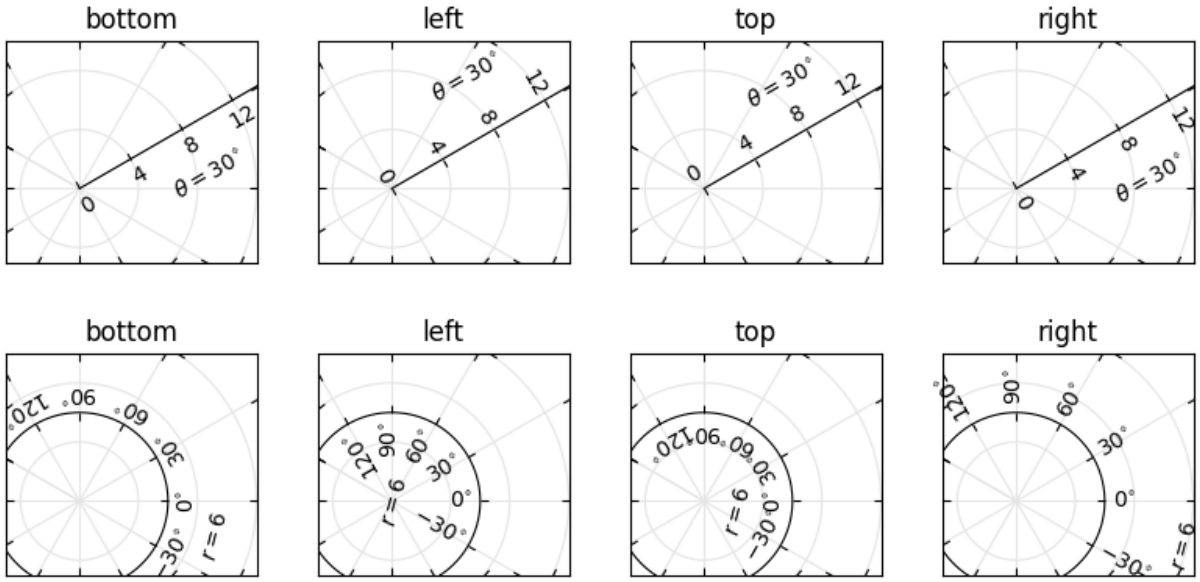
Table 3: `axislabel` property defaults

| reference direction | label direction | label rotation | horizontal alignment | vertical alignment |
|---|---|---|---|---|
| left | '-' | 180 | right | center |
| bottom | '+' | 0 | center | top |
| right | '+' | 0 | right | center |
| top | '-' | 180 | center | bottom |

Table 4: `ticklabel` property defaults

| reference direction | label direction | label rotation | horizontal alignment | vertical alignment |
|---|---|---|---|---|
| left | '-' | 90 | right | center |
| bottom | '+' | 0 | center | baseline |
| right | '+' | -90 | right | center |
| top | '-' | 180 | center | baseline |

And, 'set_axis_direction("top")' means to adjust the text rotation etc, for settings suitable for "top" axis. The concept of axis direction can be more clear with curved axis.




The axis_direction can be adjusted in the AxisArtist level, or in the level of its child artists, i.e., ticks, ticklabels, and axis-label.

```
ax1.axis["left"].set_axis_direction("top")
```

changes axis_direction of all the associated artist with the "left" axis, while

```
ax1.axis["left"].major_ticklabels.set_axis_direction("top")
```

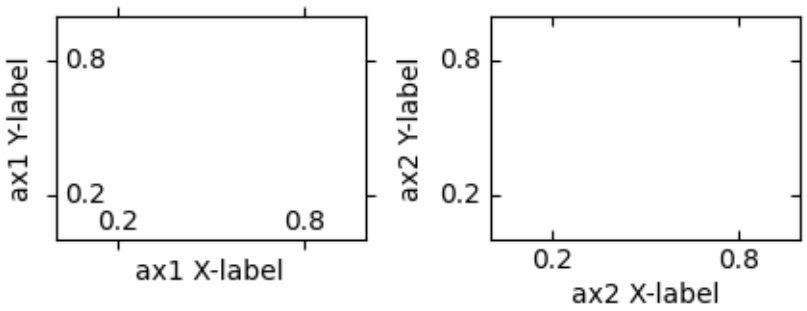
changes the axis_direction of only the major_ticklabels. Note that set_axis_direction in the AxisArtist level changes the ticklabel_direction and label_direction, while changing the axis_direction of ticks, ticklabels, and axis-label does not affect them.

If you want to make ticks outward and ticklabels inside the axes, use invert_ticklabel_direction method.

```
ax.axis[:].invert_ticklabel_direction()
```

A related method is "set_tick_out". It makes ticks outward (as a matter of fact, it makes ticks toward the opposite direction of the default direction).

```
ax.axis[:].major_ticks.set_tick_out(True)
```



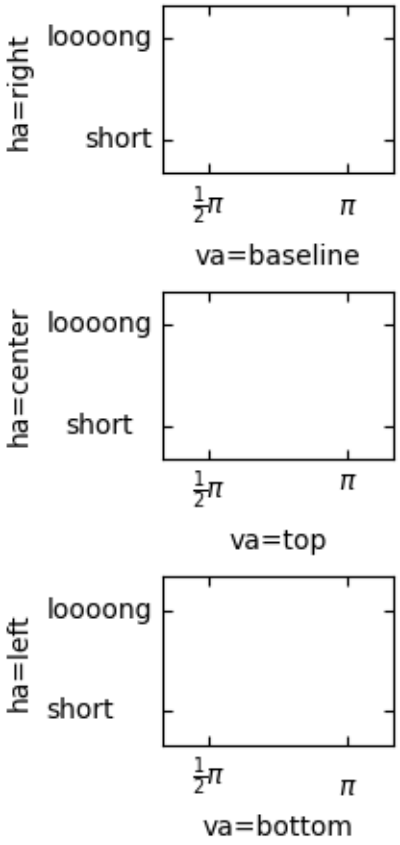

So, in summary,

- AxisArtist's methods
  - set_axis_direction: "left", "right", "bottom", or "top"
  - set_ticklabel_direction: "+" or "-"
  - set_axislabel_direction: "+" or "-"
  - invert_ticklabel_direction
- Ticks' methods (major_ticks and minor_ticks)
  - set_tick_out: True or False
  - set_ticksize: size in points
- TickLabels' methods (major_ticklabels and minor_ticklabels)
  - set_axis_direction: "left", "right", "bottom", or "top"
  - set_rotation: angle with respect to the reference direction
  - set_ha and set_va: see below
- AxisLabels' methods (label)
  - set_axis_direction: "left", "right", "bottom", or "top"
  - set_rotation: angle with respect to the reference direction

- set_ha and set_va

### Adjusting ticklabels alignment

Alignment of TickLabels are treated specially. See below



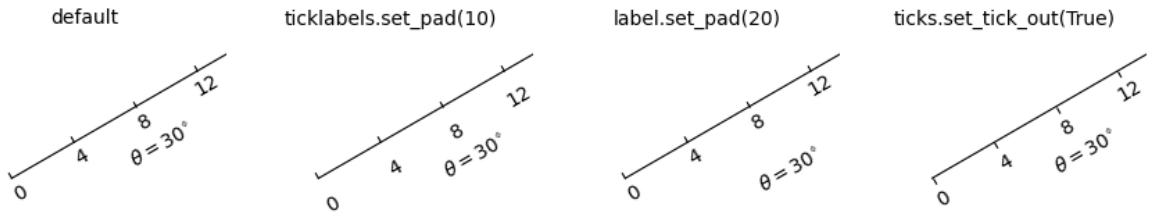

### Adjusting pad

To change the pad between ticks and ticklabels

```
ax.axis["left"].major_ticklabels.set_pad(10)
```

Or ticklabels and axis-label

```
ax.axis["left"].label.set_pad(10)
```




### GridHelper

To actually define a curvilinear coordinate, you have to use your own grid helper. A generalised version of grid helper class is supplied and this class should suffice in most of cases. A user may provide two functions which defines a transformation (and its inverse pair) from the curved coordinate to (rectilinear) image coordinate. Note that while ticks and grids are drawn for curved coordinate, the data transform of the axes itself (ax.transData) is still rectilinear (image) coordinate.

```
from mpl_toolkits.axisartist.grid_helper_curvelinear \
     import GridHelperCurveLinear
from mpl_toolkits.axisartist import Axes

# from curved coordinate to rectlinear coordinate.
def tr(x, y):
    x, y = np.asarray(x), np.asarray(y)
    return x, y-x

# from rectlinear coordinate to curved coordinate.
def inv_tr(x, y):
    x, y = np.asarray(x), np.asarray(y)
    return x, y+x

grid_helper = GridHelperCurveLinear((tr, inv_tr))

fig.add_subplot(axes_class=Axes, grid_helper=grid_helper)
```

You may use Matplotlib's Transform instance instead (but a inverse transformation must be defined). Often, coordinate range in a curved coordinate system may have a limited range, or may have cycles. In those cases, a more customized version of grid helper is required.

```
import mpl_toolkits.axisartist.angle_helper as angle_helper

# PolarAxes.PolarTransform takes radian. However, we want our coordinate
# system in degree
tr = Affine2D().scale(np.pi/180., 1.) + PolarAxes.PolarTransform()

# extreme finder: find a range of coordinate.
```

(continues on next page)

(continued from previous page)

```
# 20, 20: number of sampling points along x, y direction
# The first coordinate (longitude, but theta in polar)
#   has a cycle of 360 degree.
# The second coordinate (latitude, but radius in polar)  has a minimum of 0
extreme_finder = angle_helper.ExtremeFinderCycle(20, 20,
                                                 lon_cycle=360,
                                                 lat_cycle=None,
                                                 lon_minmax=None,
                                                 lat_minmax=(0, np.inf),
                                                 )

# Find a grid values appropriate for the coordinate (degree,
# minute, second). The argument is a approximate number of grids.
grid_locator1 = angle_helper.LocatorDMS(12)

# And also uses an appropriate formatter.  Note that the acceptable Locator
# and Formatter classes are different than that of Matplotlib's, and you
# cannot directly use Matplotlib's Locator and Formatter here (but may be
# possible in the future).
tick_formatter1 = angle_helper.FormatterDMS()

grid_helper = GridHelperCurveLinear(tr,
                                    extreme_finder=extreme_finder,
                                    grid_locator1=grid_locator1,
                                    tick_formatter1=tick_formatter1
                                    )
```

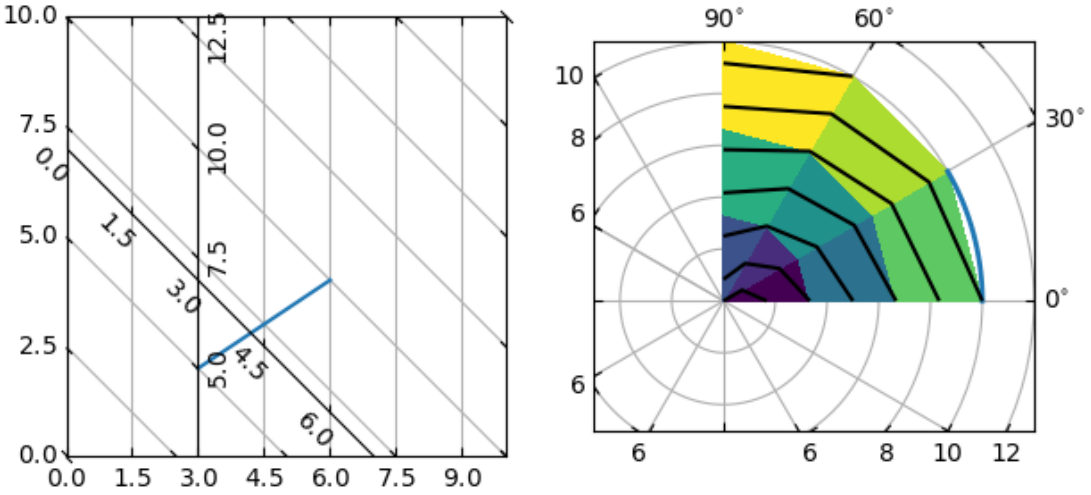
Again, the *transData* of the axes is still a rectilinear coordinate (image coordinate). You may manually do conversion between two coordinates, or you may use Parasite Axes for convenience.:

```
ax1 = SubplotHost(fig, 1, 2, 2, grid_helper=grid_helper)

# A parasite axes with given transform
ax2 = ax1.get_aux_axes(tr, "equal")
# note that ax2.transData == tr + ax1.transData
# Anything you draw in ax2 will match the ticks and grids of ax1.
```

### FloatingAxis

A floating axis is an axis one of whose data coordinate is fixed, i.e, its location is not fixed in Axes coordinate but changes as axes data limits changes. A floating axis can be created using *new_floating_axis* method. However, it is your responsibility that the resulting AxisArtist is properly added to the axes. A recommended way is to add it as an item of Axes's axis attribute.:

```
# floating axis whose first (index starts from 0) coordinate
# (theta) is fixed at 60

ax1.axis["lat"] = axis = ax1.new_floating_axis(0, 60)
axis.label.set_text(r"$\theta = 60^{\circ}$")
axis.label.set_visible(True)
```




See the first example of this page.

### Current limitations and TODO's

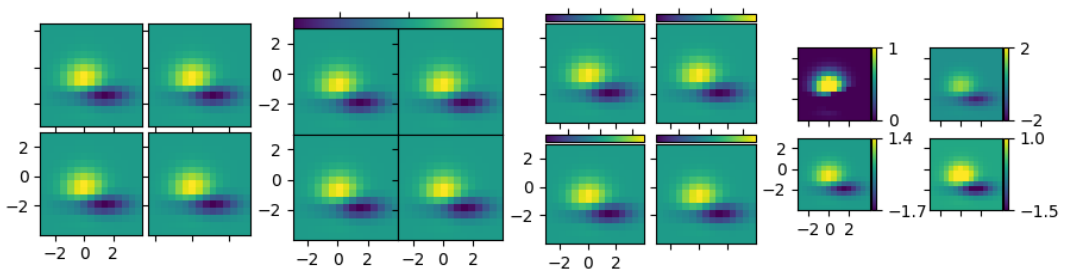
The code need more refinement. Here is a incomplete list of issues and TODO's

- No easy way to support a user customized tick location (for curvilinear grid). A new Locator class needs to be created.
- FloatingAxis may have coordinate limits, e.g., a floating axis of x = 0, but y only spans from 0 to 1.
- The location of axislabel of FloatingAxis needs to be optionally given as a coordinate value. ex, a floating axis of x=0 with label at y=1

## 3.9.2 The axes_grid1 toolkit

`axes_grid1` provides the following features:

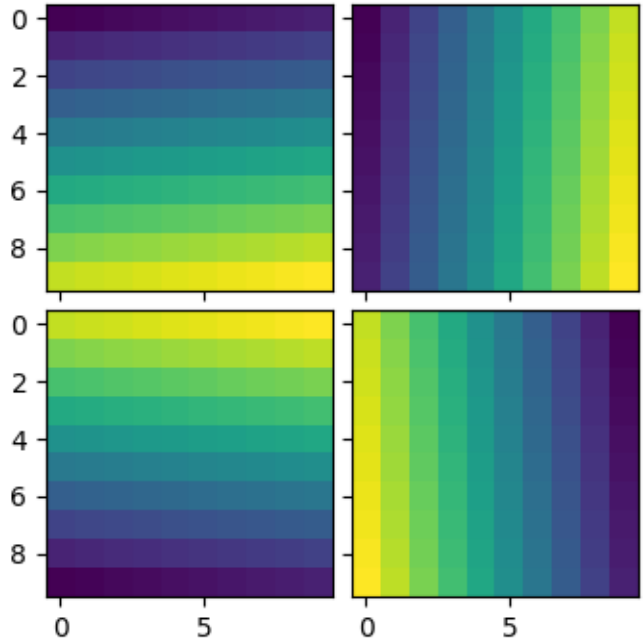
- Helper classes (*ImageGrid*, *RGBAxes*, *AxesDivider*) to ease the layout of axes displaying images with a fixed aspect ratio while satisfying additional constraints (matching the heights of a colorbar and an image, or fixing the padding between images);
- *ParasiteAxes* (twinx/twiny-like features so that you can plot different data (e.g., different y-scale) in a same Axes);
- *AnchoredArtists* (custom artists which are placed at an anchored position, similarly to legends).



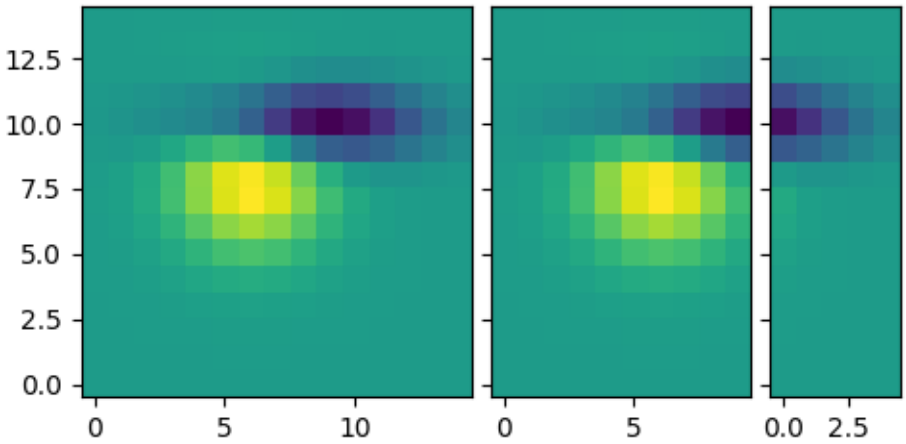
### axes_grid1

### ImageGrid

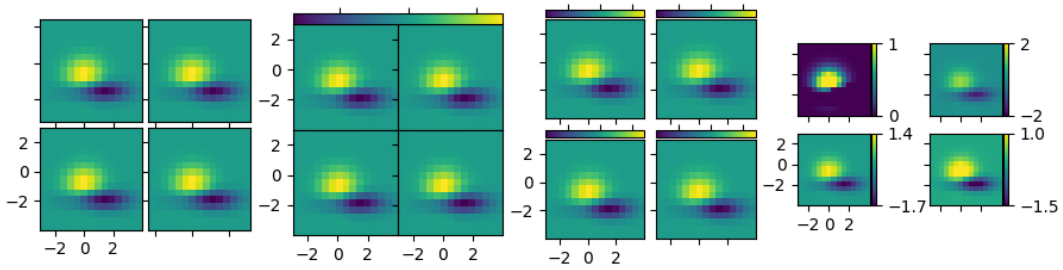
In Matplotlib, axes location and size are usually specified in normalized figure coordinates (0 = bottom left, 1 = top right), which makes it difficult to achieve a fixed (absolute) padding between images. `ImageGrid` can be used to achieve such a padding; see its docs for detailed API information.



- The position of each axes is determined at the drawing time (see *AxesDivider*), so that the size of the entire grid fits in the given rectangle (like the aspect of axes). Note that in this example, the paddings between axes are fixed even if you change the figure size.
- Axes in the same column share their x-axis, and axes in the same row share their y-axis (in the sense of `sharex`, `sharey`). Additionally, Axes in the same column all have the same width, and axes in the same row all have the same height. These widths and heights are scaled in proportion to the axes' view limits (xlim or ylim).




The examples below show what you can do with ImageGrid.



### AxesDivider class

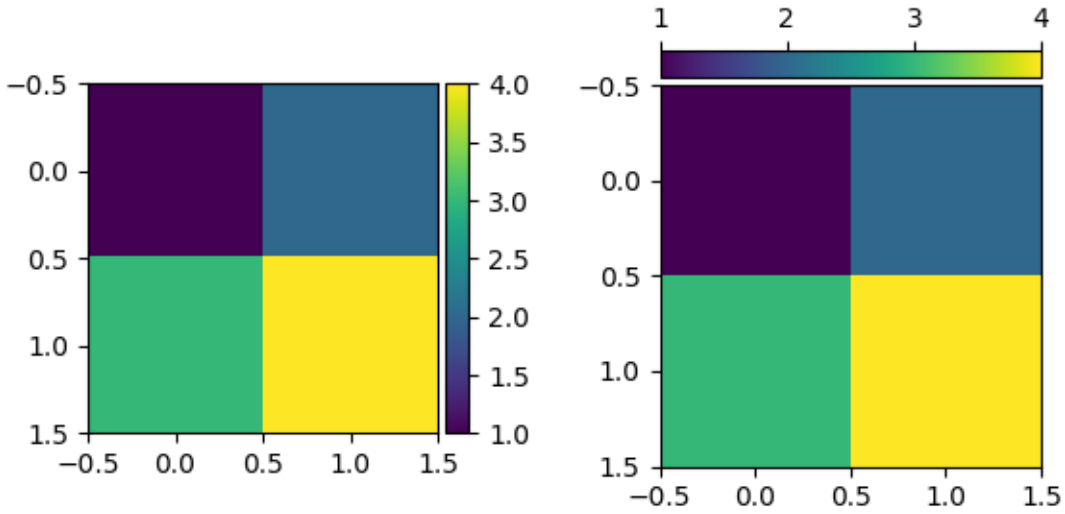
Behind the scenes, ImageGrid (and RGBAxes, described below) rely on *AxesDivider*, whose role is to calculate the location of the axes at drawing time.

Users typically do not need to directly instantiate dividers by calling *AxesDivider*; instead, *make_axes_locatable* can be used to create a divider for an Axes:

```
ax = subplot(1, 1, 1)
divider = make_axes_locatable(ax)
```

*AxesDivider.append_axes* can then be used to create a new axes on a given side ("left", "right", "top", "bottom") of the original axes.

### colorbar whose height (or width) is in sync with the main axes

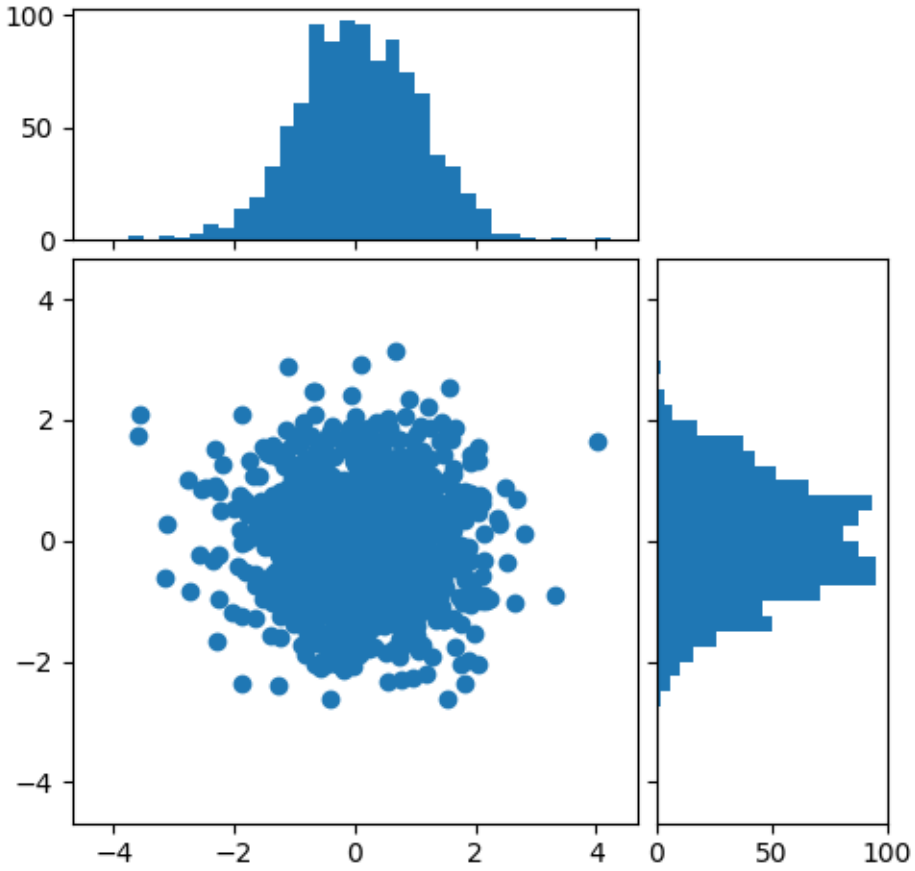



### scatter_hist.py with AxesDivider

The *Scatter plot with histograms* example can be rewritten using `make_axes_locatable`:

```
axScatter = plt.subplot()
axScatter.scatter(x, y)
axScatter.set_aspect(1.)

# create new axes on the right and on the top of the current axes.
divider = make_axes_locatable(axScatter)
axHistx = divider.append_axes("top", size=1.2, pad=0.1, sharex=axScatter)
axHisty = divider.append_axes("right", size=1.2, pad=0.1, sharey=axScatter)

# the scatter plot:
# histograms
bins = np.arange(-lim, lim + binwidth, binwidth)
axHistx.hist(x, bins=bins)
axHisty.hist(y, bins=bins, orientation='horizontal')
```

See the full source code below.




The *Scatter Histogram (Locatable Axes)* using the AxesDivider has some advantages over the original *Scatter plot with histograms* in Matplotlib. For example, you can set the aspect ratio of the scatter plot, even with the x-axis or y-axis is shared accordingly.

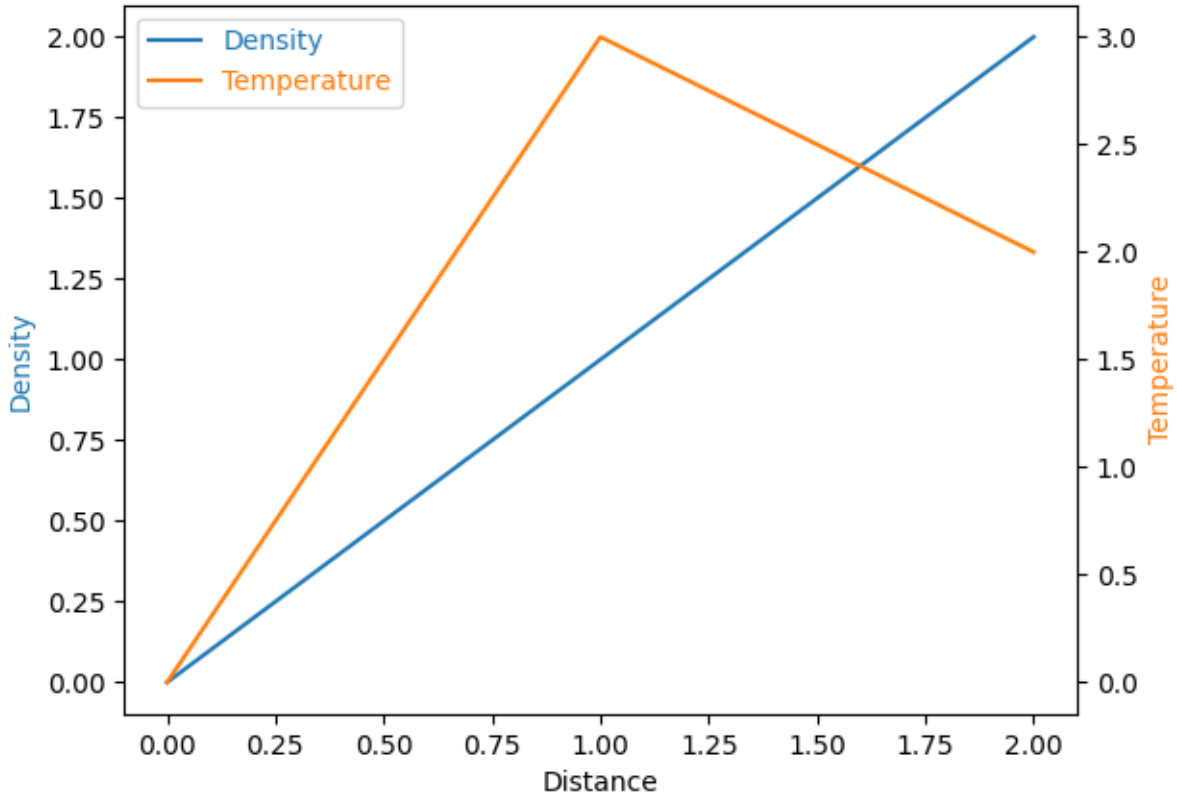
### ParasiteAxes

The ParasiteAxes is an Axes whose location is identical to its host axes. The location is adjusted in the drawing time, thus it works even if the host change its location (e.g., images).

In most cases, you first create a host axes, which provides a few methods that can be used to create parasite axes. They are `twinx`, `twiny` (which are similar to `twinx` and `twiny` in the matplotlib) and `twin`. `twin` takes an arbitrary transformation that maps between the data coordinates of the host axes and the parasite axes. The `draw` method of the parasite axes are never called. Instead, host axes collects artists in parasite axes and draws them as if they belong to the host axes, i.e., artists in parasite axes are merged to those of the host axes and then drawn according to their zorder. The host and parasite axes modifies some of the axes

behavior. For example, color cycle for plot lines are shared between host and parasites. Also, the legend command in host, creates a legend that includes lines in the parasite axes. To create a host axes, you may use `host_subplot` or `host_axes` command.

### Example 1: twinx

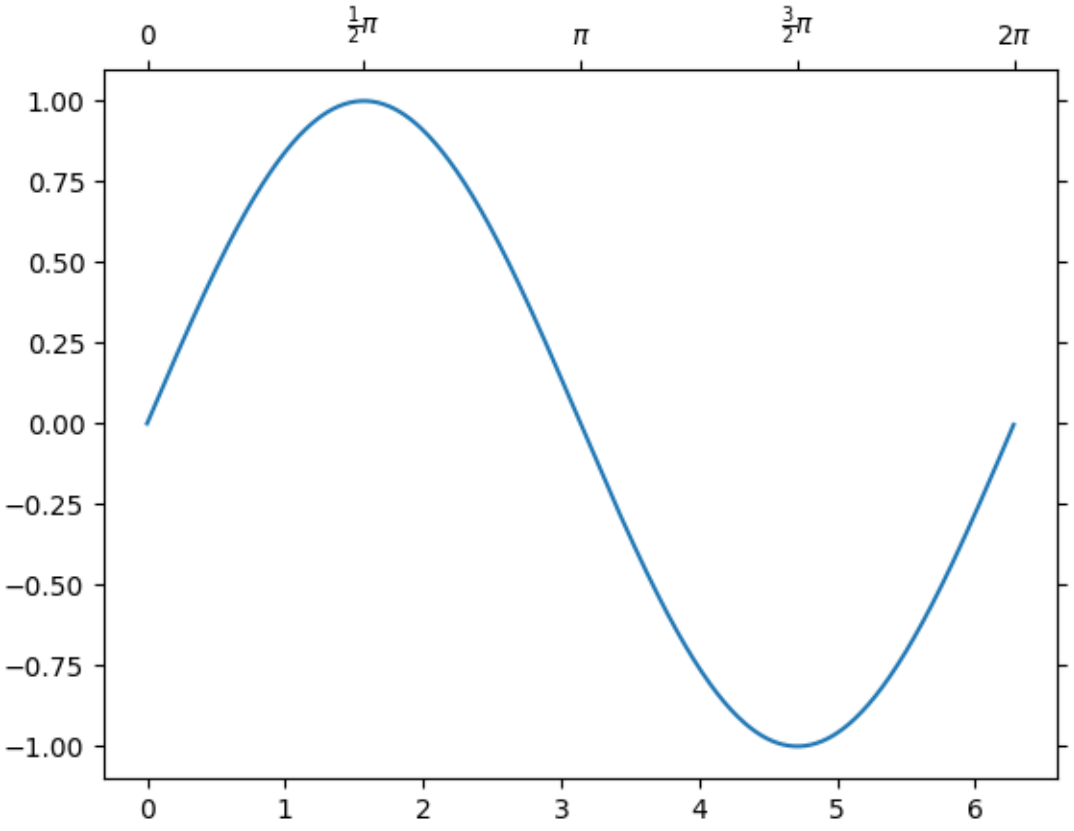


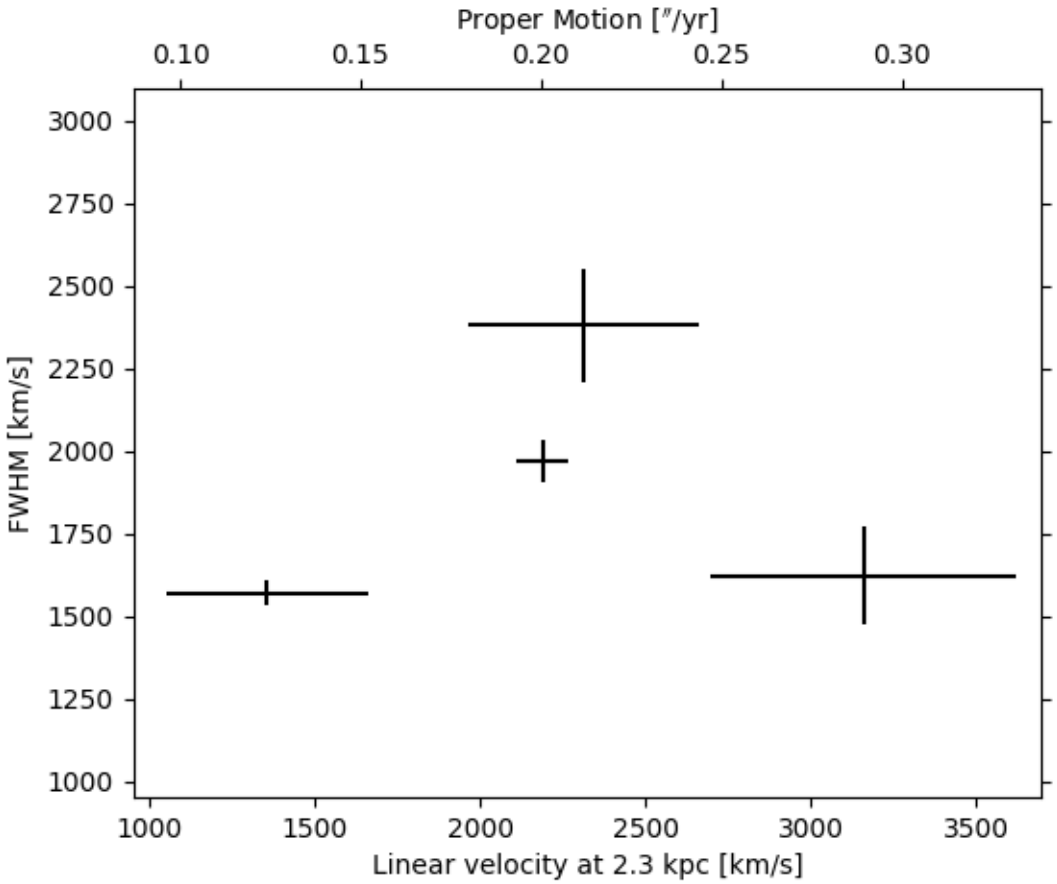

### Example 2: twin

`twin` without a transform argument assumes that the parasite axes has the same data transform as the host. This can be useful when you want the top(or right)-axis to have different tick-locations, tick-labels, or tick-formatter for bottom(or left)-axis.

```
ax2 = ax.twin() # now, ax2 is responsible for "top" axis and "right" axis
ax2.set_xticks([0., .5*np.pi, np.pi, 1.5*np.pi, 2*np.pi],
               labels=["0", r"$\frac{1}{2}\pi$",
                       r"$\pi$", r"$\frac{3}{2}\pi$", r"$2\pi$"])
```

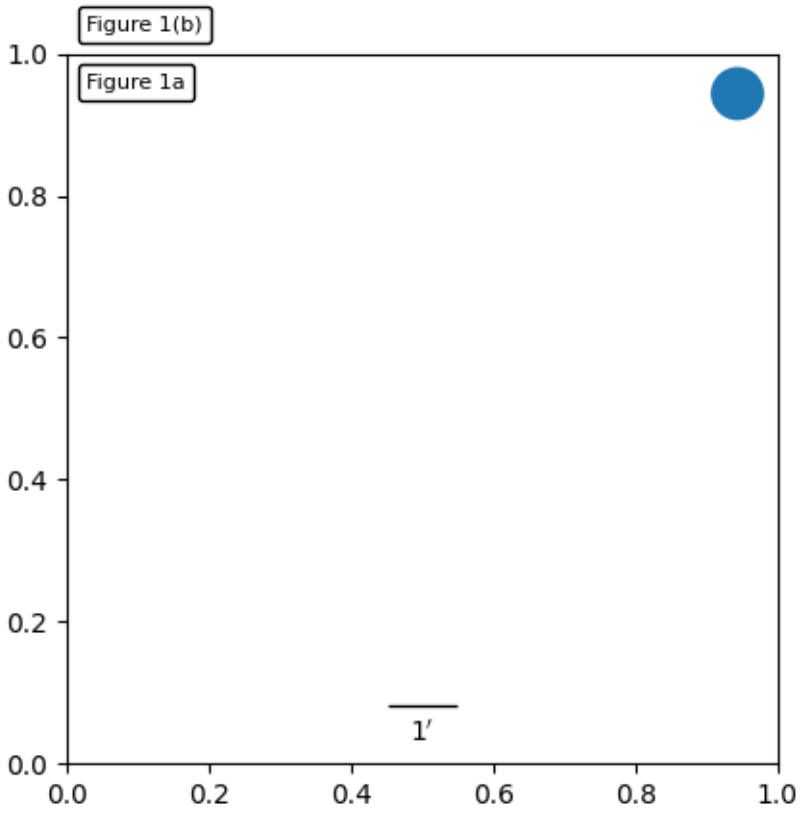
A more sophisticated example using twin. Note that if you change the x-limit in the host axes, the x-limit of the parasite axes will change accordingly.


0
$\frac{1}{2}\pi$
$\pi$
$\frac{3}{2}\pi$
$2\pi$
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
1
2
3
4
5
6


Proper Motion [″/yr]
0.10
0.15
0.20
0.25
0.30
3000
2750
2500
2250
2000
1750
1500
1250
1000
FWHM [km/s]
1000
1500
2000
2500
3000
3500
Linear velocity at 2.3 kpc [km/s]

### AnchoredArtists

*axes_grid1.anchored_artists* is a collection of artists whose location is anchored to the (axes) bbox, similarly to legends. These artists derive from *offsetbox.OffsetBox*, and the artist need to be drawn in canvas coordinates. There is limited support for arbitrary transforms. For example, the ellipse in the example below will have width and height in data coordinates.




### InsetLocator

**See also:**

*Axes.inset_axes* and *Axes.indicate_inset_zoom* in the main library.

*axes_grid1.inset_locator* provides helper classes and functions to place inset axes at an anchored position of the parent axes, similarly to AnchoredArtist.

*inset_locator.inset_axes* creates an inset axes whose size is either fixed, or a fixed proportion of the parent axes:

```
inset_axes = inset_axes(parent_axes,
                        width="30%",  # width = 30% of parent_bbox
```

(continues on next page)

(continued from previous page)

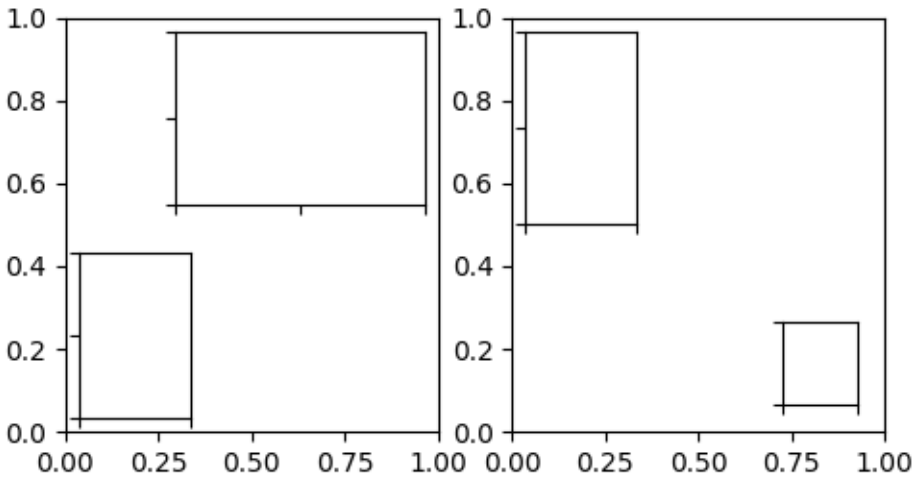
```
                                height=1.,  # height = 1 inch
                                loc='lower left')
```

creates an inset axes whose width is 30% of the parent axes and whose height is fixed at 1 inch.

*inset_locator.zoomed_inset_axes* creates an inset axes whose data scale is that of the parent axes multiplied by some factor, e.g.

```
inset_axes = zoomed_inset_axes(ax,
                               0.5,  # zoom = 0.5
                               loc='upper right')
```

creates an inset axes whose data scale is half of the parent axes. This can be useful to mark the zoomed area on the parent axes:

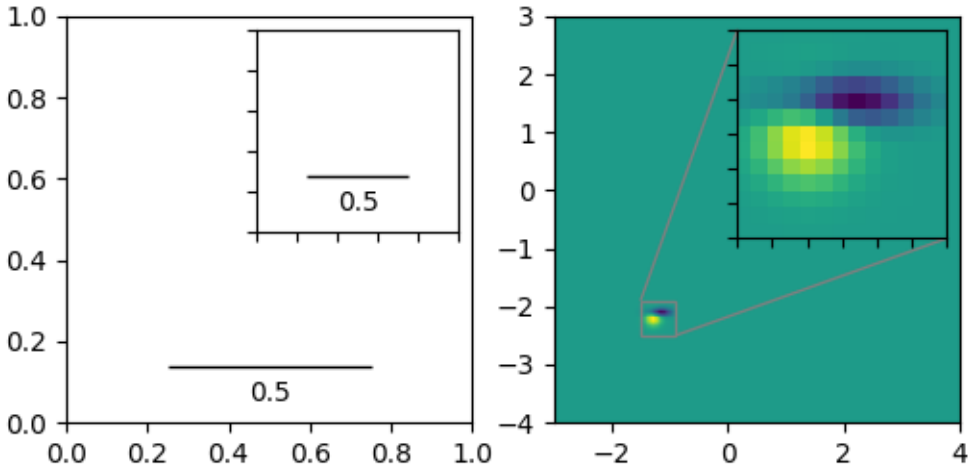


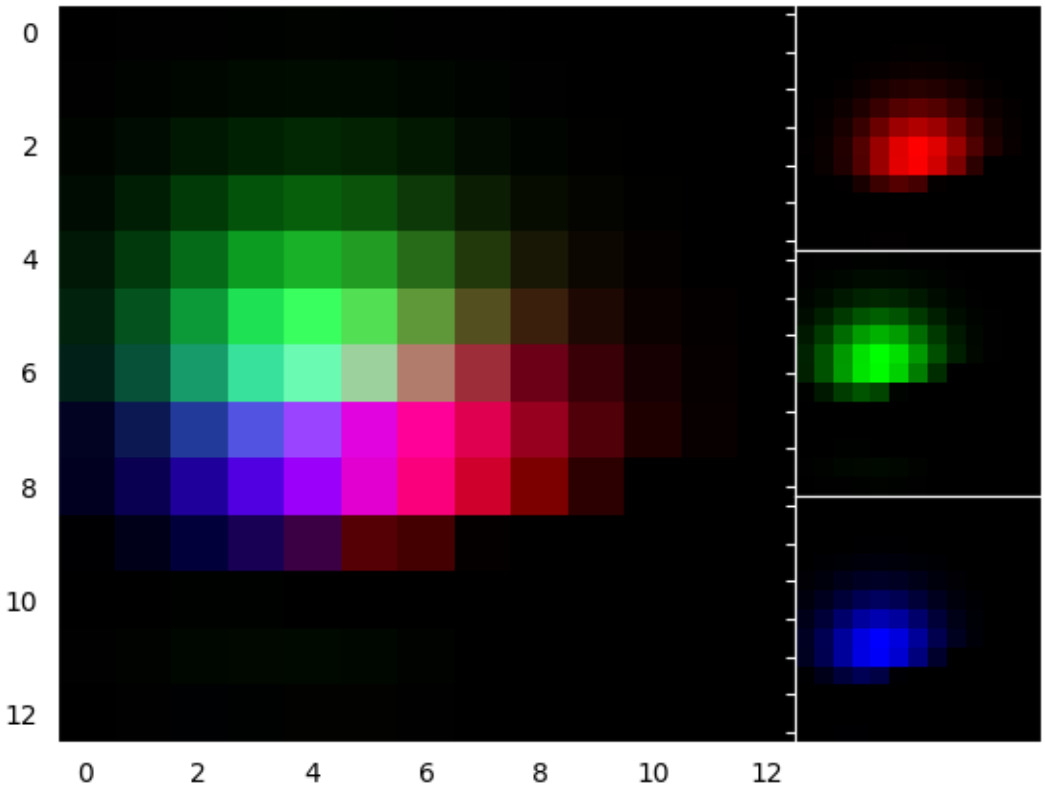

*inset_locator.mark_inset* allows marking the location of the area represented by the inset axes:

### RGBAxes

RGBAxes is a helper class to conveniently show RGB composite images. Like ImageGrid, the location of axes are adjusted so that the area occupied by them fits in a given rectangle. Also, the xaxis and yaxis of each axes are shared.

```
from mpl_toolkits.axes_grid1.axes_rgb import RGBAxes

fig = plt.figure()
ax = RGBAxes(fig, [0.1, 0.1, 0.8, 0.8], pad=0.0)
r, g, b = get_rgb()  # r, g, b are 2D images.
ax.imshow_rgb(r, g, b)
```


1.0
0.8
0.6
0.4
0.2
0.0
0.5
0.5
0.0
0.2
0.4
0.6
0.8
1.0
3
2
1
0
−1
−2
−3
−4
−2
0
2
4


0
2
4
6
8
10
12
0
2
4
6
8
10
12

### AxesDivider

The *mpl_toolkits.axes_grid1.axes_divider* module provides helper classes to adjust the axes positions of a set of images at drawing time.

- *axes_size* provides a class of units that are used to determine the size of each axes. For example, you can specify a fixed size.
- *Divider* is the class that calculates the axes position. It divides the given rectangular area into several areas. The divider is initialized by setting the lists of horizontal and vertical sizes on which the division will be based. Then use *new_locator()*, which returns a callable object that can be used to set the axes_locator of the axes.

Here, we demonstrate how to achieve the following layout: we want to position axes in a 3x4 grid (note that *Divider* makes row indices start from the *bottom*(!) of the grid):

```
┌──────┬──────┬──────┬──────┐
│ (2, 0) │ (2, 1) │ (2, 2) │ (2, 3) │
├──────┼──────┼──────┼──────┤
│ (1, 0) │ (1, 1) │ (1, 2) │ (1, 3) │
├──────┼──────┼──────┼──────┤
│ (0, 0) │ (0, 1) │ (0, 2) │ (0, 3) │
└──────┴──────┴──────┴──────┘
```

such that the bottom row has a fixed height of 2 (inches) and the top two rows have a height ratio of 2 (middle) to 3 (top). (For example, if the grid has a size of 7 inches, the bottom row will be 2 inches, the middle row also 2 inches, and the top row 3 inches.)

These constraints are specified using classes from the *axes_size* module, namely:

```
from mpl_toolkits.axes_grid1.axes_size import Fixed, Scaled
vert = [Fixed(2), Scaled(2), Scaled(3)]
```

(More generally, *axes_size* classes define a `get_size(renderer)` method that returns a pair of floats -- a relative size, and an absolute size. `Fixed(2).get_size(renderer)` returns `(0, 2)`; `Scaled(2).get_size(renderer)` returns `(2, 0)`.)

We use these constraints to initialize a *Divider* object:

```
rect = [0.2, 0.2, 0.6, 0.6]  # Position of the grid in the figure.
vert = [Fixed(2), Scaled(2), Scaled(3)]  # As above.
horiz = [...]  # Some other horizontal constraints.
divider = Divider(fig, rect, horiz, vert)
```

then use *Divider.new_locator* to create an axes locator callable for a given grid entry:

```
locator = divider.new_locator(nx=0, ny=1)  # Grid entry (1, 0).
```

and make it responsible for locating the axes:

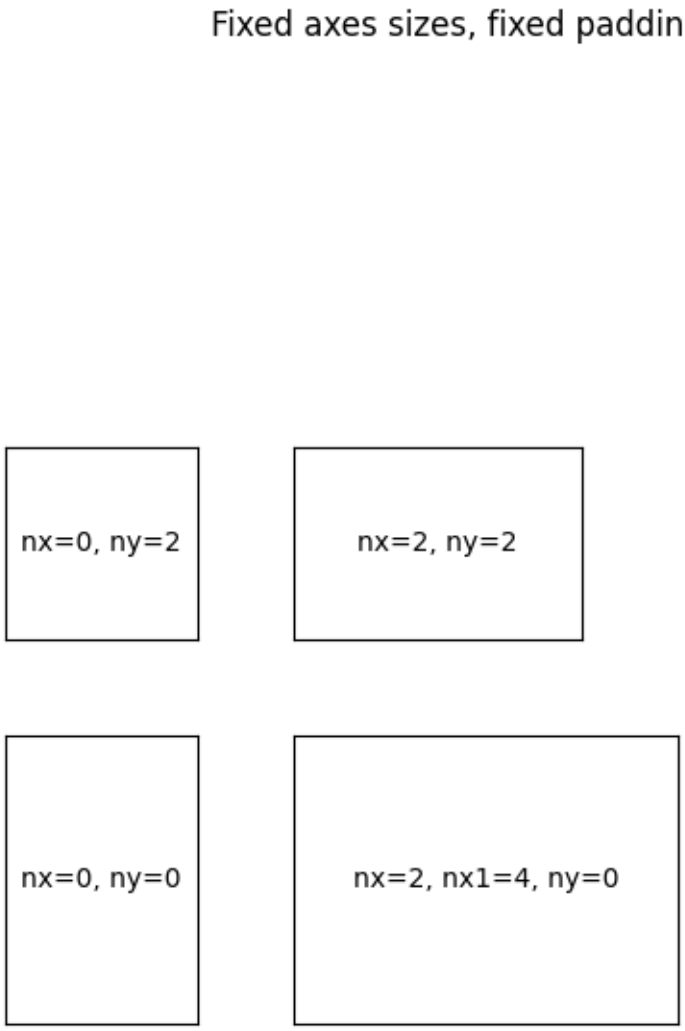
```
ax.set_axes_locator(locator)
```

The axes locator callable returns the location and size of the cell at the first column and the second row.

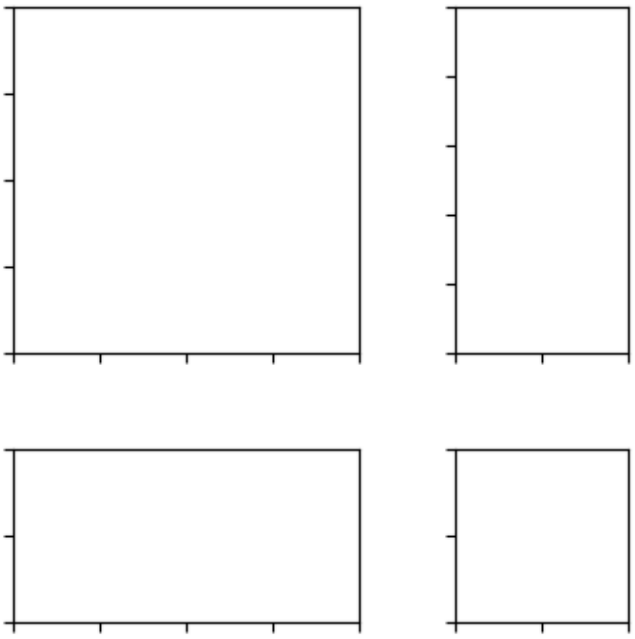
Locators that spans over multiple cells can be created with, e.g.:

```
# Columns #0 and #1 ("0-2 range"), row #1.
locator = divider.new_locator(nx=0, nx1=2, ny=1)
```

See the example,




You can also adjust the size of each axes according to its x or y data limits (AxesX and AxesY).



### 3.9.3 The mplot3d toolkit

Generating 3D plots using the mplot3d toolkit.

This tutorial showcases various 3D plots. Click on the figures to see each full gallery example with the code that generates the figures.

**Contents**

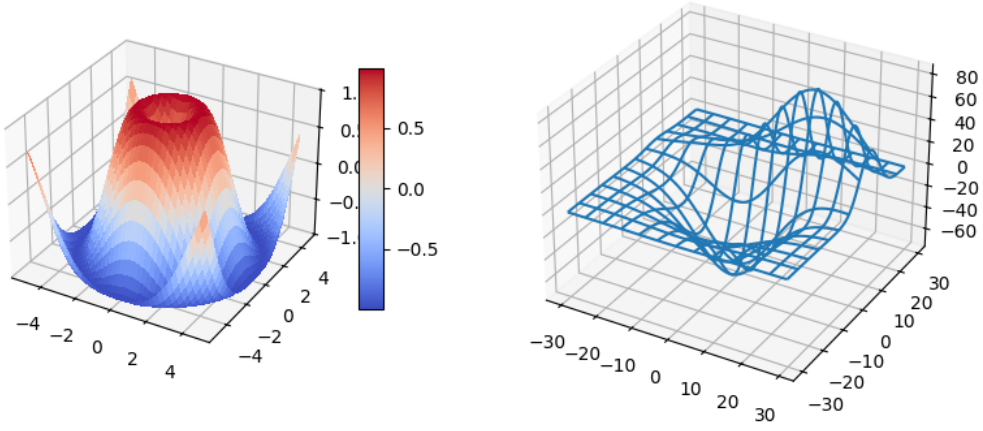

- *The mplot3d toolkit*
  - *Line plots*
  - *Scatter plots*
  - *Wireframe plots*
  - *Surface plots*
  - *Tri-Surface plots*
  - *Contour plots*
  - *Filled contour plots*
  - *Polygon plots*
  - *Bar plots*

- *Quiver*
- *2D plots in 3D*
- *Text*

3D Axes (of class *Axes3D*) are created by passing the `projection="3d"` keyword argument to *Figure.add_subplot*:

```
import matplotlib.pyplot as plt
fig = plt.figure()
ax = fig.add_subplot(projection='3d')
```

Multiple 3D subplots can be added on the same figure, as for 2D subplots.

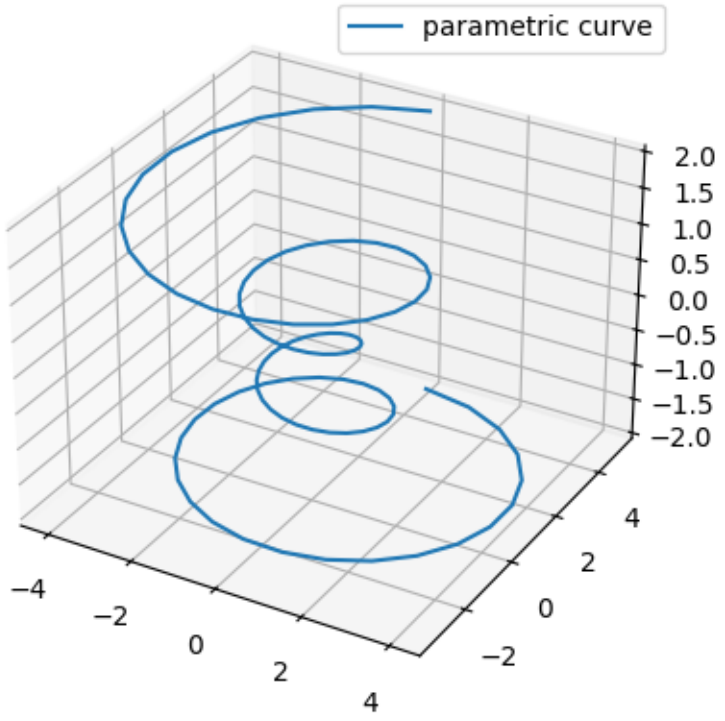


Changed in version 3.2.0: Prior to Matplotlib 3.2.0, it was necessary to explicitly import the *mpl_toolkits.mplot3d* module to make the '3d' projection to *Figure.add_subplot*.

See the *mplot3d FAQ* for more information about the mplot3d toolkit.

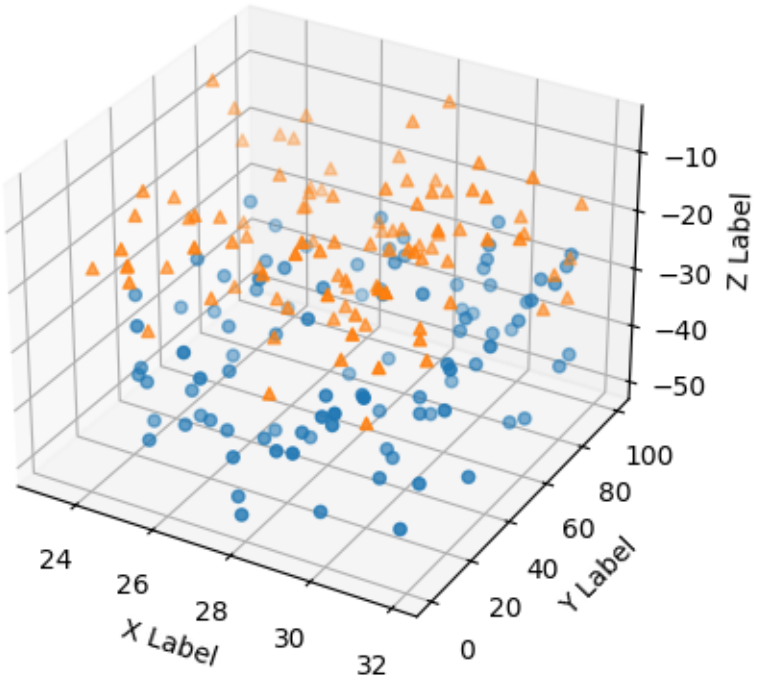
### Line plots

See *Axes3D.plot* for API documentation.


parametric curve
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−4
−2
0
2
4
4
2
0
−2

### Scatter plots

See *Axes3D.scatter* for API documentation.

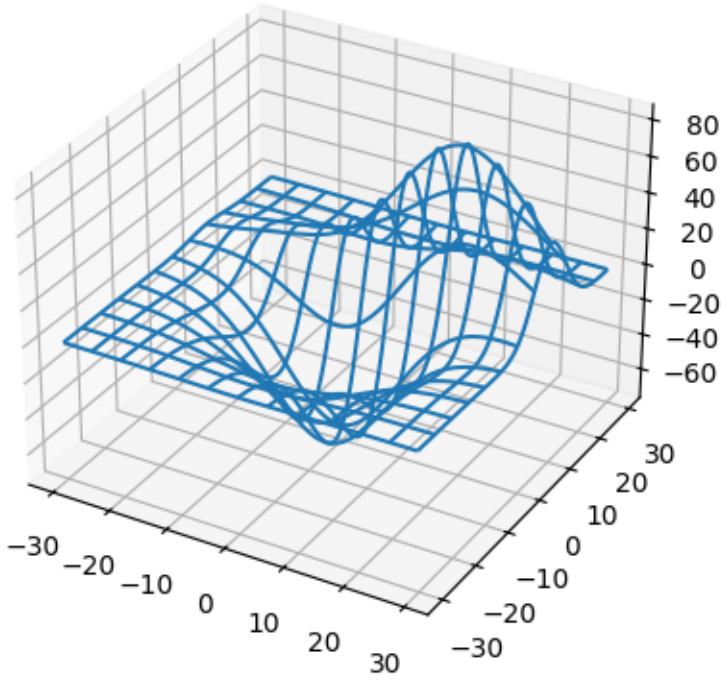


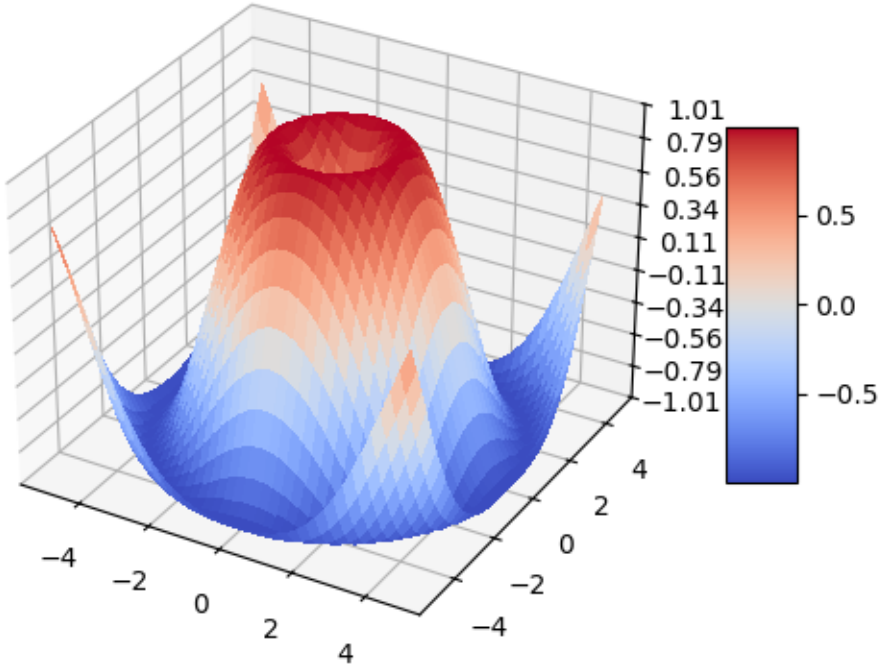

### Wireframe plots

See *Axes3D.plot_wireframe* for API documentation.

### Surface plots

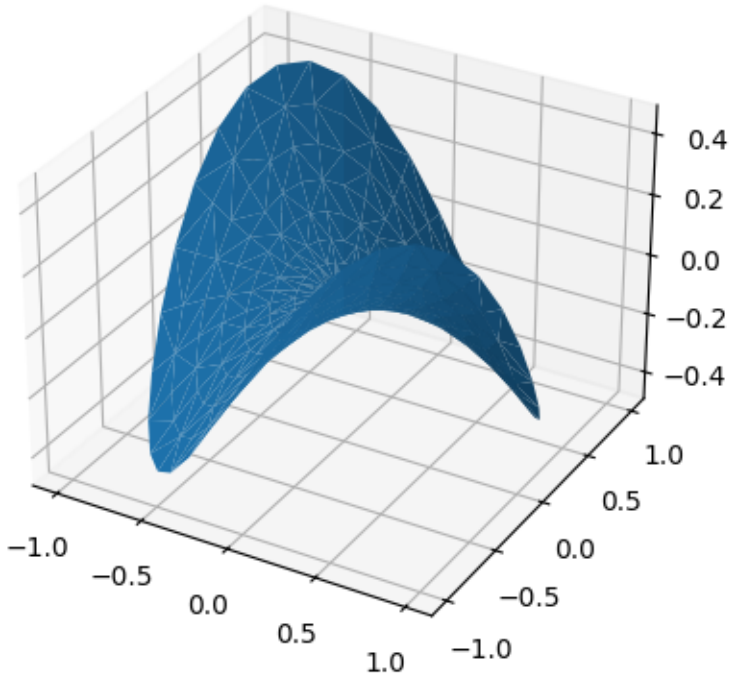
See *Axes3D.plot_surface* for API documentation.


80
60
40
20
0
−20
−40
−60
−30
−20
−10
0
10
20
30
30
20
10
0
−10
−20
−30


1.01
0.79
0.56
0.34
0.11
−0.11
−0.34
−0.56
−0.79
−1.01
0.5
0.0
−0.5
−4
−2
0
2
4
−4
−2
0
2
4

### Tri-Surface plots

See *Axes3D.plot_trisurf* for API documentation.



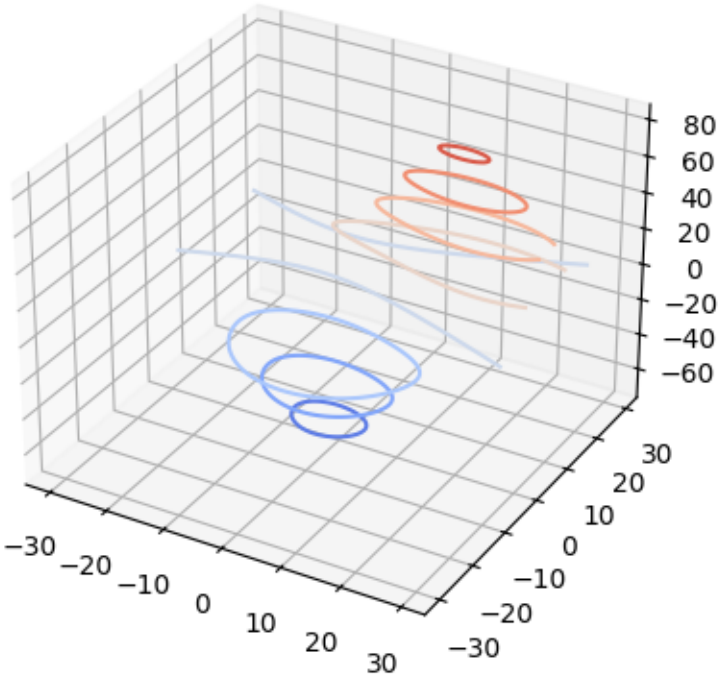

### Contour plots

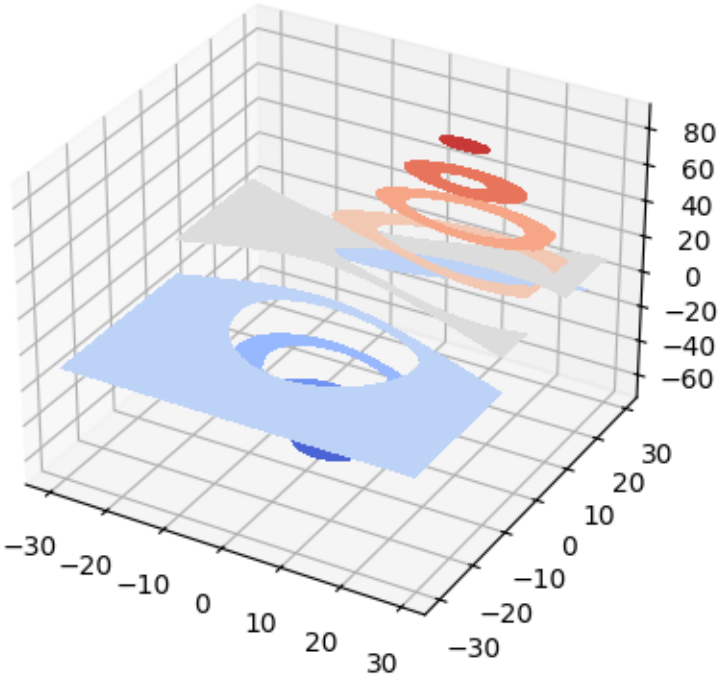
See *Axes3D.contour* for API documentation.

### Filled contour plots

See *Axes3D.contourf* for API documentation.

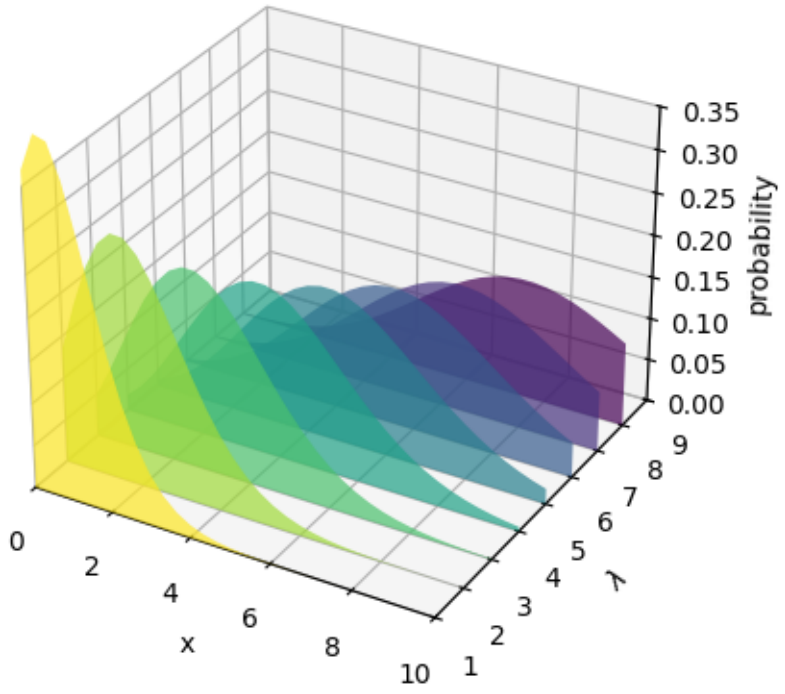
New in version 1.1.0: The feature demoed in the second contourf3d example was enabled as a result of a bugfix for version 1.1.0.


80
60
40
20
0
−20
−40
−60
−30
−20
−10
0
10
20
30
30
20
10
0
−10
−20
−30


80
60
40
20
0
−20
−40
−60
−30
−20
−10
0
10
20
30
30
20
10
0
−10
−20
−30

### Polygon plots

See *Axes3D.add_collection3d* for API documentation.

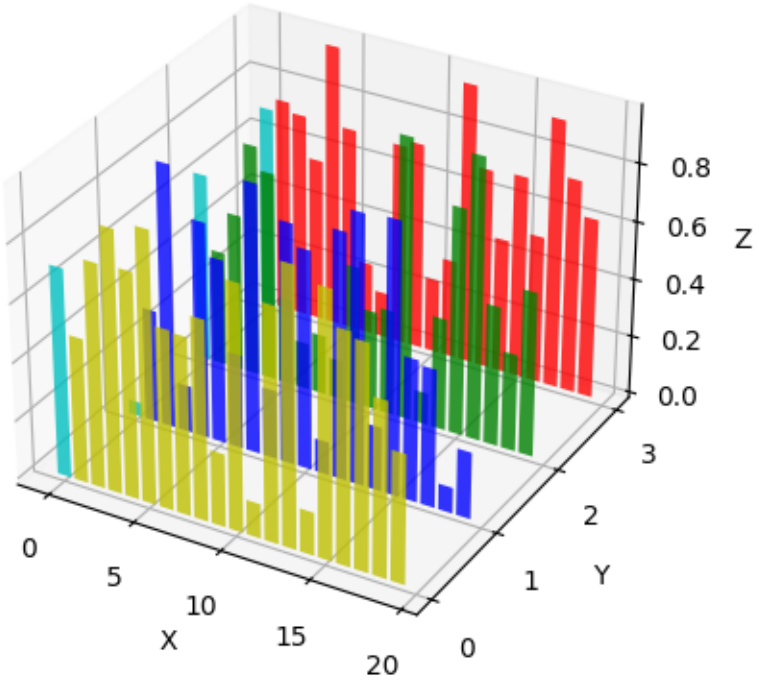


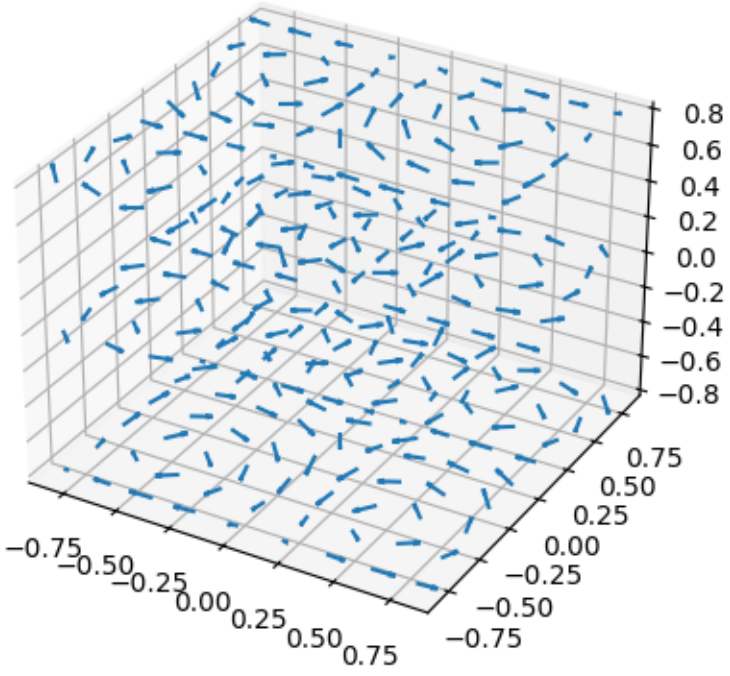

### Bar plots

See *Axes3D.bar* for API documentation.

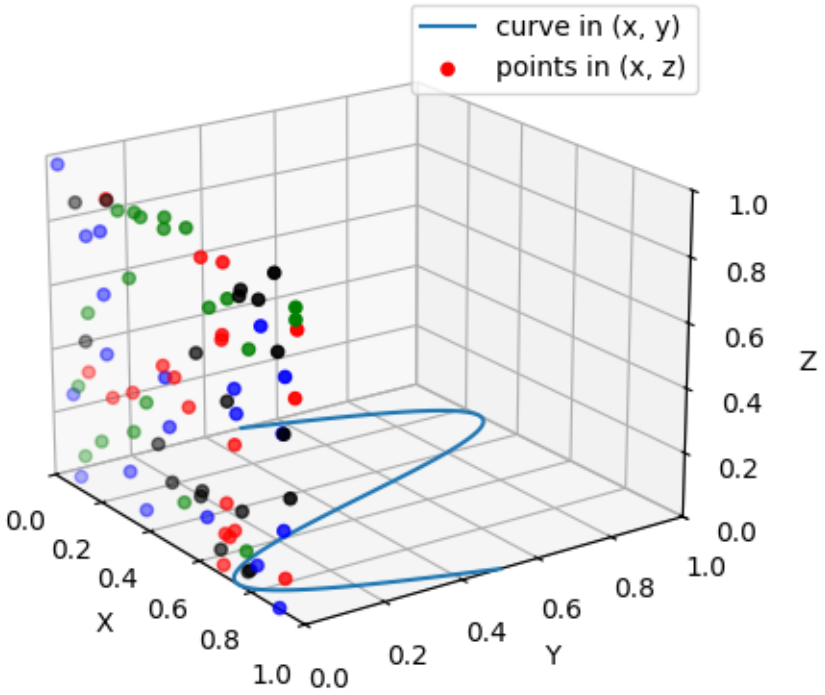
### Quiver

See *Axes3D.quiver* for API documentation.


0.8
0.6
Z
0.4
0.2
0.0
3
2
1
Y
0
0
5
10
15
20
X


0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75

### 2D plots in 3D




### Text

See `Axes3D.text` for API documentation.

## 3.10 Getting started

### 3.10.1 Installation quick-start

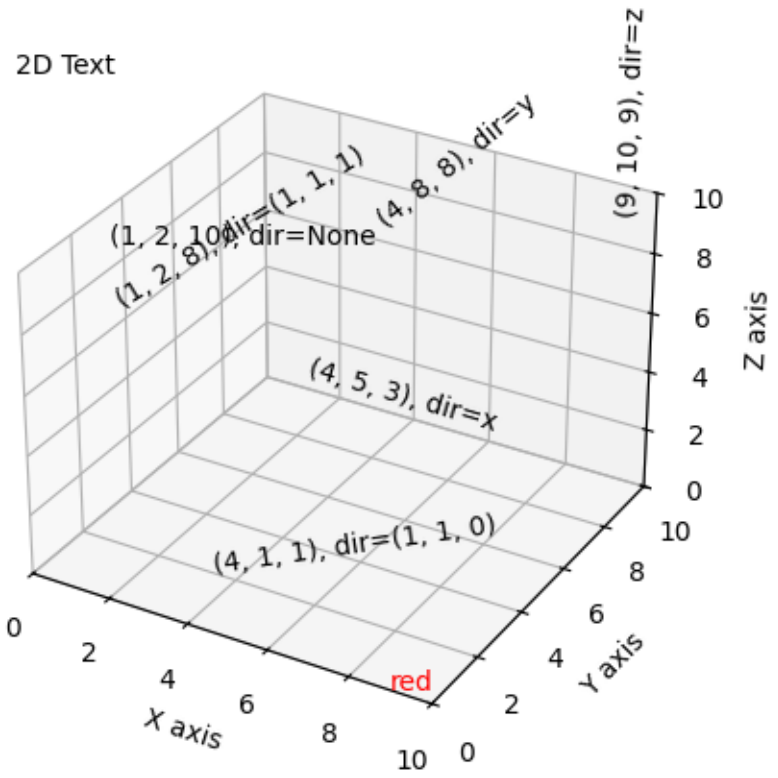
Install using pip:

```
pip install matplotlib
```

Install using conda:

```
conda install -c conda-forge matplotlib
```

Further details are available in the *Installation Guide*.


2D Text
(1, 2, 10), dir=(1, 1, 1)
(1, 2, 8), dir=None
(4, 8, 8), dir=y
(9, 10, 9), dir=z
(4, 5, 3), dir=x
(4, 1, 1), dir=(1, 1, 0)
red
X axis
Y axis
Z axis
0
2
4
6
8
10

### 3.10.2 Draw a first plot

Here is a minimal example plot:

```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 2 * np.pi, 200)
y = np.sin(x)

fig, ax = plt.subplots()
ax.plot(x, y)
plt.show()
```

If a plot does not show up please check *Troubleshooting*.

### 3.10.3 Where to go next

- Check out *Plot types* to get an overview of the types of plots you can create with Matplotlib.
- Learn Matplotlib from the ground up in the *Quick-start guide*.

CHAPTER

FOUR

# TUTORIALS

This page contains a few tutorials for using Matplotlib. For the old tutorials, see *below*.

For shorter examples, see our *examples page*. You can also find *external resources* and a FAQ in our *user guide*.

## 4.1 Pyplot tutorial

An introduction to the pyplot interface. Please also see *Quick start guide* for an overview of how Matplotlib works and *Matplotlib Application Interfaces (APIs)* for an explanation of the trade-offs between the supported user APIs.

### 4.1.1 Introduction to pyplot

`matplotlib.pyplot` is a collection of functions that make matplotlib work like MATLAB. Each `pyplot` function makes some change to a figure: e.g., creates a figure, creates a plotting area in a figure, plots some lines in a plotting area, decorates the plot with labels, etc.

In `matplotlib.pyplot` various states are preserved across function calls, so that it keeps track of things like the current figure and plotting area, and the plotting functions are directed to the current Axes (please note that we use uppercase Axes to refer to the `Axes` concept, which is a central *part of a figure* and not only the plural of *axis*).

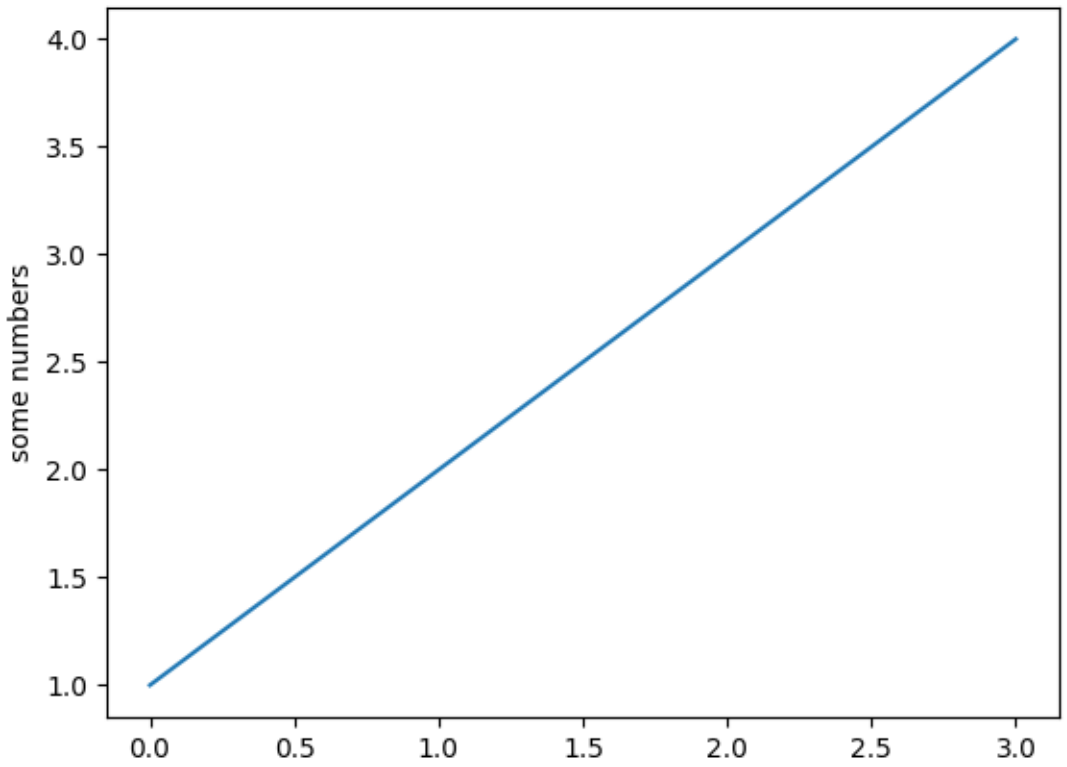
---

**Note:** The implicit pyplot API is generally less verbose but also not as flexible as the explicit API. Most of the function calls you see here can also be called as methods from an `Axes` object. We recommend browsing the tutorials and examples to see how this works. See *Matplotlib Application Interfaces (APIs)* for an explanation of the trade-off of the supported user APIs.

---

Generating visualizations with pyplot is very quick:

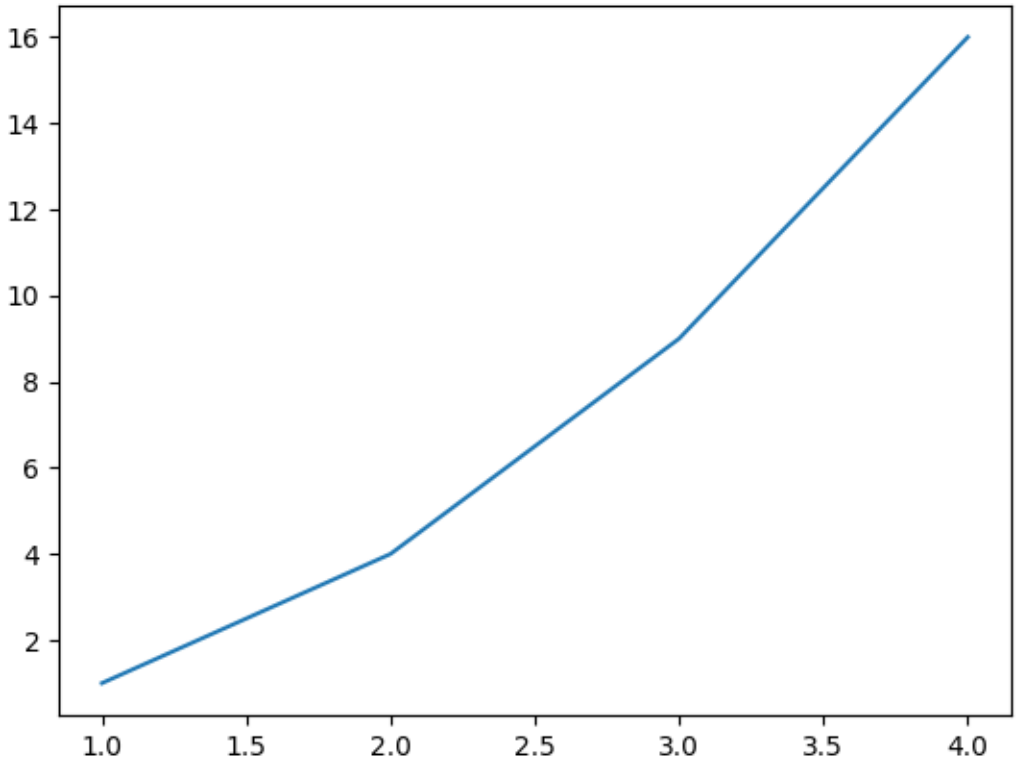
```
import matplotlib.pyplot as plt

plt.plot([1, 2, 3, 4])
plt.ylabel('some numbers')
plt.show()
```




You may be wondering why the x-axis ranges from 0-3 and the y-axis from 1-4. If you provide a single list or array to *plot*, matplotlib assumes it is a sequence of y values, and automatically generates the x values for you. Since python ranges start with 0, the default x vector has the same length as y but starts with 0; therefore, the x data are `[0, 1, 2, 3]`.

*plot* is a versatile function, and will take an arbitrary number of arguments. For example, to plot x versus y, you can write:

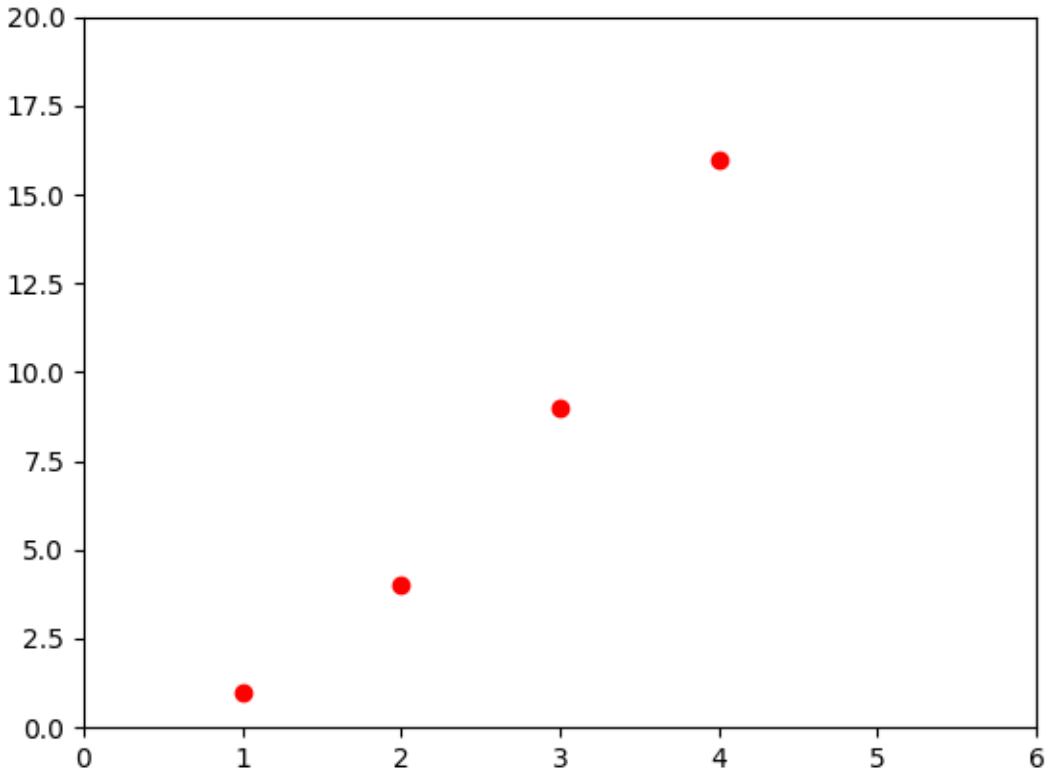
```
plt.plot([1, 2, 3, 4], [1, 4, 9, 16])
```




### Formatting the style of your plot

For every x, y pair of arguments, there is an optional third argument which is the format string that indicates the color and line type of the plot. The letters and symbols of the format string are from MATLAB, and you concatenate a color string with a line style string. The default format string is 'b-', which is a solid blue line. For example, to plot the above with red circles, you would issue

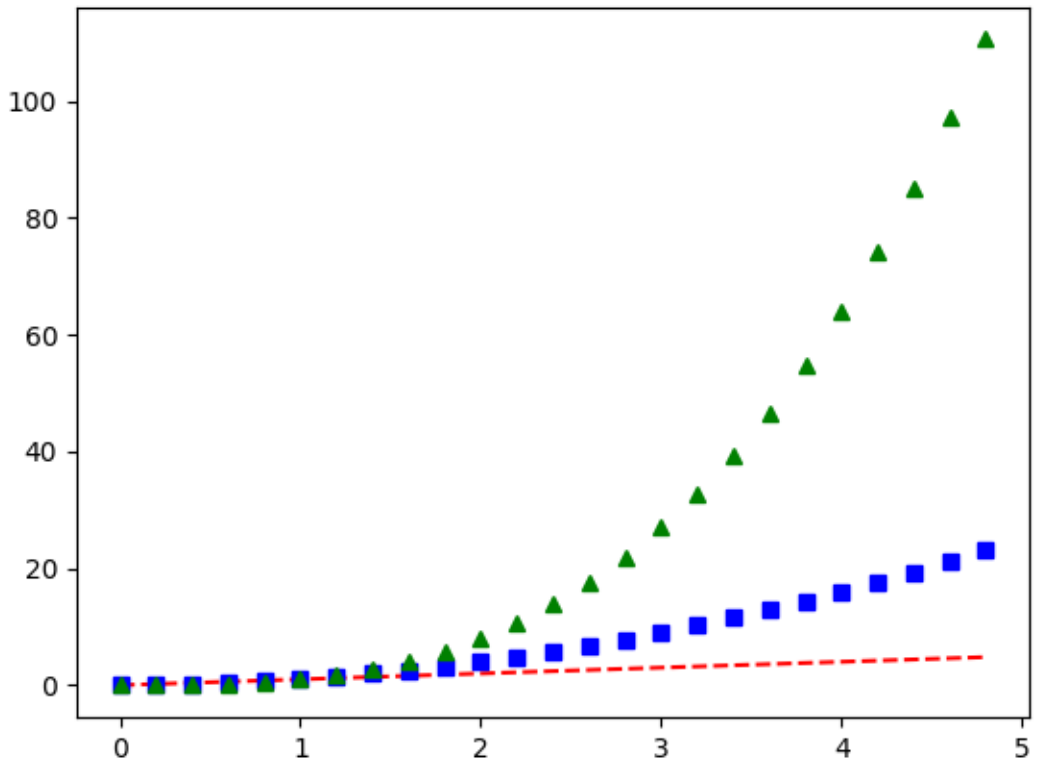
```
plt.plot([1, 2, 3, 4], [1, 4, 9, 16], 'ro')
plt.axis((0, 6, 0, 20))
plt.show()
```




See the *plot* documentation for a complete list of line styles and format strings. The *axis* function in the example above takes a list of `[xmin, xmax, ymin, ymax]` and specifies the viewport of the Axes.

If matplotlib were limited to working with lists, it would be fairly useless for numeric processing. Generally, you will use numpy arrays. In fact, all sequences are converted to numpy arrays internally. The example below illustrates plotting several lines with different format styles in one function call using arrays.

```
import numpy as np

# evenly sampled time at 200ms intervals
t = np.arange(0., 5., 0.2)

# red dashes, blue squares and green triangles
plt.plot(t, t, 'r--', t, t**2, 'bs', t, t**3, 'g^')
plt.show()
```

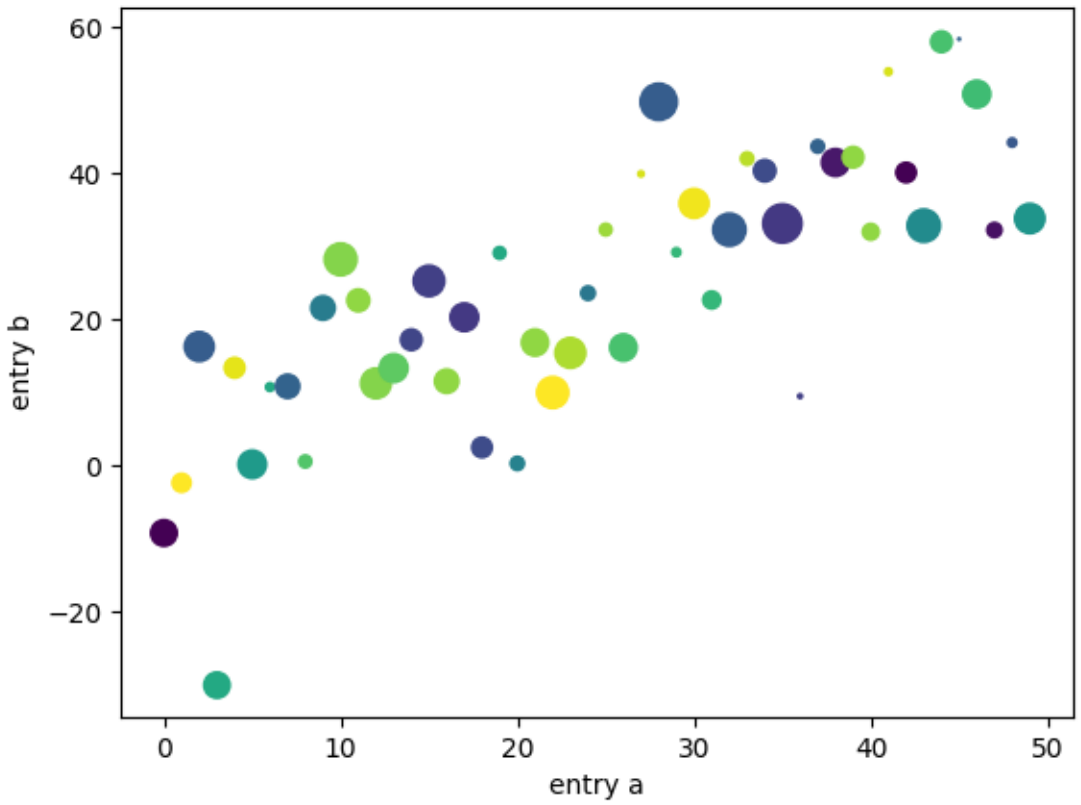



### 4.1.2 Plotting with keyword strings

There are some instances where you have data in a format that lets you access particular variables with strings. For example, with structured arrays or `pandas.DataFrame`.

Matplotlib allows you to provide such an object with the `data` keyword argument. If provided, then you may generate plots with the strings corresponding to these variables.

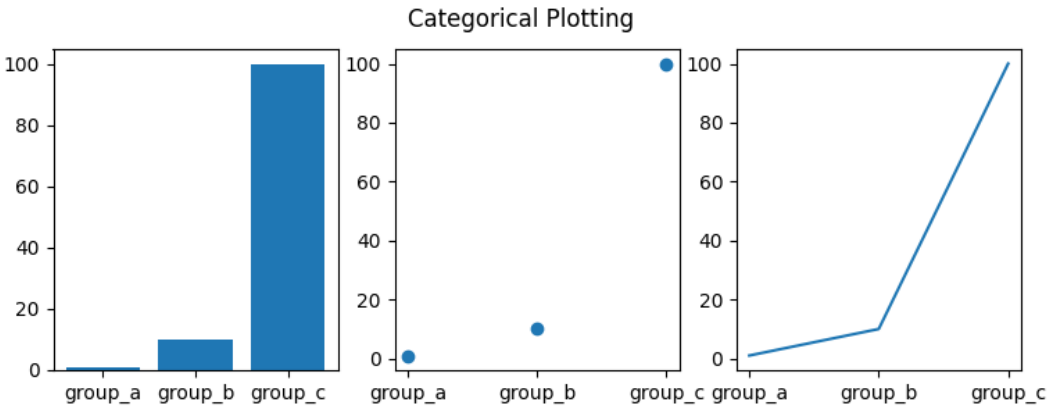
```
data = {'a': np.arange(50),
        'c': np.random.randint(0, 50, 50),
        'd': np.random.randn(50)}
data['b'] = data['a'] + 10 * np.random.randn(50)
data['d'] = np.abs(data['d']) * 100

plt.scatter('a', 'b', c='c', s='d', data=data)
plt.xlabel('entry a')
plt.ylabel('entry b')
plt.show()
```




### 4.1.3 Plotting with categorical variables

It is also possible to create a plot using categorical variables. Matplotlib allows you to pass categorical variables directly to many plotting functions. For example:

```
names = ['group_a', 'group_b', 'group_c']
values = [1, 10, 100]

plt.figure(figsize=(9, 3))

plt.subplot(131)
plt.bar(names, values)
plt.subplot(132)
plt.scatter(names, values)
plt.subplot(133)
plt.plot(names, values)
plt.suptitle('Categorical Plotting')
plt.show()
```




### 4.1.4 Controlling line properties

Lines have many attributes that you can set: linewidth, dash style, antialiased, etc; see *matplotlib.lines.Line2D*. There are several ways to set line properties

- Use keyword arguments:

```
plt.plot(x, y, linewidth=2.0)
```

- Use the setter methods of a `Line2D` instance. `plot` returns a list of `Line2D` objects; e.g., `line1, line2 = plot(x1, y1, x2, y2)`. In the code below we will suppose that we have only one line so that the list returned is of length 1. We use tuple unpacking with `line,` to get the first element of that list:

```
line, = plt.plot(x, y, '-')
line.set_antialiased(False) # turn off antialiasing
```

- Use *setp*. The example below uses a MATLAB-style function to set multiple properties on a list of lines. `setp` works transparently with a list of objects or a single object. You can either use python keyword arguments or MATLAB-style string/value pairs:

```
lines = plt.plot(x1, y1, x2, y2)
# use keyword arguments
plt.setp(lines, color='r', linewidth=2.0)
# or MATLAB style string value pairs
plt.setp(lines, 'color', 'r', 'linewidth', 2.0)
```

Here are the available *Line2D* properties.

| Property | Value Type |
|---|---|
| alpha | float |
| animated | [True \| False] |
| antialiased or aa | [True \| False] |
| clip_box | a matplotlib.transform.Bbox instance |

continues on next page

Table 1 – continued from previous page

| Property | Value Type |
|---|---|
| clip_on | [True \| False] |
| clip_path | a Path instance and a Transform instance, a Patch |
| color or c | any matplotlib color |
| contains | the hit testing function |
| dash_capstyle | ['butt' \| 'round' \| 'projecting'] |
| dash_joinstyle | ['miter' \| 'round' \| 'bevel'] |
| dashes | sequence of on/off ink in points |
| data | (np.array xdata, np.array ydata) |
| figure | a matplotlib.figure.Figure instance |
| label | any string |
| linestyle or ls | [ '-' \| '--' \| '-.' \| ':' \| 'steps' \| ...] |
| linewidth or lw | float value in points |
| marker | [ '+' \| ',' \| '.' \| '1' \| '2' \| '3' \| '4' ] |
| markeredgecolor or mec | any matplotlib color |
| markeredgewidth or mew | float value in points |
| markerfacecolor or mfc | any matplotlib color |
| markersize or ms | float |
| markevery | [ None \| integer \| (startind, stride) ] |
| picker | used in interactive line selection |
| pickradius | the line pick selection radius |
| solid_capstyle | ['butt' \| 'round' \| 'projecting'] |
| solid_joinstyle | ['miter' \| 'round' \| 'bevel'] |
| transform | a matplotlib.transforms.Transform instance |
| visible | [True \| False] |
| xdata | np.array |
| ydata | np.array |
| zorder | any number |

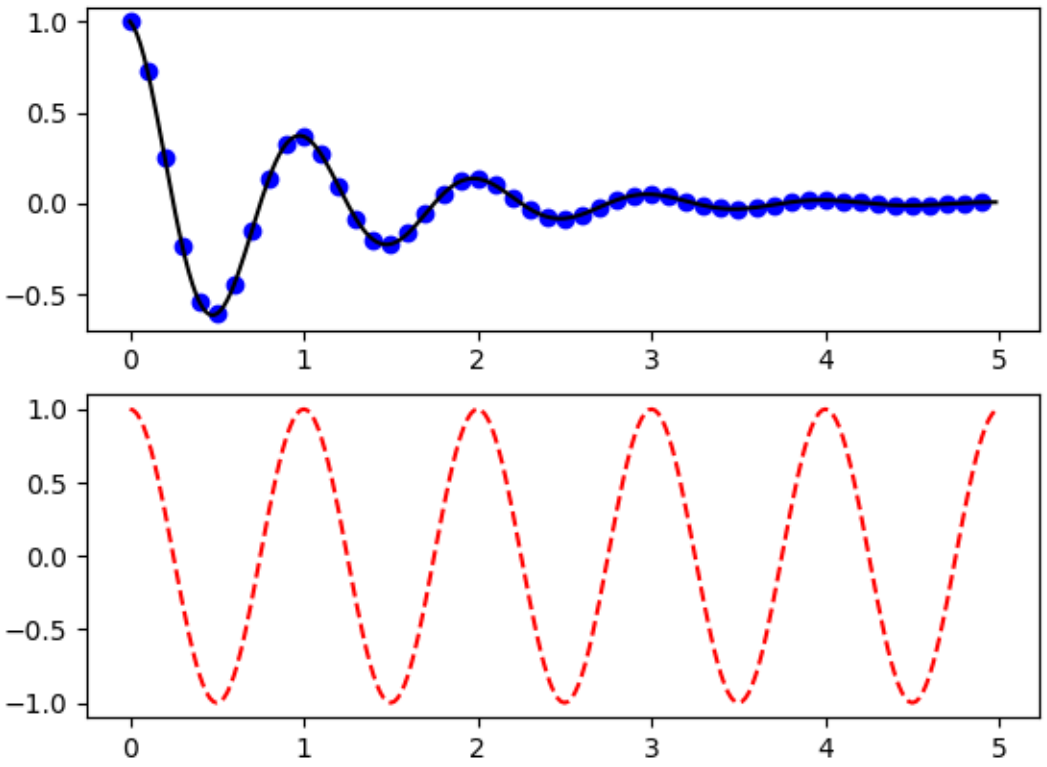
To get a list of settable line properties, call the *setp* function with a line or lines as argument

```
In [69]: lines = plt.plot([1, 2, 3])

In [70]: plt.setp(lines)
  alpha: float
  animated: [True | False]
  antialiased or aa: [True | False]
  ...snip
```

### 4.1.5 Working with multiple figures and Axes

MATLAB, and *pyplot*, have the concept of the current figure and the current Axes. All plotting functions apply to the current Axes. The function *gca* returns the current Axes (a *matplotlib.axes.Axes* instance), and *gcf* returns the current figure (a *matplotlib.figure.Figure* instance). Normally, you don't have to worry about this, because it is all taken care of behind the scenes. Below is a script to create two subplots.

```
def f(t):
    return np.exp(-t) * np.cos(2*np.pi*t)

t1 = np.arange(0.0, 5.0, 0.1)
t2 = np.arange(0.0, 5.0, 0.02)

plt.figure()
plt.subplot(211)
plt.plot(t1, f(t1), 'bo', t2, f(t2), 'k')

plt.subplot(212)
plt.plot(t2, np.cos(2*np.pi*t2), 'r--')
plt.show()
```




The *figure* call here is optional because a figure will be created if none exists, just as an Axes will be

created (equivalent to an explicit `subplot()` call) if none exists. The *subplot* call specifies `numrows, numcols, plot_number` where `plot_number` ranges from 1 to `numrows*numcols`. The commas in the `subplot` call are optional if `numrows*numcols<10`. So `subplot(211)` is identical to `subplot(2, 1, 1)`.

You can create an arbitrary number of subplots and Axes. If you want to place an Axes manually, i.e., not on a rectangular grid, use *axes*, which allows you to specify the location as `axes([left, bottom, width, height])` where all values are in fractional (0 to 1) coordinates. See *Axes Demo* for an example of placing Axes manually and *Multiple subplots* for an example with lots of subplots.

You can create multiple figures by using multiple *figure* calls with an increasing figure number. Of course, each figure can contain as many Axes and subplots as your heart desires:

```
import matplotlib.pyplot as plt
plt.figure(1)                # the first figure
plt.subplot(211)             # the first subplot in the first figure
plt.plot([1, 2, 3])
plt.subplot(212)             # the second subplot in the first figure
plt.plot([4, 5, 6])


plt.figure(2)                # a second figure
plt.plot([4, 5, 6])          # creates a subplot() by default

plt.figure(1)                # first figure current;
                             # subplot(212) still current
plt.subplot(211)             # make subplot(211) in the first figure
                             # current
plt.title('Easy as 1, 2, 3') # subplot 211 title
```

You can clear the current figure with *clf* and the current Axes with *cla*. If you find it annoying that states (specifically the current image, figure and Axes) are being maintained for you behind the scenes, don't despair: this is just a thin stateful wrapper around an object-oriented API, which you can use instead (see *Artist tutorial*)

If you are making lots of figures, you need to be aware of one more thing: the memory required for a figure is not completely released until the figure is explicitly closed with *close*. Deleting all references to the figure, and/or using the window manager to kill the window in which the figure appears on the screen, is not enough, because pyplot maintains internal references until *close* is called.

### 4.1.6 Working with text

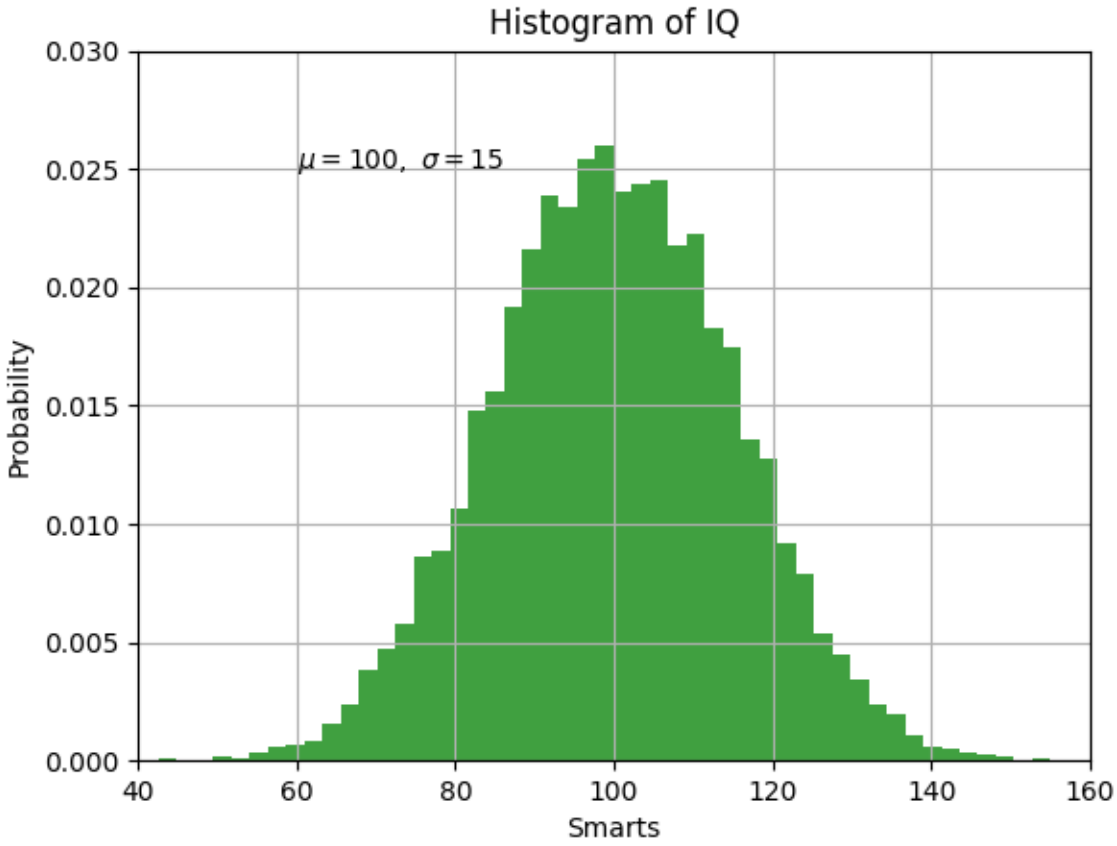
*text* can be used to add text in an arbitrary location, and *xlabel*, *ylabel* and *title* are used to add text in the indicated locations (see *Text in Matplotlib* for a more detailed example)

```
mu, sigma = 100, 15
x = mu + sigma * np.random.randn(10000)

# the histogram of the data
n, bins, patches = plt.hist(x, 50, density=True, facecolor='g', alpha=0.75)
```

(continues on next page)

(continued from previous page)

```
plt.xlabel('Smarts')
plt.ylabel('Probability')
plt.title('Histogram of IQ')
plt.text(60, .025, r'$\mu=100,\ \sigma=15$')
plt.axis([40, 160, 0, 0.03])
plt.grid(True)
plt.show()
```




All of the *text* functions return a *matplotlib.text.Text* instance. Just as with lines above, you can customize the properties by passing keyword arguments into the text functions or using *setp*:

```
t = plt.xlabel('my data', fontsize=14, color='red')
```

These properties are covered in more detail in *Text properties and layout*.

### Using mathematical expressions in text

Matplotlib accepts TeX equation expressions in any text expression. For example to write the expression $\sigma_i = 15$ in the title, you can write a TeX expression surrounded by dollar signs:

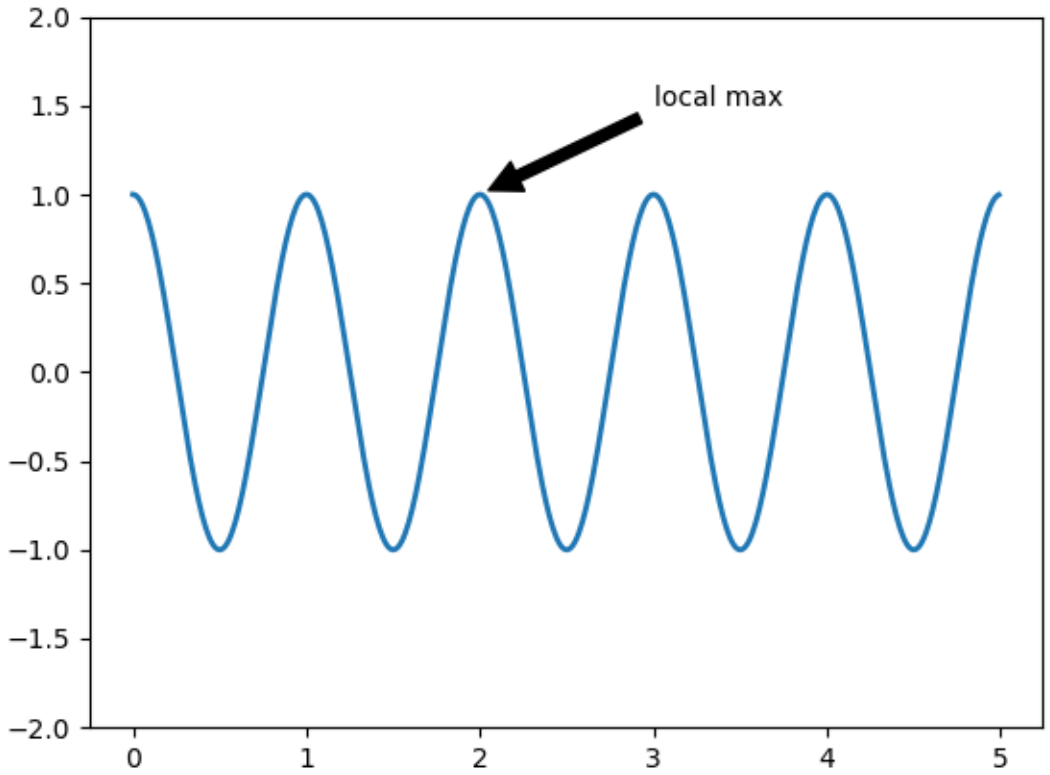
```
plt.title(r'$\sigma_i=15$')
```

The `r` preceding the title string is important -- it signifies that the string is a *raw* string and not to treat backslashes as python escapes. matplotlib has a built-in TeX expression parser and layout engine, and ships its own math fonts -- for details see *Writing mathematical expressions*. Thus, you can use mathematical text across platforms without requiring a TeX installation. For those who have LaTeX and dvipng installed, you can also use LaTeX to format your text and incorporate the output directly into your display figures or saved postscript -- see *Text rendering with LaTeX*.

### Annotating text

The uses of the basic *`text`* function above place text at an arbitrary position on the Axes. A common use for text is to annotate some feature of the plot, and the *`annotate`* method provides helper functionality to make annotations easy. In an annotation, there are two points to consider: the location being annotated represented by the argument `xy` and the location of the text `xytext`. Both of these arguments are `(x, y)` tuples.

```
ax = plt.subplot()

t = np.arange(0.0, 5.0, 0.01)
s = np.cos(2*np.pi*t)
line, = plt.plot(t, s, lw=2)

plt.annotate('local max', xy=(2, 1), xytext=(3, 1.5),
             arrowprops=dict(facecolor='black', shrink=0.05),
             )

plt.ylim(-2, 2)
plt.show()
```




In this basic example, both the `xy` (arrow tip) and `xytext` locations (text location) are in data coordinates. There are a variety of other coordinate systems one can choose -- see *Basic annotation* and *Advanced annotation* for details. More examples can be found in *Annotating Plots*.

### 4.1.7 Logarithmic and other nonlinear axes

*matplotlib.pyplot* supports not only linear axis scales, but also logarithmic and logit scales. This is commonly used if data spans many orders of magnitude. Changing the scale of an axis is easy:

```
plt.xscale('log')
```

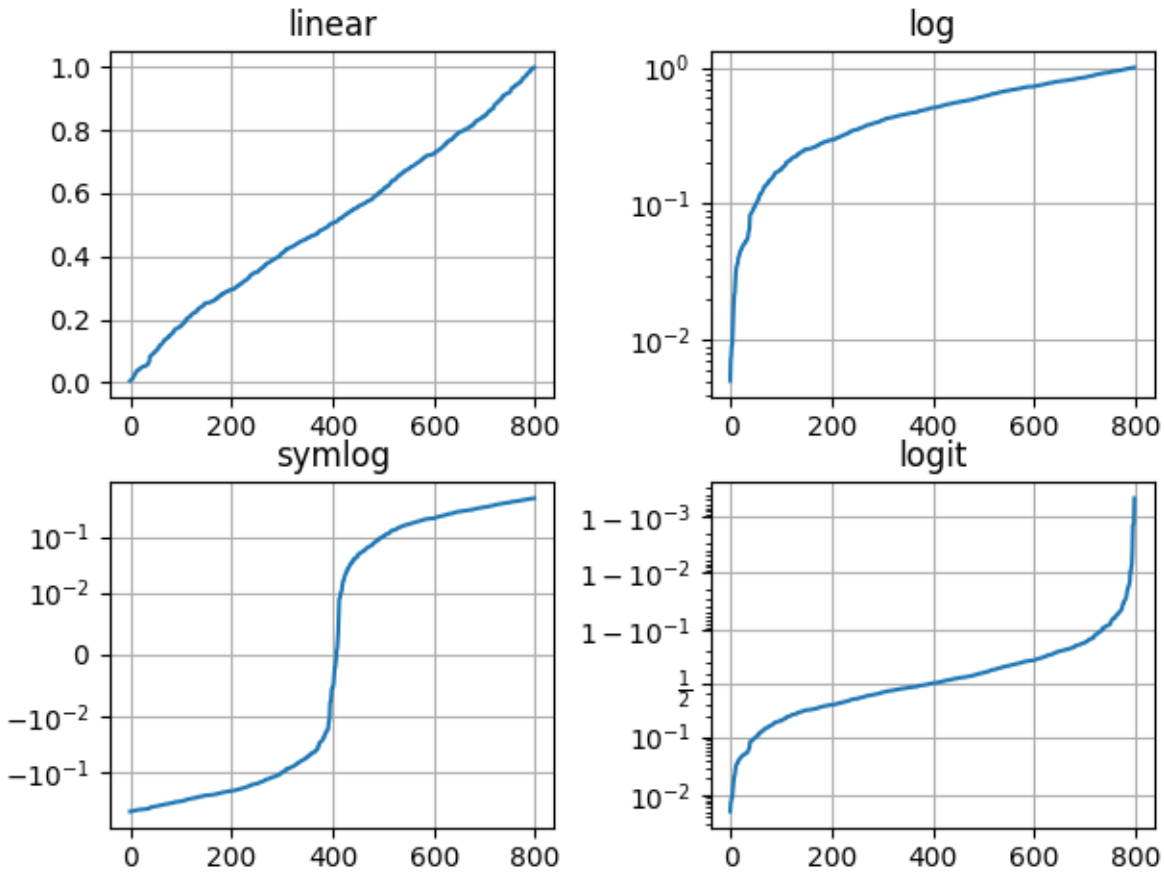
An example of four plots with the same data and different scales for the y-axis is shown below.

```
# Fixing random state for reproducibility
np.random.seed(19680801)

# make up some data in the open interval (0, 1)
y = np.random.normal(loc=0.5, scale=0.4, size=1000)
y = y[(y > 0) & (y < 1)]
y.sort()
x = np.arange(len(y))
```

(continues on next page)

(continued from previous page)

```
# plot with various axes scales
plt.figure()

# linear
plt.subplot(221)
plt.plot(x, y)
plt.yscale('linear')
plt.title('linear')
plt.grid(True)

# log
plt.subplot(222)
plt.plot(x, y)
plt.yscale('log')
plt.title('log')
plt.grid(True)

# symmetric log
plt.subplot(223)
plt.plot(x, y - y.mean())
plt.yscale('symlog', linthresh=0.01)
plt.title('symlog')
plt.grid(True)

# logit
plt.subplot(224)
plt.plot(x, y)
plt.yscale('logit')
plt.title('logit')
plt.grid(True)
# Adjust the subplot layout, because the logit one may take more space
# than usual, due to y-tick labels like "1 - 10^{-3}"
plt.subplots_adjust(top=0.92, bottom=0.08, left=0.10, right=0.95, hspace=0.25,
                    wspace=0.35)

plt.show()
```




It is also possible to add your own scale, see `matplotlib.scale` for details.

**Total running time of the script:** (0 minutes 3.137 seconds)

## 4.2 Image tutorial

A short tutorial on plotting images with Matplotlib.

### 4.2.1 Startup commands

First, let's start IPython. It is a most excellent enhancement to the standard Python prompt, and it ties in especially well with Matplotlib. Start IPython either directly at a shell, or with the Jupyter Notebook (where IPython as a running kernel).

With IPython started, we now need to connect to a GUI event loop. This tells IPython where (and how) to display plots. To connect to a GUI loop, execute the **%matplotlib** magic at your IPython prompt. There's more detail on exactly what this does at IPython's documentation on GUI event loops.

If you're using Jupyter Notebook, the same commands are available, but people commonly use a specific argument to the %matplotlib magic:

```
In [1]: %matplotlib inline
```

This turns on inline plotting, where plot graphics will appear in your notebook. This has important implications for interactivity. For inline plotting, commands in cells below the cell that outputs a plot will not affect the plot. For example, changing the colormap is not possible from cells below the cell that creates a plot. However, for other backends, such as Qt, that open a separate window, cells below those that create the plot will change the plot - it is a live object in memory.

This tutorial will use Matplotlib's implicit plotting interface, pyplot. This interface maintains global state, and is very useful for quickly and easily experimenting with various plot settings. The alternative is the explicit, which is more suitable for large application development. For an explanation of the tradeoffs between the implicit and explicit interfaces see *Matplotlib Application Interfaces (APIs)* and the *Quick start guide* to start using the explicit interface. For now, let's get on with the implicit approach:

```
from PIL import Image

import matplotlib.pyplot as plt
import numpy as np
```

### 4.2.2 Importing image data into Numpy arrays

Matplotlib relies on the Pillow library to load image data.

Here's the image we're going to play with:

It's a 24-bit RGB PNG image (8 bits for each of R, G, B). Depending on where you get your data, the other kinds of image that you'll most likely encounter are RGBA images, which allow for transparency, or single-channel grayscale (luminosity) images. Download stinkbug.png to your computer for the rest of this tutorial.

We use Pillow to open an image (with `PIL.Image.open`), and immediately convert the `PIL.Image.Image` object into an 8-bit (`dtype=uint8`) numpy array.

```
img = np.asarray(Image.open('../../doc/_static/stinkbug.png'))
print(repr(img))
```

```
array([[[104, 104, 104],
        [104, 104, 104],
        [104, 104, 104],
        ...,
        [109, 109, 109],
        [109, 109, 109],
        [109, 109, 109]],

       [[105, 105, 105],
        [105, 105, 105],
        [105, 105, 105],
        ...,
```

(continues on next page)

(continued from previous page)

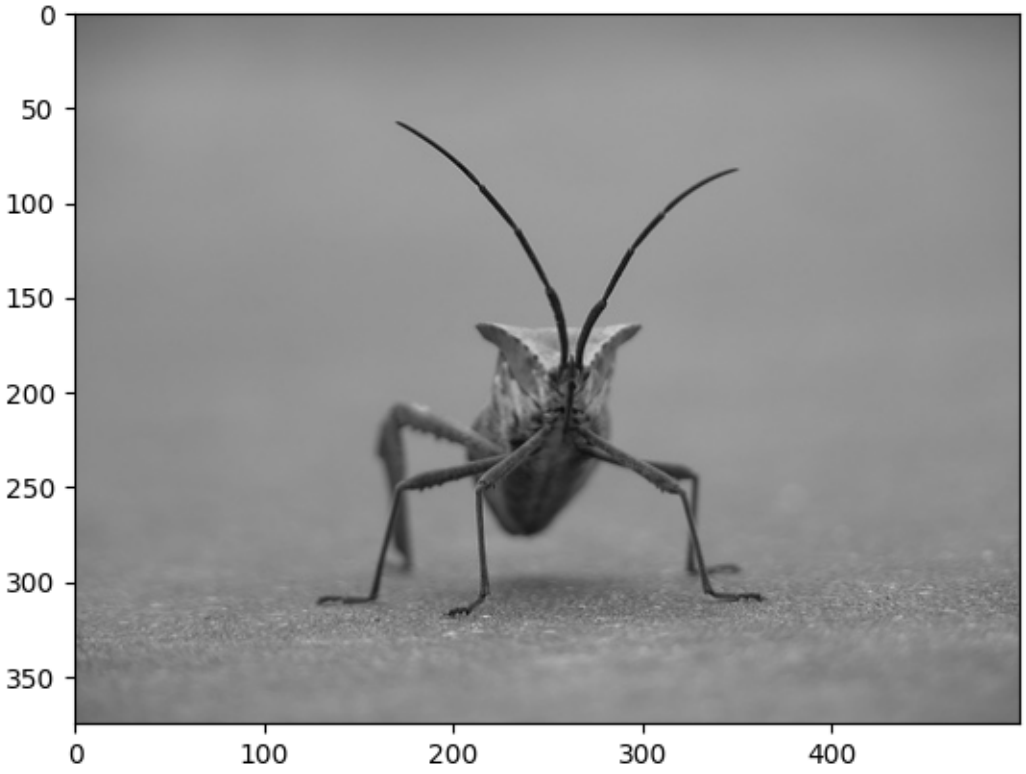
```
        [109, 109, 109],
        [109, 109, 109],
        [109, 109, 109]],

       [[107, 107, 107],
        [106, 106, 106],
        [106, 106, 106],
        ...,
        [110, 110, 110],
        [110, 110, 110],
        [110, 110, 110]],

       ...,

       [[112, 112, 112],
        [111, 111, 111],
        [110, 110, 110],
        ...,
        [116, 116, 116],
        [115, 115, 115],
        [115, 115, 115]],

       [[113, 113, 113],
        [113, 113, 113],
        [112, 112, 112],
        ...,
        [115, 115, 115],
        [114, 114, 114],
        [114, 114, 114]],

       [[113, 113, 113],
        [115, 115, 115],
        [115, 115, 115],
        ...,
        [114, 114, 114],
        [114, 114, 114],
        [113, 113, 113]]], dtype=uint8)
```

Each inner list represents a pixel. Here, with an RGB image, there are 3 values. Since it's a black and white image, R, G, and B are all similar. An RGBA (where A is alpha, or transparency) has 4 values per inner list, and a simple luminance image just has one value (and is thus only a 2-D array, not a 3-D array). For RGB and RGBA images, Matplotlib supports float32 and uint8 data types. For grayscale, Matplotlib supports only float32. If your array data does not meet one of these descriptions, you need to rescale it.

### 4.2.3 Plotting numpy arrays as images

So, you have your data in a numpy array (either by importing it, or by generating it). Let's render it. In Matplotlib, this is performed using the *imshow()* function. Here we'll grab the plot object. This object gives you an easy way to manipulate the plot from the prompt.

```
imgplot = plt.imshow(img)
```



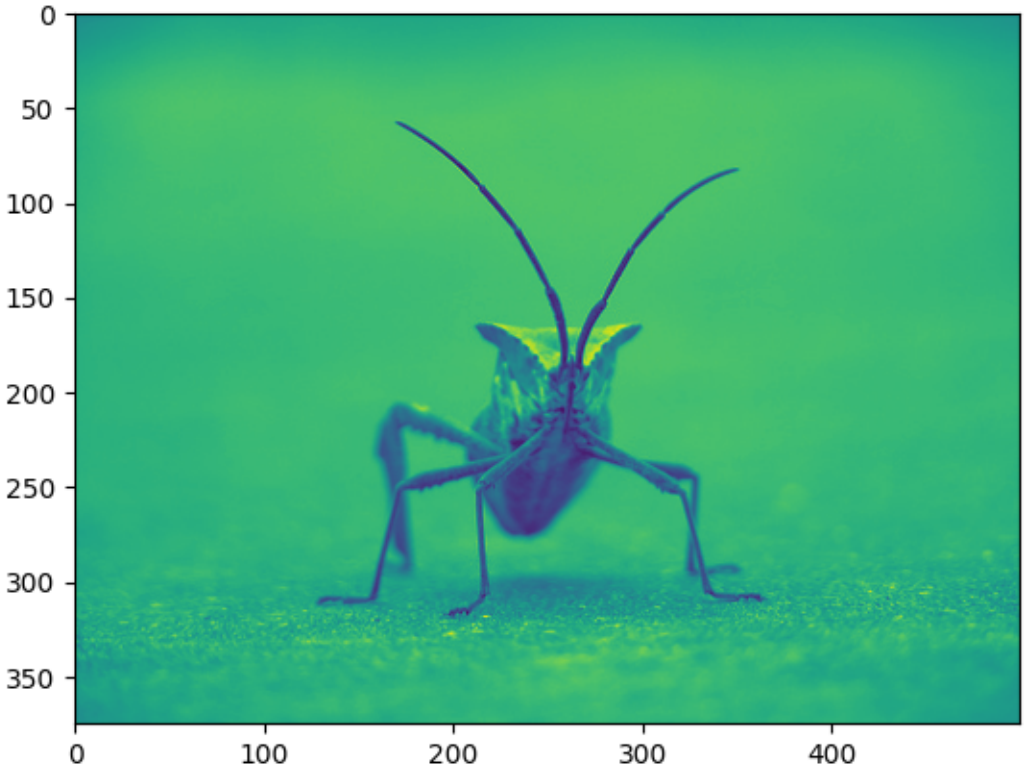

You can also plot any numpy array.

#### Applying pseudocolor schemes to image plots

Pseudocolor can be a useful tool for enhancing contrast and visualizing your data more easily. This is especially useful when making presentations of your data using projectors - their contrast is typically quite poor.

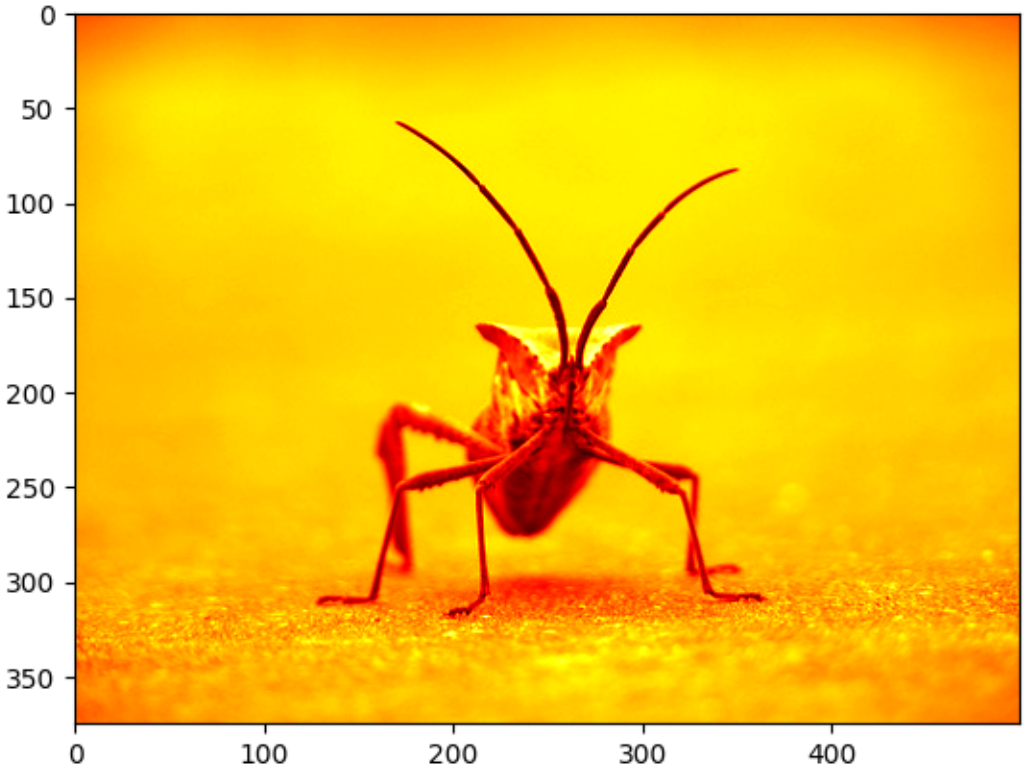
Pseudocolor is only relevant to single-channel, grayscale, luminosity images. We currently have an RGB image. Since R, G, and B are all similar (see for yourself above or in your data), we can just pick one channel of our data using array slicing (you can read more in the Numpy tutorial):

```
lum_img = img[:, :, 0]
plt.imshow(lum_img)
```



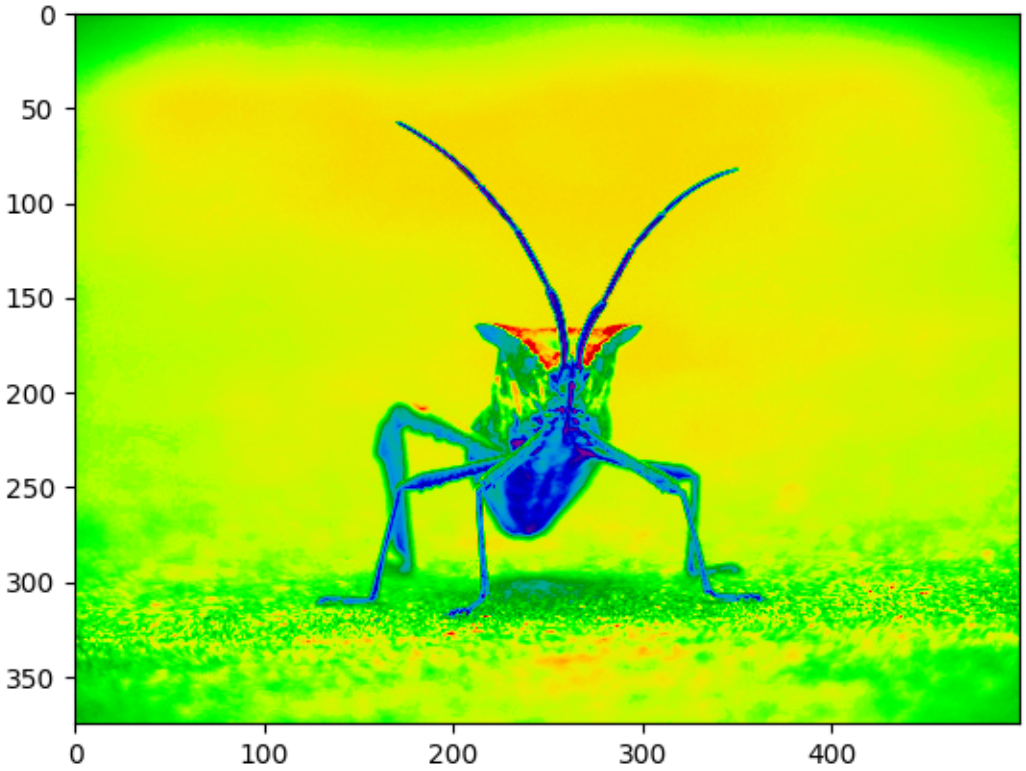

Now, with a luminosity (2D, no color) image, the default colormap (aka lookup table, LUT), is applied. The default is called viridis. There are plenty of others to choose from.

```
plt.imshow(lum_img, cmap="hot")
```




Note that you can also change colormaps on existing plot objects using the *set_cmap()* method:

```
imgplot = plt.imshow(lum_img)
imgplot.set_cmap('nipy_spectral')
```



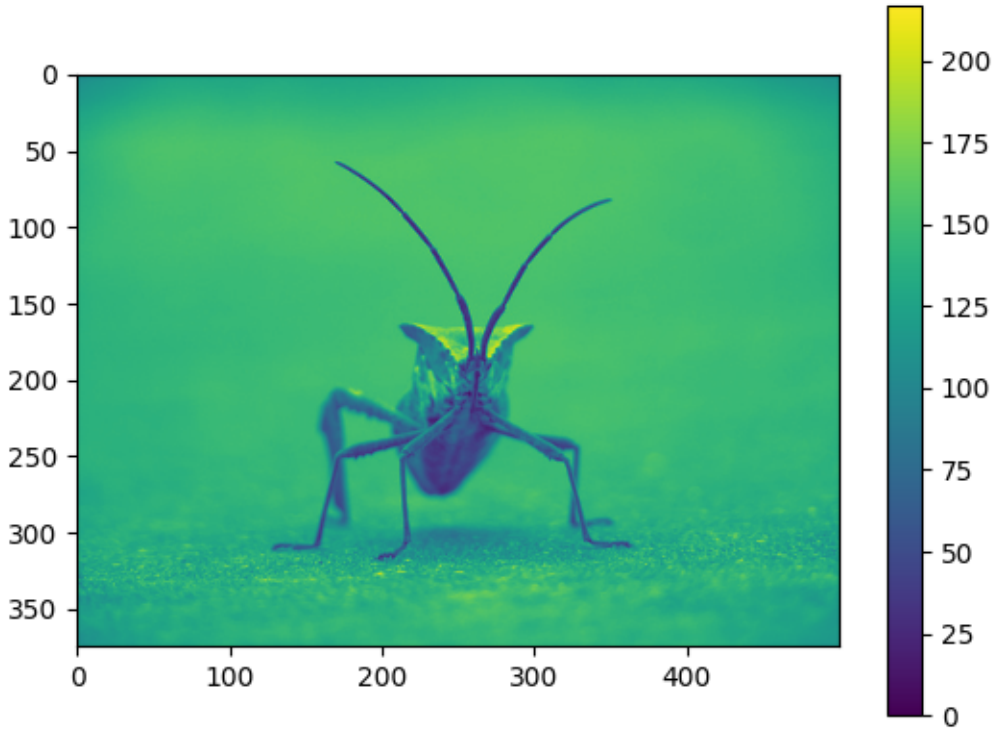

**Note:** However, remember that in the Jupyter Notebook with the inline backend, you can't make changes to plots that have already been rendered. If you create imgplot here in one cell, you cannot call set_cmap() on it in a later cell and expect the earlier plot to change. Make sure that you enter these commands together in one cell. plt commands will not change plots from earlier cells.

There are many other colormap schemes available. See the *list and images of the colormaps*.

### Color scale reference

It's helpful to have an idea of what value a color represents. We can do that by adding a color bar to your figure:

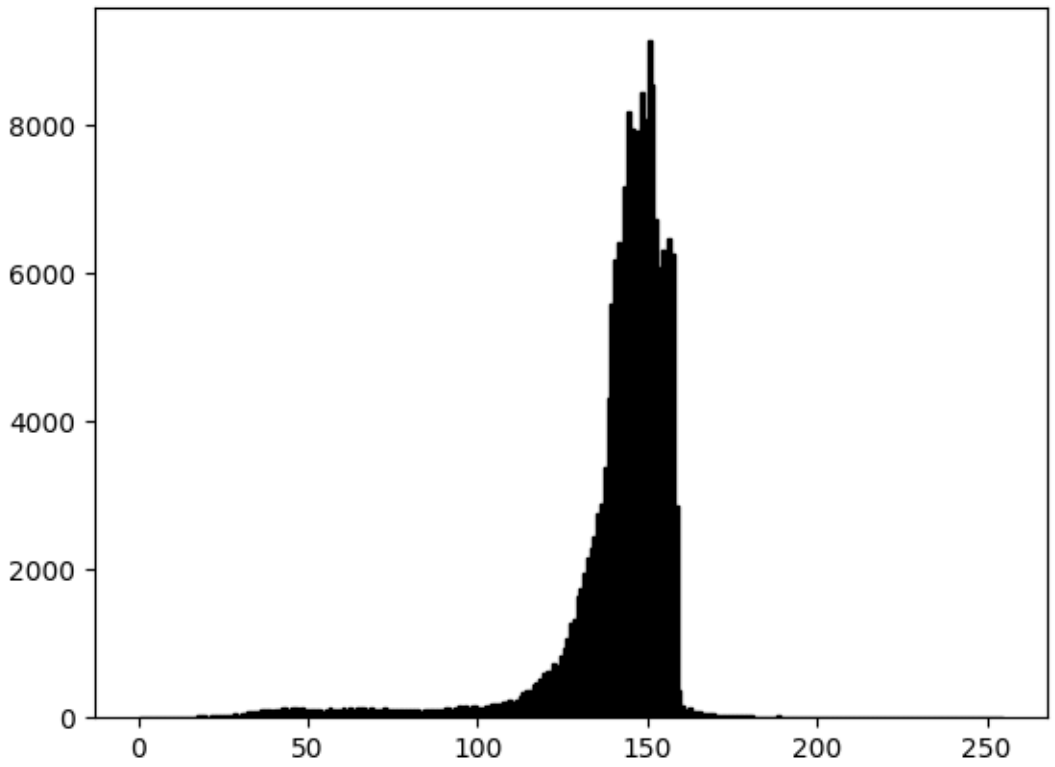
```
imgplot = plt.imshow(lum_img)
plt.colorbar()
```




### Examining a specific data range

Sometimes you want to enhance the contrast in your image, or expand the contrast in a particular region while sacrificing the detail in colors that don't vary much, or don't matter. A good tool to find interesting regions is the histogram. To create a histogram of our image data, we use the *hist()* function.

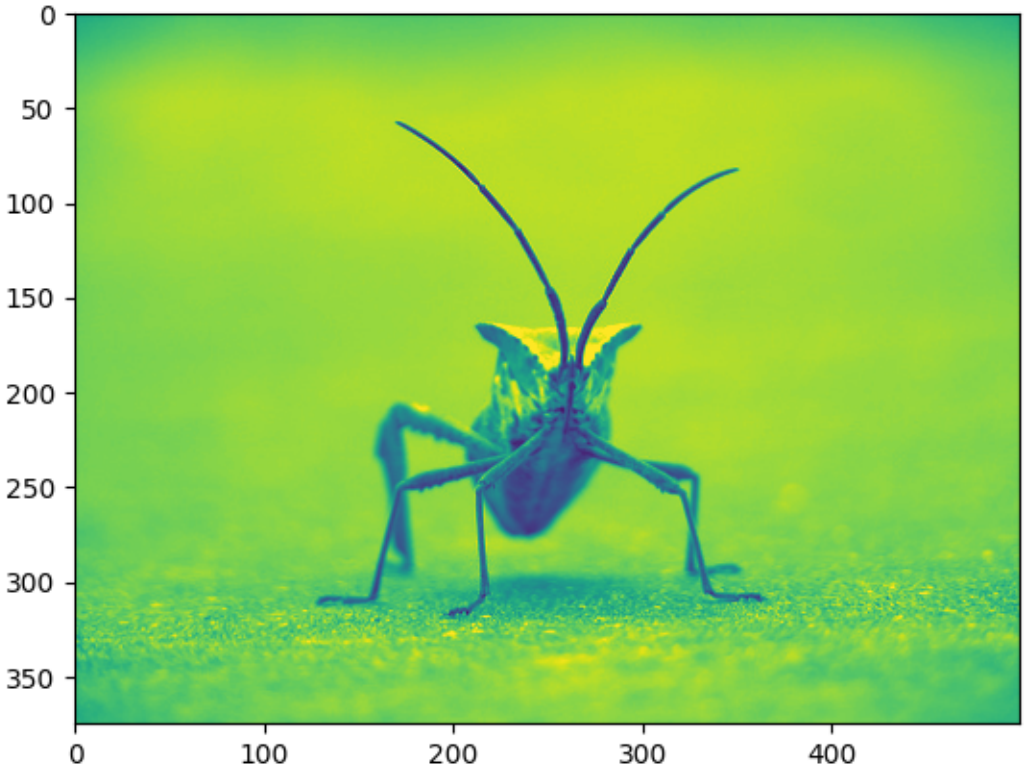
```
plt.hist(lum_img.ravel(), bins=range(256), fc='k', ec='k')
```




Most often, the "interesting" part of the image is around the peak, and you can get extra contrast by clipping the regions above and/or below the peak. In our histogram, it looks like there's not much useful information in the high end (not many white things in the image). Let's adjust the upper limit, so that we effectively "zoom in on" part of the histogram. We do this by setting *clim*, the colormap limits.

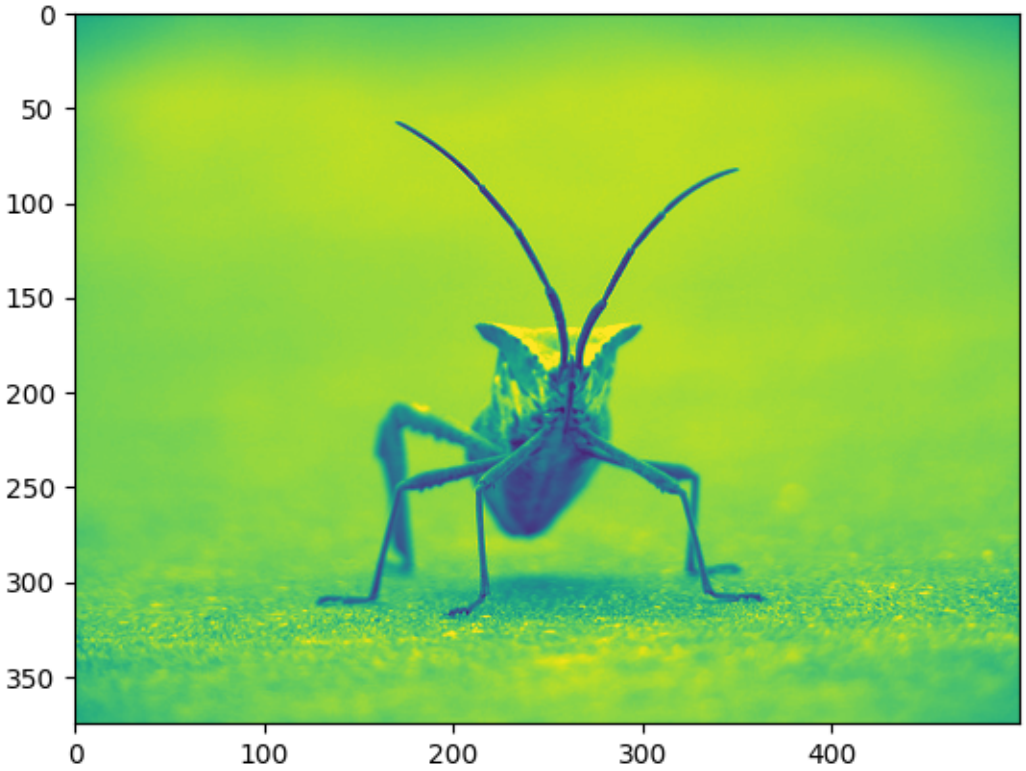
This can be done by passing a *clim* keyword argument in the call to `imshow`.

```
plt.imshow(lum_img, clim=(0, 175))
```




This can also be done by calling the *set_clim()* method of the returned image plot object, but make sure that you do so in the same cell as your plot command when working with the Jupyter Notebook - it will not change plots from earlier cells.

```
imgplot = plt.imshow(lum_img)
imgplot.set_clim(0, 175)
```

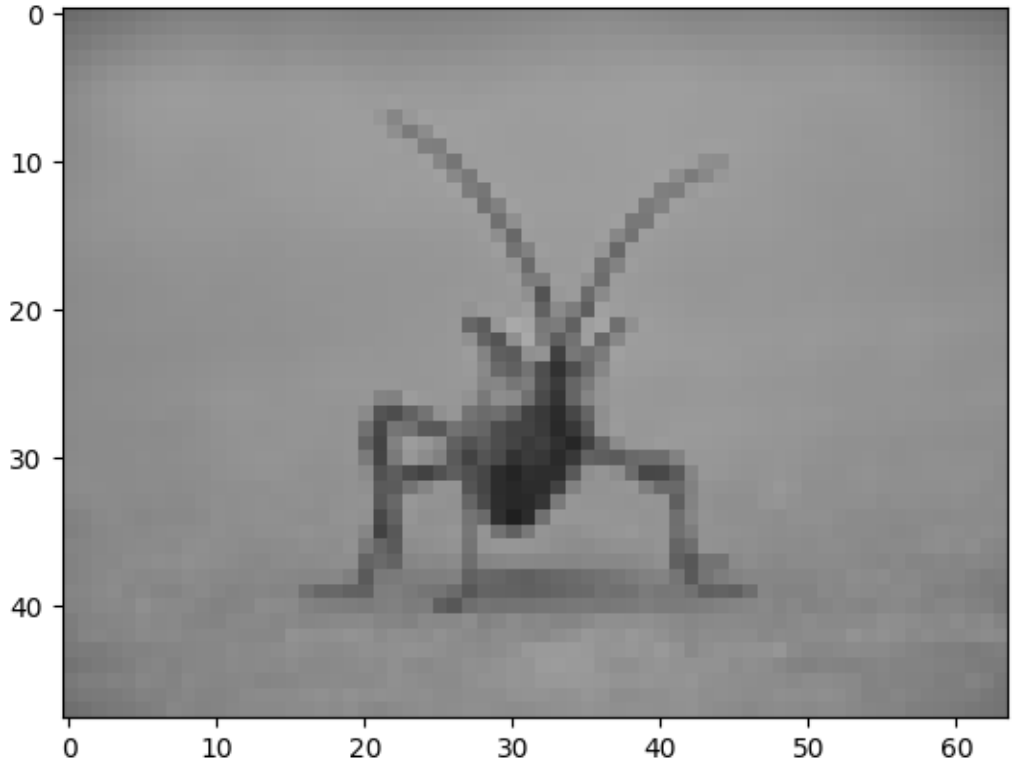



### Array Interpolation schemes

Interpolation calculates what the color or value of a pixel "should" be, according to different mathematical schemes. One common place that this happens is when you resize an image. The number of pixels change, but you want the same information. Since pixels are discrete, there's missing space. Interpolation is how you fill that space. This is why your images sometimes come out looking pixelated when you blow them up. The effect is more pronounced when the difference between the original image and the expanded image is greater. Let's take our image and shrink it. We're effectively discarding pixels, only keeping a select few. Now when we plot it, that data gets blown up to the size on your screen. The old pixels aren't there anymore, and the computer has to draw in pixels to fill that space.

We'll use the Pillow library that we used to load the image also to resize the image.

```
img = Image.open('../../doc/_static/stinkbug.png')
img.thumbnail((64, 64))  # resizes image in-place
imgplot = plt.imshow(img)
```




Here we use the default interpolation ("nearest"), since we did not give *imshow()* any interpolation argument.

Let's try some others. Here's "bilinear":

```
imgplot = plt.imshow(img, interpolation="bilinear")
```

and bicubic:

```
imgplot = plt.imshow(img, interpolation="bicubic")
```

Bicubic interpolation is often used when blowing up photos - people tend to prefer blurry over pixelated.

**Total running time of the script:** (0 minutes 7.193 seconds)

## 4.3 The Lifecycle of a Plot

This tutorial aims to show the beginning, middle, and end of a single visualization using Matplotlib. We'll begin with some raw data and end by saving a figure of a customized visualization. Along the way we try to highlight some neat features and best-practices using Matplotlib.

---

**Note:** This tutorial is based on this excellent blog post by Chris Moffitt. It was transformed into this tutorial by Chris Holdgraf.

---

### 4.3.1 A note on the explicit vs. implicit interfaces

Matplotlib has two interfaces. For an explanation of the trade-offs between the explicit and implicit interfaces see *Matplotlib Application Interfaces (APIs)*.

In the explicit object-oriented (OO) interface we directly utilize instances of `axes.Axes` to build up the visualization in an instance of `figure.Figure`. In the implicit interface, inspired by and modeled on MATLAB, we use a global state-based interface which is encapsulated in the `pyplot` module to plot to the "current Axes". See the *pyplot tutorials* for a more in-depth look at the pyplot interface.

Most of the terms are straightforward but the main thing to remember is that:

- The `Figure` is the final image, and may contain one or more `Axes`.
- **The `Axes` represents an individual plot (not to be confused with**

  `Axis`, which refers to the x-, y-, or z-axis of a plot).

We call methods that do the plotting directly from the Axes, which gives us much more flexibility and power in customizing our plot.

---

**Note:** In general, use the explicit interface over the implicit pyplot interface for plotting.

---

### 4.3.2 Our data

We'll use the data from the post from which this tutorial was derived. It contains sales information for a number of companies.

```
import matplotlib.pyplot as plt
import numpy as np

data = {'Barton LLC': 109438.50,
        'Frami, Hills and Schmidt': 103569.59,
        'Fritsch, Russel and Anderson': 112214.71,
        'Jerde-Hilpert': 112591.43,
        'Keeling LLC': 100934.30,
        'Koepp Ltd': 103660.54,
        'Kulas Inc': 137351.96,
        'Trantow-Barrows': 123381.38,
        'White-Trantow': 135841.99,
        'Will LLC': 104437.60}
group_data = list(data.values())
group_names = list(data.keys())
group_mean = np.mean(group_data)
```

### 4.3.3 Getting started

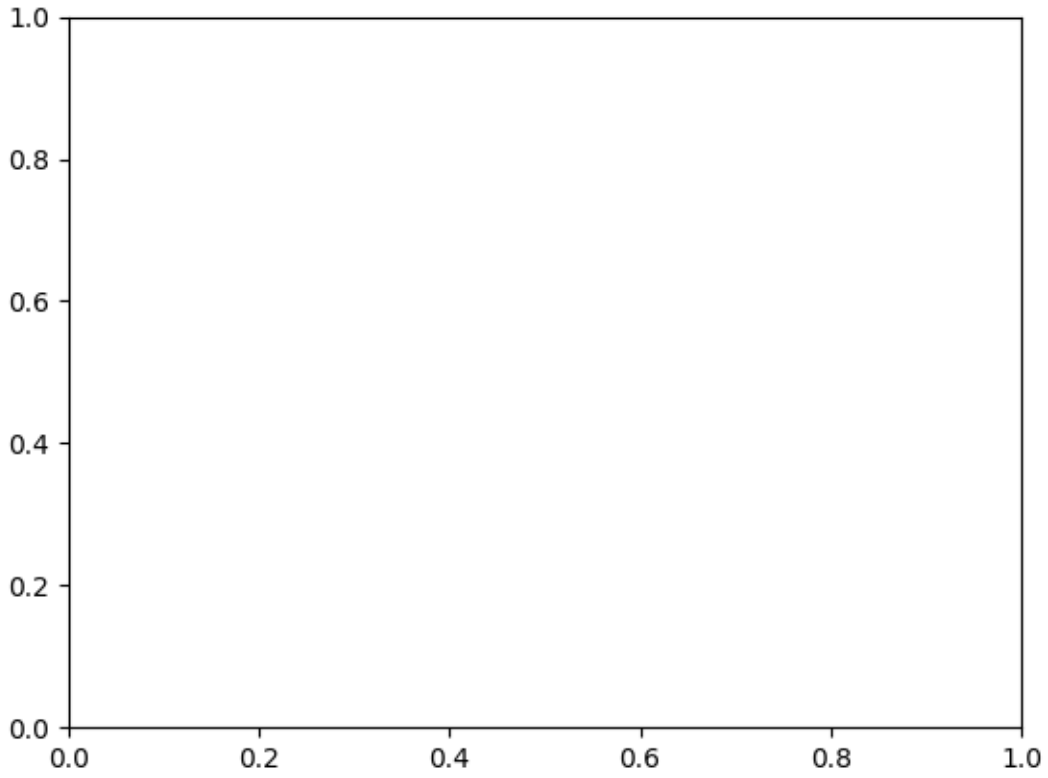
This data is naturally visualized as a barplot, with one bar per group. To do this with the object-oriented approach, we first generate an instance of *figure.Figure* and *axes.Axes*. The Figure is like a canvas, and the Axes is a part of that canvas on which we will make a particular visualization.

---

**Note:** Figures can have multiple Axes on them. For information on how to do this, see the *Tight Layout tutorial*.

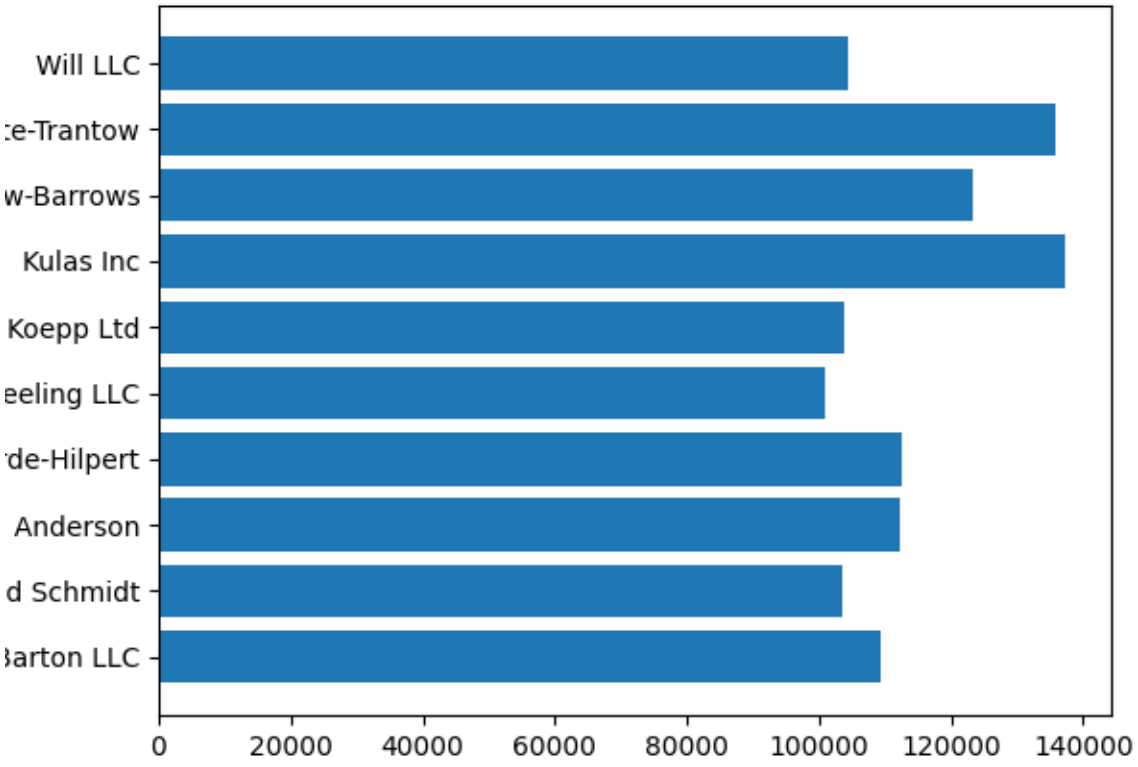
---

```
fig, ax = plt.subplots()
```




Now that we have an Axes instance, we can plot on top of it.

```
fig, ax = plt.subplots()
ax.barh(group_names, group_data)
```




### 4.3.4 Controlling the style

There are many styles available in Matplotlib in order to let you tailor your visualization to your needs. To see a list of styles, we can use *style*.

```
print(plt.style.available)
```

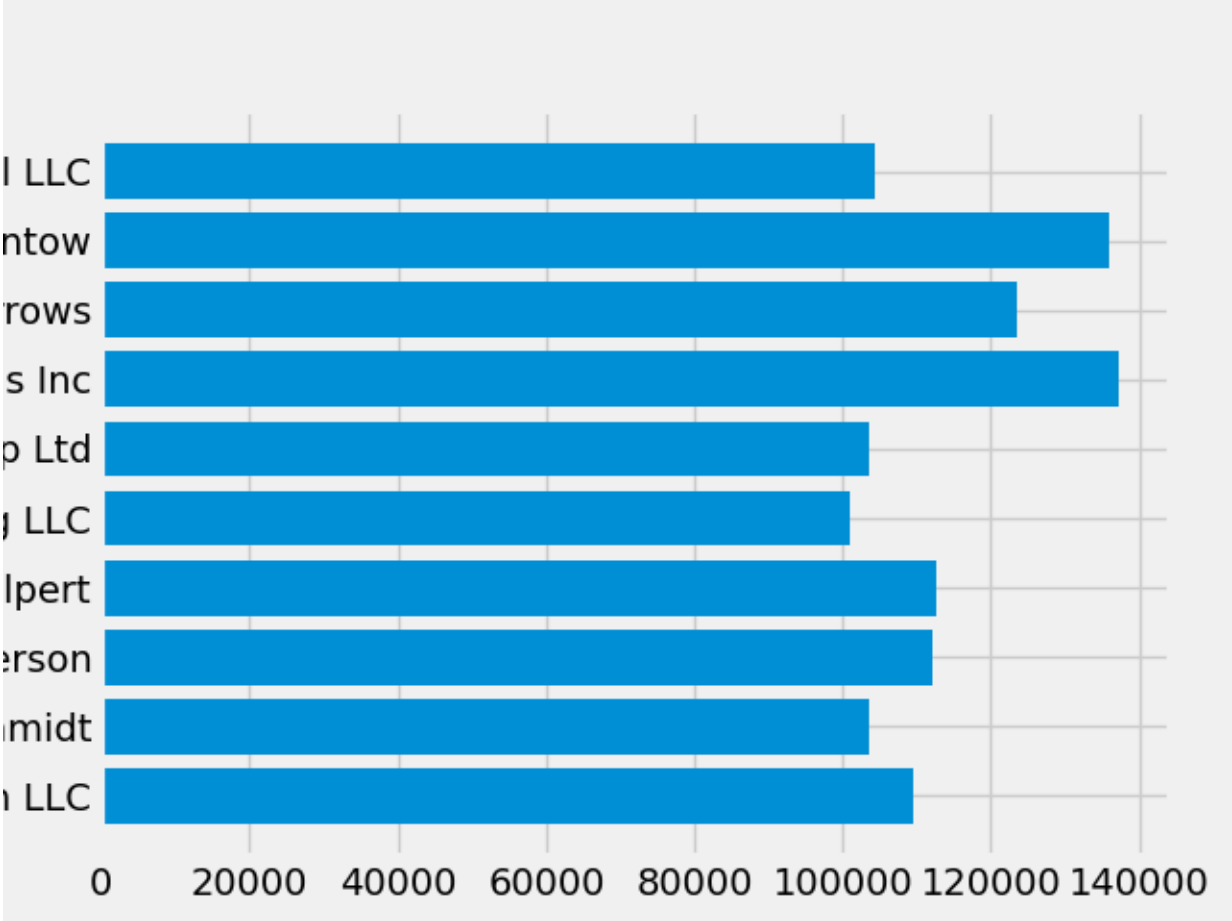
```
['Solarize_Light2', '_classic_test_patch', '_mpl-gallery', '_mpl-gallery-
 ↪nogrid', 'bmh', 'classic', 'dark_background', 'fast', 'fivethirtyeight',
 ↪'ggplot', 'grayscale', 'seaborn-v0_8', 'seaborn-v0_8-bright', 'seaborn-v0_8-
 ↪colorblind', 'seaborn-v0_8-dark', 'seaborn-v0_8-dark-palette', 'seaborn-v0_
 ↪8-darkgrid', 'seaborn-v0_8-deep', 'seaborn-v0_8-muted', 'seaborn-v0_8-
 ↪notebook', 'seaborn-v0_8-paper', 'seaborn-v0_8-pastel', 'seaborn-v0_8-poster
 ↪', 'seaborn-v0_8-talk', 'seaborn-v0_8-ticks', 'seaborn-v0_8-white',
 ↪'seaborn-v0_8-whitegrid', 'tableau-colorblind10']
```

You can activate a style with the following:

```
plt.style.use('fivethirtyeight')
```

Now let's remake the above plot to see how it looks:

```
fig, ax = plt.subplots()
ax.barh(group_names, group_data)
```

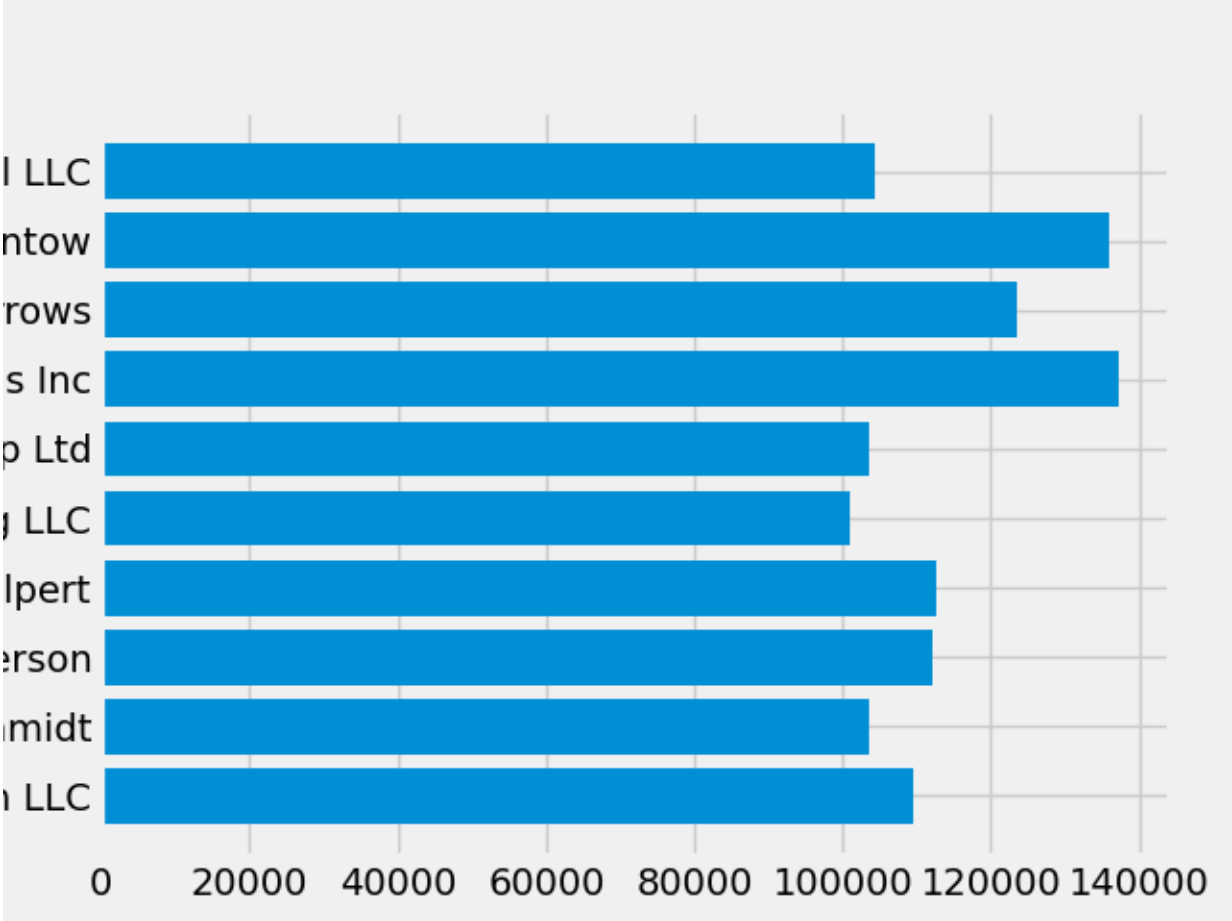



The style controls many things, such as color, linewidths, backgrounds, etc.

### 4.3.5 Customizing the plot

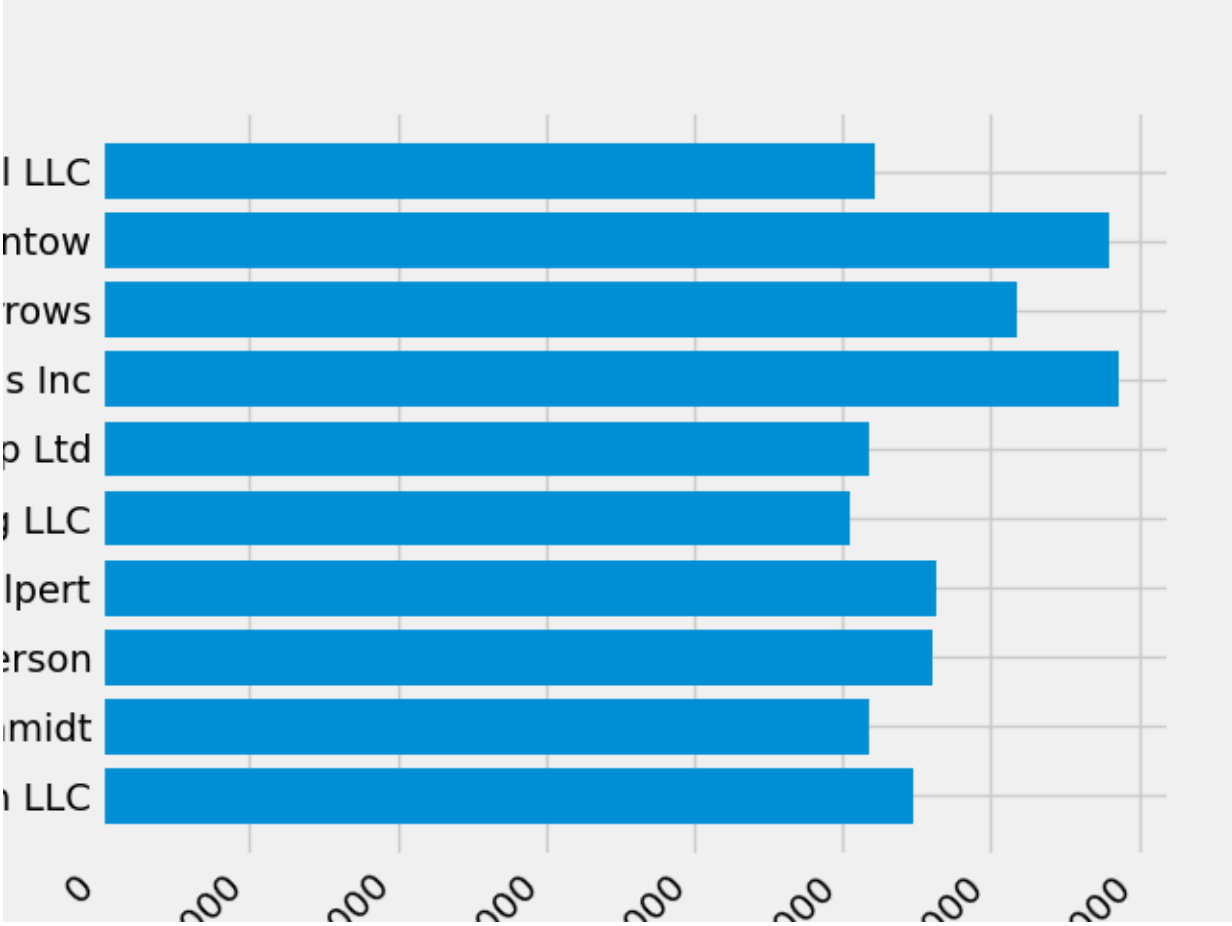
Now we've got a plot with the general look that we want, so let's fine-tune it so that it's ready for print. First let's rotate the labels on the x-axis so that they show up more clearly. We can gain access to these labels with the *axes.Axes.get_xticklabels()* method:

```
fig, ax = plt.subplots()
ax.barh(group_names, group_data)
labels = ax.get_xticklabels()
```



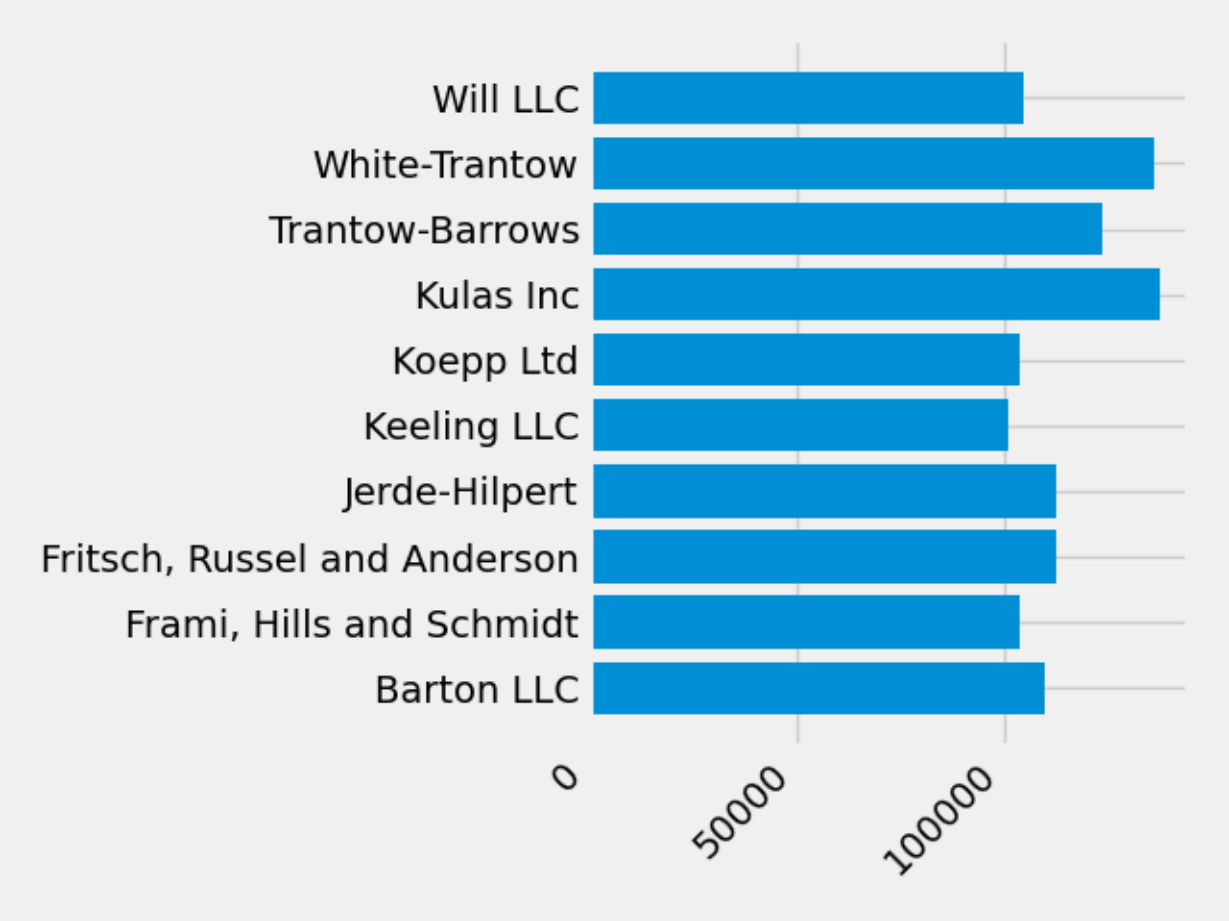

If we'd like to set the property of many items at once, it's useful to use the *pyplot.setp()* function. This will take a list (or many lists) of Matplotlib objects, and attempt to set some style element of each one.

```
fig, ax = plt.subplots()
ax.barh(group_names, group_data)
labels = ax.get_xticklabels()
plt.setp(labels, rotation=45, horizontalalignment='right')
```



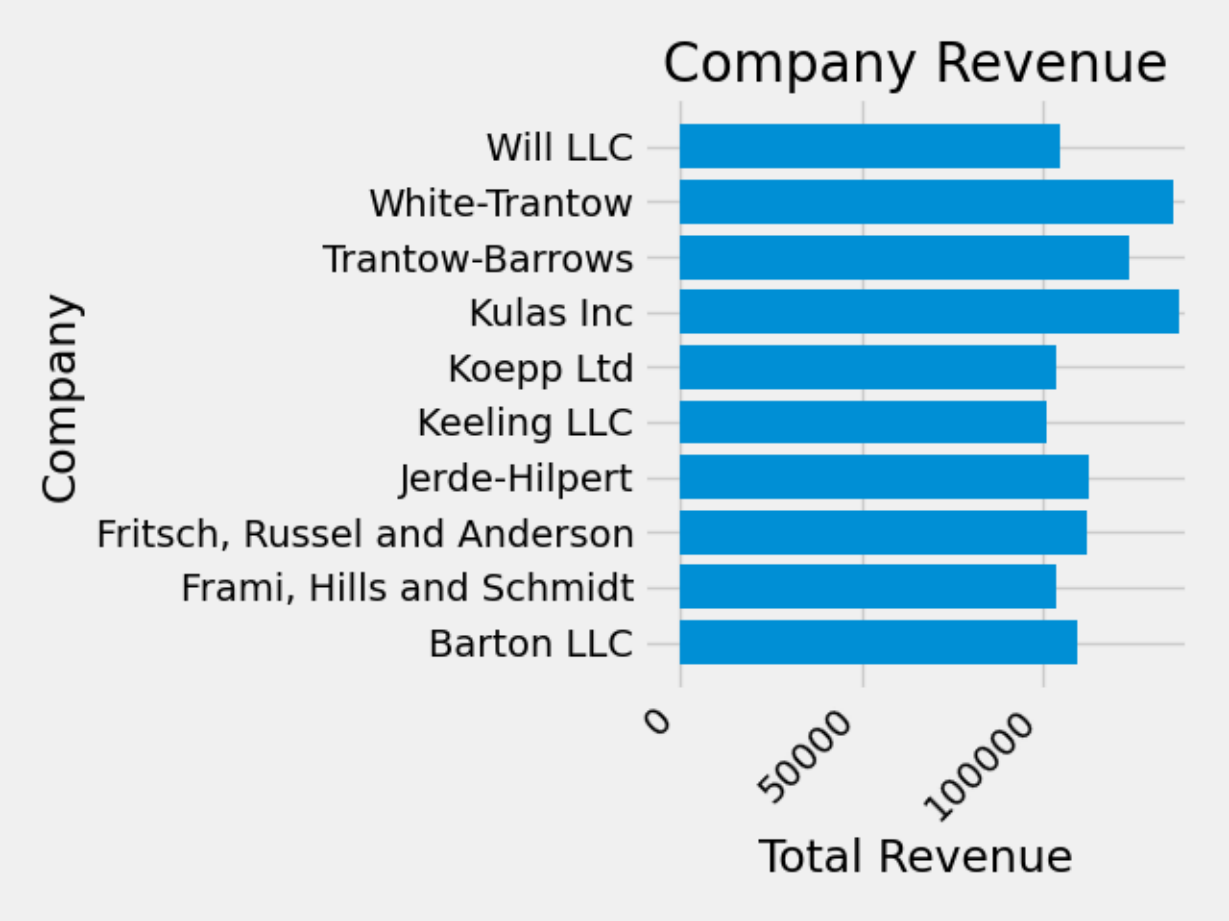

It looks like this cut off some of the labels on the bottom. We can tell Matplotlib to automatically make room for elements in the figures that we create. To do this we set the `autolayout` value of our rcParams. For more information on controlling the style, layout, and other features of plots with rcParams, see *Customizing Matplotlib with style sheets and rcParams*.

```
plt.rcParams.update({'figure.autolayout': True})

fig, ax = plt.subplots()
ax.barh(group_names, group_data)
labels = ax.get_xticklabels()
plt.setp(labels, rotation=45, horizontalalignment='right')
```




Next, we add labels to the plot. To do this with the OO interface, we can use the *Artist.set()* method to set properties of this Axes object.

```
fig, ax = plt.subplots()
ax.barh(group_names, group_data)
labels = ax.get_xticklabels()
plt.setp(labels, rotation=45, horizontalalignment='right')
ax.set(xlim=[-10000, 140000], xlabel='Total Revenue', ylabel='Company',
       title='Company Revenue')
```



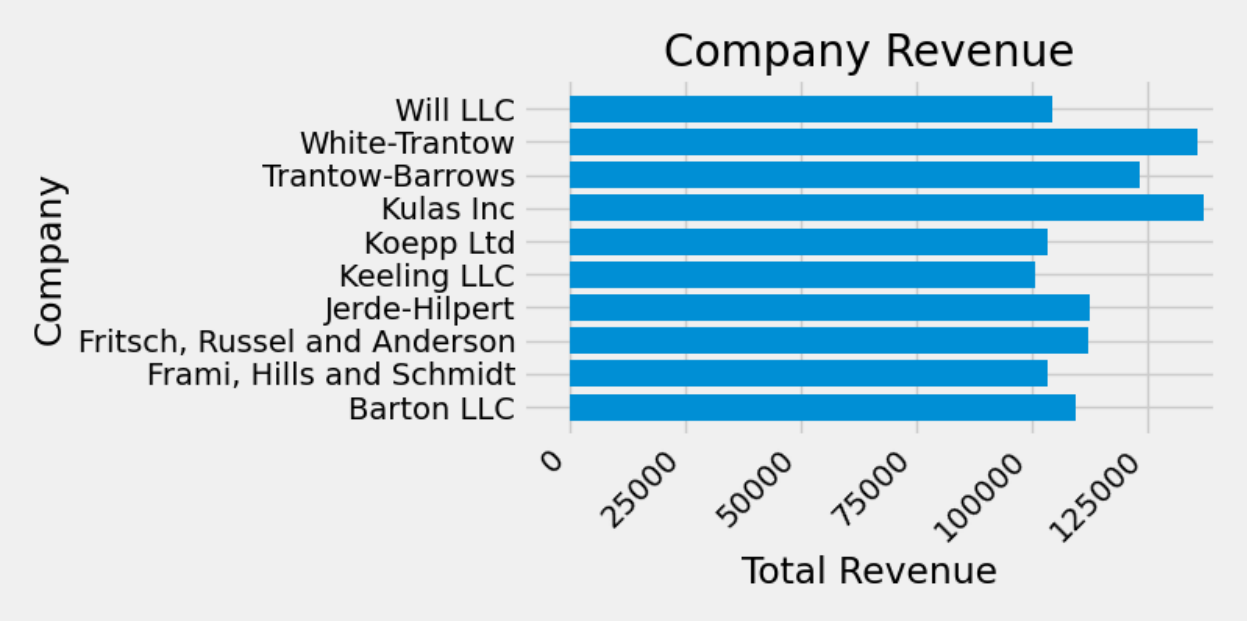

We can also adjust the size of this plot using the *pyplot.subplots()* function. We can do this with the *figsize* keyword argument.

---

**Note:** While indexing in NumPy follows the form (row, column), the *figsize* keyword argument follows the form (width, height). This follows conventions in visualization, which unfortunately are different from those of linear algebra.

---

```
fig, ax = plt.subplots(figsize=(8, 4))
ax.barh(group_names, group_data)
labels = ax.get_xticklabels()
plt.setp(labels, rotation=45, horizontalalignment='right')
ax.set(xlim=[-10000, 140000], xlabel='Total Revenue', ylabel='Company',
       title='Company Revenue')
```



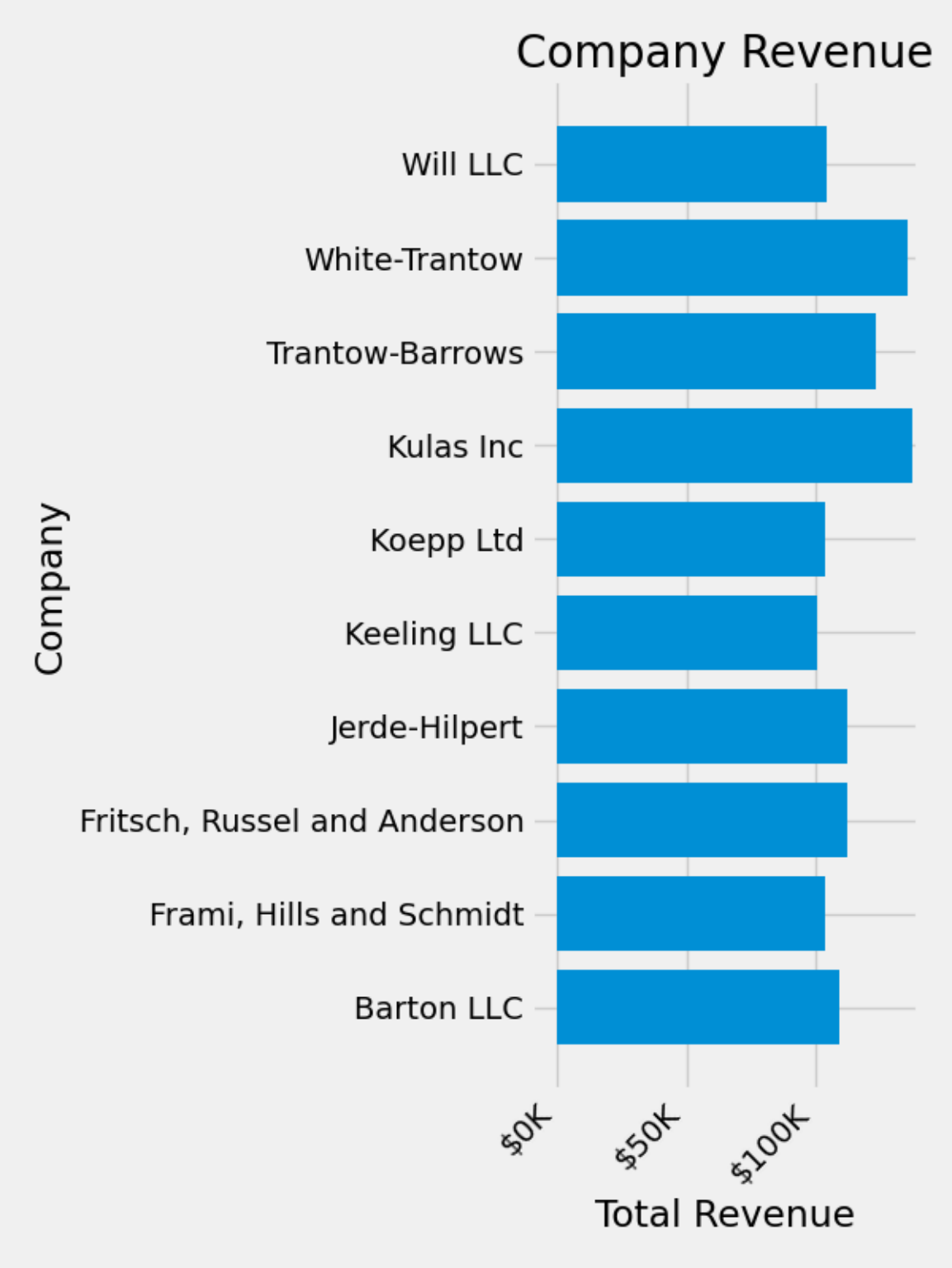

For labels, we can specify custom formatting guidelines in the form of functions. Below we define a function that takes an integer as input, and returns a string as an output. When used with *Axis.set_major_formatter* or *Axis.set_minor_formatter*, they will automatically create and use a *ticker.FuncFormatter* class.

For this function, the `x` argument is the original tick label and `pos` is the tick position. We will only use `x` here but both arguments are needed.

```
def currency(x, pos):
    """The two arguments are the value and tick position"""
    if x >= 1e6:
        s = f'${x*1e-6:1.1f}M'
    else:
        s = f'${x*1e-3:1.0f}K'
    return s
```

We can then apply this function to the labels on our plot. To do this, we use the `xaxis` attribute of our Axes. This lets you perform actions on a specific axis on our plot.

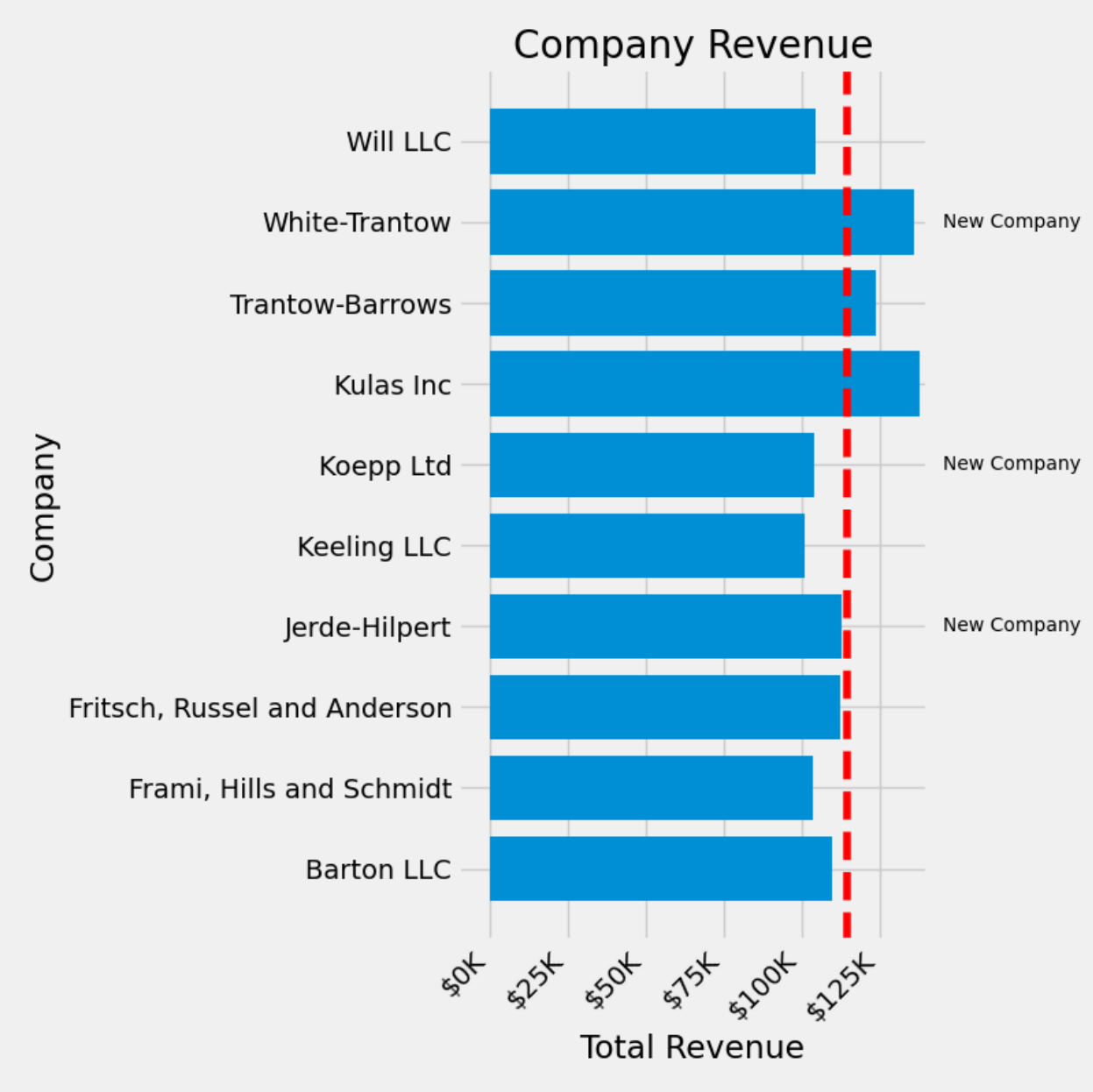
```
fig, ax = plt.subplots(figsize=(6, 8))
ax.barh(group_names, group_data)
labels = ax.get_xticklabels()
plt.setp(labels, rotation=45, horizontalalignment='right')

ax.set(xlim=[-10000, 140000], xlabel='Total Revenue', ylabel='Company',
       title='Company Revenue')
ax.xaxis.set_major_formatter(currency)
```


Company Revenue
Will LLC
White-Trantow
Trantow-Barrows
Kulas Inc
Koepp Ltd
Keeling LLC
Jerde-Hilpert
Fritsch, Russel and Anderson
Frami, Hills and Schmidt
Barton LLC
Company
$0K
$50K
$100K
Total Revenue

### 4.3.6 Combining multiple visualizations

It is possible to draw multiple plot elements on the same instance of *axes.Axes*. To do this we simply need to call another one of the plot methods on that Axes object.

```
fig, ax = plt.subplots(figsize=(8, 8))
ax.barh(group_names, group_data)
labels = ax.get_xticklabels()
plt.setp(labels, rotation=45, horizontalalignment='right')

# Add a vertical line, here we set the style in the function call
ax.axvline(group_mean, ls='--', color='r')

# Annotate new companies
for group in [3, 5, 8]:
    ax.text(145000, group, "New Company", fontsize=10,
            verticalalignment="center")

# Now we move our title up since it's getting a little cramped
ax.title.set(y=1.05)

ax.set(xlim=[-10000, 140000], xlabel='Total Revenue', ylabel='Company',
       title='Company Revenue')
ax.xaxis.set_major_formatter(currency)
ax.set_xticks([0, 25e3, 50e3, 75e3, 100e3, 125e3])
fig.subplots_adjust(right=.1)

plt.show()
```




### 4.3.7 Saving our plot

Now that we're happy with the outcome of our plot, we want to save it to disk. There are many file formats we can save to in Matplotlib. To see a list of available options, use:

```
print(fig.canvas.get_supported_filetypes())
```

```
{'eps': 'Encapsulated Postscript', 'jpg': 'Joint Photographic Experts Group',
 ↪'jpeg': 'Joint Photographic Experts Group', 'pdf': 'Portable Document Format
 ↪', 'pgf': 'PGF code for LaTeX', 'png': 'Portable Network Graphics', 'ps':
 ↪'Postscript', 'raw': 'Raw RGBA bitmap', 'rgba': 'Raw RGBA bitmap', 'svg':
```

(continues on next page)

(continued from previous page)

```
↪'Scalable Vector Graphics', 'svgz': 'Scalable Vector Graphics', 'tif':
↪'Tagged Image File Format', 'tiff': 'Tagged Image File Format', 'webp':
↪'WebP Image Format'}
```

We can then use the *figure.Figure.savefig()* in order to save the figure to disk. Note that there are several useful flags we show below:

- `transparent=True` makes the background of the saved figure transparent if the format supports it.
- `dpi=80` controls the resolution (dots per square inch) of the output.
- `bbox_inches="tight"` fits the bounds of the figure to our plot.

```
# Uncomment this line to save the figure.
# fig.savefig('sales.png', transparent=False, dpi=80, bbox_inches="tight")
```

**Total running time of the script:** (0 minutes 2.696 seconds)

## 4.4 Artist tutorial

Using Artist objects to render on the canvas.

There are three layers to the Matplotlib API.

- the `matplotlib.backend_bases.FigureCanvas` is the area onto which the figure is drawn
- the `matplotlib.backend_bases.Renderer` is the object which knows how to draw on the `FigureCanvas`
- and the *matplotlib.artist.Artist* is the object that knows how to use a renderer to paint onto the canvas.

The `FigureCanvas` and `Renderer` handle all the details of talking to user interface toolkits like wxPython or drawing languages like PostScript®, and the `Artist` handles all the high level constructs like representing and laying out the figure, text, and lines. The typical user will spend 95% of their time working with the `Artists`.

There are two types of `Artists`: primitives and containers. The primitives represent the standard graphical objects we want to paint onto our canvas: *Line2D*, *Rectangle*, *Text*, *AxesImage*, etc., and the containers are places to put them (*Axis*, *Axes* and *Figure*). The standard use is to create a *Figure* instance, use the `Figure` to create one or more *Axes* instances, and use the `Axes` instance helper methods to create the primitives. In the example below, we create a `Figure` instance using *matplotlib.pyplot.figure()*, which is a convenience method for instantiating `Figure` instances and connecting them with your user interface or drawing toolkit `FigureCanvas`. As we will discuss below, this is not necessary -- you can work directly with PostScript, PDF Gtk+, or wxPython `FigureCanvas` instances, instantiate your `Figures` directly and connect them yourselves -- but since we are focusing here on the `Artist` API we'll let *pyplot* handle some of those details for us:

```
import matplotlib.pyplot as plt
fig = plt.figure()
ax = fig.add_subplot(2, 1, 1) # two rows, one column, first plot
```

The *Axes* is probably the most important class in the Matplotlib API, and the one you will be working with most of the time. This is because the `Axes` is the plotting area into which most of the objects go, and the `Axes` has many special helper methods (*plot()*, *text()*, *hist()*, *imshow()*) to create the most common graphics primitives (*Line2D*, *Text*, *Rectangle*, *AxesImage*, respectively). These helper methods will take your data (e.g., `numpy` arrays and strings) and create primitive `Artist` instances as needed (e.g., `Line2D`), add them to the relevant containers, and draw them when requested. If you want to create an `Axes` at an arbitrary location, simply use the *add_axes()* method which takes a list of `[left, bottom, width, height]` values in 0-1 relative figure coordinates:

```
fig2 = plt.figure()
ax2 = fig2.add_axes([0.15, 0.1, 0.7, 0.3])
```

Continuing with our example:

```
import numpy as np
t = np.arange(0.0, 1.0, 0.01)
s = np.sin(2*np.pi*t)
line, = ax.plot(t, s, color='blue', lw=2)
```

In this example, `ax` is the `Axes` instance created by the `fig.add_subplot` call above and when you call `ax.plot`, it creates a `Line2D` instance and adds it to the `Axes`. In the interactive IPython session below, you can see that the `Axes.lines` list is length one and contains the same line that was returned by the `line, = ax.plot...` call:

```
In [101]: ax.lines[0]
Out[101]: <matplotlib.lines.Line2D at 0x19a95710>

In [102]: line
Out[102]: <matplotlib.lines.Line2D at 0x19a95710>
```

If you make subsequent calls to `ax.plot` (and the hold state is "on" which is the default) then additional lines will be added to the list. You can remove a line later by calling its `remove` method:

```
line = ax.lines[0]
line.remove()
```

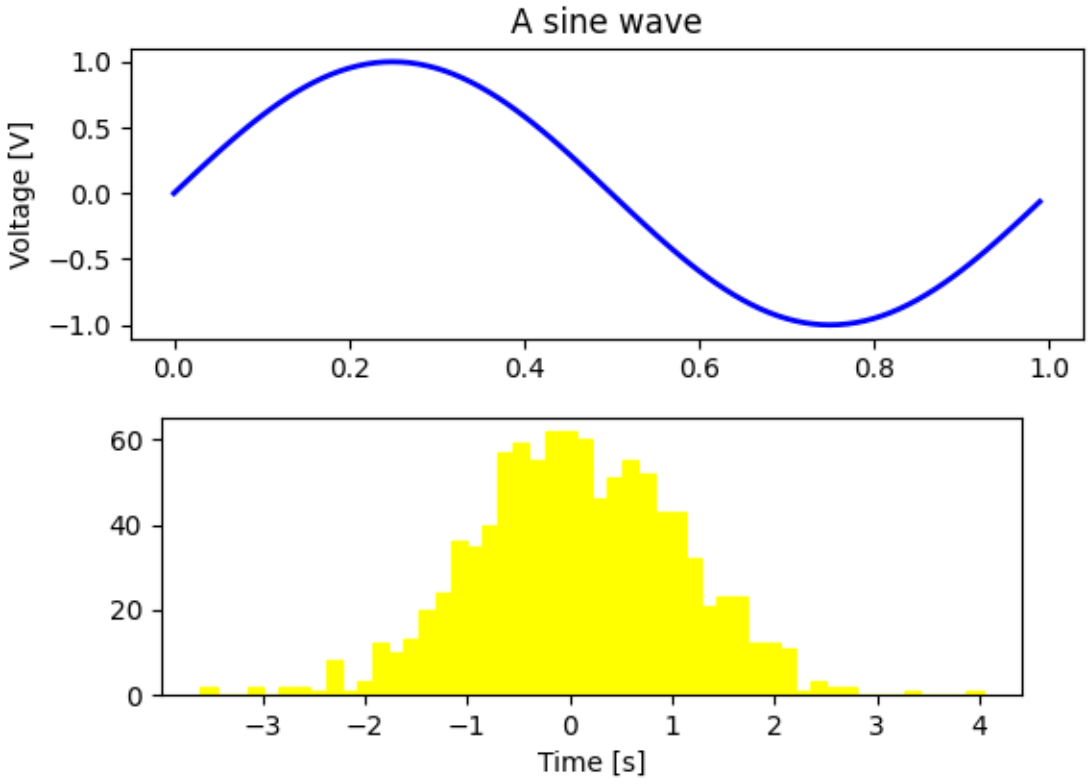
The Axes also has helper methods to configure and decorate the x-axis and y-axis tick, tick labels and axis labels:

```
xtext = ax.set_xlabel('my xdata')  # returns a Text instance
ytext = ax.set_ylabel('my ydata')
```

When you call *ax.set_xlabel*, it passes the information on the *Text* instance of the *XAxis*. Each `Axes` instance contains an *XAxis* and a *YAxis* instance, which handle the layout and drawing of the ticks, tick labels and axis labels.

Try creating the figure below.

```
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure()
fig.subplots_adjust(top=0.8)
ax1 = fig.add_subplot(211)
ax1.set_ylabel('Voltage [V]')
ax1.set_title('A sine wave')

t = np.arange(0.0, 1.0, 0.01)
s = np.sin(2*np.pi*t)
line, = ax1.plot(t, s, color='blue', lw=2)

# Fixing random state for reproducibility
np.random.seed(19680801)

ax2 = fig.add_axes([0.15, 0.1, 0.7, 0.3])
n, bins, patches = ax2.hist(np.random.randn(1000), 50,
                            facecolor='yellow', edgecolor='yellow')
ax2.set_xlabel('Time [s]')

plt.show()
```



### 4.4.1 Customizing your objects

Every element in the figure is represented by a Matplotlib *Artist*, and each has an extensive list of properties to configure its appearance. The figure itself contains a *Rectangle* exactly the size of the figure, which you can use to set the background color and transparency of the figures. Likewise, each *Axes* bounding box (the standard white box with black edges in the typical Matplotlib plot, has a `Rectangle` instance that determines the color, transparency, and other properties of the Axes. These instances are stored as member variables `Figure.patch` and `Axes.patch` ("Patch" is a name inherited from MATLAB, and is a 2D "patch" of color on the figure, e.g., rectangles, circles and polygons). Every Matplotlib `Artist` has the following properties

| Property | Description |
|---|---|
| alpha | The transparency - a scalar from 0-1 |
| animated | A boolean that is used to facilitate animated drawing |
| axes | The Axes that the Artist lives in, possibly None |
| clip_box | The bounding box that clips the Artist |
| clip_on | Whether clipping is enabled |
| clip_path | The path the artist is clipped to |
| contains | A picking function to test whether the artist contains the pick point |
| figure | The figure instance the artist lives in, possibly None |
| label | A text label (e.g., for auto-labeling) |
| picker | A python object that controls object picking |
| transform | The transformation |
| visible | A boolean whether the artist should be drawn |
| zorder | A number which determines the drawing order |
| rasterized | Boolean; Turns vectors into raster graphics (for compression & EPS transparency) |

Each of the properties is accessed with an old-fashioned setter or getter (yes we know this irritates Pythonistas and we plan to support direct access via properties or traits but it hasn't been done yet). For example, to multiply the current alpha by a half:

```
a = o.get_alpha()
o.set_alpha(0.5*a)
```

If you want to set a number of properties at once, you can also use the `set` method with keyword arguments. For example:

```
o.set(alpha=0.5, zorder=2)
```

If you are working interactively at the python shell, a handy way to inspect the `Artist` properties is to use the *matplotlib.artist.getp()* function (simply *getp()* in pyplot), which lists the properties and their values. This works for classes derived from `Artist` as well, e.g., `Figure` and `Rectangle`. Here are the `Figure` rectangle properties mentioned above:

```
In [149]: matplotlib.artist.getp(fig.patch)
  agg_filter = None
  alpha = None
```

(continues on next page)

(continued from previous page)

```
  animated = False
  antialiased or aa = False
  bbox = Bbox(x0=0.0, y0=0.0, x1=1.0, y1=1.0)
  capstyle = butt
  children = []
  clip_box = None
  clip_on = True
  clip_path = None
  contains = None
  data_transform = BboxTransformTo(     TransformedBbox(         Bbox...
  edgecolor or ec = (1.0, 1.0, 1.0, 1.0)
  extents = Bbox(x0=0.0, y0=0.0, x1=640.0, y1=480.0)
  facecolor or fc = (1.0, 1.0, 1.0, 1.0)
  figure = Figure(640x480)
  fill = True
  gid = None
  hatch = None
  height = 1
  in_layout = False
  joinstyle = miter
  label =
  linestyle or ls = solid
  linewidth or lw = 0.0
  patch_transform = CompositeGenericTransform(     BboxTransformTo(   ...
  path = Path(array([[0., 0.],        [1., 0.],        [1.,...
  path_effects = []
  picker = None
  rasterized = None
  sketch_params = None
  snap = None
  transform = CompositeGenericTransform(     CompositeGenericTra...
  transformed_clip_path_and_affine = (None, None)
  url = None
  verts = [[  0.   0.]  [640.   0.]  [640. 480.]  [  0. 480....
  visible = True
  width = 1
  window_extent = Bbox(x0=0.0, y0=0.0, x1=640.0, y1=480.0)
  x = 0
  xy = (0, 0)
  y = 0
  zorder = 1
```

The docstrings for all of the classes also contain the `Artist` properties, so you can consult the interactive "help" or the *matplotlib.artist* for a listing of properties for a given object.

### 4.4.2 Object containers

Now that we know how to inspect and set the properties of a given object we want to configure, we need to know how to get at that object. As mentioned in the introduction, there are two kinds of objects: primitives and containers. The primitives are usually the things you want to configure (the font of a *Text* instance, the width of a *Line2D*) although the containers also have some properties as well -- for example the *Axes Artist* is a container that contains many of the primitives in your plot, but it also has properties like the `xscale` to control whether the xaxis is 'linear' or 'log'. In this section we'll review where the various container objects store the `Artists` that you want to get at.

#### Figure container

The top level container `Artist` is the *matplotlib.figure.Figure*, and it contains everything in the figure. The background of the figure is a *Rectangle* which is stored in `Figure.patch`. As you add subplots (*add_subplot()*) and Axes (*add_axes()*) to the figure these will be appended to the *Figure.axes*. These are also returned by the methods that create them:

```
In [156]: fig = plt.figure()

In [157]: ax1 = fig.add_subplot(211)

In [158]: ax2 = fig.add_axes([0.1, 0.1, 0.7, 0.3])

In [159]: ax1
Out[159]: <Axes:>

In [160]: print(fig.axes)
[<Axes:>, <matplotlib.axes._axes.Axes object at 0x7f0768702be0>]
```

Because the figure maintains the concept of the "current Axes" (see *Figure.gca* and *Figure.sca*) to support the pylab/pyplot state machine, you should not insert or remove Axes directly from the Axes list, but rather use the *add_subplot()* and *add_axes()* methods to insert, and the *Axes.remove* method to delete. You are free however, to iterate over the list of Axes or index into it to get access to `Axes` instances you want to customize. Here is an example which turns all the Axes grids on:

```
for ax in fig.axes:
    ax.grid(True)
```

The figure also has its own `images`, `lines`, `patches` and `text` attributes, which you can use to add primitives directly. When doing so, the default coordinate system for the `Figure` will simply be in pixels (which is not usually what you want). If you instead use Figure-level methods to add Artists (e.g., using *Figure.text* to add text), then the default coordinate system will be "figure coordinates" where (0, 0) is the bottom-left of the figure and (1, 1) is the top-right of the figure.

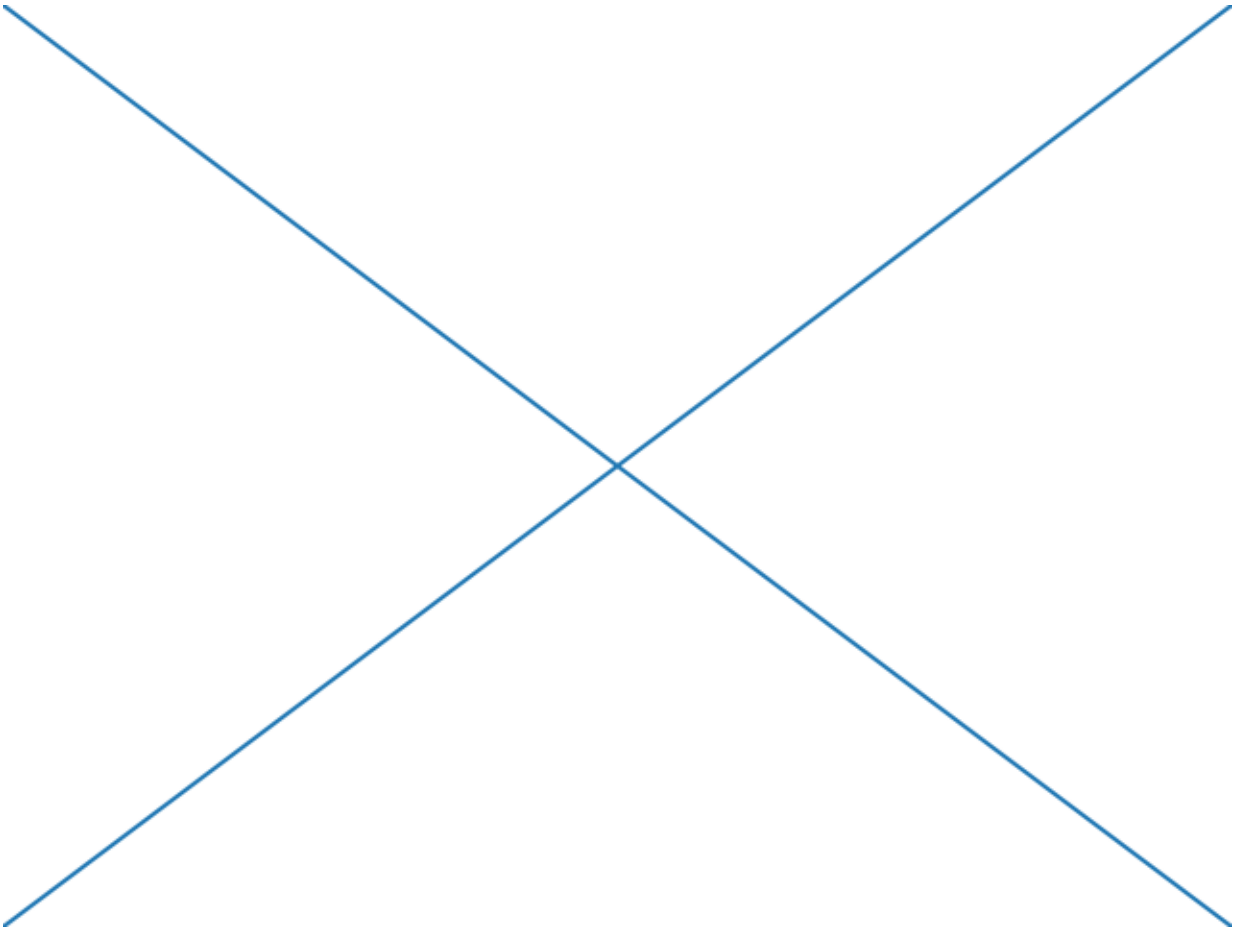
As with all `Artists`, you can control this coordinate system by setting the transform property. You can explicitly use "figure coordinates" by setting the `Artist` transform to `fig.transFigure`:

```
import matplotlib.lines as lines
```

(continues on next page)

(continued from previous page)

```
fig = plt.figure()

l1 = lines.Line2D([0, 1], [0, 1], transform=fig.transFigure, figure=fig)
l2 = lines.Line2D([0, 1], [1, 0], transform=fig.transFigure, figure=fig)
fig.lines.extend([l1, l2])

plt.show()
```



Here is a summary of the Artists the Figure contains

| Figure attribute | Description |
|---|---|
| axes | A list of *Axes* instances |
| patch | The *Rectangle* background |
| images | A list of *FigureImage* patches - useful for raw pixel display |
| legends | A list of Figure *Legend* instances (different from `Axes.get_legend()`) |
| lines | A list of Figure *Line2D* instances (rarely used, see `Axes.lines`) |
| patches | A list of Figure *Patch*s (rarely used, see `Axes.patches`) |
| texts | A list Figure *Text* instances |

### Axes container

The *matplotlib.axes.Axes* is the center of the Matplotlib universe -- it contains the vast majority of all the `Artists` used in a figure with many helper methods to create and add these `Artists` to itself, as well as helper methods to access and customize the `Artists` it contains. Like the *Figure*, it contains a *Patch* `patch` which is a *Rectangle* for Cartesian coordinates and a *Circle* for polar coordinates; this patch determines the shape, background and border of the plotting region:

```
ax = fig.add_subplot()
rect = ax.patch  # a Rectangle instance
rect.set_facecolor('green')
```

When you call a plotting method, e.g., the canonical *plot* and pass in arrays or lists of values, the method will create a *matplotlib.lines.Line2D* instance, update the line with all the `Line2D` properties passed as keyword arguments, add the line to the `Axes`, and return it to you:

```
In [213]: x, y = np.random.rand(2, 100)

In [214]: line, = ax.plot(x, y, '-', color='blue', linewidth=2)
```

`plot` returns a list of lines because you can pass in multiple x, y pairs to plot, and we are unpacking the first element of the length one list into the line variable. The line has been added to the `Axes.lines` list:

```
In [229]: print(ax.lines)
[<matplotlib.lines.Line2D at 0xd378b0c>]
```

Similarly, methods that create patches, like *bar()* creates a list of rectangles, will add the patches to the `Axes.patches` list:

```
In [233]: n, bins, rectangles = ax.hist(np.random.randn(1000), 50)

In [234]: rectangles
Out[234]: <BarContainer object of 50 artists>

In [235]: print(len(ax.patches))
Out[235]: 50
```

You should not add objects directly to the `Axes.lines` or `Axes.patches` lists, because the `Axes` needs to do a few things when it creates and adds an object:

- It sets the `figure` and `axes` property of the `Artist`;
- It sets the default `Axes` transformation (unless one is already set);
- It inspects the data contained in the `Artist` to update the data structures controlling auto-scaling, so that the view limits can be adjusted to contain the plotted data.

You can, nonetheless, create objects yourself and add them directly to the `Axes` using helper methods like *add_line* and *add_patch*. Here is an annotated interactive session illustrating what is going on:

```
In [262]: fig, ax = plt.subplots()
```

(continues on next page)

(continued from previous page)

```
# create a rectangle instance
In [263]: rect = matplotlib.patches.Rectangle((1, 1), width=5, height=12)

# by default the Axes instance is None
In [264]: print(rect.axes)
None

# and the transformation instance is set to the "identity transform"
In [265]: print(rect.get_data_transform())
IdentityTransform()

# now we add the Rectangle to the Axes
In [266]: ax.add_patch(rect)

# and notice that the ax.add_patch method has set the Axes
# instance
In [267]: print(rect.axes)
Axes(0.125,0.1;0.775x0.8)

# and the transformation has been set too
In [268]: print(rect.get_data_transform())
CompositeGenericTransform(
    TransformWrapper(
        BlendedAffine2D(
            IdentityTransform(),
            IdentityTransform())),
    CompositeGenericTransform(
        BboxTransformFrom(
            TransformedBbox(
                Bbox(x0=0.0, y0=0.0, x1=1.0, y1=1.0),
                TransformWrapper(
                    BlendedAffine2D(
                        IdentityTransform(),
                        IdentityTransform())))),
        BboxTransformTo(
            TransformedBbox(
                Bbox(x0=0.125, y0=0.10999999999999999, x1=0.9, y1=0.88),
                BboxTransformTo(
                    TransformedBbox(
                        Bbox(x0=0.0, y0=0.0, x1=6.4, y1=4.8),
                        Affine2D(
                            [[100.   0.   0.]
                             [  0. 100.   0.]
                             [  0.   0.   1.]]))))))))

# the default Axes transformation is ax.transData
In [269]: print(ax.transData)
CompositeGenericTransform(
    TransformWrapper(
        BlendedAffine2D(
            IdentityTransform(),
            IdentityTransform())),
```

(continues on next page)

(continued from previous page)

```
    CompositeGenericTransform(
        BboxTransformFrom(
            TransformedBbox(
                Bbox(x0=0.0, y0=0.0, x1=1.0, y1=1.0),
                TransformWrapper(
                    BlendedAffine2D(
                        IdentityTransform(),
                        IdentityTransform())))),
        BboxTransformTo(
            TransformedBbox(
                Bbox(x0=0.125, y0=0.10999999999999999, x1=0.9, y1=0.88),
                BboxTransformTo(
                    TransformedBbox(
                        Bbox(x0=0.0, y0=0.0, x1=6.4, y1=4.8),
                        Affine2D(
                            [[100.   0.   0.]
                             [  0. 100.   0.]
                             [  0.   0.   1.]]))))))))

# notice that the xlimits of the Axes have not been changed
In [270]: print(ax.get_xlim())
(0.0, 1.0)

# but the data limits have been updated to encompass the rectangle
In [271]: print(ax.dataLim.bounds)
(1.0, 1.0, 5.0, 12.0)

# we can manually invoke the auto-scaling machinery
In [272]: ax.autoscale_view()

# and now the xlim are updated to encompass the rectangle, plus margins
In [273]: print(ax.get_xlim())
(0.75, 6.25)

# we have to manually force a figure draw
In [274]: fig.canvas.draw()
```

There are many, many `Axes` helper methods for creating primitive `Artists` and adding them to their respective containers. The table below summarizes a small sampling of them, the kinds of `Artist` they create, and where they store them

| Axes helper method | Artist | Container |
|---|---|---|
| *annotate* - text annotations | *Annotation* | ax.texts |
| *bar* - bar charts | *Rectangle* | ax.patches |
| *errorbar* - error bar plots | *Line2D* and *Rectangle* | ax.lines and ax.patches |
| *fill* - shared area | *Polygon* | ax.patches |
| *hist* - histograms | *Rectangle* | ax.patches |
| *imshow* - image data | *AxesImage* | ax.images |
| *legend* - Axes legend | *Legend* | ax.get_legend() |
| *plot* - xy plots | *Line2D* | ax.lines |
| *scatter* - scatter charts | *PolyCollection* | ax.collections |
| *text* - text | *Text* | ax.texts |

In addition to all of these `Artists`, the `Axes` contains two important `Artist` containers: the *XAxis* and *YAxis*, which handle the drawing of the ticks and labels. These are stored as instance variables `xaxis` and `yaxis`. The `XAxis` and `YAxis` containers will be detailed below, but note that the `Axes` contains many helper methods which forward calls on to the *Axis* instances, so you often do not need to work with them directly unless you want to. For example, you can set the font color of the `XAxis` ticklabels using the `Axes` helper method:

```
ax.tick_params(axis='x', labelcolor='orange')
```

Below is a summary of the Artists that the *Axes* contains

| Axes attribute | Description |
|---|---|
| artists | An *ArtistList* of *Artist* instances |
| patch | *Rectangle* instance for Axes background |
| collections | An *ArtistList* of *Collection* instances |
| images | An *ArtistList* of *AxesImage* |
| lines | An *ArtistList* of *Line2D* instances |
| patches | An *ArtistList* of *Patch* instances |
| texts | An *ArtistList* of *Text* instances |
| xaxis | A *matplotlib.axis.XAxis* instance |
| yaxis | A *matplotlib.axis.YAxis* instance |

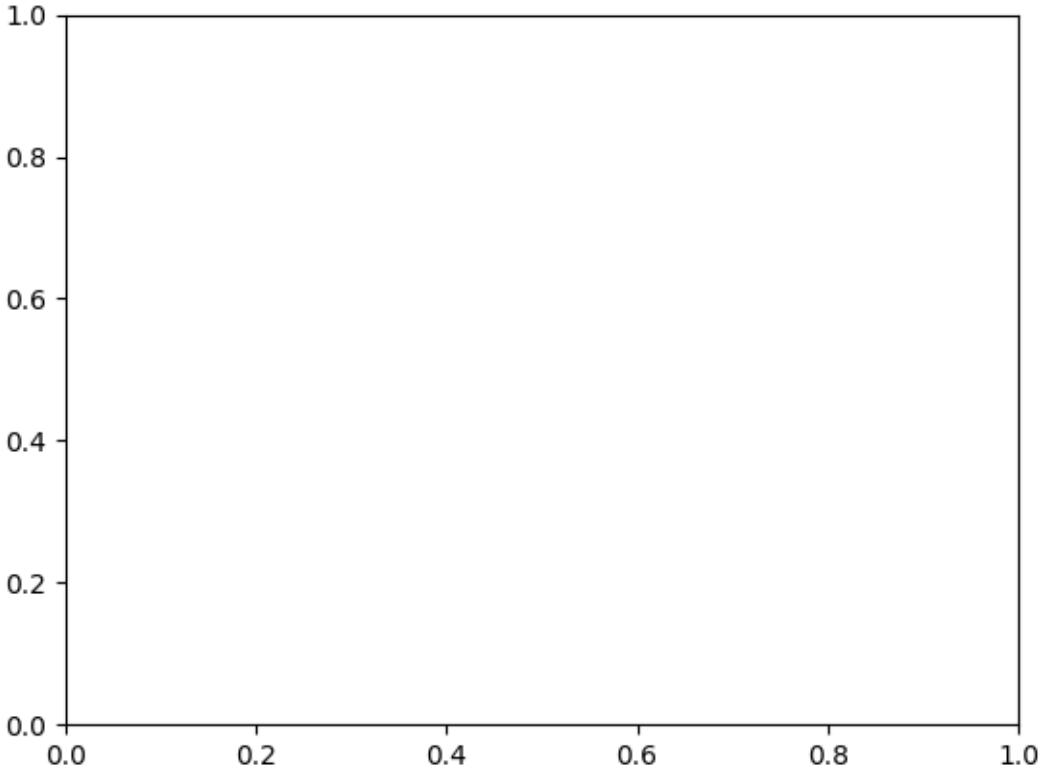
The legend can be accessed by *get_legend*,

### Axis containers

The *matplotlib.axis.Axis* instances handle the drawing of the tick lines, the grid lines, the tick labels and the axis label. You can configure the left and right ticks separately for the y-axis, and the upper and lower ticks separately for the x-axis. The `Axis` also stores the data and view intervals used in auto-scaling, panning and zooming, as well as the *Locator* and *Formatter* instances which control where the ticks are placed and how they are represented as strings.

Each `Axis` object contains a `label` attribute (this is what *pyplot* modifies in calls to *xlabel* and *ylabel*) as well as a list of major and minor ticks. The ticks are *axis.XTick* and *axis.YTick* instances, which contain the actual line and text primitives that render the ticks and ticklabels. Because the ticks are dynamically created as needed (e.g., when panning and zooming), you should access the lists of major and minor ticks through their accessor methods *axis.Axis.get_major_ticks* and *axis.Axis.get_minor_ticks*. Although the ticks contain all the primitives and will be covered below, `Axis` instances have accessor methods that return the tick lines, tick labels, tick locations etc.:

```
fig, ax = plt.subplots()
axis = ax.xaxis
axis.get_ticklocs()
```




```
array([0. , 0.2, 0.4, 0.6, 0.8, 1. ])
```

```
axis.get_ticklabels()
```

```
[Text(0.0, 0, '0.0'), Text(0.2, 0, '0.2'), Text(0.4, 0, '0.4'), Text(0.
↪6000000000000001, 0, '0.6'), Text(0.8, 0, '0.8'), Text(1.0, 0, '1.0')]
```

note there are twice as many ticklines as labels because by default there are tick lines at the top and bottom but only tick labels below the xaxis; however, this can be customized.

```
axis.get_ticklines()
```

```
<a list of 12 Line2D ticklines objects>
```

And with the above methods, you only get lists of major ticks back by default, but you can also ask for the minor ticks:

```
axis.get_ticklabels(minor=True)
axis.get_ticklines(minor=True)
```

```
<a list of 0 Line2D ticklines objects>
```

Here is a summary of some of the useful accessor methods of the `Axis` (these have corresponding setters where useful, such as *`set_major_formatter()`*.)

| Axis accessor method | Description |
|---|---|
| *`get_scale`* | The scale of the Axis, e.g., 'log' or 'linear' |
| *`get_view_interval`* | The interval instance of the Axis view limits |
| *`get_data_interval`* | The interval instance of the Axis data limits |
| *`get_gridlines`* | A list of grid lines for the Axis |
| *`get_label`* | The Axis label - a *`Text`* instance |
| *`get_offset_text`* | The Axis offset text - a *`Text`* instance |
| *`get_ticklabels`* | A list of *`Text`* instances - keyword minor=True\|False |
| *`get_ticklines`* | A list of *`Line2D`* instances - keyword minor=True\|False |
| *`get_ticklocs`* | A list of Tick locations - keyword minor=True\|False |
| *`get_major_locator`* | The *`ticker.Locator`* instance for major ticks |
| *`get_major_formatter`* | The *`ticker.Formatter`* instance for major ticks |
| *`get_minor_locator`* | The *`ticker.Locator`* instance for minor ticks |
| *`get_minor_formatter`* | The *`ticker.Formatter`* instance for minor ticks |
| *`get_major_ticks`* | A list of *`Tick`* instances for major ticks |
| *`get_minor_ticks`* | A list of *`Tick`* instances for minor ticks |
| *`grid`* | Turn the grid on or off for the major or minor ticks |

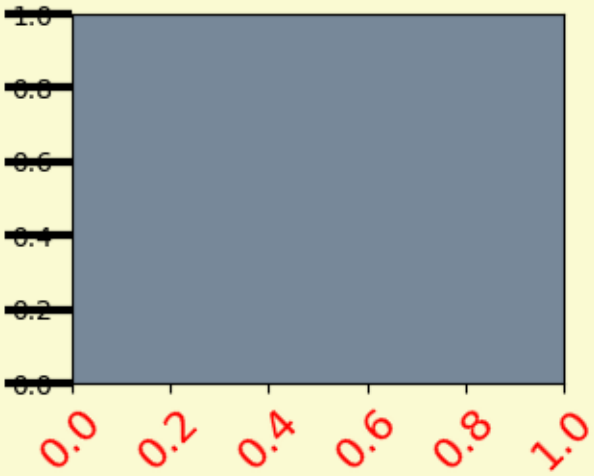
Here is an example, not recommended for its beauty, which customizes the Axes and Tick properties.

```
# plt.figure creates a matplotlib.figure.Figure instance
fig = plt.figure()
rect = fig.patch  # a rectangle instance
```

(continues on next page)

(continued from previous page)

```
rect.set_facecolor('lightgoldenrodyellow')

ax1 = fig.add_axes([0.1, 0.3, 0.4, 0.4])
rect = ax1.patch
rect.set_facecolor('lightslategray')


for label in ax1.xaxis.get_ticklabels():
    # label is a Text instance
    label.set_color('red')
    label.set_rotation(45)
    label.set_fontsize(16)

for line in ax1.yaxis.get_ticklines():
    # line is a Line2D instance
    line.set_color('green')
    line.set_markersize(25)
    line.set_markeredgewidth(3)

plt.show()
```



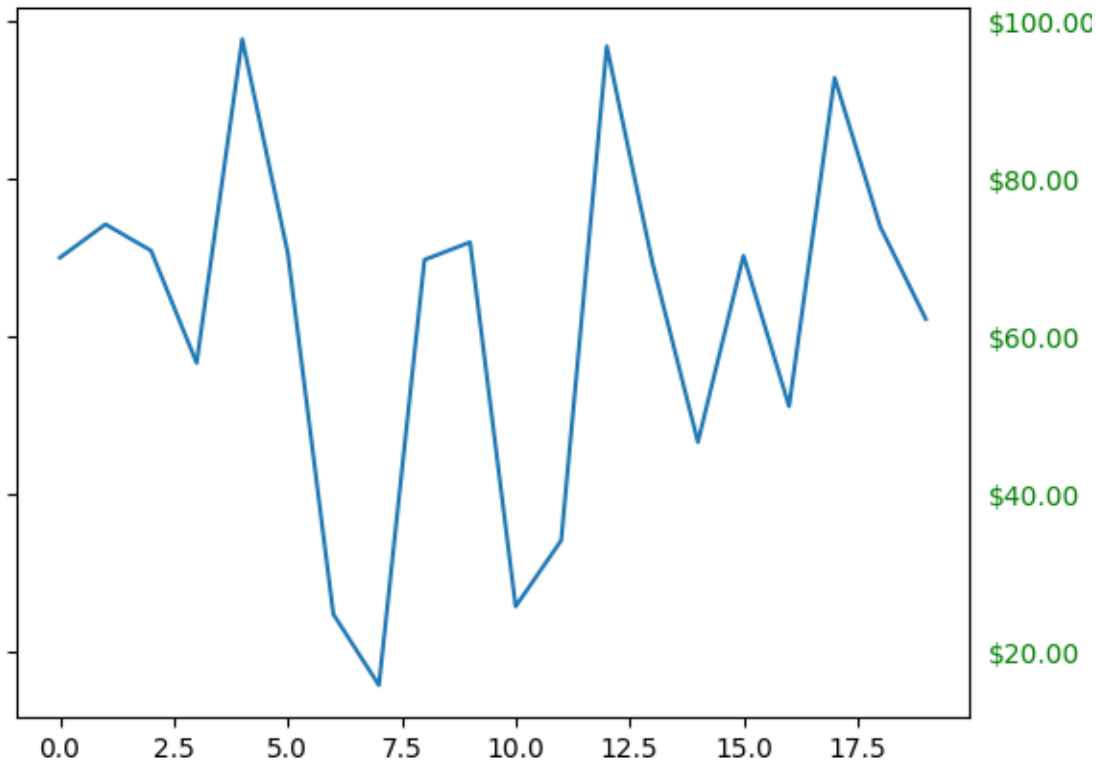
### Tick containers

The *matplotlib.axis.Tick* is the final container object in our descent from the *Figure* to the *Axes* to the *Axis* to the *Tick*. The `Tick` contains the tick and grid line instances, as well as the label instances for the upper and lower ticks. Each of these is accessible directly as an attribute of the `Tick`.

| Tick attribute | Description |
|---|---|
| tick1line | A *Line2D* instance |
| tick2line | A *Line2D* instance |
| gridline | A *Line2D* instance |
| label1 | A *Text* instance |
| label2 | A *Text* instance |

Here is an example which sets the formatter for the right side ticks with dollar signs and colors them green on the right side of the yaxis.

```
import matplotlib.pyplot as plt
import numpy as np

# Fixing random state for reproducibility
np.random.seed(19680801)

fig, ax = plt.subplots()
ax.plot(100*np.random.rand(20))

# Use automatic StrMethodFormatter
ax.yaxis.set_major_formatter('${x:1.2f}')

ax.yaxis.set_tick_params(which='major', labelcolor='green',
                         labelleft=False, labelright=True)

plt.show()
```




**Total running time of the script:** (0 minutes 1.925 seconds)

## 4.5 User guide tutorials

Many of our tutorials were moved from this section to *Using Matplotlib*:

### 4.5.1 Introductory

- *Quick start guide*
- *Customizing Matplotlib with style sheets and rcParams*
- *Animations using Matplotlib*

### 4.5.2 Intermediate

- *Legend guide*
- *Styling with cycler*
- *Constrained layout guide*
- *Tight layout guide*
- *Arranging multiple Axes in a Figure*
- *Autoscaling Axis*
- *origin and extent in imshow*

### 4.5.3 Advanced

- *Faster rendering by using blitting*
- *Path Tutorial*
- *Path effects guide*
- *Transformations Tutorial*

### 4.5.4 Colors

See *Colors*.

### 4.5.5 Text

See *Text*.

### 4.5.6 Toolkits

See *User Toolkits*.

**What can Matplotlib do?**

CHAPTER

FIVE

# PLOT TYPES

Overview of many common plotting commands provided by Matplotlib.

See the gallery for more examples and the tutorials page for longer examples.

## 5.1 Pairwise data

Plots of pairwise $(x, y)$, tabular $(var\_0, \cdots, var\_n)$, and functional $f(x) = y$ data.

## 5.2 Statistical distributions

Plots of the distribution of at least one variable in a dataset. Some of these methods also compute the distributions.

## 5.3 Gridded data:

Plots of arrays and images $Z_{i,j}$ and fields $U_{i,j}, V_{i,j}$ on regular grids and corresponding coordinate grids $X_{i,j}, Y_{i,j}$.

## 5.4 Irregularly gridded data

Plots of data $Z_{x,y}$ on unstructured grids , unstructured coordinate grids $(x, y)$, and 2D functions $f(x, y) = z$.

## 5.5 3D and volumetric data

Plots of three-dimensional $(x, y, z)$, surface $f(x, y) = z$, and volumetric $V_{x,y,z}$ data using the *mpl_toolkits.mplot3d* library.

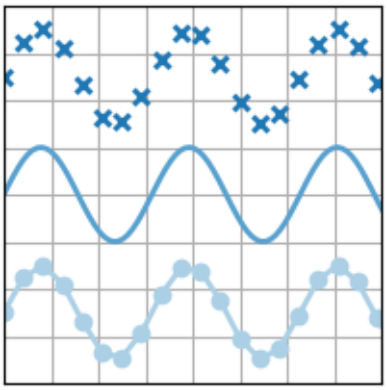
### 5.5.1 Pairwise data

Plots of pairwise $(x, y)$, tabular $(var\_0, \cdots, var\_n)$, and functional $f(x) = y$ data.

#### plot(x, y)

Plot y versus x as lines and/or markers.

See *plot*.

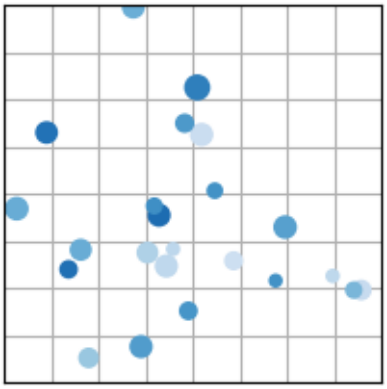


```
import matplotlib.pyplot as plt
import numpy as np

plt.style.use('_mpl-gallery')

# make data
x = np.linspace(0, 10, 100)
y = 4 + 1 * np.sin(2 * x)
x2 = np.linspace(0, 10, 25)
y2 = 4 + 1 * np.sin(2 * x2)

# plot
fig, ax = plt.subplots()

ax.plot(x2, y2 + 2.5, 'x', markeredgewidth=2)
ax.plot(x, y, linewidth=2.0)
ax.plot(x2, y2 - 2.5, 'o-', linewidth=2)

ax.set(xlim=(0, 8), xticks=np.arange(1, 8),
       ylim=(0, 8), yticks=np.arange(1, 8))

plt.show()
```

### scatter(x, y)

A scatter plot of y vs. x with varying marker size and/or color.

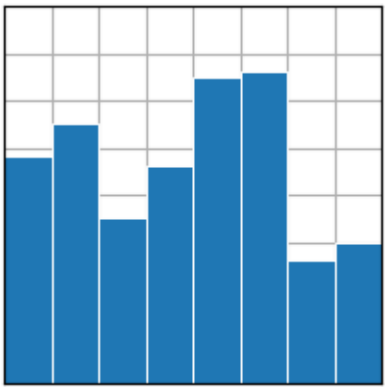
See *scatter*.



```
import matplotlib.pyplot as plt
import numpy as np

plt.style.use('_mpl-gallery')

# make the data
np.random.seed(3)
x = 4 + np.random.normal(0, 2, 24)
y = 4 + np.random.normal(0, 2, len(x))
# size and color:
sizes = np.random.uniform(15, 80, len(x))
colors = np.random.uniform(15, 80, len(x))

# plot
fig, ax = plt.subplots()

ax.scatter(x, y, s=sizes, c=colors, vmin=0, vmax=100)

ax.set(xlim=(0, 8), xticks=np.arange(1, 8),
       ylim=(0, 8), yticks=np.arange(1, 8))

plt.show()
```

### bar(x, height)

See *bar*.

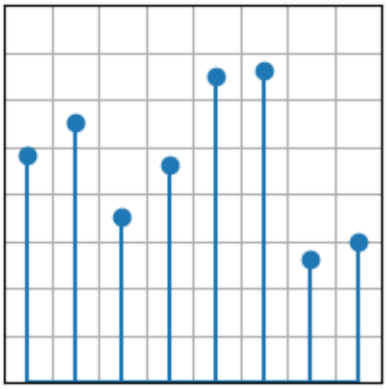


```
import matplotlib.pyplot as plt
import numpy as np

plt.style.use('_mpl-gallery')

# make data:
x = 0.5 + np.arange(8)
y = [4.8, 5.5, 3.5, 4.6, 6.5, 6.6, 2.6, 3.0]

# plot
fig, ax = plt.subplots()

ax.bar(x, y, width=1, edgecolor="white", linewidth=0.7)

ax.set(xlim=(0, 8), xticks=np.arange(1, 8),
       ylim=(0, 8), yticks=np.arange(1, 8))

plt.show()
```

### stem(x, y)

Create a stem plot.

See *stem*.

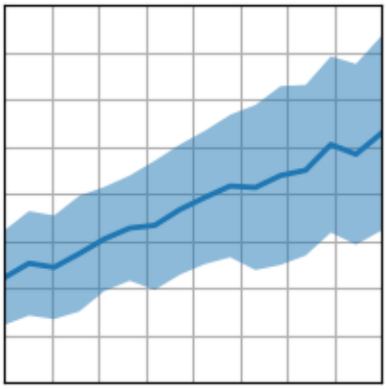


```
import matplotlib.pyplot as plt
import numpy as np

plt.style.use('_mpl-gallery')

# make data
x = 0.5 + np.arange(8)
y = [4.8, 5.5, 3.5, 4.6, 6.5, 6.6, 2.6, 3.0]

# plot
fig, ax = plt.subplots()

ax.stem(x, y)

ax.set(xlim=(0, 8), xticks=np.arange(1, 8),
       ylim=(0, 8), yticks=np.arange(1, 8))

plt.show()
```

### fill_between(x, y1, y2)

Fill the area between two horizontal curves.

See *fill_between*.

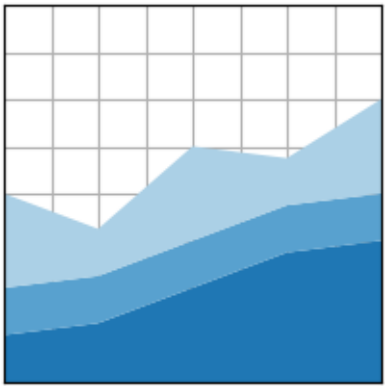


```
import matplotlib.pyplot as plt
import numpy as np

plt.style.use('_mpl-gallery')

# make data
np.random.seed(1)
x = np.linspace(0, 8, 16)
y1 = 3 + 4*x/8 + np.random.uniform(0.0, 0.5, len(x))
y2 = 1 + 2*x/8 + np.random.uniform(0.0, 0.5, len(x))

# plot
fig, ax = plt.subplots()

ax.fill_between(x, y1, y2, alpha=.5, linewidth=0)
ax.plot(x, (y1 + y2)/2, linewidth=2)

ax.set(xlim=(0, 8), xticks=np.arange(1, 8),
       ylim=(0, 8), yticks=np.arange(1, 8))

plt.show()
```

### stackplot(x, y)

Draw a stacked area plot or a streamgraph.

See *stackplot*



```
import matplotlib.pyplot as plt
import numpy as np

plt.style.use('_mpl-gallery')

# make data
x = np.arange(0, 10, 2)
ay = [1, 1.25, 2, 2.75, 3]
by = [1, 1, 1, 1, 1]
cy = [2, 1, 2, 1, 2]
```

(continues on next page)

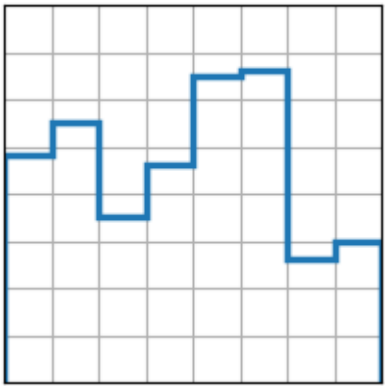
(continued from previous page)

```
y = np.vstack([ay, by, cy])

# plot
fig, ax = plt.subplots()

ax.stackplot(x, y)

ax.set(xlim=(0, 8), xticks=np.arange(1, 8),
       ylim=(0, 8), yticks=np.arange(1, 8))

plt.show()
```

### stairs(values)

Draw a stepwise constant function as a line or a filled plot.

See *stairs* when plotting $y$ between $(x_i, x_{i+1})$. For plotting $y$ at $x$, see *step*.



```
import matplotlib.pyplot as plt
import numpy as np

plt.style.use('_mpl-gallery')

# make data
y = [4.8, 5.5, 3.5, 4.6, 6.5, 6.6, 2.6, 3.0]

# plot
fig, ax = plt.subplots()

ax.stairs(y, linewidth=2.5)

ax.set(xlim=(0, 8), xticks=np.arange(1, 8),
       ylim=(0, 8), yticks=np.arange(1, 8))

plt.show()
```

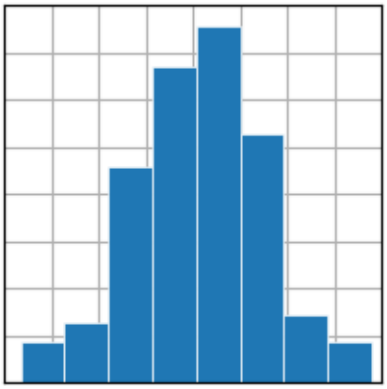
### 5.5.2 Statistical distributions

Plots of the distribution of at least one variable in a dataset. Some of these methods also compute the distributions.

#### hist(x)

Compute and plot a histogram.

See *hist*.

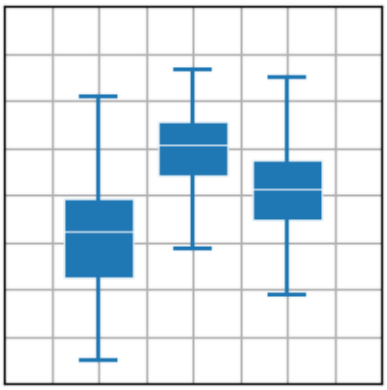


```
import matplotlib.pyplot as plt
import numpy as np

plt.style.use('_mpl-gallery')

# make data
np.random.seed(1)
x = 4 + np.random.normal(0, 1.5, 200)

# plot:
fig, ax = plt.subplots()

ax.hist(x, bins=8, linewidth=0.5, edgecolor="white")

ax.set(xlim=(0, 8), xticks=np.arange(1, 8),
       ylim=(0, 56), yticks=np.linspace(0, 56, 9))

plt.show()
```

### boxplot(X)

Draw a box and whisker plot.

See *boxplot*.

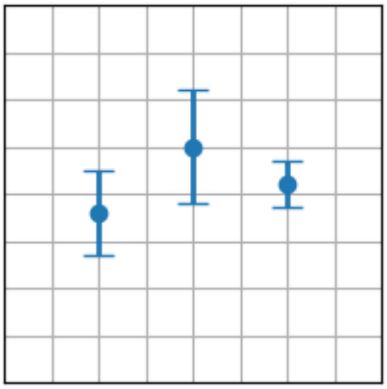


```
import matplotlib.pyplot as plt
import numpy as np

plt.style.use('_mpl-gallery')

# make data:
np.random.seed(10)
D = np.random.normal((3, 5, 4), (1.25, 1.00, 1.25), (100, 3))

# plot
fig, ax = plt.subplots()
VP = ax.boxplot(D, positions=[2, 4, 6], widths=1.5, patch_artist=True,
                showmeans=False, showfliers=False,
                medianprops={"color": "white", "linewidth": 0.5},
                boxprops={"facecolor": "C0", "edgecolor": "white",
                          "linewidth": 0.5},
                whiskerprops={"color": "C0", "linewidth": 1.5},
                capprops={"color": "C0", "linewidth": 1.5})

ax.set(xlim=(0, 8), xticks=np.arange(1, 8),
       ylim=(0, 8), yticks=np.arange(1, 8))

plt.show()
```

### errorbar(x, y, yerr, xerr)

Plot y versus x as lines and/or markers with attached errorbars.

See *errorbar*.

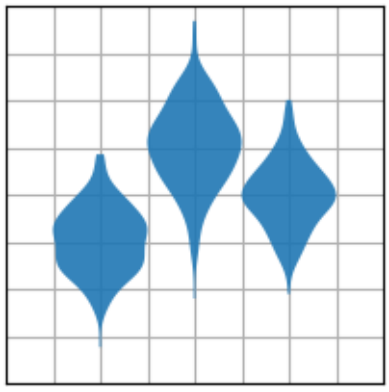


```
import matplotlib.pyplot as plt
import numpy as np

plt.style.use('_mpl-gallery')

# make data:
np.random.seed(1)
x = [2, 4, 6]
y = [3.6, 5, 4.2]
yerr = [0.9, 1.2, 0.5]

# plot:
fig, ax = plt.subplots()

ax.errorbar(x, y, yerr, fmt='o', linewidth=2, capsize=6)

ax.set(xlim=(0, 8), xticks=np.arange(1, 8),
       ylim=(0, 8), yticks=np.arange(1, 8))

plt.show()
```

### violinplot(D)

Make a violin plot.

See *violinplot*.

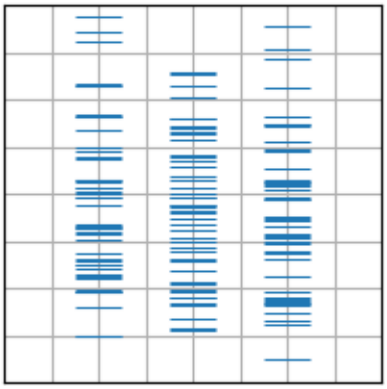


```
import matplotlib.pyplot as plt
import numpy as np

plt.style.use('_mpl-gallery')

# make data:
np.random.seed(10)
D = np.random.normal((3, 5, 4), (0.75, 1.00, 0.75), (200, 3))

# plot:
fig, ax = plt.subplots()

vp = ax.violinplot(D, [2, 4, 6], widths=2,
                   showmeans=False, showmedians=False, showextrema=False)
# styling:
for body in vp['bodies']:
    body.set_alpha(0.9)
ax.set(xlim=(0, 8), xticks=np.arange(1, 8),
       ylim=(0, 8), yticks=np.arange(1, 8))

plt.show()
```

### eventplot(D)

Plot identical parallel lines at the given positions.

See *eventplot*.

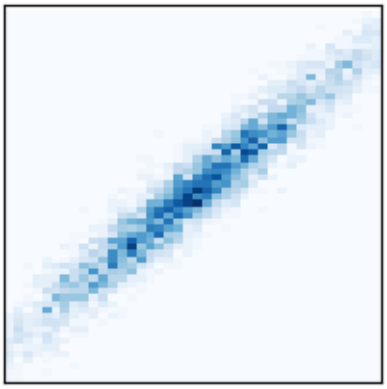


```
import matplotlib.pyplot as plt
import numpy as np

plt.style.use('_mpl-gallery')

# make data:
np.random.seed(1)
x = [2, 4, 6]
D = np.random.gamma(4, size=(3, 50))

# plot:
fig, ax = plt.subplots()

ax.eventplot(D, orientation="vertical", lineoffsets=x, linewidth=0.75)

ax.set(xlim=(0, 8), xticks=np.arange(1, 8),
       ylim=(0, 8), yticks=np.arange(1, 8))

plt.show()
```

### hist2d(x, y)

Make a 2D histogram plot.

See *hist2d*.

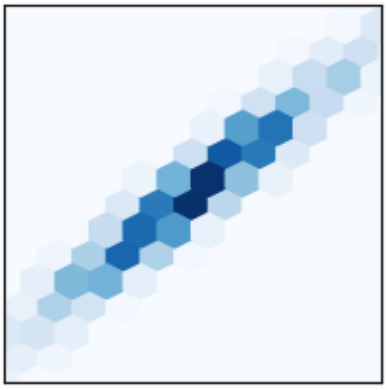


```
import matplotlib.pyplot as plt
import numpy as np

plt.style.use('_mpl-gallery-nogrid')

# make data: correlated + noise
np.random.seed(1)
x = np.random.randn(5000)
y = 1.2 * x + np.random.randn(5000) / 3

# plot:
fig, ax = plt.subplots()

ax.hist2d(x, y, bins=(np.arange(-3, 3, 0.1), np.arange(-3, 3, 0.1)))

ax.set(xlim=(-2, 2), ylim=(-3, 3))

plt.show()
```

### hexbin(x, y, C)

Make a 2D hexagonal binning plot of points x, y.

See *hexbin*.



```
import matplotlib.pyplot as plt
import numpy as np

plt.style.use('_mpl-gallery-nogrid')

# make data: correlated + noise
np.random.seed(1)
x = np.random.randn(5000)
y = 1.2 * x + np.random.randn(5000) / 3

# plot:
fig, ax = plt.subplots()
```

(continues on next page)

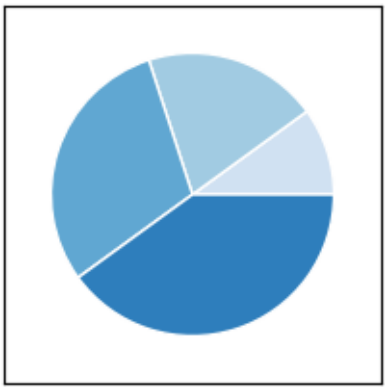
(continued from previous page)

```
ax.hexbin(x, y, gridsize=20)

ax.set(xlim=(-2, 2), ylim=(-3, 3))

plt.show()
```

### pie(x)

Plot a pie chart.

See *pie*.

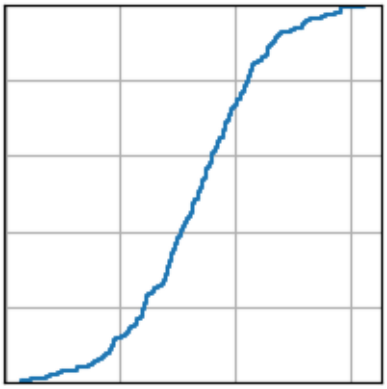


```
import matplotlib.pyplot as plt
import numpy as np

plt.style.use('_mpl-gallery-nogrid')


# make data
x = [1, 2, 3, 4]
colors = plt.get_cmap('Blues')(np.linspace(0.2, 0.7, len(x)))

# plot
fig, ax = plt.subplots()
ax.pie(x, colors=colors, radius=3, center=(4, 4),
       wedgeprops={"linewidth": 1, "edgecolor": "white"}, frame=True)

ax.set(xlim=(0, 8), xticks=np.arange(1, 8),
       ylim=(0, 8), yticks=np.arange(1, 8))

plt.show()
```

#### ecdf(x)

Compute and plot the empirical cumulative distribution function of x.

See *ecdf*.



```
import matplotlib.pyplot as plt
import numpy as np

plt.style.use('_mpl-gallery')

# make data
np.random.seed(1)
x = 4 + np.random.normal(0, 1.5, 200)

# plot:
fig, ax = plt.subplots()
ax.ecdf(x)
plt.show()
```

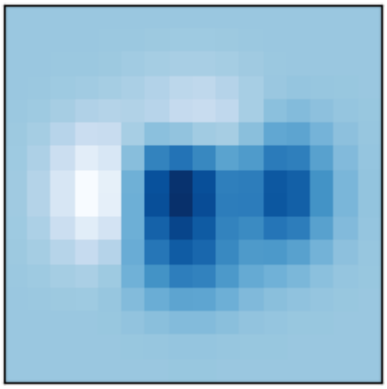
### 5.5.3 Gridded data:

Plots of arrays and images $Z_{i,j}$ and fields $U_{i,j}, V_{i,j}$ on regular grids and corresponding coordinate grids $X_{i,j}, Y_{i,j}$.

#### imshow(Z)

Display data as an image, i.e., on a 2D regular raster.

See *imshow*.

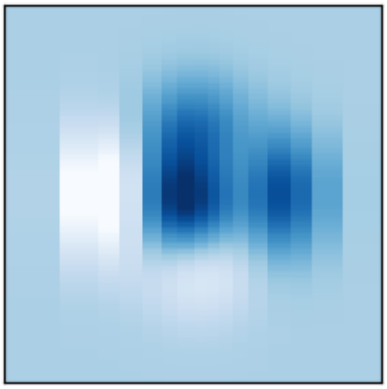


```
import matplotlib.pyplot as plt
import numpy as np

plt.style.use('_mpl-gallery-nogrid')

# make data
X, Y = np.meshgrid(np.linspace(-3, 3, 16), np.linspace(-3, 3, 16))
Z = (1 - X/2 + X**5 + Y**3) * np.exp(-X**2 - Y**2)

# plot
fig, ax = plt.subplots()

ax.imshow(Z)

plt.show()
```

### pcolormesh(X, Y, Z)

Create a pseudocolor plot with a non-regular rectangular grid.

*pcolormesh* is more flexible than *imshow* in that the x and y vectors need not be equally spaced (indeed they can be skewed).



```
import matplotlib.pyplot as plt
import numpy as np
```

(continues on next page)

(continued from previous page)

```
plt.style.use('_mpl-gallery-nogrid')

# make data with uneven sampling in x
x = [-3, -2, -1.6, -1.2, -.8, -.5, -.2, .1, .3, .5, .8, 1.1, 1.5, 1.9, 2.3, 3]
X, Y = np.meshgrid(x, np.linspace(-3, 3, 128))
Z = (1 - X/2 + X**5 + Y**3) * np.exp(-X**2 - Y**2)

# plot
fig, ax = plt.subplots()

ax.pcolormesh(X, Y, Z, vmin=-0.5, vmax=1.0)

plt.show()
```

### contour(X, Y, Z)

Plot contour lines.

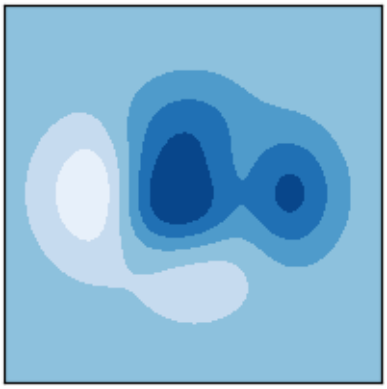
See *contour*.

```
import matplotlib.pyplot as plt
import numpy as np

plt.style.use('_mpl-gallery-nogrid')

# make data
X, Y = np.meshgrid(np.linspace(-3, 3, 256), np.linspace(-3, 3, 256))
Z = (1 - X/2 + X**5 + Y**3) * np.exp(-X**2 - Y**2)
levels = np.linspace(np.min(Z), np.max(Z), 7)

# plot
fig, ax = plt.subplots()

ax.contour(X, Y, Z, levels=levels)

plt.show()
```

### contourf(X, Y, Z)

Plot filled contours.

See *contourf*.

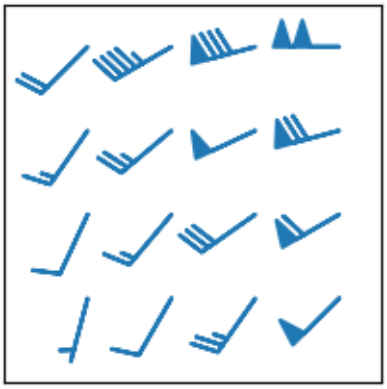


```
import matplotlib.pyplot as plt
import numpy as np

plt.style.use('_mpl-gallery-nogrid')

# make data
X, Y = np.meshgrid(np.linspace(-3, 3, 256), np.linspace(-3, 3, 256))
Z = (1 - X/2 + X**5 + Y**3) * np.exp(-X**2 - Y**2)
levels = np.linspace(Z.min(), Z.max(), 7)

# plot
fig, ax = plt.subplots()

ax.contourf(X, Y, Z, levels=levels)

plt.show()
```

### barbs(X, Y, U, V)

Plot a 2D field of wind barbs.

See *barbs*.

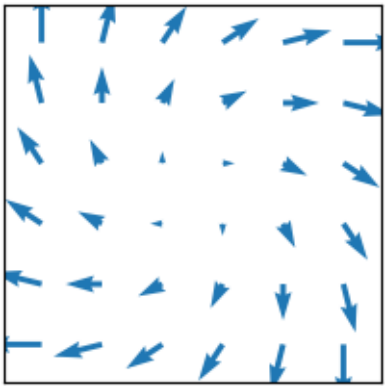


```
import matplotlib.pyplot as plt
import numpy as np

plt.style.use('_mpl-gallery-nogrid')

# make data:
X, Y = np.meshgrid([1, 2, 3, 4], [1, 2, 3, 4])
angle = np.pi / 180 * np.array([[15., 30, 35, 45],
                                [25., 40, 55, 60],
                                [35., 50, 65, 75],
                                [45., 60, 75, 90]])
amplitude = np.array([[5, 10, 25, 50],
                      [10, 15, 30, 60],
                      [15, 26, 50, 70],
                      [20, 45, 80, 100]])
U = amplitude * np.sin(angle)
V = amplitude * np.cos(angle)

# plot:
fig, ax = plt.subplots()

ax.barbs(X, Y, U, V, barbcolor='C0', flagcolor='C0', length=7, linewidth=1.5)

ax.set(xlim=(0, 4.5), ylim=(0, 4.5))

plt.show()
```

### quiver(X, Y, U, V)

Plot a 2D field of arrows.

See *quiver*.



```
import matplotlib.pyplot as plt
import numpy as np

plt.style.use('_mpl-gallery-nogrid')

# make data
x = np.linspace(-4, 4, 6)
y = np.linspace(-4, 4, 6)
X, Y = np.meshgrid(x, y)
U = X + Y
V = Y - X

# plot
fig, ax = plt.subplots()

ax.quiver(X, Y, U, V, color="C0", angles='xy',
          scale_units='xy', scale=5, width=.015)

ax.set(xlim=(-5, 5), ylim=(-5, 5))

plt.show()
```

### streamplot(X, Y, U, V)

Draw streamlines of a vector flow.

See *streamplot*.

```
import matplotlib.pyplot as plt
import numpy as np

plt.style.use('_mpl-gallery-nogrid')

# make a stream function:
X, Y = np.meshgrid(np.linspace(-3, 3, 256), np.linspace(-3, 3, 256))
Z = (1 - X/2 + X**5 + Y**3) * np.exp(-X**2 - Y**2)
# make U and V out of the streamfunction:
V = np.diff(Z[1:, :], axis=1)
U = -np.diff(Z[:, 1:], axis=0)

# plot:
fig, ax = plt.subplots()

ax.streamplot(X[1:, 1:], Y[1:, 1:], U, V)

plt.show()
```

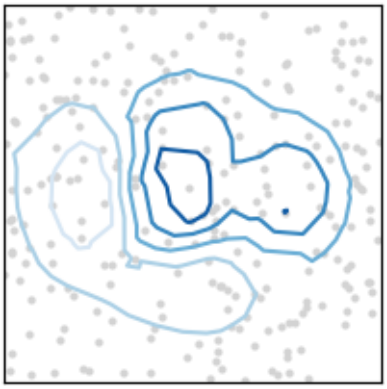
### 5.5.4 Irregularly gridded data

Plots of data $Z_{x,y}$ on unstructured grids , unstructured coordinate grids $(x, y)$, and 2D functions $f(x, y) = z$.

### tricontour(x, y, z)

Draw contour lines on an unstructured triangular grid.

See *tricontour*.

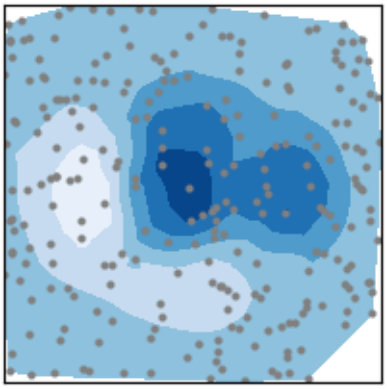


```
import matplotlib.pyplot as plt
import numpy as np

plt.style.use('_mpl-gallery-nogrid')

# make data:
np.random.seed(1)
x = np.random.uniform(-3, 3, 256)
y = np.random.uniform(-3, 3, 256)
z = (1 - x/2 + x**5 + y**3) * np.exp(-x**2 - y**2)
levels = np.linspace(z.min(), z.max(), 7)

# plot:
fig, ax = plt.subplots()

ax.plot(x, y, 'o', markersize=2, color='lightgrey')
ax.tricontour(x, y, z, levels=levels)

ax.set(xlim=(-3, 3), ylim=(-3, 3))

plt.show()
```

### tricontourf(x, y, z)

Draw contour regions on an unstructured triangular grid.

See *tricontourf*.

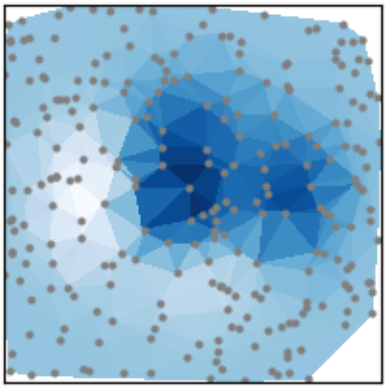


```
import matplotlib.pyplot as plt
import numpy as np

plt.style.use('_mpl-gallery-nogrid')

# make data:
np.random.seed(1)
x = np.random.uniform(-3, 3, 256)
y = np.random.uniform(-3, 3, 256)
z = (1 - x/2 + x**5 + y**3) * np.exp(-x**2 - y**2)
levels = np.linspace(z.min(), z.max(), 7)

# plot:
fig, ax = plt.subplots()

ax.plot(x, y, 'o', markersize=2, color='grey')
ax.tricontourf(x, y, z, levels=levels)

ax.set(xlim=(-3, 3), ylim=(-3, 3))

plt.show()
```

### tripcolor(x, y, z)

Create a pseudocolor plot of an unstructured triangular grid.

See *tripcolor*.

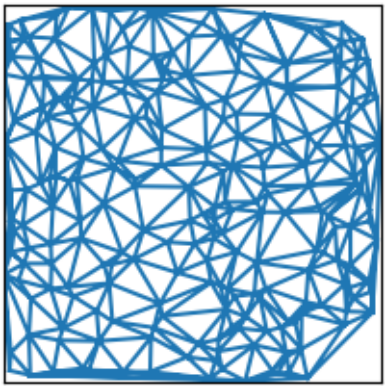


```
import matplotlib.pyplot as plt
import numpy as np

plt.style.use('_mpl-gallery-nogrid')

# make data:
np.random.seed(1)
x = np.random.uniform(-3, 3, 256)
y = np.random.uniform(-3, 3, 256)
z = (1 - x/2 + x**5 + y**3) * np.exp(-x**2 - y**2)

# plot:
fig, ax = plt.subplots()

ax.plot(x, y, 'o', markersize=2, color='grey')
ax.tripcolor(x, y, z)

ax.set(xlim=(-3, 3), ylim=(-3, 3))

plt.show()
```

### triplot(x, y)

Draw an unstructured triangular grid as lines and/or markers.

See *triplot*.



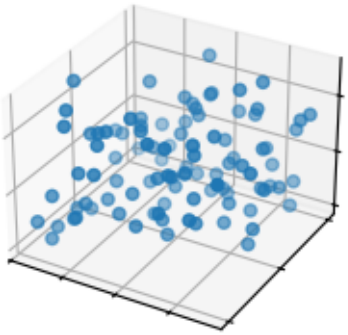
```
import matplotlib.pyplot as plt
import numpy as np

plt.style.use('_mpl-gallery-nogrid')

# make data:
np.random.seed(1)
x = np.random.uniform(-3, 3, 256)
y = np.random.uniform(-3, 3, 256)
z = (1 - x/2 + x**5 + y**3) * np.exp(-x**2 - y**2)

# plot:
fig, ax = plt.subplots()

ax.triplot(x, y)

ax.set(xlim=(-3, 3), ylim=(-3, 3))

plt.show()
```

### 5.5.5 3D and volumetric data

Plots of three-dimensional $(x, y, z)$, surface $f(x, y) = z$, and volumetric $V_{x,y,z}$ data using the *mpl_toolkits.mplot3d* library.

#### scatter(xs, ys, zs)

See *scatter*.



```
import matplotlib.pyplot as plt
import numpy as np

plt.style.use('_mpl-gallery')

# Make data
np.random.seed(19680801)
```

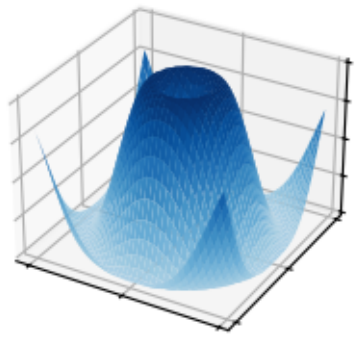
(continues on next page)

(continued from previous page)

```
n = 100
rng = np.random.default_rng()
xs = rng.uniform(23, 32, n)
ys = rng.uniform(0, 100, n)
zs = rng.uniform(-50, -25, n)

# Plot
fig, ax = plt.subplots(subplot_kw={"projection": "3d"})
ax.scatter(xs, ys, zs)

ax.set(xticklabels=[],
       yticklabels=[],
       zticklabels=[])

plt.show()
```

### plot_surface(X, Y, Z)

See *plot_surface*.



```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib import cm

plt.style.use('_mpl-gallery')

# Make data
X = np.arange(-5, 5, 0.25)
Y = np.arange(-5, 5, 0.25)
X, Y = np.meshgrid(X, Y)
R = np.sqrt(X**2 + Y**2)
Z = np.sin(R)

# Plot the surface
fig, ax = plt.subplots(subplot_kw={"projection": "3d"})
ax.plot_surface(X, Y, Z, vmin=Z.min() * 2, cmap=cm.Blues)
```

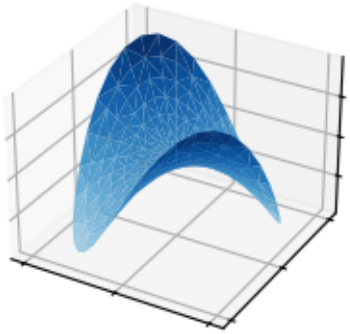
(continues on next page)

(continued from previous page)

```
ax.set(xticklabels=[],
       yticklabels=[],
       zticklabels=[])

plt.show()
```

### plot_trisurf(x, y, z)

See *plot_trisurf*.



```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib import cm

plt.style.use('_mpl-gallery')

n_radii = 8
n_angles = 36

# Make radii and angles spaces
radii = np.linspace(0.125, 1.0, n_radii)
angles = np.linspace(0, 2*np.pi, n_angles, endpoint=False)[..., np.newaxis]

# Convert polar (radii, angles) coords to cartesian (x, y) coords.
x = np.append(0, (radii*np.cos(angles)).flatten())
y = np.append(0, (radii*np.sin(angles)).flatten())
z = np.sin(-x*y)

# Plot
fig, ax = plt.subplots(subplot_kw={'projection': '3d'})
ax.plot_trisurf(x, y, z, vmin=z.min() * 2, cmap=cm.Blues)

ax.set(xticklabels=[],
       yticklabels=[],
       zticklabels=[])
```

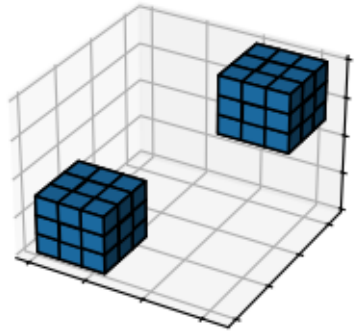
(continues on next page)

(continued from previous page)

```
plt.show()
```

### voxels([x, y, z], filled)

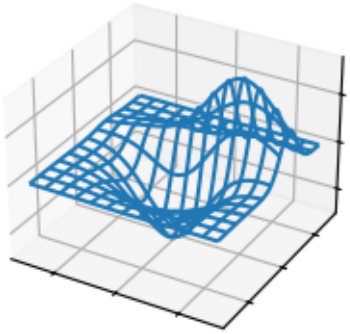
See *voxels*.



```
import matplotlib.pyplot as plt
import numpy as np

plt.style.use('_mpl-gallery')

# Prepare some coordinates
x, y, z = np.indices((8, 8, 8))

# Draw cuboids in the top left and bottom right corners
cube1 = (x < 3) & (y < 3) & (z < 3)
cube2 = (x >= 5) & (y >= 5) & (z >= 5)

# Combine the objects into a single boolean array
voxelarray = cube1 | cube2

# Plot
fig, ax = plt.subplots(subplot_kw={"projection": "3d"})
ax.voxels(voxelarray, edgecolor='k')

ax.set(xticklabels=[],
       yticklabels=[],
       zticklabels=[])

plt.show()
```

### plot_wireframe(X, Y, Z)

See *plot_wireframe*.



```
import matplotlib.pyplot as plt

from mpl_toolkits.mplot3d import axes3d

plt.style.use('_mpl-gallery')

# Make data
X, Y, Z = axes3d.get_test_data(0.05)

# Plot
fig, ax = plt.subplots(subplot_kw={"projection": "3d"})
ax.plot_wireframe(X, Y, Z, rstride=10, cstride=10)

ax.set(xticklabels=[],
       yticklabels=[],
       zticklabels=[])

plt.show()
```

CHAPTER

SIX

# EXAMPLES

For an overview of the plotting methods we provide, see *Plot types*

This page contains example plots. Click on any image to see the full image and source code.

For longer tutorials, see our *tutorials page*. You can also find *external resources* and a FAQ in our *user guide*.

**Tagging!**

You can also browse the example gallery by tags.

## 6.1 Lines, bars and markers

## 6.2 Images, contours and fields

## 6.3 Subplots, axes and figures

## 6.4 Statistics

## 6.5 Pie and polar charts

## 6.6 Text, labels and annotations

## 6.7 Color

For a description of the colormaps available in Matplotlib, see the *colormaps tutorial*.

## 6.8 Shapes and collections

## 6.9 Style sheets

## 6.10 Module - pyplot

## 6.11 Module - axes_grid1

## 6.12 Module - axisartist

## 6.13 Showcase

## 6.14 Animation

## 6.15 Event handling

Matplotlib supports *event handling* with a GUI neutral event model, so you can connect to Matplotlib events without knowledge of what user interface Matplotlib will ultimately be plugged in to. This has two advantages: the code you write will be more portable, and Matplotlib events are aware of things like data coordinate space and which axes the event occurs in so you don't have to mess with low level transformation details to go from canvas space to data space. Object picking examples are also included.

## 6.16 Miscellaneous

## 6.17 3D plotting

## 6.18 Scales

These examples cover how different scales are handled in Matplotlib.

## 6.19 Specialty plots

## 6.20 Spines

## 6.21 Ticks

## 6.22 Units

These examples cover the many representations of units in Matplotlib.

## 6.23 Embedding Matplotlib in graphical user interfaces

You can embed Matplotlib directly into a user interface application by following the embedding_in_SOMEGUI.py examples here. Currently Matplotlib supports PyQt/PySide, PyGObject, Tkinter, and wxPython.

When embedding Matplotlib in a GUI, you must use the Matplotlib API directly rather than the pylab/pyplot procedural interface, so take a look at the examples/api directory for some example code working with the API.

## 6.24 Widgets

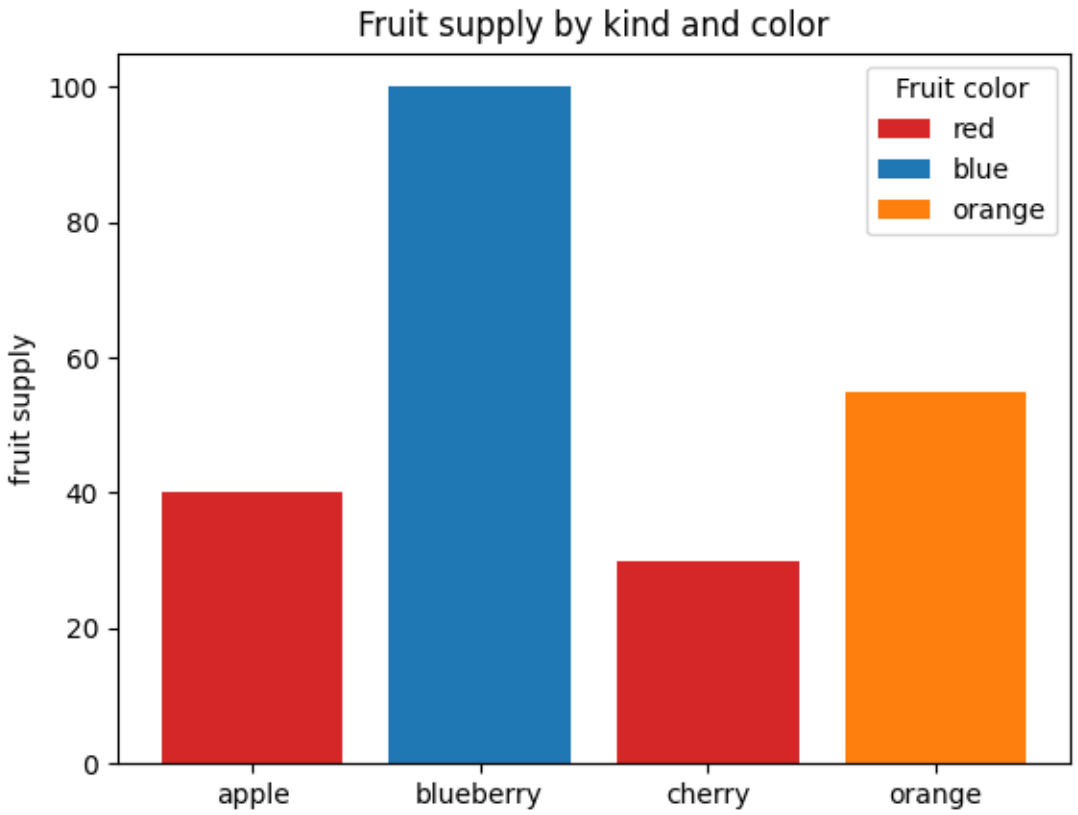
Examples of how to write primitive, but GUI agnostic, widgets in matplotlib

## 6.25 Userdemo

### 6.25.1 Lines, bars and markers

#### Bar color demo

This is an example showing how to control bar color and legend entries using the *color* and *label* parameters of `bar`. Note that labels with a preceding underscore won't show up in the legend.




```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()

fruits = ['apple', 'blueberry', 'cherry', 'orange']
counts = [40, 100, 30, 55]
bar_labels = ['red', 'blue', '_red', 'orange']
bar_colors = ['tab:red', 'tab:blue', 'tab:red', 'tab:orange']

ax.bar(fruits, counts, label=bar_labels, color=bar_colors)

ax.set_ylabel('fruit supply')
ax.set_title('Fruit supply by kind and color')
ax.legend(title='Fruit color')

plt.show()
```

### Bar Label Demo

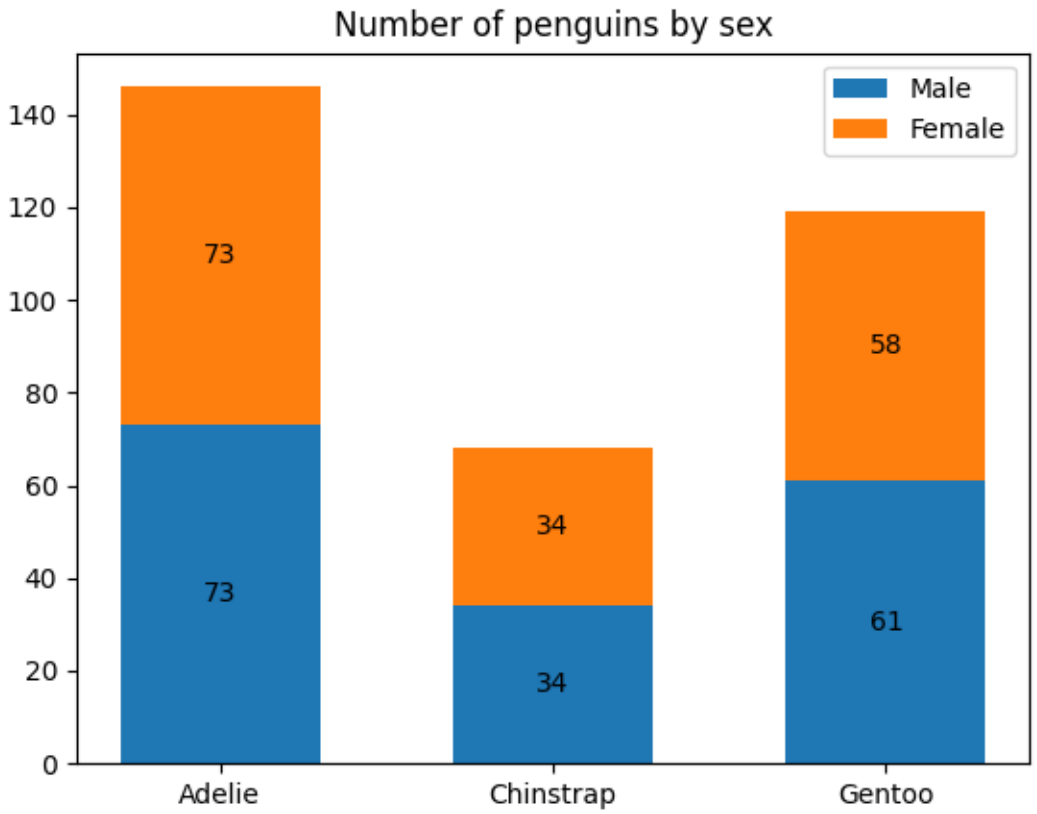
This example shows how to use the *bar_label* helper function to create bar chart labels.

See also the *grouped bar*, *stacked bar* and *horizontal bar chart* examples.

```
import matplotlib.pyplot as plt
import numpy as np
```

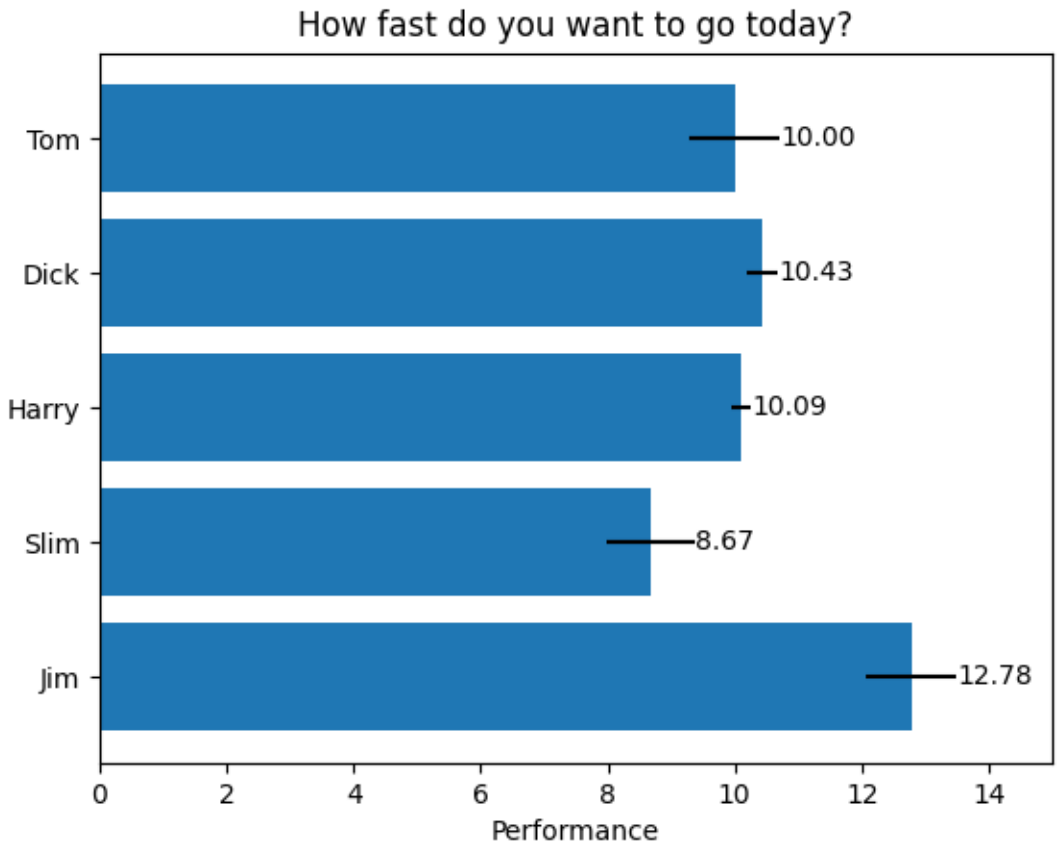
data from https://allisonhorst.github.io/palmerpenguins/

```
species = ('Adelie', 'Chinstrap', 'Gentoo')
sex_counts = {
    'Male': np.array([73, 34, 61]),
    'Female': np.array([73, 34, 58]),
}
width = 0.6  # the width of the bars: can also be len(x) sequence


fig, ax = plt.subplots()
bottom = np.zeros(3)

for sex, sex_count in sex_counts.items():
    p = ax.bar(species, sex_count, width, label=sex, bottom=bottom)
    bottom += sex_count

    ax.bar_label(p, label_type='center')

ax.set_title('Number of penguins by sex')
ax.legend()

plt.show()
```



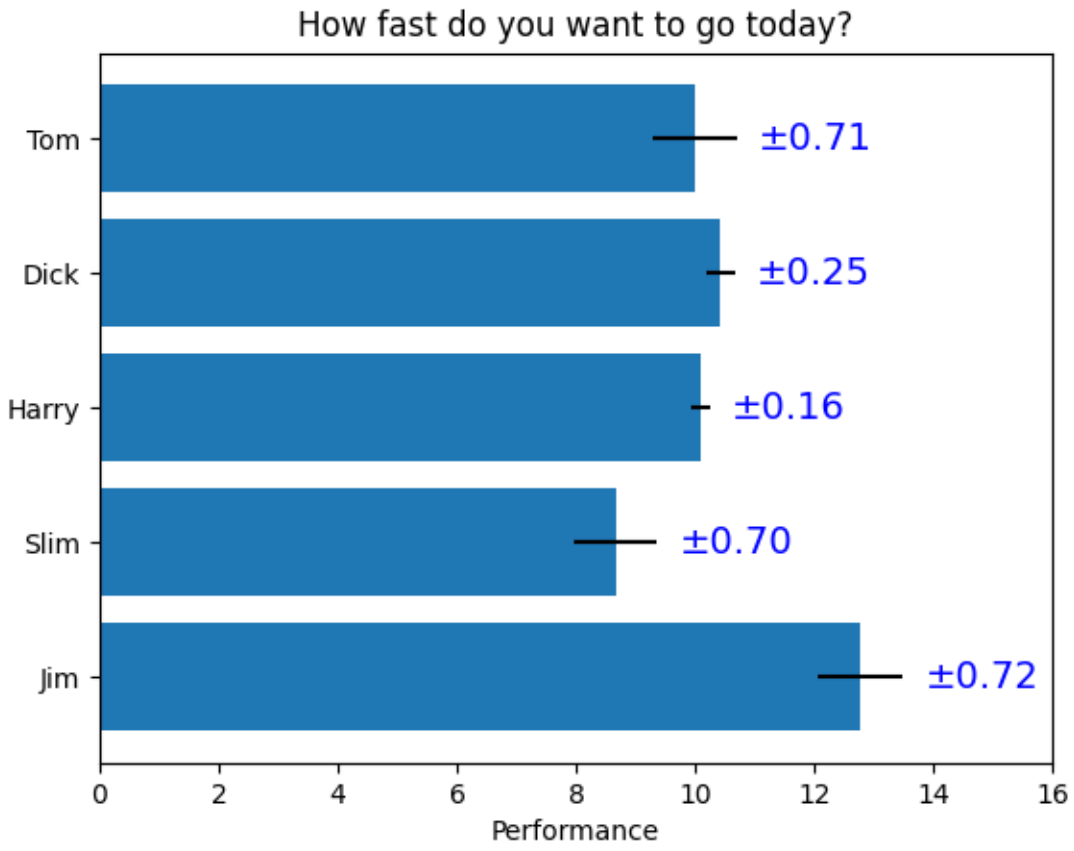

Horizontal bar chart

```
# Fixing random state for reproducibility
np.random.seed(19680801)

# Example data
people = ('Tom', 'Dick', 'Harry', 'Slim', 'Jim')
y_pos = np.arange(len(people))
performance = 3 + 10 * np.random.rand(len(people))
error = np.random.rand(len(people))

fig, ax = plt.subplots()

hbars = ax.barh(y_pos, performance, xerr=error, align='center')
ax.set_yticks(y_pos, labels=people)
ax.invert_yaxis()  # labels read top-to-bottom
ax.set_xlabel('Performance')
ax.set_title('How fast do you want to go today?')

# Label with specially formatted floats
ax.bar_label(hbars, fmt='%.2f')
ax.set_xlim(right=15)  # adjust xlim to fit labels

plt.show()
```



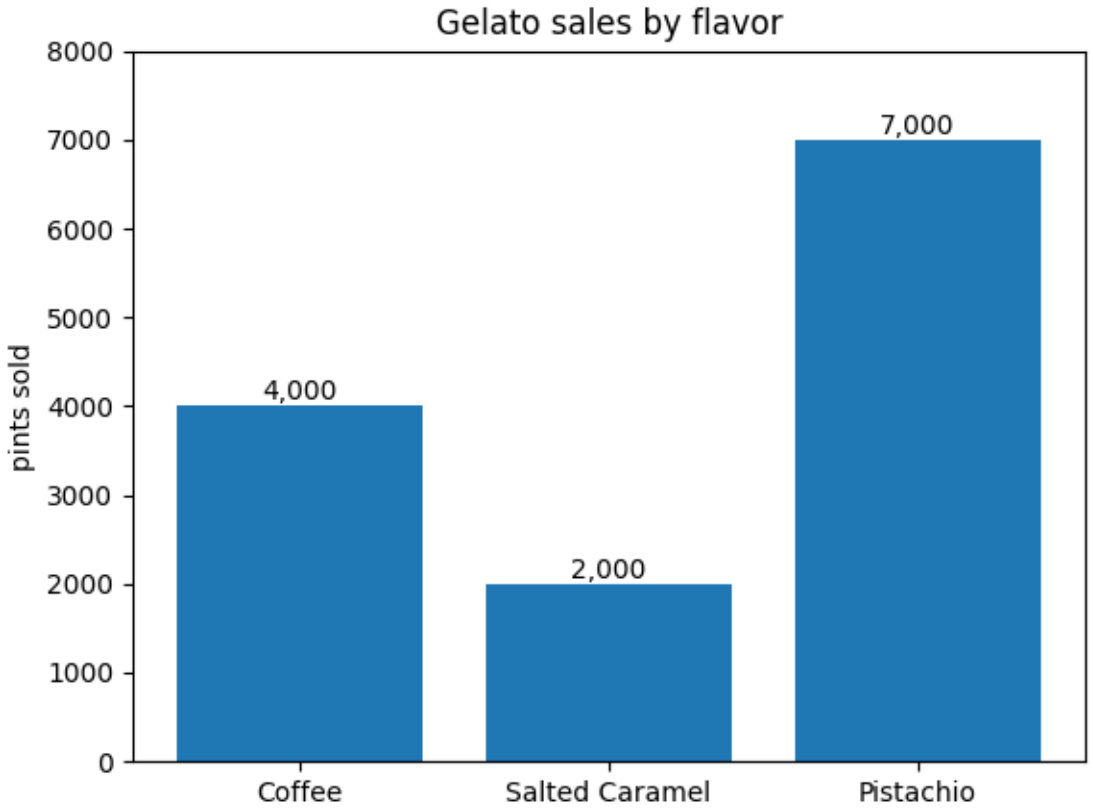

Some of the more advanced things that one can do with bar labels

```
fig, ax = plt.subplots()

hbars = ax.barh(y_pos, performance, xerr=error, align='center')
ax.set_yticks(y_pos, labels=people)
ax.invert_yaxis()  # labels read top-to-bottom
ax.set_xlabel('Performance')
ax.set_title('How fast do you want to go today?')

# Label with given captions, custom padding and annotate options
ax.bar_label(hbars, labels=[f'±{e:.2f}' for e in error],
             padding=8, color='b', fontsize=14)
ax.set_xlim(right=16)

plt.show()
```



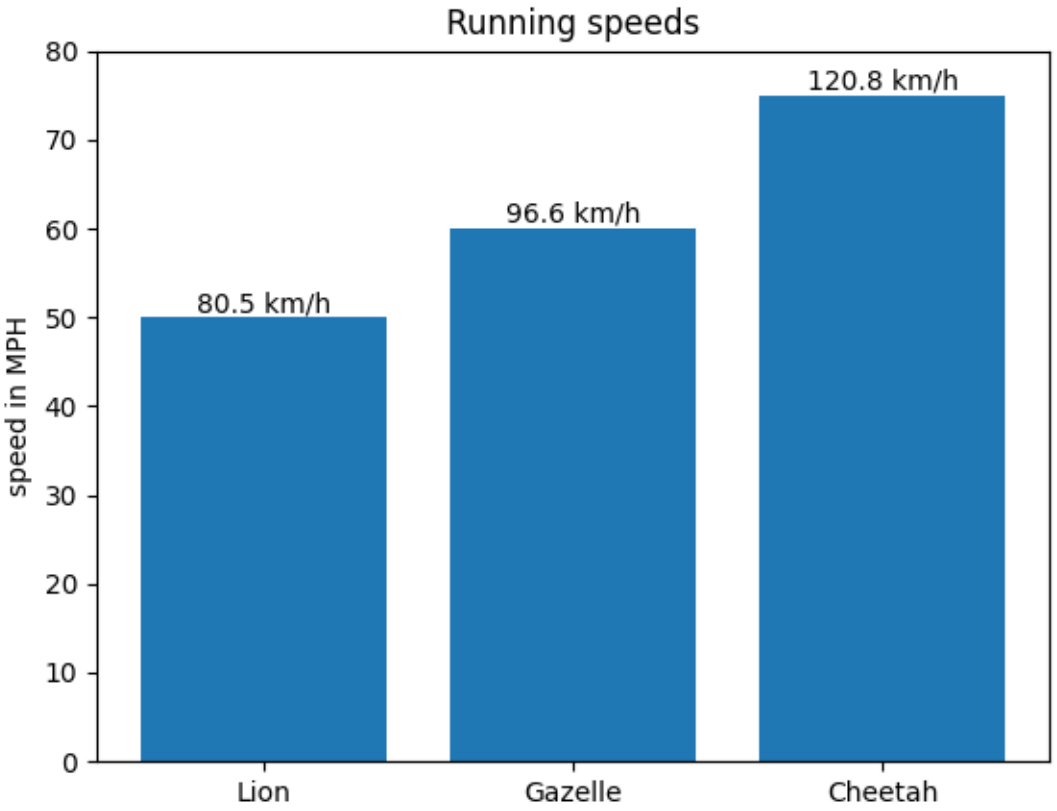

Bar labels using {}-style format string

```
fruit_names = ['Coffee', 'Salted Caramel', 'Pistachio']
fruit_counts = [4000, 2000, 7000]

fig, ax = plt.subplots()
bar_container = ax.bar(fruit_names, fruit_counts)
ax.set(ylabel='pints sold', title='Gelato sales by flavor', ylim=(0, 8000))
ax.bar_label(bar_container, fmt='{:,.0f}')
```




Bar labels using a callable

```
animal_names = ['Lion', 'Gazelle', 'Cheetah']
mph_speed = [50, 60, 75]

fig, ax = plt.subplots()
bar_container = ax.bar(animal_names, mph_speed)
ax.set(ylabel='speed in MPH', title='Running speeds', ylim=(0, 80))
ax.bar_label(bar_container, fmt=lambda x: f'{x * 1.61:.1f} km/h')
```




**References**

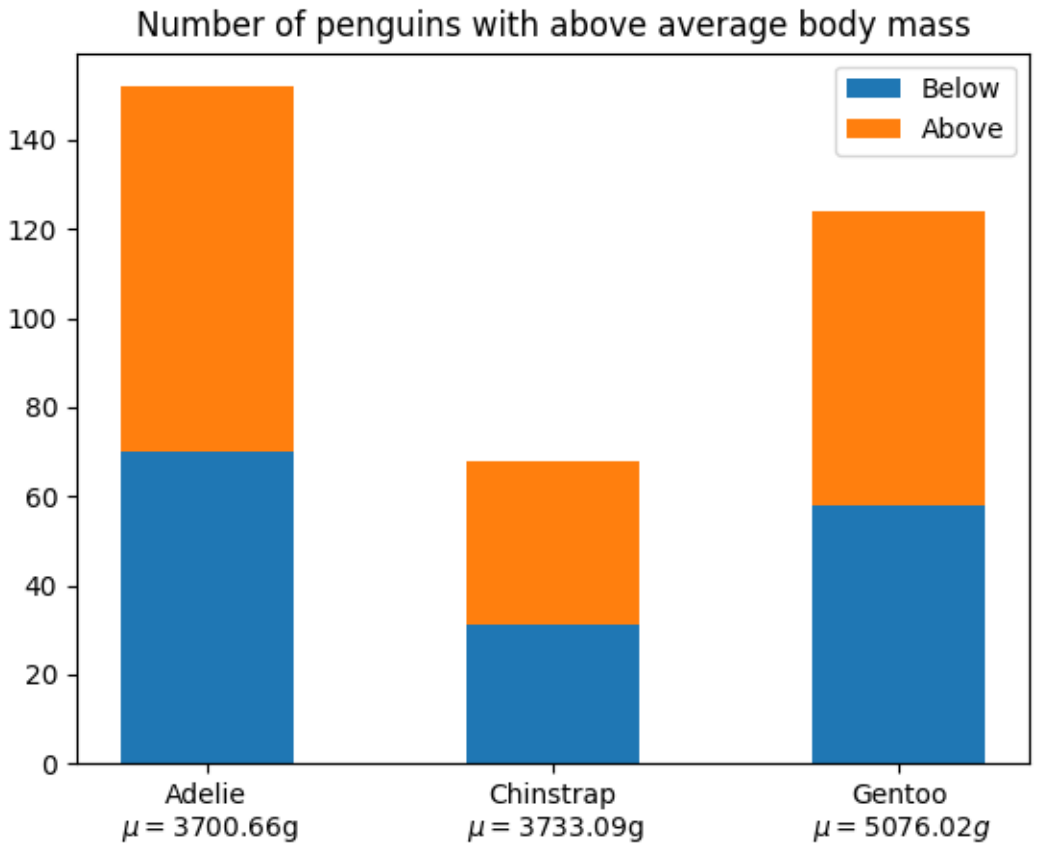
The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.bar* / *matplotlib.pyplot.bar*
- *matplotlib.axes.Axes.barh* / *matplotlib.pyplot.barh*
- *matplotlib.axes.Axes.bar_label* / *matplotlib.pyplot.bar_label*

**Total running time of the script:** (0 minutes 1.248 seconds)

### Stacked bar chart

This is an example of creating a stacked bar plot using *bar*.




```
import matplotlib.pyplot as plt
import numpy as np

# data from https://allisonhorst.github.io/palmerpenguins/

species = (
    "Adelie\n $\\mu=$3700.66g",
    "Chinstrap\n $\\mu=$3733.09g",
    "Gentoo\n $\\mu=5076.02g$",
)
weight_counts = {
    "Below": np.array([70, 31, 58]),
    "Above": np.array([82, 37, 66]),
}
width = 0.5

fig, ax = plt.subplots()
bottom = np.zeros(3)

for boolean, weight_count in weight_counts.items():
    p = ax.bar(species, weight_count, width, label=boolean, bottom=bottom)
    bottom += weight_count
```

(continues on next page)

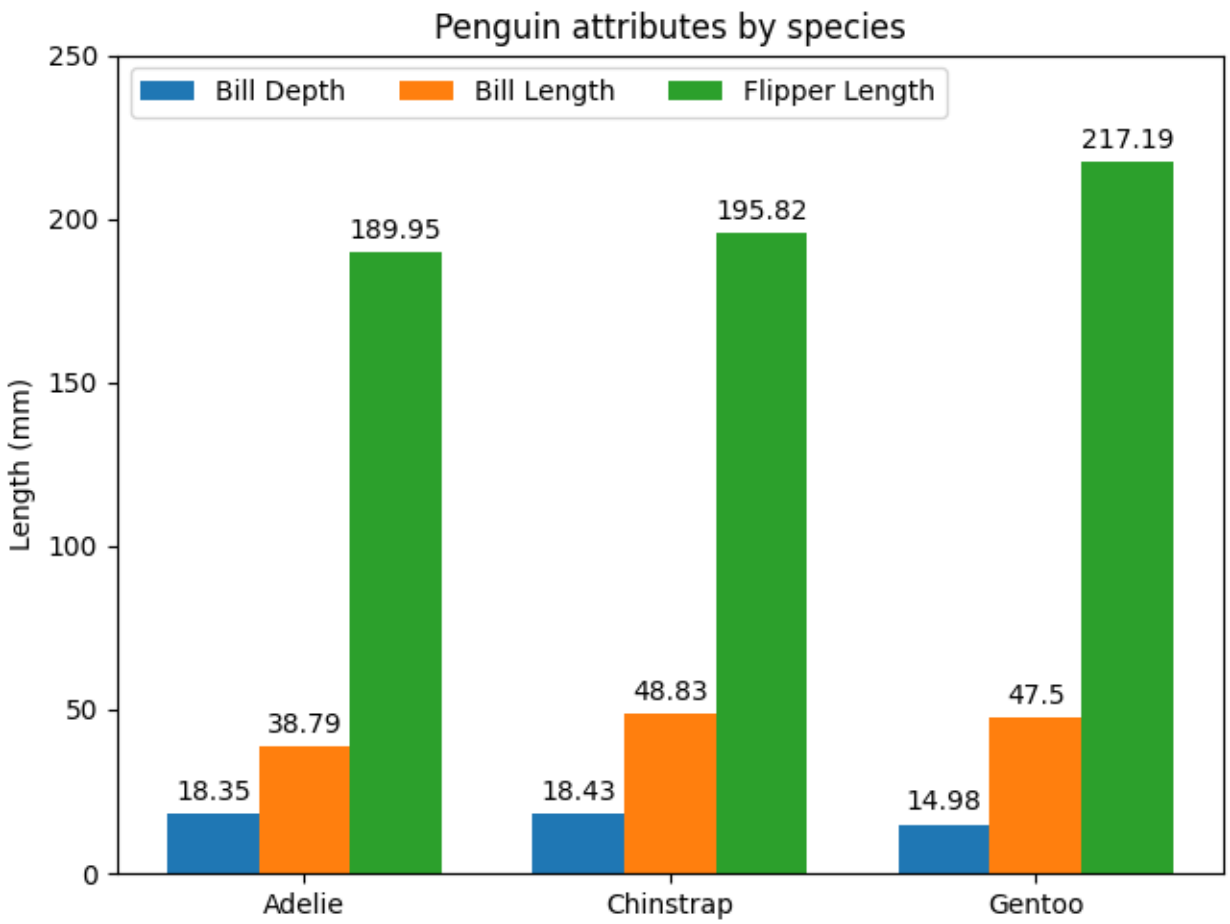
(continued from previous page)

```
ax.set_title("Number of penguins with above average body mass")
ax.legend(loc="upper right")

plt.show()
```

### Grouped bar chart with labels

This example shows a how to create a grouped bar chart and how to annotate bars with labels.

```
# data from https://allisonhorst.github.io/palmerpenguins/

import matplotlib.pyplot as plt
import numpy as np

species = ("Adelie", "Chinstrap", "Gentoo")
penguin_means = {
    'Bill Depth': (18.35, 18.43, 14.98),
    'Bill Length': (38.79, 48.83, 47.50),
    'Flipper Length': (189.95, 195.82, 217.19),
}

x = np.arange(len(species))  # the label locations
width = 0.25  # the width of the bars
multiplier = 0

fig, ax = plt.subplots(layout='constrained')

for attribute, measurement in penguin_means.items():
    offset = width * multiplier
    rects = ax.bar(x + offset, measurement, width, label=attribute)
    ax.bar_label(rects, padding=3)
    multiplier += 1

# Add some text for labels, title and custom x-axis tick labels, etc.
ax.set_ylabel('Length (mm)')
ax.set_title('Penguin attributes by species')
ax.set_xticks(x + width, species)
ax.legend(loc='upper left', ncols=3)
ax.set_ylim(0, 250)

plt.show()
```



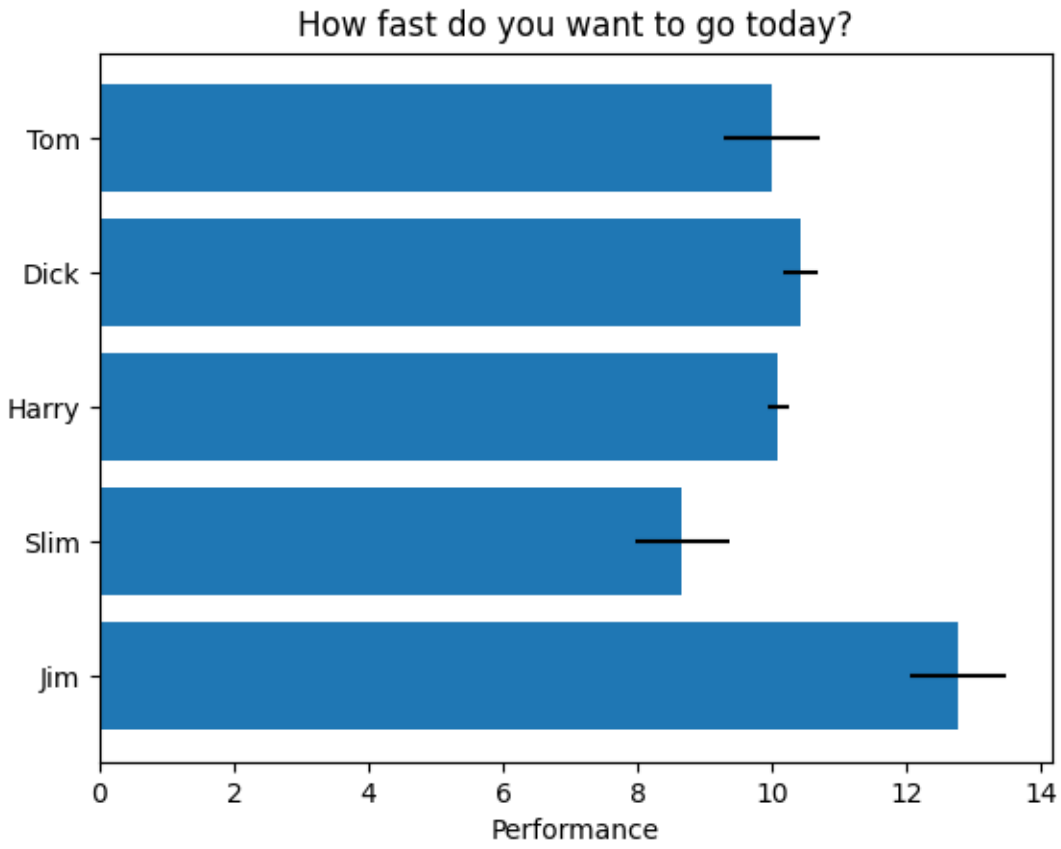

**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.bar* / *matplotlib.pyplot.bar*
- *matplotlib.axes.Axes.bar_label* / *matplotlib.pyplot.bar_label*

### Horizontal bar chart

This example showcases a simple horizontal bar chart.




```
import matplotlib.pyplot as plt
import numpy as np

# Fixing random state for reproducibility
np.random.seed(19680801)

fig, ax = plt.subplots()

# Example data
people = ('Tom', 'Dick', 'Harry', 'Slim', 'Jim')
y_pos = np.arange(len(people))
performance = 3 + 10 * np.random.rand(len(people))
error = np.random.rand(len(people))

ax.barh(y_pos, performance, xerr=error, align='center')
ax.set_yticks(y_pos, labels=people)
ax.invert_yaxis()  # labels read top-to-bottom
ax.set_xlabel('Performance')
ax.set_title('How fast do you want to go today?')
```

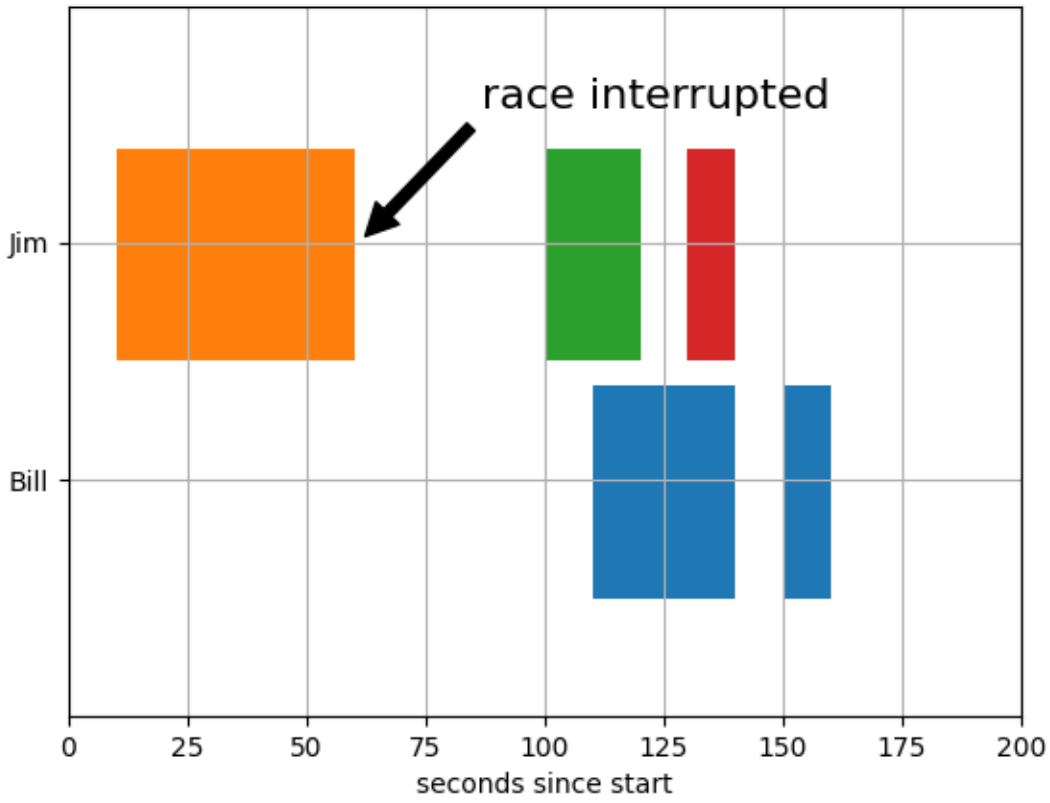
(continues on next page)

(continued from previous page)

```
plt.show()
```

### Broken Barh

Make a "broken" horizontal bar plot, i.e., one with gaps




```
import matplotlib.pyplot as plt

# Horizontal bar plot with gaps
fig, ax = plt.subplots()
ax.broken_barh([(110, 30), (150, 10)], (10, 9), facecolors='tab:blue')
ax.broken_barh([(10, 50), (100, 20), (130, 10)], (20, 9),
               facecolors=('tab:orange', 'tab:green', 'tab:red'))
ax.set_ylim(5, 35)
ax.set_xlim(0, 200)
ax.set_xlabel('seconds since start')
ax.set_yticks([15, 25], labels=['Bill', 'Jim'])     # Modify y-axis tick␣
 ↪labels
ax.grid(True)                                       # Make grid lines visible
```

(continues on next page)

(continued from previous page)

```
ax.annotate('race interrupted', (61, 25),
            xytext=(0.8, 0.9), textcoords='axes fraction',
            arrowprops=dict(facecolor='black', shrink=0.05),
            fontsize=16,
            horizontalalignment='right', verticalalignment='top')

plt.show()
```

### CapStyle

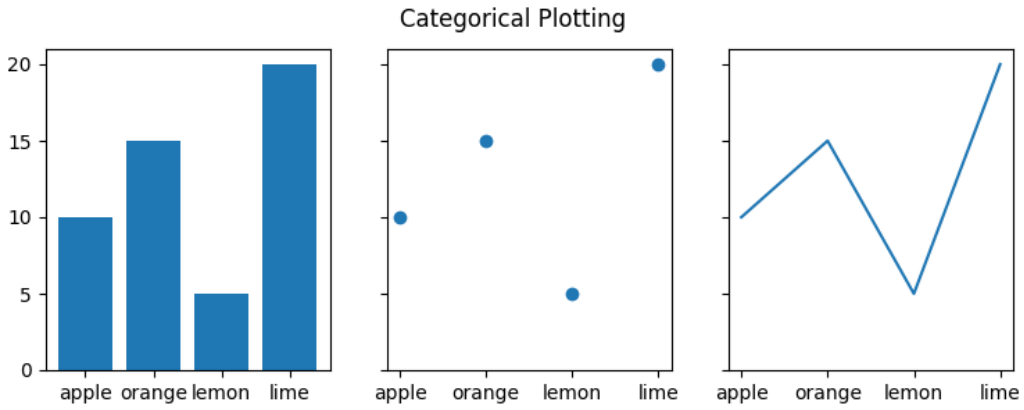
The *matplotlib._enums.CapStyle* controls how Matplotlib draws the two endpoints (caps) of an unclosed line. For more details, see the *CapStyle* docs.

```
import matplotlib.pyplot as plt

from matplotlib._enums import CapStyle

CapStyle.demo()
plt.show()
```

### Plotting categorical variables

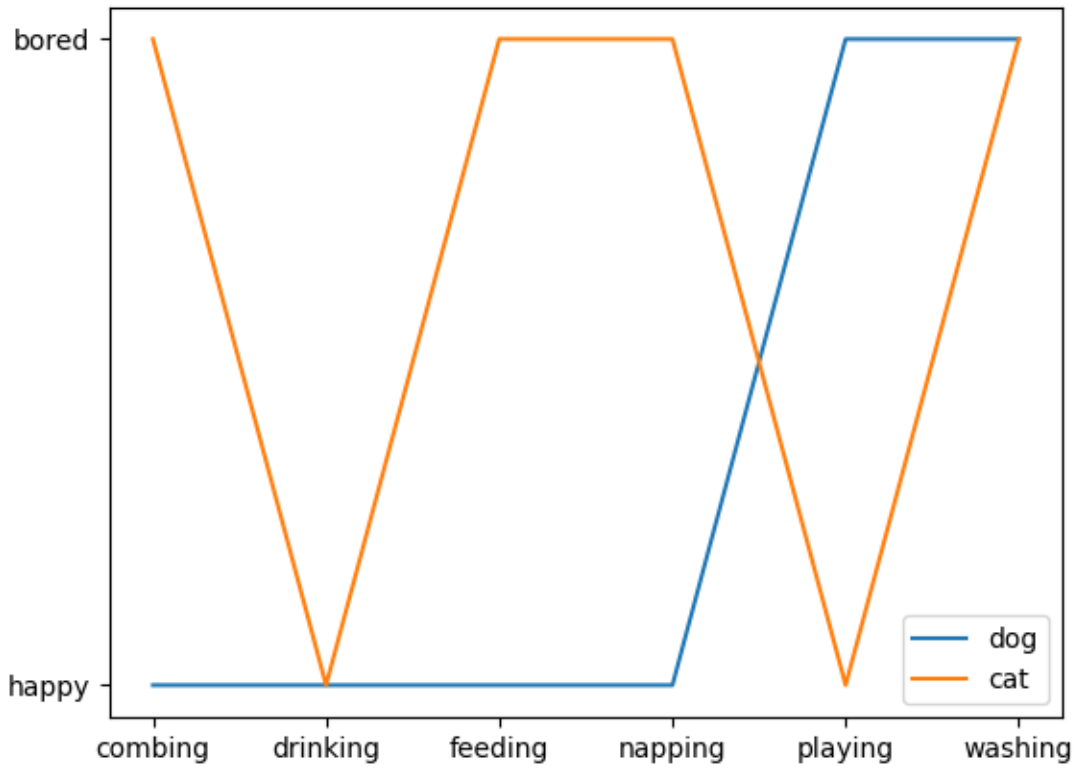
You can pass categorical values (i.e. strings) directly as x- or y-values to many plotting functions:

```
import matplotlib.pyplot as plt

data = {'apple': 10, 'orange': 15, 'lemon': 5, 'lime': 20}
names = list(data.keys())
values = list(data.values())

fig, axs = plt.subplots(1, 3, figsize=(9, 3), sharey=True)
axs[0].bar(names, values)
axs[1].scatter(names, values)
axs[2].plot(names, values)
fig.suptitle('Categorical Plotting')
```



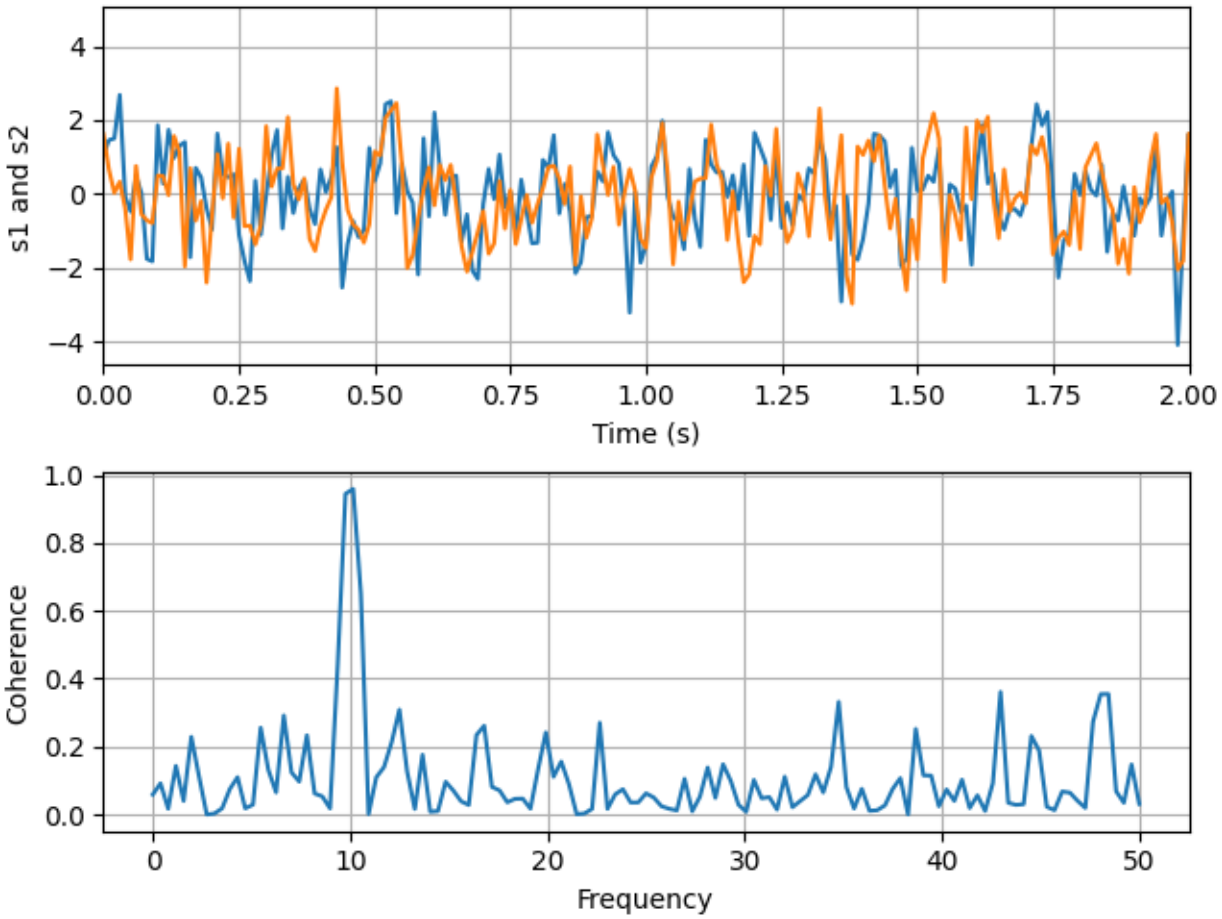

This works on both Axes:

```
cat = ["bored", "happy", "bored", "bored", "happy", "bored"]
dog = ["happy", "happy", "happy", "happy", "bored", "bored"]
activity = ["combing", "drinking", "feeding", "napping", "playing", "washing"]

fig, ax = plt.subplots()
ax.plot(activity, dog, label="dog")
ax.plot(activity, cat, label="cat")
ax.legend()

plt.show()
```




### Plotting the coherence of two signals

An example showing how to plot the coherence of two signals using `cohere`.




```
import matplotlib.pyplot as plt
import numpy as np

# Fixing random state for reproducibility
np.random.seed(19680801)

dt = 0.01
t = np.arange(0, 30, dt)
nse1 = np.random.randn(len(t))                 # white noise 1
nse2 = np.random.randn(len(t))                 # white noise 2

# Two signals with a coherent part at 10 Hz and a random part
s1 = np.sin(2 * np.pi * 10 * t) + nse1
s2 = np.sin(2 * np.pi * 10 * t) + nse2

fig, axs = plt.subplots(2, 1, layout='constrained')
axs[0].plot(t, s1, t, s2)
axs[0].set_xlim(0, 2)
axs[0].set_xlabel('Time (s)')
axs[0].set_ylabel('s1 and s2')
axs[0].grid(True)

cxy, f = axs[1].cohere(s1, s2, 256, 1. / dt)
```

(continues on next page)

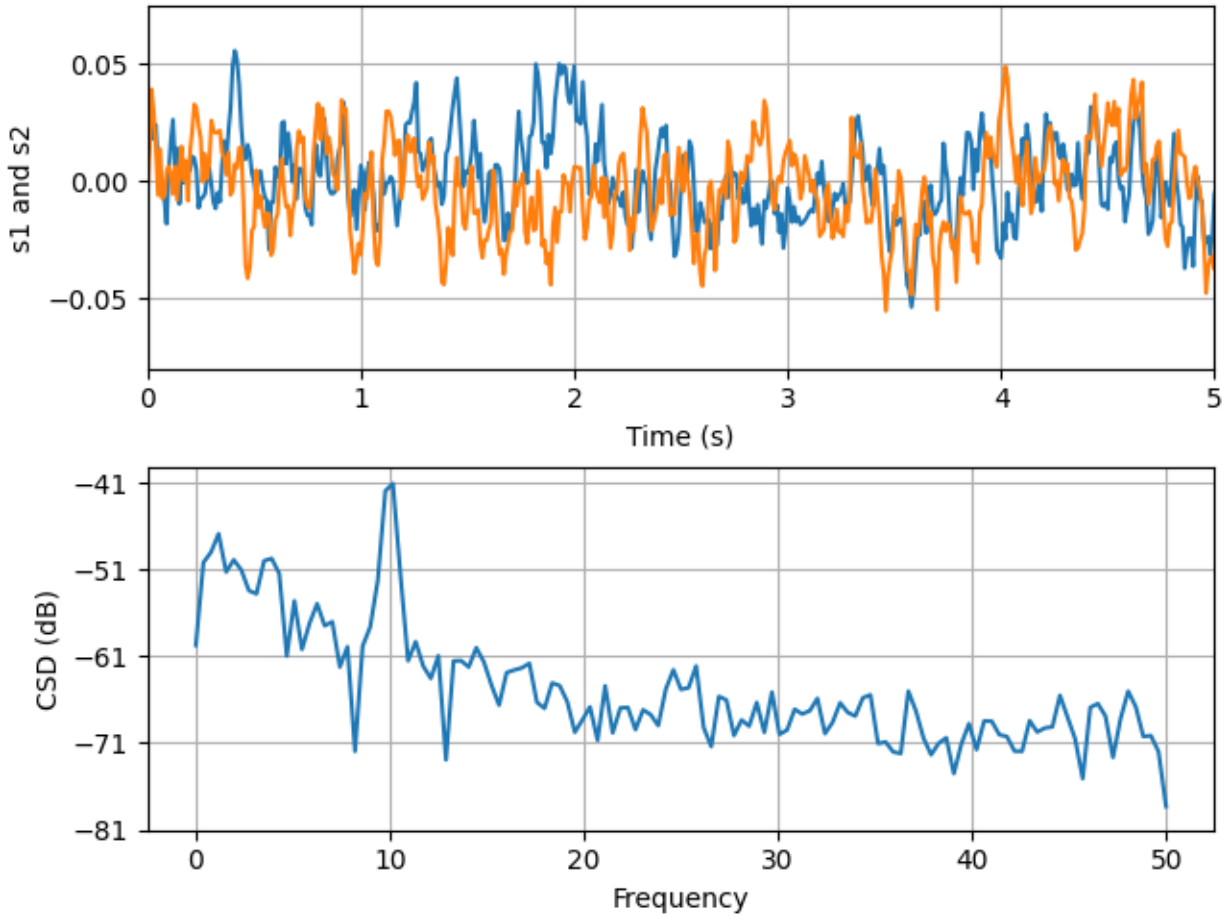
(continued from previous page)

```
axs[1].set_ylabel('Coherence')

plt.show()
```

**Total running time of the script:** (0 minutes 1.591 seconds)

### Cross spectral density (CSD)

Plot the cross spectral density (CSD) of two signals using *csd*.




```
import matplotlib.pyplot as plt
import numpy as np

fig, (ax1, ax2) = plt.subplots(2, 1, layout='constrained')

dt = 0.01
t = np.arange(0, 30, dt)

# Fixing random state for reproducibility
np.random.seed(19680801)
```

(continues on next page)

(continued from previous page)

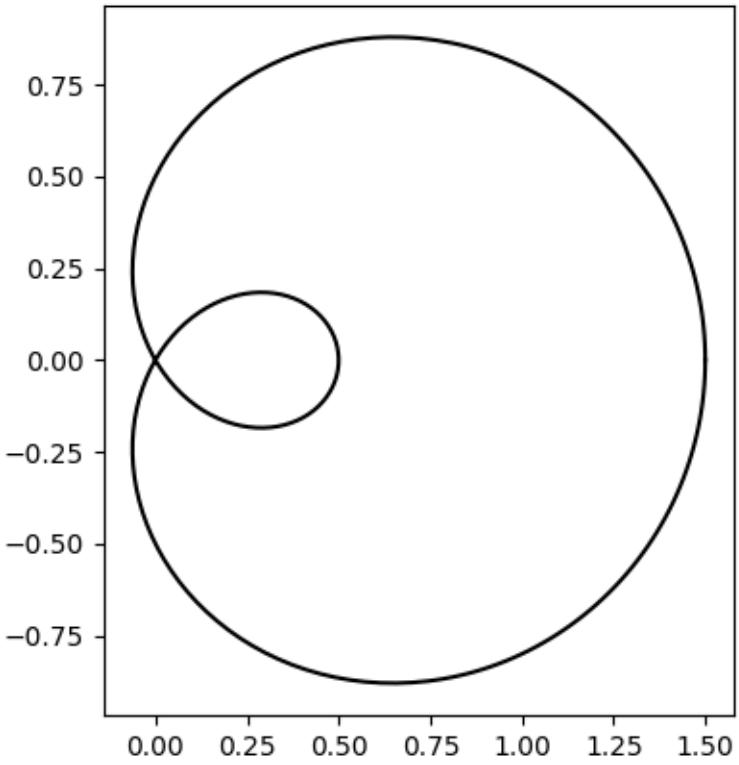
```
nse1 = np.random.randn(len(t))                 # white noise 1
nse2 = np.random.randn(len(t))                 # white noise 2
r = np.exp(-t / 0.05)

cnse1 = np.convolve(nse1, r, mode='same') * dt   # colored noise 1
cnse2 = np.convolve(nse2, r, mode='same') * dt   # colored noise 2

# two signals with a coherent part and a random part
s1 = 0.01 * np.sin(2 * np.pi * 10 * t) + cnse1
s2 = 0.01 * np.sin(2 * np.pi * 10 * t) + cnse2

ax1.plot(t, s1, t, s2)
ax1.set_xlim(0, 5)
ax1.set_xlabel('Time (s)')
ax1.set_ylabel('s1 and s2')
ax1.grid(True)

cxy, f = ax2.csd(s1, s2, 256, 1. / dt)
ax2.set_ylabel('CSD (dB)')

plt.show()
```

### Curve with error band

This example illustrates how to draw an error band around a parametrized curve.

A parametrized curve x(t), y(t) can directly be drawn using *plot*.

```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.patches import PathPatch
from matplotlib.path import Path

N = 400
t = np.linspace(0, 2 * np.pi, N)
r = 0.5 + np.cos(t)
x, y = r * np.cos(t), r * np.sin(t)

fig, ax = plt.subplots()
ax.plot(x, y, "k")
ax.set(aspect=1)
```




An error band can be used to indicate the uncertainty of the curve. In this example we assume that the error can be given as a scalar *err* that describes the uncertainty perpendicular to the curve in every point.

We visualize this error as a colored band around the path using a `PathPatch`. The patch is created from two path segments *(xp, yp)*, and *(xn, yn)* that are shifted by +/- *err* perpendicular to the curve *(x, y)*.

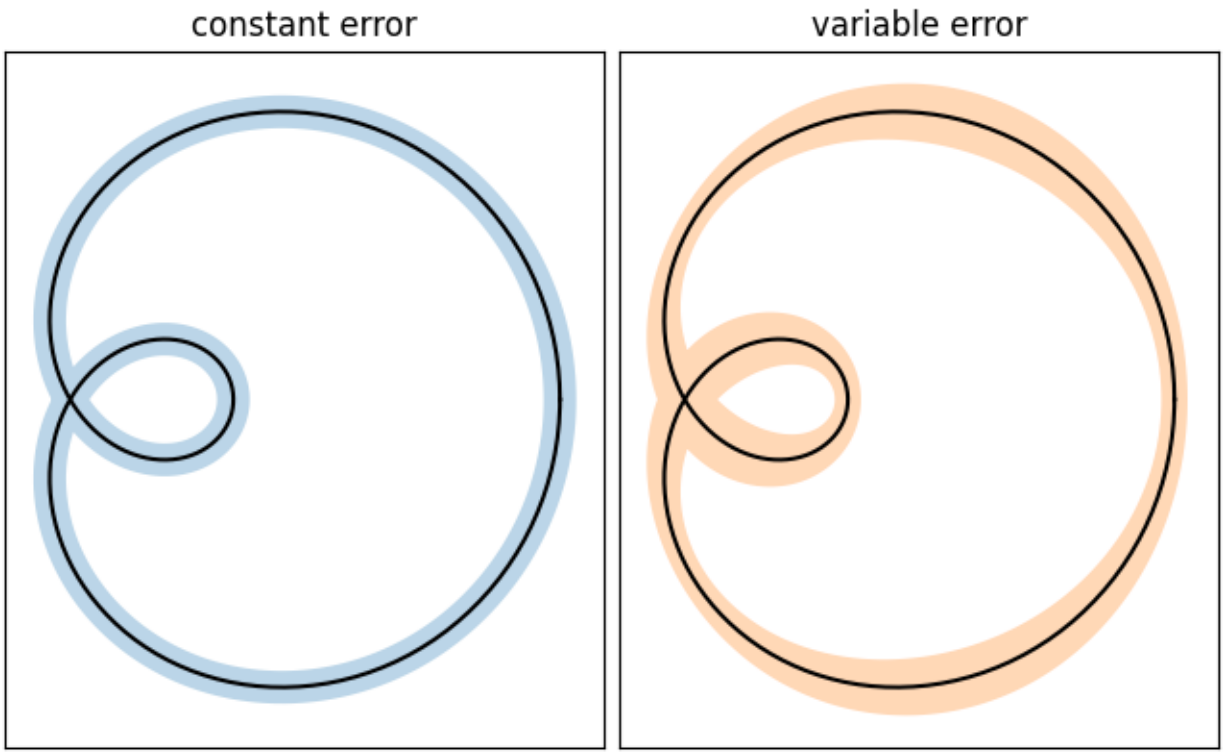
Note: This method of using a `PathPatch` is suited to arbitrary curves in 2D. If you just have a standard y-vs.-x plot, you can use the simpler `fill_between` method (see also *Filling the area between lines*).

```
def draw_error_band(ax, x, y, err, **kwargs):
    # Calculate normals via centered finite differences (except the first␣
 ↪point
    # which uses a forward difference and the last point which uses a backward
    # difference).
    dx = np.concatenate([[x[1] - x[0]], x[2:] - x[:-2], [x[-1] - x[-2]]])
    dy = np.concatenate([[y[1] - y[0]], y[2:] - y[:-2], [y[-1] - y[-2]]])
    l = np.hypot(dx, dy)
    nx = dy / l
    ny = -dx / l

    # end points of errors
    xp = x + nx * err
    yp = y + ny * err
    xn = x - nx * err
```

(continues on next page)

(continued from previous page)

```
    yn = y - ny * err

    vertices = np.block([[xp, xn[::-1]],
                         [yp, yn[::-1]]]).T
    codes = np.full(len(vertices), Path.LINETO)
    codes[0] = codes[len(xp)] = Path.MOVETO
    path = Path(vertices, codes)
    ax.add_patch(PathPatch(path, **kwargs))


_, axs = plt.subplots(1, 2, layout='constrained', sharex=True, sharey=True)
errs = [
    (axs[0], "constant error", 0.05),
    (axs[1], "variable error", 0.05 * np.sin(2 * t) ** 2 + 0.04),
]
for i, (ax, title, err) in enumerate(errs):
    ax.set(title=title, aspect=1, xticks=[], yticks=[])
    ax.plot(x, y, "k")
    draw_error_band(ax, x, y, err=err,
                    facecolor=f"C{i}", edgecolor="none", alpha=.3)

plt.show()
```



**References**

The use of the following functions, methods, classes and modules is shown in this example:

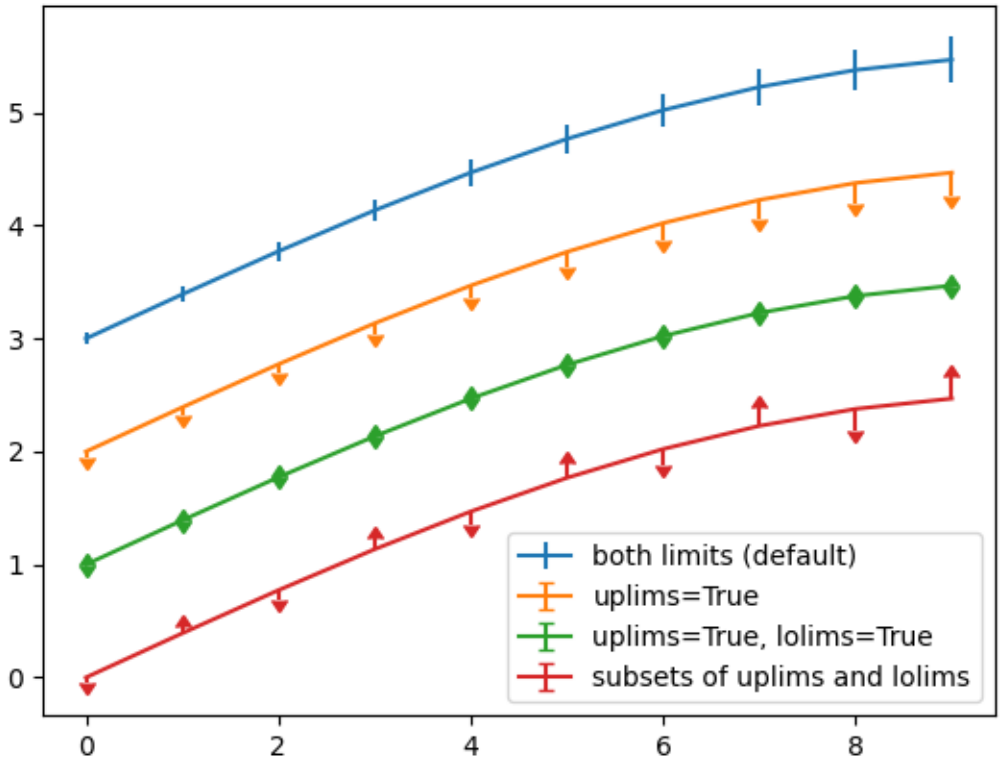
- *matplotlib.patches.PathPatch*
- *matplotlib.path.Path*

## Errorbar limit selection

Illustration of selectively drawing lower and/or upper limit symbols on errorbars using the parameters `uplims`, `lolims` of *errorbar*.

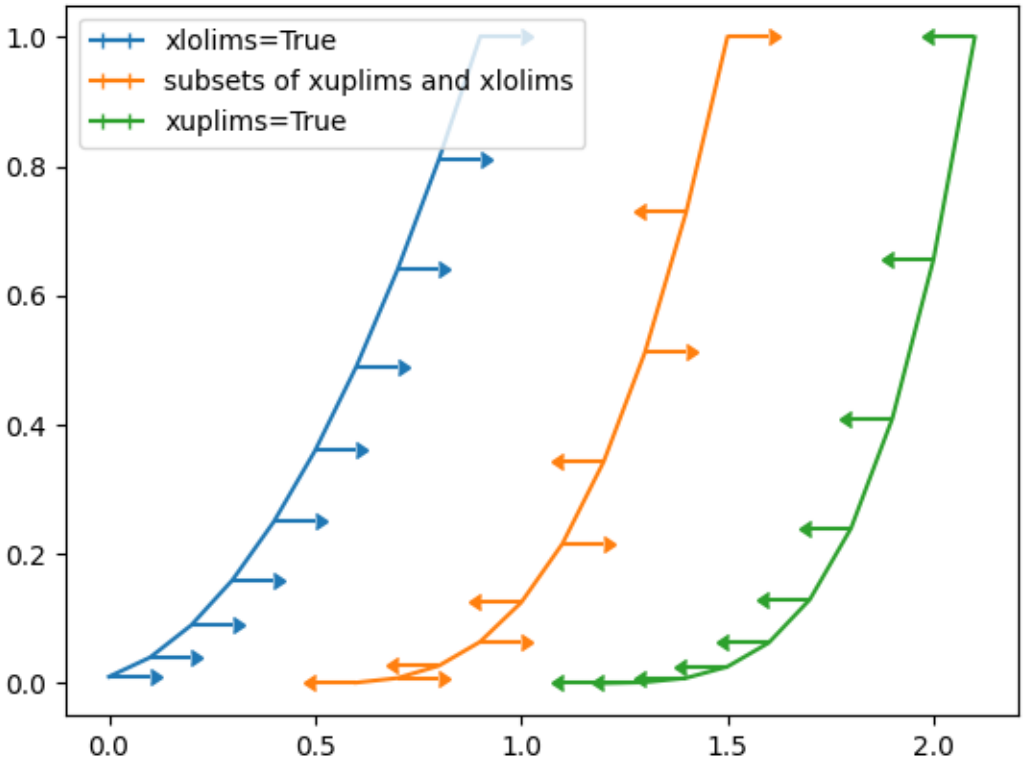
Alternatively, you can use 2xN values to draw errorbars in only one direction.

```
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure()
x = np.arange(10)
y = 2.5 * np.sin(x / 20 * np.pi)
yerr = np.linspace(0.05, 0.2, 10)

plt.errorbar(x, y + 3, yerr=yerr, label='both limits (default)')

plt.errorbar(x, y + 2, yerr=yerr, uplims=True, label='uplims=True')

plt.errorbar(x, y + 1, yerr=yerr, uplims=True, lolims=True,
             label='uplims=True, lolims=True')

upperlimits = [True, False] * 5
lowerlimits = [False, True] * 5
plt.errorbar(x, y, yerr=yerr, uplims=upperlimits, lolims=lowerlimits,
             label='subsets of uplims and lolims')

plt.legend(loc='lower right')
```




Similarly `xuplims` and `xlolims` can be used on the horizontal `xerr` errorbars.

```
fig = plt.figure()
x = np.arange(10) / 10
y = (x + 0.1)**2

plt.errorbar(x, y, xerr=0.1, xlolims=True, label='xlolims=True')
y = (x + 0.1)**3

plt.errorbar(x + 0.6, y, xerr=0.1, xuplims=upperlimits, xlolims=lowerlimits,
             label='subsets of xuplims and xlolims')

y = (x + 0.1)**4
plt.errorbar(x + 1.2, y, xerr=0.1, xuplims=True, label='xuplims=True')

plt.legend()
plt.show()
```



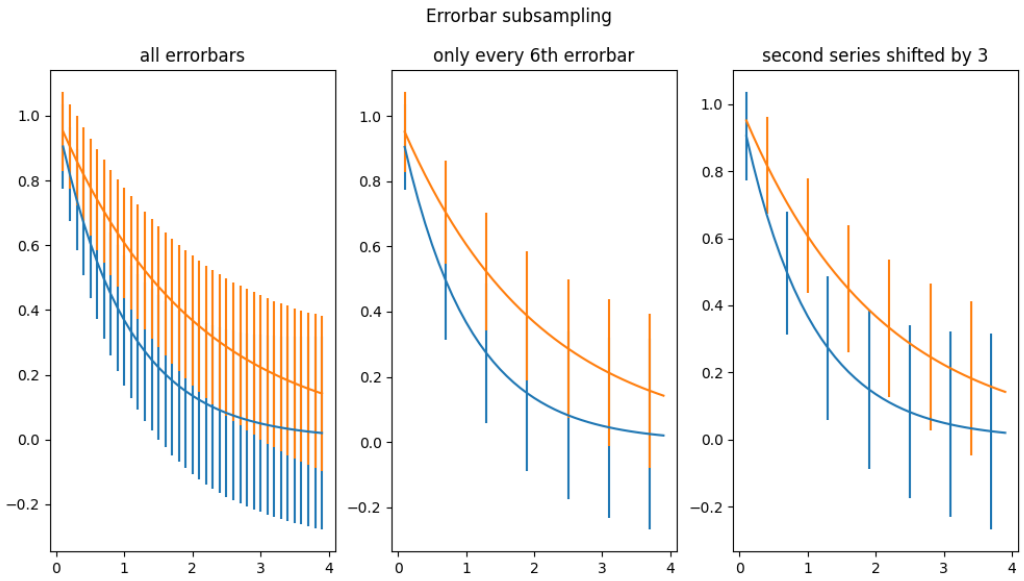

**References**

The use of the following functions, methods, classes and modules is shown in this example:

- `matplotlib.axes.Axes.errorbar` / `matplotlib.pyplot.errorbar`

## Errorbar subsampling

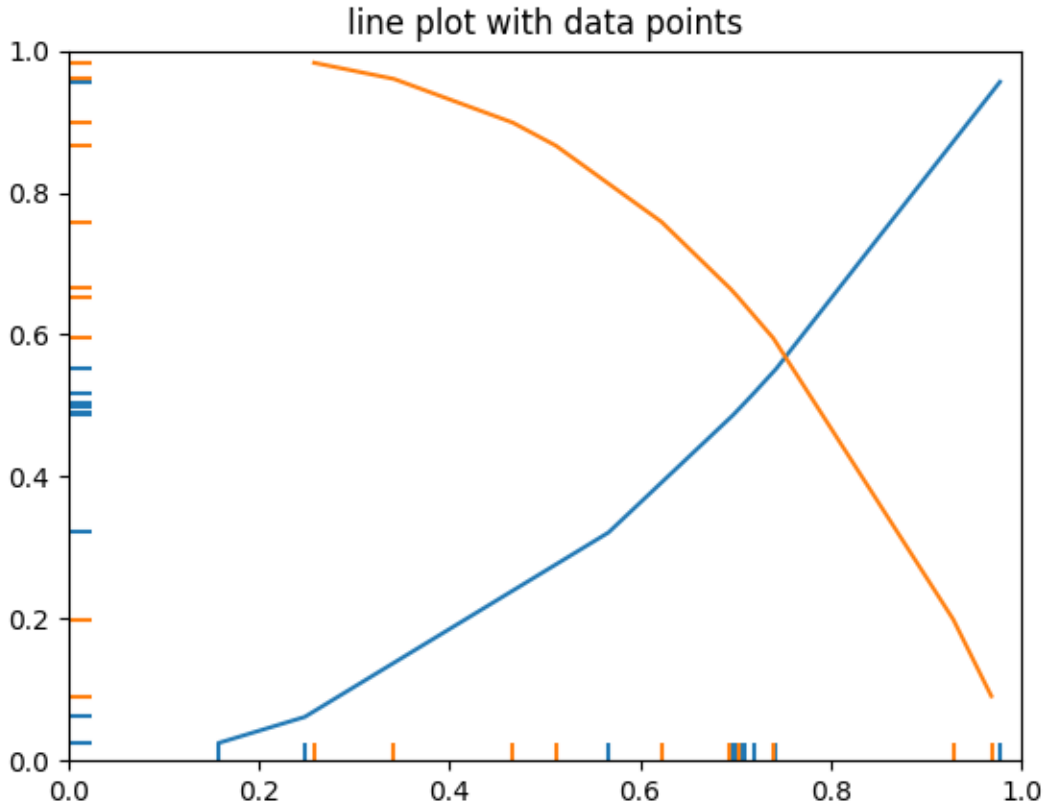
The parameter *errorevery* of `Axes.errorbar` can be used to draw error bars only on a subset of data points. This is particularly useful if there are many data points with similar errors.




```
import matplotlib.pyplot as plt
import numpy as np

# example data
x = np.arange(0.1, 4, 0.1)
y1 = np.exp(-1.0 * x)
y2 = np.exp(-0.5 * x)

# example variable error bar values
y1err = 0.1 + 0.1 * np.sqrt(x)
y2err = 0.1 + 0.1 * np.sqrt(x/2)


fig, (ax0, ax1, ax2) = plt.subplots(nrows=1, ncols=3, sharex=True,
                                    figsize=(12, 6))

ax0.set_title('all errorbars')
ax0.errorbar(x, y1, yerr=y1err)
ax0.errorbar(x, y2, yerr=y2err)

ax1.set_title('only every 6th errorbar')
ax1.errorbar(x, y1, yerr=y1err, errorevery=6)
ax1.errorbar(x, y2, yerr=y2err, errorevery=6)

ax2.set_title('second series shifted by 3')
ax2.errorbar(x, y1, yerr=y1err, errorevery=(0, 6))
ax2.errorbar(x, y2, yerr=y2err, errorevery=(3, 6))

fig.suptitle('Errorbar subsampling')
plt.show()
```

### EventCollection Demo

Plot two curves, then use *EventCollection*s to mark the locations of the x and y data points on the respective Axes for each curve.




```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.collections import EventCollection

# Fixing random state for reproducibility
np.random.seed(19680801)

# create random data
xdata = np.random.random([2, 10])

# split the data into two parts
xdata1 = xdata[0, :]
xdata2 = xdata[1, :]

# sort the data so it makes clean curves
xdata1.sort()
xdata2.sort()
```

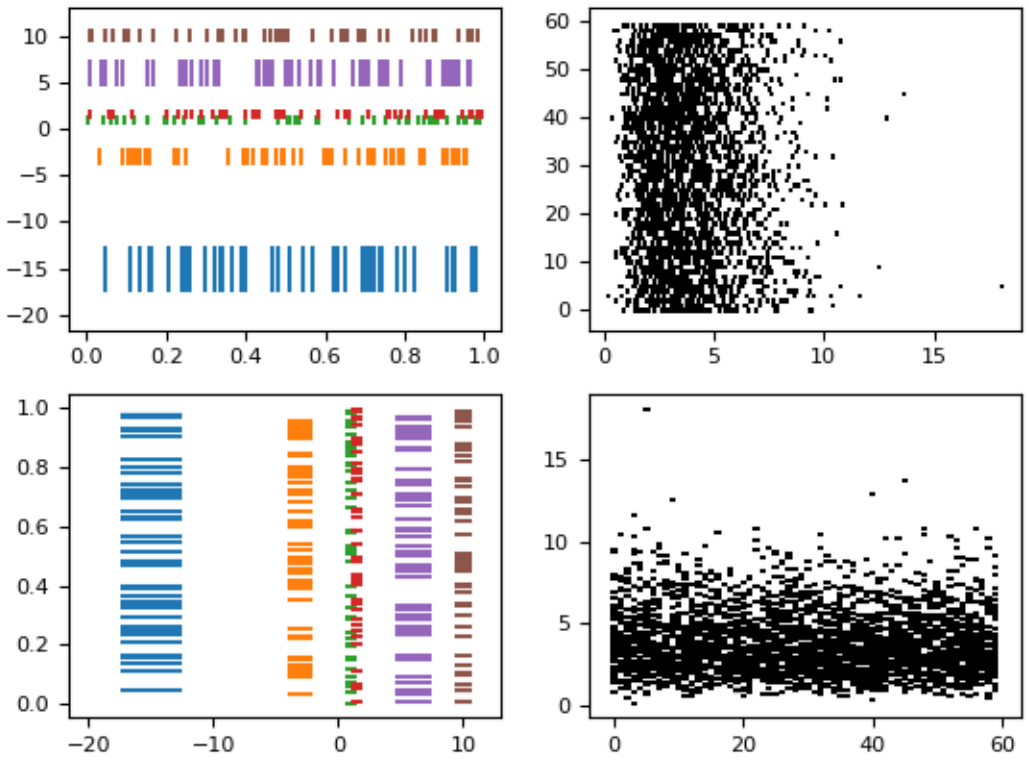
(continues on next page)

(continued from previous page)

```
# create some y data points
ydata1 = xdata1 ** 2
ydata2 = 1 - xdata2 ** 3

# plot the data
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
ax.plot(xdata1, ydata1, color='tab:blue')
ax.plot(xdata2, ydata2, color='tab:orange')

# create the events marking the x data points
xevents1 = EventCollection(xdata1, color='tab:blue', linelength=0.05)
xevents2 = EventCollection(xdata2, color='tab:orange', linelength=0.05)

# create the events marking the y data points
yevents1 = EventCollection(ydata1, color='tab:blue', linelength=0.05,
                           orientation='vertical')
yevents2 = EventCollection(ydata2, color='tab:orange', linelength=0.05,
                           orientation='vertical')

# add the events to the axis
ax.add_collection(xevents1)
ax.add_collection(xevents2)
ax.add_collection(yevents1)
ax.add_collection(yevents2)

# set the limits
ax.set_xlim([0, 1])
ax.set_ylim([0, 1])

ax.set_title('line plot with data points')

# display the plot
plt.show()
```

### Eventplot demo

An *eventplot* showing sequences of events with various line properties. The plot is shown in both horizontal and vertical orientations.




```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib

matplotlib.rcParams['font.size'] = 8.0

# Fixing random state for reproducibility
np.random.seed(19680801)


# create random data
data1 = np.random.random([6, 50])

# set different colors for each set of positions
colors1 = [f'C{i}' for i in range(6)]

# set different line properties for each set of positions
# note that some overlap
lineoffsets1 = [-15, -3, 1, 1.5, 6, 10]
linelengths1 = [5, 2, 1, 1, 3, 1.5]

fig, axs = plt.subplots(2, 2)
```

(continues on next page)

(continued from previous page)

```
# create a horizontal plot
axs[0, 0].eventplot(data1, colors=colors1, lineoffsets=lineoffsets1,
                    linelengths=linelengths1)

# create a vertical plot
axs[1, 0].eventplot(data1, colors=colors1, lineoffsets=lineoffsets1,
                    linelengths=linelengths1, orientation='vertical')

# create another set of random data.
# the gamma distribution is only used for aesthetic purposes
data2 = np.random.gamma(4, size=[60, 50])

# use individual values for the parameters this time
# these values will be used for all data sets (except lineoffsets2, which
# sets the increment between each data set in this usage)
colors2 = 'black'
lineoffsets2 = 1
linelengths2 = 1

# create a horizontal plot
axs[0, 1].eventplot(data2, colors=colors2, lineoffsets=lineoffsets2,
                    linelengths=linelengths2)


# create a vertical plot
axs[1, 1].eventplot(data2, colors=colors2, lineoffsets=lineoffsets2,
                    linelengths=linelengths2, orientation='vertical')

plt.show()
```

### Filled polygon

*fill()* draws a filled polygon based on lists of point coordinates *x*, *y*.

This example uses the Koch snowflake as an example polygon.

```
import matplotlib.pyplot as plt
import numpy as np


def koch_snowflake(order, scale=10):
    """
    Return two lists x, y of point coordinates of the Koch snowflake.

    Parameters
    ----------
    order : int
        The recursion depth.
    scale : float
```

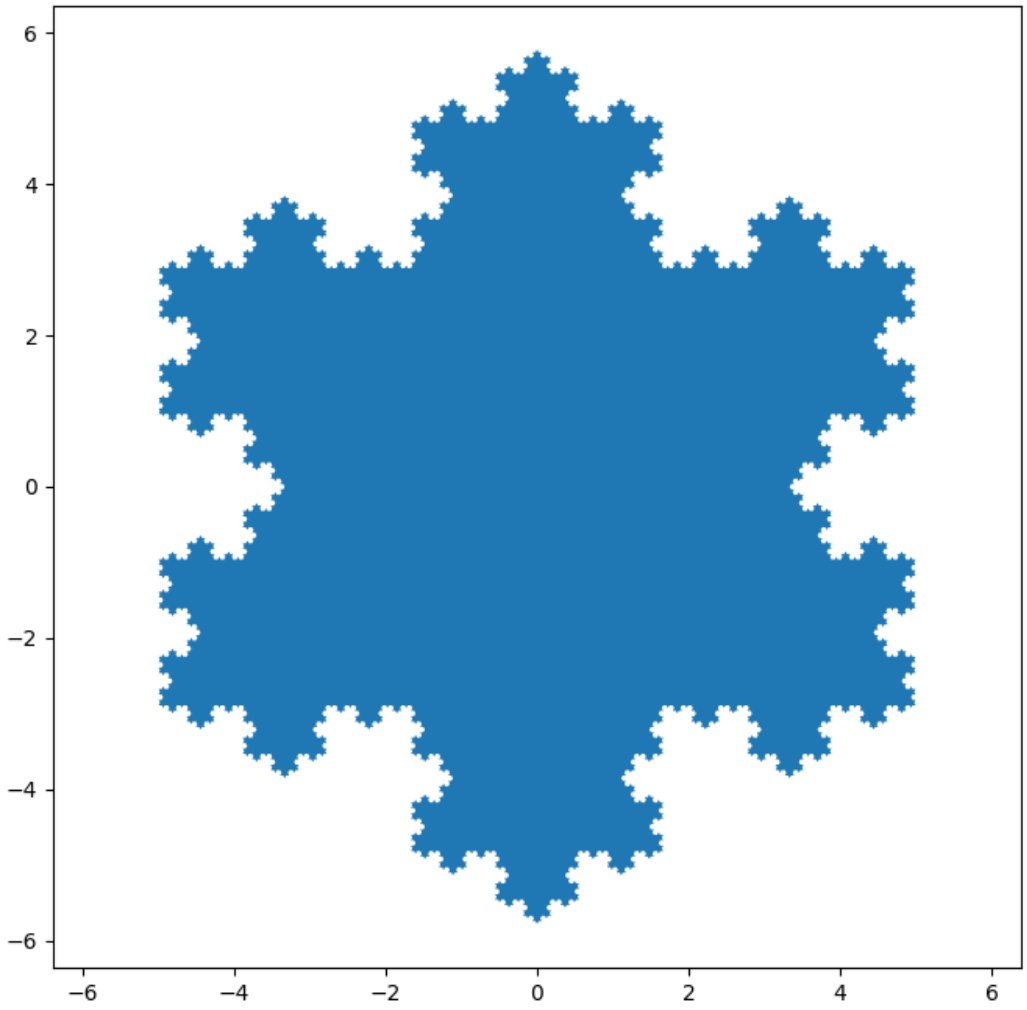
(continues on next page)

(continued from previous page)

```
        The extent of the snowflake (edge length of the base triangle).
    """
    def _koch_snowflake_complex(order):
        if order == 0:
            # initial triangle
            angles = np.array([0, 120, 240]) + 90
            return scale / np.sqrt(3) * np.exp(np.deg2rad(angles) * 1j)
        else:
            ZR = 0.5 - 0.5j * np.sqrt(3) / 3

            p1 = _koch_snowflake_complex(order - 1)  # start points
            p2 = np.roll(p1, shift=-1)  # end points
            dp = p2 - p1  # connection vectors

            new_points = np.empty(len(p1) * 4, dtype=np.complex128)
            new_points[::4] = p1
            new_points[1::4] = p1 + dp / 3
            new_points[2::4] = p1 + dp * ZR
            new_points[3::4] = p1 + dp / 3 * 2
            return new_points

    points = _koch_snowflake_complex(order)
    x, y = points.real, points.imag
    return x, y
```

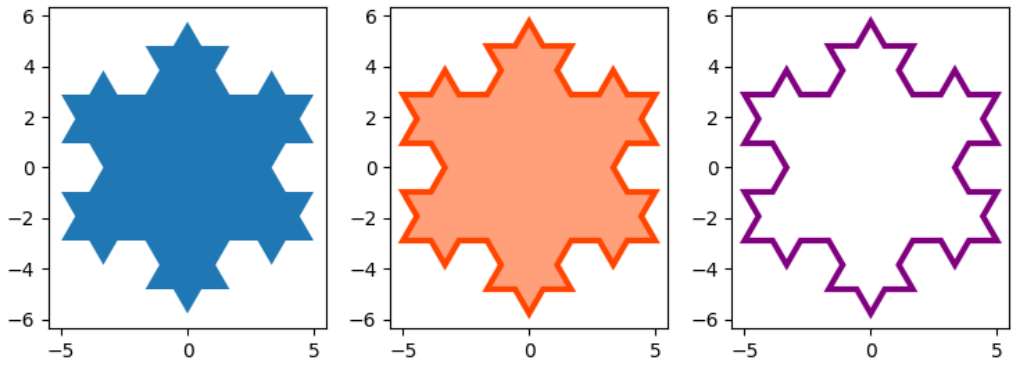
Basic usage:

```
x, y = koch_snowflake(order=5)

plt.figure(figsize=(8, 8))
plt.axis('equal')
plt.fill(x, y)
plt.show()
```




Use keyword arguments *facecolor* and *edgecolor* to modify the colors of the polygon. Since the *linewidth* of the edge is 0 in the default Matplotlib style, we have to set it as well for the edge to become visible.

```
x, y = koch_snowflake(order=2)

fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(9, 3),
                                    subplot_kw={'aspect': 'equal'})
ax1.fill(x, y)
ax2.fill(x, y, facecolor='lightsalmon', edgecolor='orangered', linewidth=3)
ax3.fill(x, y, facecolor='none', edgecolor='purple', linewidth=3)

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.fill* / *matplotlib.pyplot.fill*
- *matplotlib.axes.Axes.axis* / *matplotlib.pyplot.axis*

## Fill Between and Alpha

The *fill_between* function generates a shaded region between a min and max boundary that is useful for illustrating ranges. It has a very handy `where` argument to combine filling with logical ranges, e.g., to just fill in a curve over some threshold value.

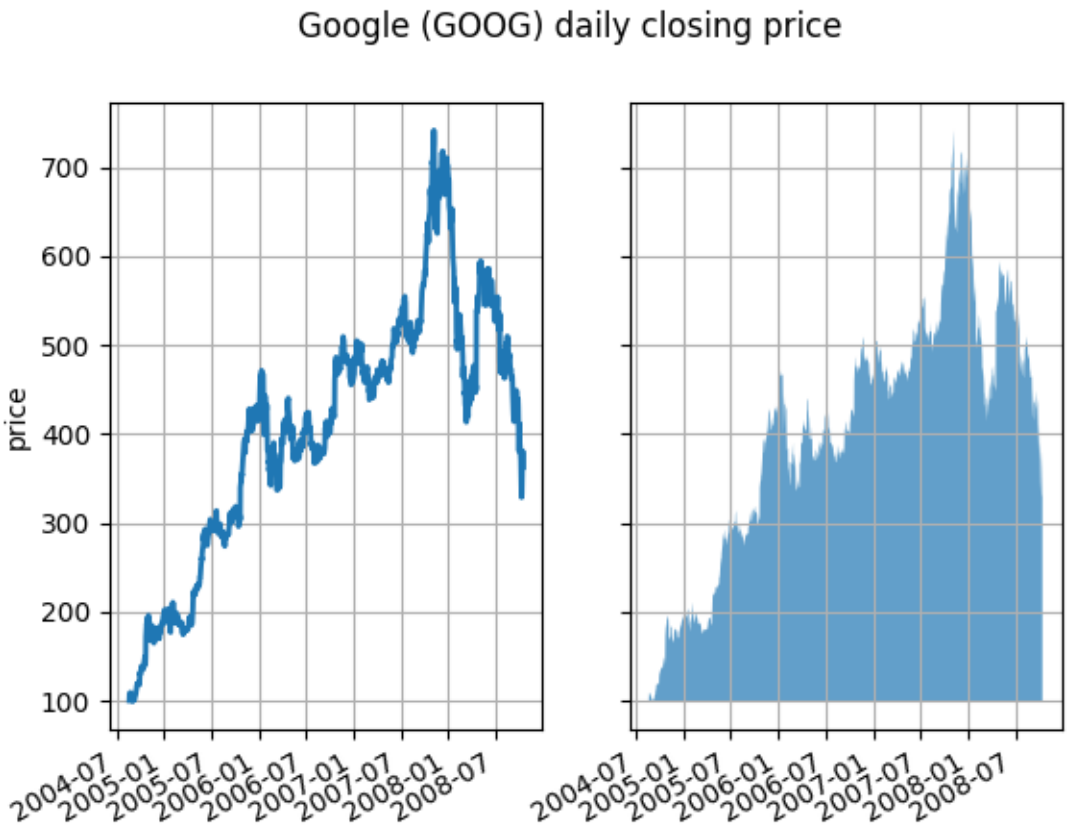
At its most basic level, `fill_between` can be used to enhance a graph's visual appearance. Let's compare two graphs of financial data with a simple line plot on the left and a filled line on the right.

```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib.cbook as cbook

# load up some sample financial data
r = cbook.get_sample_data('goog.npz')['price_data']
# create two subplots with the shared x and y axes
fig, (ax1, ax2) = plt.subplots(1, 2, sharex=True, sharey=True)

pricemin = r["close"].min()

ax1.plot(r["date"], r["close"], lw=2)
ax2.fill_between(r["date"], pricemin, r["close"], alpha=0.7)

for ax in ax1, ax2:
    ax.grid(True)
    ax.label_outer()
```

(continues on next page)

(continued from previous page)

```
ax1.set_ylabel('price')

fig.suptitle('Google (GOOG) daily closing price')
fig.autofmt_xdate()
```




The alpha channel is not necessary here, but it can be used to soften colors for more visually appealing plots. In other examples, as we'll see below, the alpha channel is functionally useful as the shaded regions can overlap and alpha allows you to see both. Note that the postscript format does not support alpha (this is a postscript limitation, not a matplotlib limitation), so when using alpha save your figures in PNG, PDF or SVG.

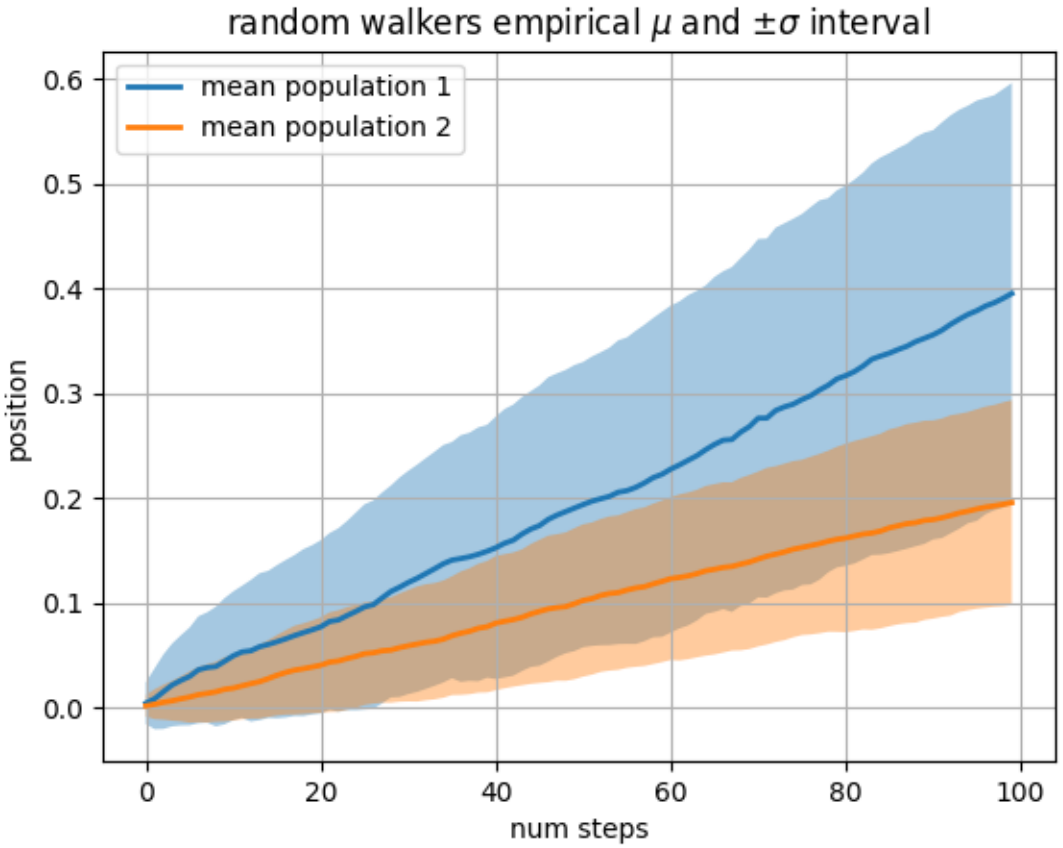
Our next example computes two populations of random walkers with a different mean and standard deviation of the normal distributions from which the steps are drawn. We use filled regions to plot +/- one standard deviation of the mean position of the population. Here the alpha channel is useful, not just aesthetic.

```
# Fixing random state for reproducibility
np.random.seed(19680801)

Nsteps, Nwalkers = 100, 250
t = np.arange(Nsteps)

# an (Nsteps x Nwalkers) array of random walk steps
```

(continues on next page)

(continued from previous page)

```
S1 = 0.004 + 0.02*np.random.randn(Nsteps, Nwalkers)
S2 = 0.002 + 0.01*np.random.randn(Nsteps, Nwalkers)

# an (Nsteps x Nwalkers) array of random walker positions
X1 = S1.cumsum(axis=0)
X2 = S2.cumsum(axis=0)


# Nsteps length arrays empirical means and standard deviations of both
# populations over time
mu1 = X1.mean(axis=1)
sigma1 = X1.std(axis=1)
mu2 = X2.mean(axis=1)
sigma2 = X2.std(axis=1)

# plot it!
fig, ax = plt.subplots(1)
ax.plot(t, mu1, lw=2, label='mean population 1')
ax.plot(t, mu2, lw=2, label='mean population 2')
ax.fill_between(t, mu1+sigma1, mu1-sigma1, facecolor='C0', alpha=0.4)
ax.fill_between(t, mu2+sigma2, mu2-sigma2, facecolor='C1', alpha=0.4)
ax.set_title(r'random walkers empirical $\mu$ and $\pm \sigma$ interval')
ax.legend(loc='upper left')
ax.set_xlabel('num steps')
ax.set_ylabel('position')
ax.grid()
```



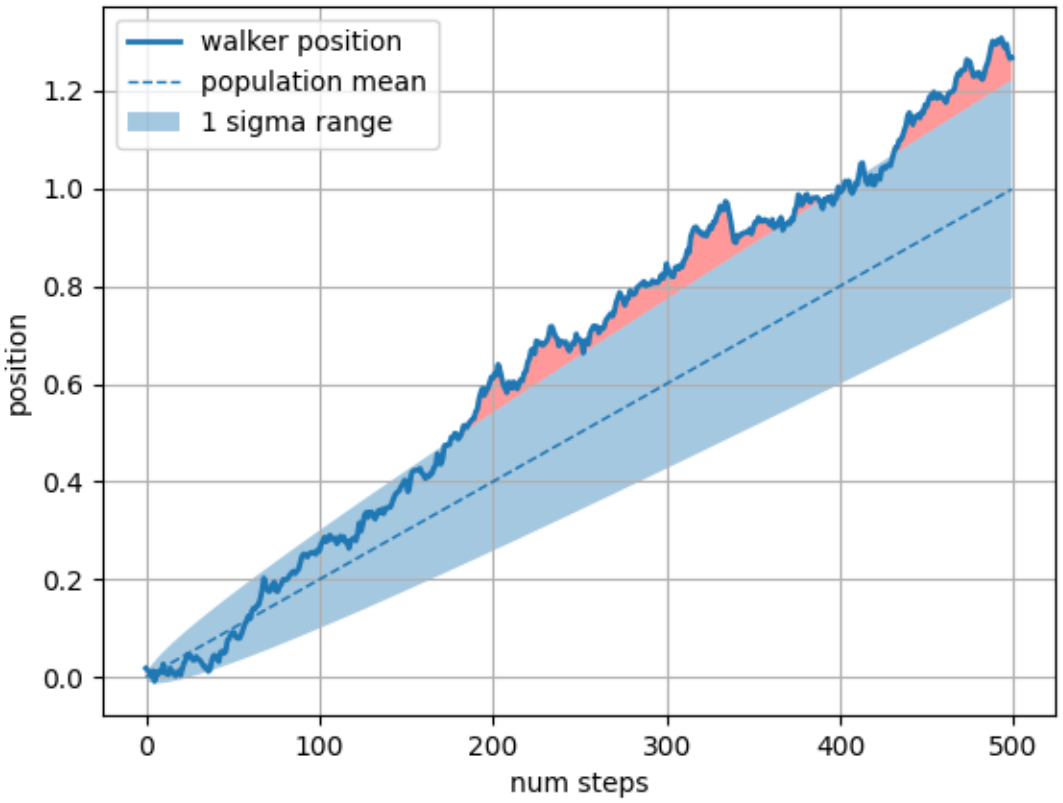

The `where` keyword argument is very handy for highlighting certain regions of the graph. `where` takes a boolean mask the same length as the x, ymin and ymax arguments, and only fills in the region where the boolean mask is True. In the example below, we simulate a single random walker and compute the analytic mean and standard deviation of the population positions. The population mean is shown as the dashed line, and the plus/minus one sigma deviation from the mean is shown as the filled region. We use the where mask `X > upper_bound` to find the region where the walker is outside the one sigma boundary, and shade that region red.

```
# Fixing random state for reproducibility
np.random.seed(1)

Nsteps = 500
t = np.arange(Nsteps)

mu = 0.002
sigma = 0.01

# the steps and position
S = mu + sigma*np.random.randn(Nsteps)
X = S.cumsum()

# the 1 sigma upper and lower analytic population bounds
lower_bound = mu*t - sigma*np.sqrt(t)
```

(continues on next page)

(continued from previous page)

```
upper_bound = mu*t + sigma*np.sqrt(t)

fig, ax = plt.subplots(1)
ax.plot(t, X, lw=2, label='walker position')
ax.plot(t, mu*t, lw=1, label='population mean', color='C0', ls='--')
ax.fill_between(t, lower_bound, upper_bound, facecolor='C0', alpha=0.4,
                label='1 sigma range')
ax.legend(loc='upper left')

# here we use the where argument to only fill the region where the
# walker is above the population 1 sigma boundary
ax.fill_between(t, upper_bound, X, where=X > upper_bound, fc='red', alpha=0.4)
ax.fill_between(t, lower_bound, X, where=X < lower_bound, fc='red', alpha=0.4)
ax.set_xlabel('num steps')
ax.set_ylabel('position')
ax.grid()
```




Another handy use of filled regions is to highlight horizontal or vertical spans of an Axes -- for that Matplotlib has the helper functions *axhspan* and *axvspan*. See *axhspan Demo*.

```
plt.show()
```

**Total running time of the script:** (0 minutes 1.107 seconds)

### Filling the area between lines

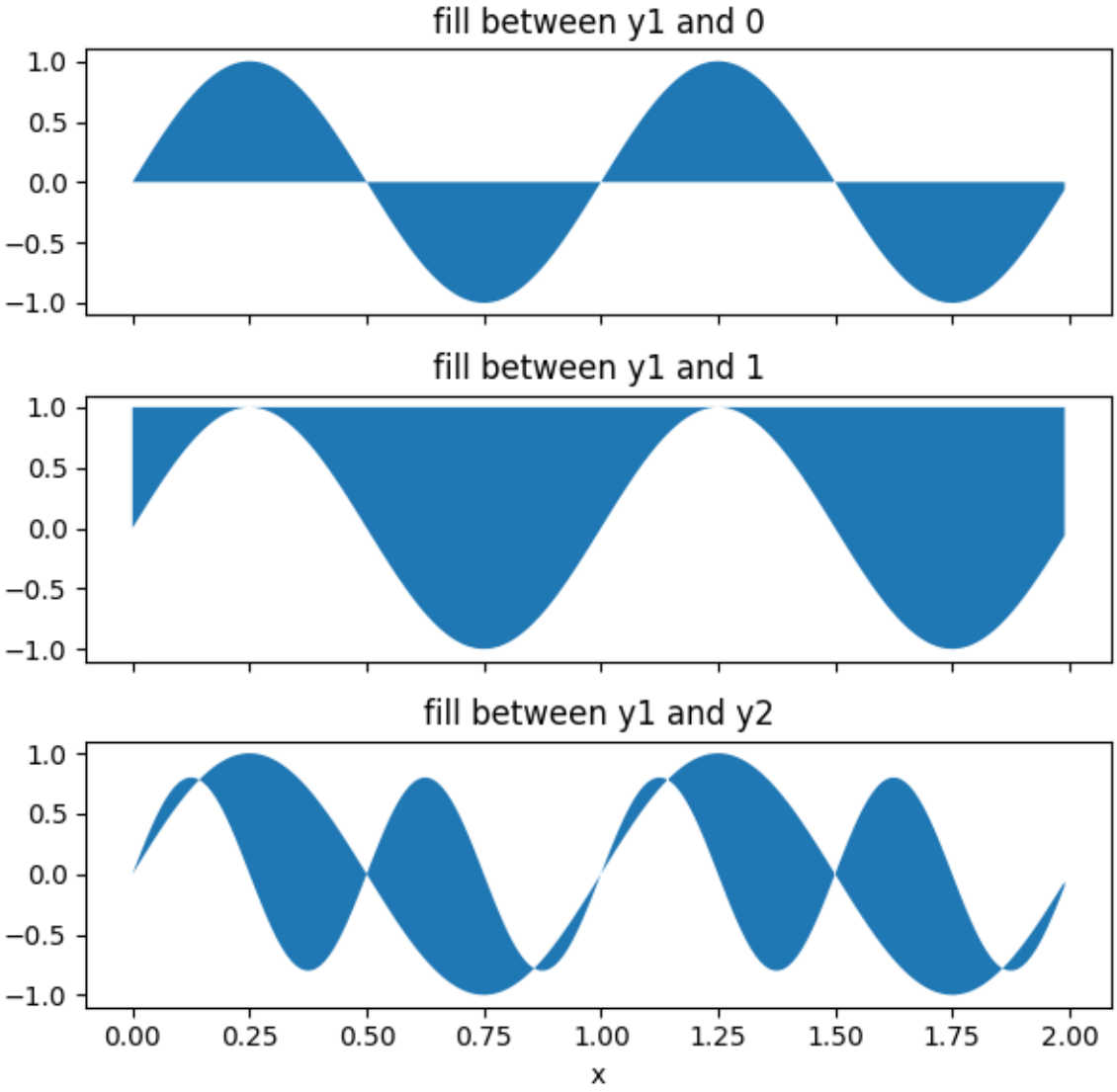
This example shows how to use *fill_between* to color the area between two lines.

```
import matplotlib.pyplot as plt
import numpy as np
```

### Basic usage

The parameters *y1* and *y2* can be scalars, indicating a horizontal boundary at the given y-values. If only *y1* is given, *y2* defaults to 0.

```
x = np.arange(0.0, 2, 0.01)
y1 = np.sin(2 * np.pi * x)
y2 = 0.8 * np.sin(4 * np.pi * x)

fig, (ax1, ax2, ax3) = plt.subplots(3, 1, sharex=True, figsize=(6, 6))

ax1.fill_between(x, y1)
ax1.set_title('fill between y1 and 0')

ax2.fill_between(x, y1, 1)
ax2.set_title('fill between y1 and 1')

ax3.fill_between(x, y1, y2)
ax3.set_title('fill between y1 and y2')
ax3.set_xlabel('x')
fig.tight_layout()
```




### Example: Confidence bands

A common application for *fill_between* is the indication of confidence bands.

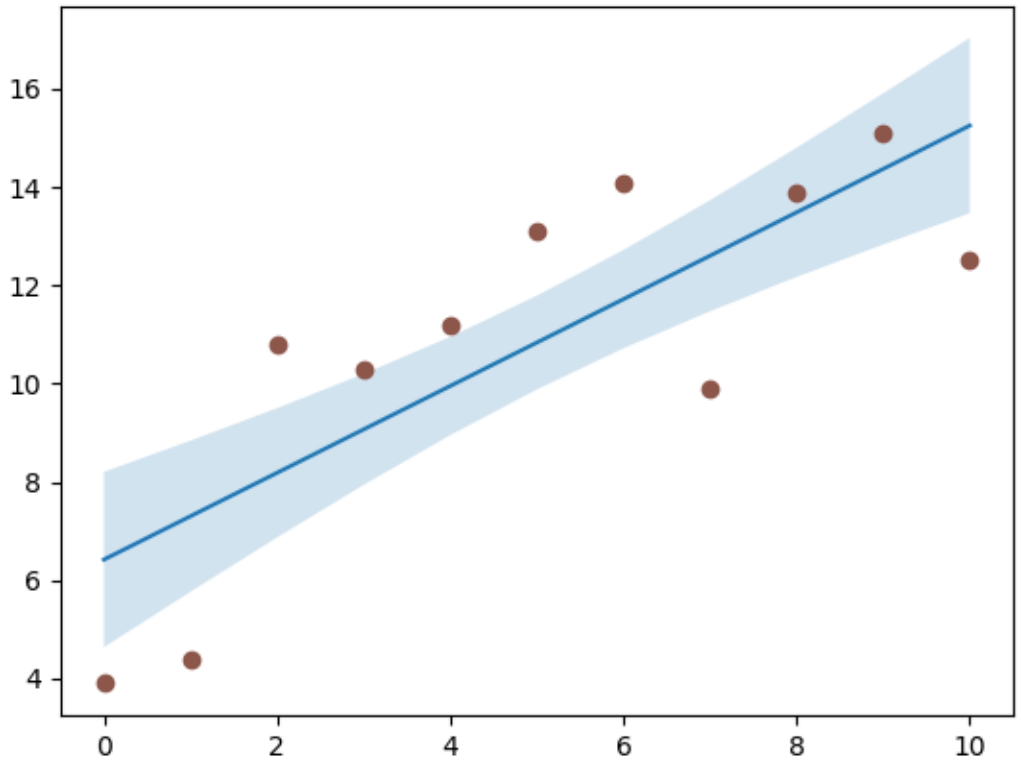
*fill_between* uses the colors of the color cycle as the fill color. These may be a bit strong when applied to fill areas. It is therefore often a good practice to lighten the color by making the area semi-transparent using *alpha*.

```
N = 21
x = np.linspace(0, 10, 11)
y = [3.9, 4.4, 10.8, 10.3, 11.2, 13.1, 14.1,  9.9, 13.9, 15.1, 12.5]

# fit a linear curve and estimate its y-values and their error.
a, b = np.polyfit(x, y, deg=1)
```

(continues on next page)

(continued from previous page)

```
y_est = a * x + b
y_err = x.std() * np.sqrt(1/len(x) +
                          (x - x.mean())**2 / np.sum((x - x.mean())**2))

fig, ax = plt.subplots()
ax.plot(x, y_est, '-')
ax.fill_between(x, y_est - y_err, y_est + y_err, alpha=0.2)
ax.plot(x, y, 'o', color='tab:brown')
```

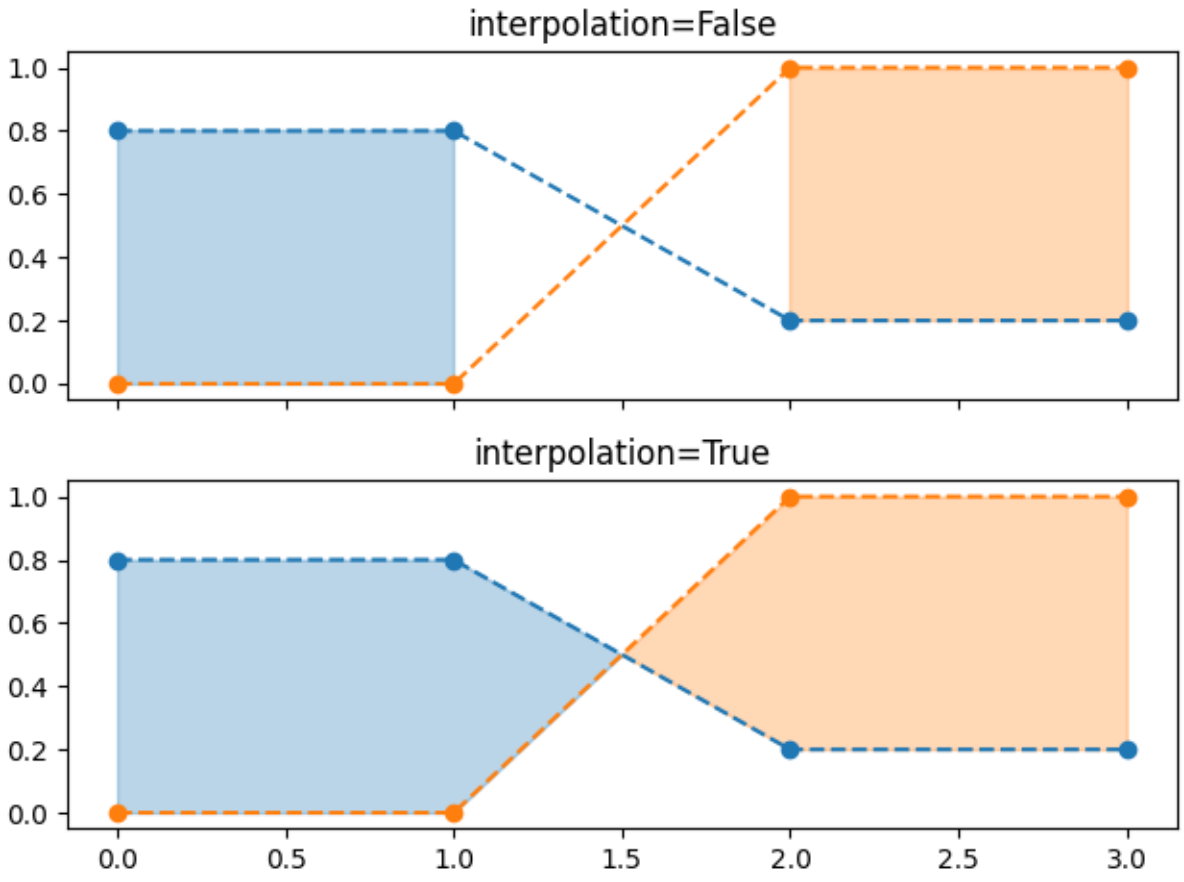



### Selectively filling horizontal regions

The parameter *where* allows to specify the x-ranges to fill. It's a boolean array with the same size as *x*.

Only x-ranges of contiguous *True* sequences are filled. As a result the range between neighboring *True* and *False* values is never filled. This often undesired when the data points should represent a contiguous quantity. It is therefore recommended to set `interpolate=True` unless the x-distance of the data points is fine enough so that the above effect is not noticeable. Interpolation approximates the actual x position at which the *where* condition will change and extends the filling up to there.

```
x = np.array([0, 1, 2, 3])
y1 = np.array([0.8, 0.8, 0.2, 0.2])
y2 = np.array([0, 0, 1, 1])

fig, (ax1, ax2) = plt.subplots(2, 1, sharex=True)

ax1.set_title('interpolation=False')
ax1.plot(x, y1, 'o--')
ax1.plot(x, y2, 'o--')
ax1.fill_between(x, y1, y2, where=(y1 > y2), color='C0', alpha=0.3)
ax1.fill_between(x, y1, y2, where=(y1 < y2), color='C1', alpha=0.3)

ax2.set_title('interpolation=True')
ax2.plot(x, y1, 'o--')
ax2.plot(x, y2, 'o--')
ax2.fill_between(x, y1, y2, where=(y1 > y2), color='C0', alpha=0.3,
                 interpolate=True)
ax2.fill_between(x, y1, y2, where=(y1 <= y2), color='C1', alpha=0.3,
                 interpolate=True)
fig.tight_layout()
```




**Note:** Similar gaps will occur if *y1* or *y2* are masked arrays. Since missing values cannot be approximated,

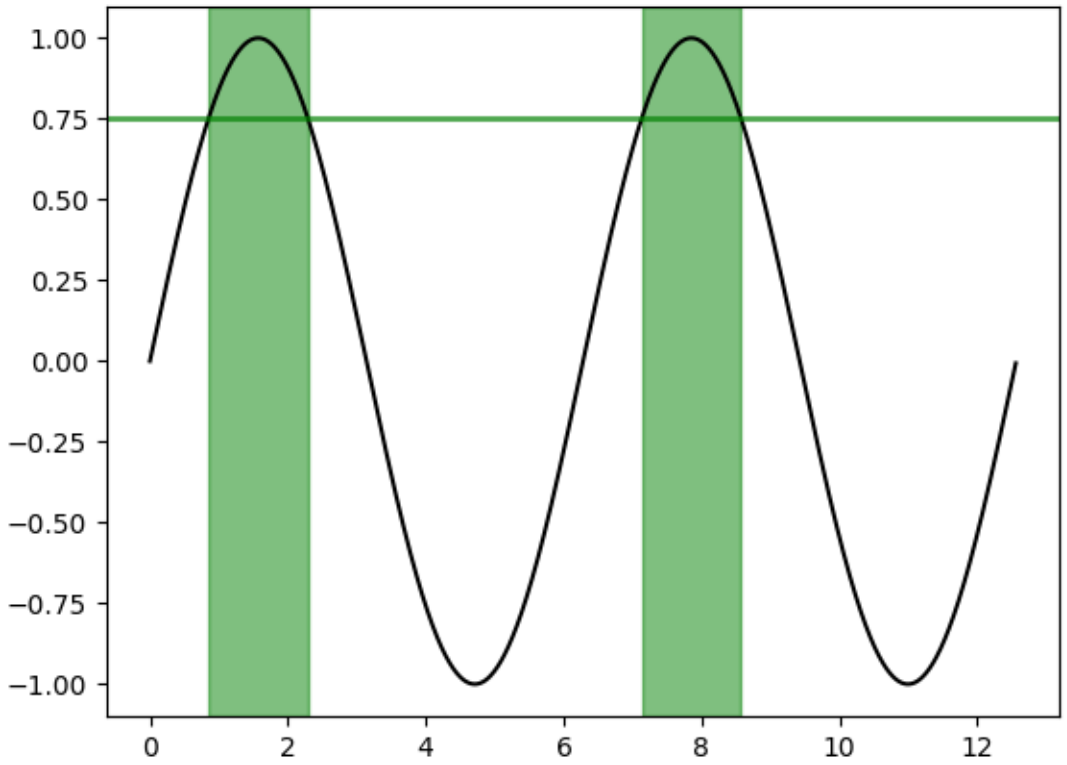
*interpolate* has no effect in this case. The gaps around masked values can only be reduced by adding more data points close to the masked values.

### Selectively marking horizontal regions across the whole Axes

The same selection mechanism can be applied to fill the full vertical height of the Axes. To be independent of y-limits, we add a transform that interprets the x-values in data coordinates and the y-values in Axes coordinates.

The following example marks the regions in which the y-data are above a given threshold.

```
fig, ax = plt.subplots()
x = np.arange(0, 4 * np.pi, 0.01)
y = np.sin(x)
ax.plot(x, y, color='black')

threshold = 0.75
ax.axhline(threshold, color='green', lw=2, alpha=0.7)
ax.fill_between(x, 0, 1, where=y > threshold,
                color='green', alpha=0.5, transform=ax.get_xaxis_transform())
```



**References**

The use of the following functions, methods, classes and modules is shown in this example:

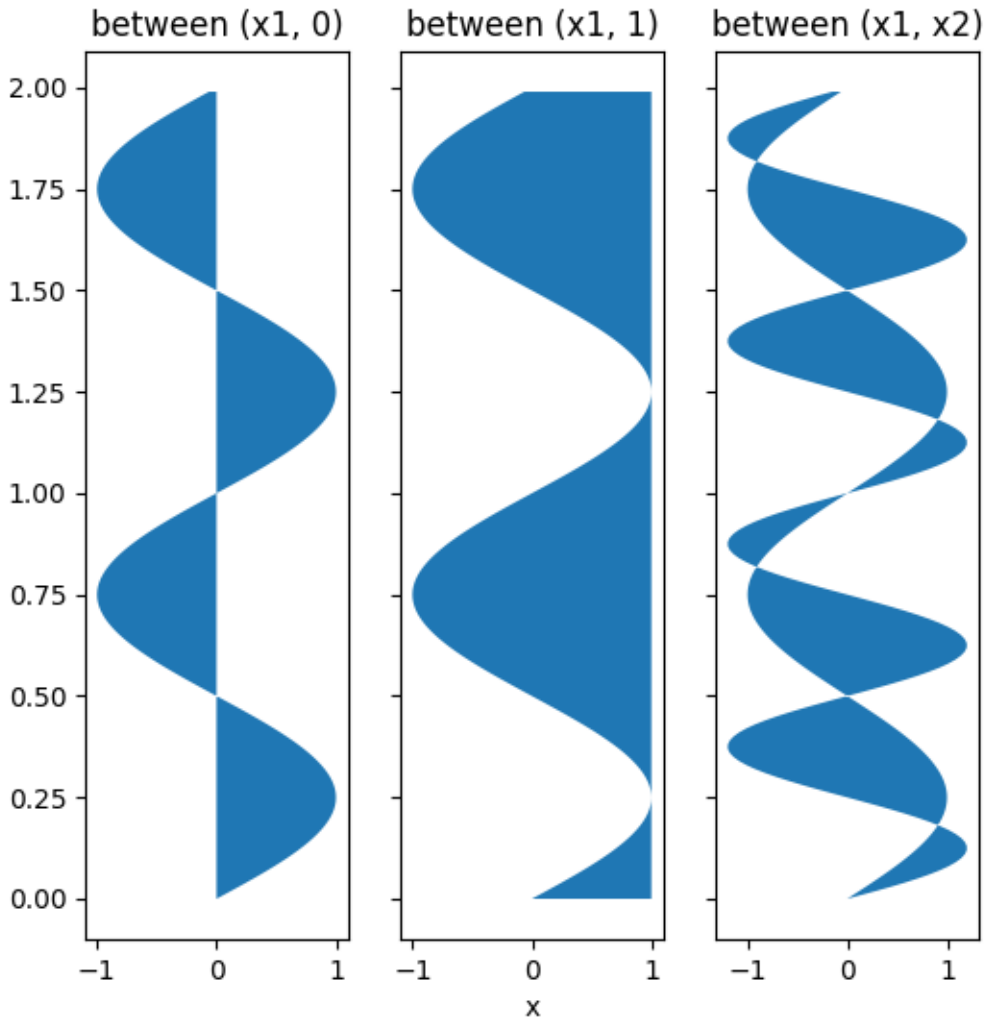
- *matplotlib.axes.Axes.fill_between* / *matplotlib.pyplot.fill_between*
- *matplotlib.axes.Axes.get_xaxis_transform*

**Total running time of the script:** (0 minutes 1.659 seconds)

## Fill Betweenx Demo

Using *fill_betweenx* to color along the horizontal direction between two curves.

```
import matplotlib.pyplot as plt
import numpy as np

y = np.arange(0.0, 2, 0.01)
x1 = np.sin(2 * np.pi * y)
x2 = 1.2 * np.sin(4 * np.pi * y)

fig, [ax1, ax2, ax3] = plt.subplots(1, 3, sharey=True, figsize=(6, 6))

ax1.fill_betweenx(y, 0, x1)
ax1.set_title('between (x1, 0)')

ax2.fill_betweenx(y, x1, 1)
ax2.set_title('between (x1, 1)')
ax2.set_xlabel('x')

ax3.fill_betweenx(y, x1, x2)
ax3.set_title('between (x1, x2)')
```




Now fill between x1 and x2 where a logical condition is met. Note this is different than calling:

```
fill_between(y[where], x1[where], x2[where])
```

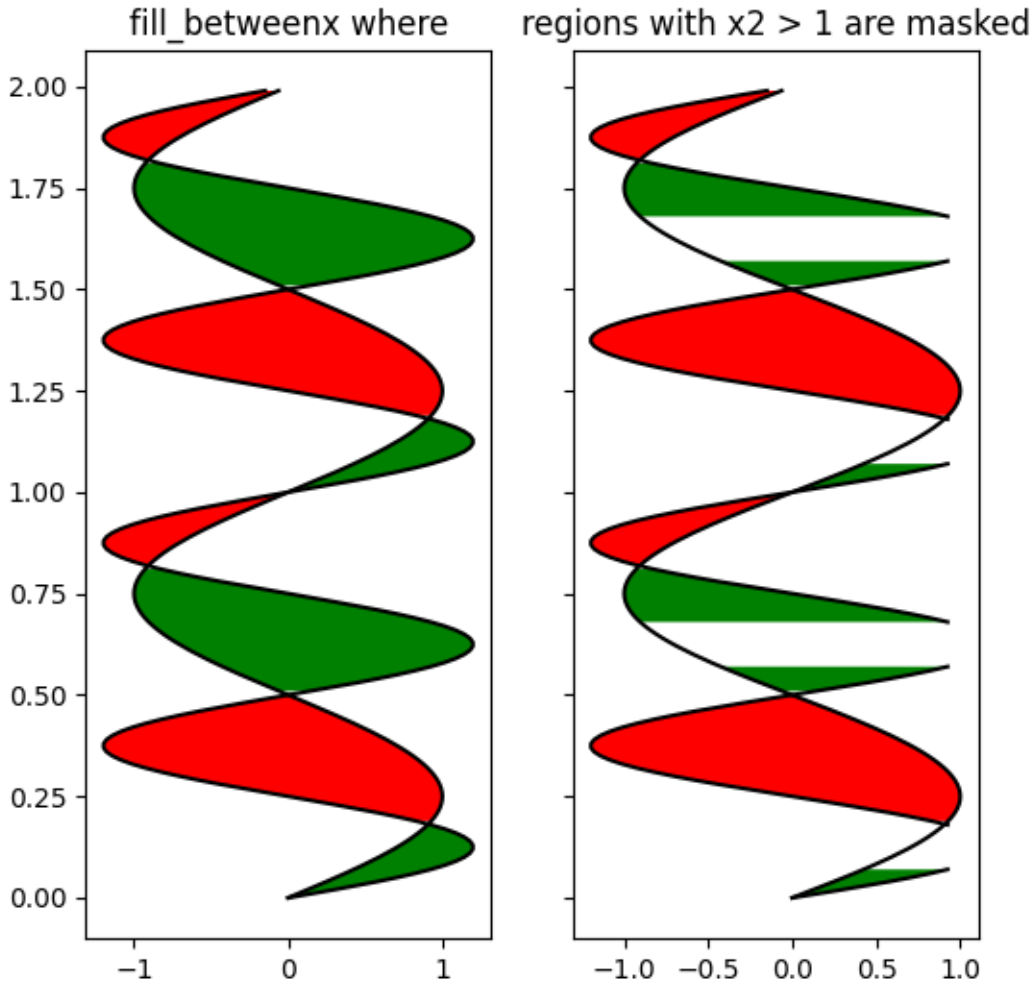
because of edge effects over multiple contiguous regions.

```
fig, [ax, ax1] = plt.subplots(1, 2, sharey=True, figsize=(6, 6))
ax.plot(x1, y, x2, y, color='black')
ax.fill_betweenx(y, x1, x2, where=x2 >= x1, facecolor='green')
ax.fill_betweenx(y, x1, x2, where=x2 <= x1, facecolor='red')
ax.set_title('fill_betweenx where')

# Test support for masked arrays.
x2 = np.ma.masked_greater(x2, 1.0)
ax1.plot(x1, y, x2, y, color='black')
ax1.fill_betweenx(y, x1, x2, where=x2 >= x1, facecolor='green')
```

(continues on next page)

(continued from previous page)

```
ax1.fill_betweenx(y, x1, x2, where=x2 <= x1, facecolor='red')
ax1.set_title('regions with x2 > 1 are masked')
```




This example illustrates a problem; because of the data gridding, there are undesired unfilled triangles at the crossover points. A brute-force solution would be to interpolate all arrays to a very fine grid before plotting.

```
plt.show()
```

### Hatch-filled histograms

Hatching capabilities for plotting histograms.

```
from functools import partial
import itertools

from cycler import cycler

import matplotlib.pyplot as plt
import numpy as np

import matplotlib.ticker as mticker


def filled_hist(ax, edges, values, bottoms=None, orientation='v',
                **kwargs):
    """
    Draw a histogram as a stepped patch.

    Parameters
    ----------
    ax : Axes
        The Axes to plot to.

    edges : array
        A length n+1 array giving the left edges of each bin and the
        right edge of the last bin.

    values : array
        A length n array of bin counts or values

    bottoms : float or array, optional
        A length n array of the bottom of the bars.  If None, zero is used.

    orientation : {'v', 'h'}
       Orientation of the histogram.  'v' (default) has
       the bars increasing in the positive y-direction.

    **kwargs
        Extra keyword arguments are passed through to `.fill_between`.

    Returns
    -------
    ret : PolyCollection
        Artist added to the Axes
    """
    print(orientation)
    if orientation not in 'hv':
        raise ValueError(f"orientation must be in {{'h', 'v'}} "
                         f"not {orientation}")

    kwargs.setdefault('step', 'post')
```

(continues on next page)

(continued from previous page)

```
    kwargs.setdefault('alpha', 0.7)
    edges = np.asarray(edges)
    values = np.asarray(values)
    if len(edges) - 1 != len(values):
        raise ValueError(f'Must provide one more bin edge than value not: '
                         f'{len(edges)=} {len(values)=}')

    if bottoms is None:
        bottoms = 0
    bottoms = np.broadcast_to(bottoms, values.shape)

    values = np.append(values, values[-1])
    bottoms = np.append(bottoms, bottoms[-1])
    if orientation == 'h':
        return ax.fill_betweenx(edges, values, bottoms,
                                **kwargs)
    elif orientation == 'v':
        return ax.fill_between(edges, values, bottoms,
                               **kwargs)
    else:
        raise AssertionError("you should never be here")


def stack_hist(ax, stacked_data, sty_cycle, bottoms=None,
               hist_func=None, labels=None,
               plot_func=None, plot_kwargs=None):
    """
    Parameters
    ----------
    ax : axes.Axes
        The Axes to add artists to.

    stacked_data : array or Mapping
        A (M, N) shaped array.  The first dimension will be iterated over to
        compute histograms row-wise

    sty_cycle : Cycler or operable of dict
        Style to apply to each set

    bottoms : array, default: 0
        The initial positions of the bottoms.

    hist_func : callable, optional
        Must have signature `bin_vals, bin_edges = f(data)`.
        `bin_edges` expected to be one longer than `bin_vals`

    labels : list of str, optional
        The label for each set.

        If not given and stacked data is an array defaults to 'default set {n}
↪'
```

(continues on next page)

(continued from previous page)

```
        If *stacked_data* is a mapping, and *labels* is None, default to the
        keys.

        If *stacked_data* is a mapping and *labels* is given then only the
        columns listed will be plotted.

    plot_func : callable, optional
        Function to call to draw the histogram must have signature:

          ret = plot_func(ax, edges, top, bottoms=bottoms,
                          label=label, **kwargs)

    plot_kwargs : dict, optional
        Any extra keyword arguments to pass through to the plotting function.
        This will be the same for all calls to the plotting function and will
        override the values in *sty_cycle*.

    Returns
    -------
    arts : dict
        Dictionary of artists keyed on their labels
    """
    # deal with default binning function
    if hist_func is None:
        hist_func = np.histogram

    # deal with default plotting function
    if plot_func is None:
        plot_func = filled_hist

    # deal with default
    if plot_kwargs is None:
        plot_kwargs = {}
    print(plot_kwargs)
    try:
        l_keys = stacked_data.keys()
        label_data = True
        if labels is None:
            labels = l_keys

    except AttributeError:
        label_data = False
        if labels is None:
            labels = itertools.repeat(None)

    if label_data:
        loop_iter = enumerate((stacked_data[lab], lab, s)
                              for lab, s in zip(labels, sty_cycle))
    else:
        loop_iter = enumerate(zip(stacked_data, labels, sty_cycle))

    arts = {}
```

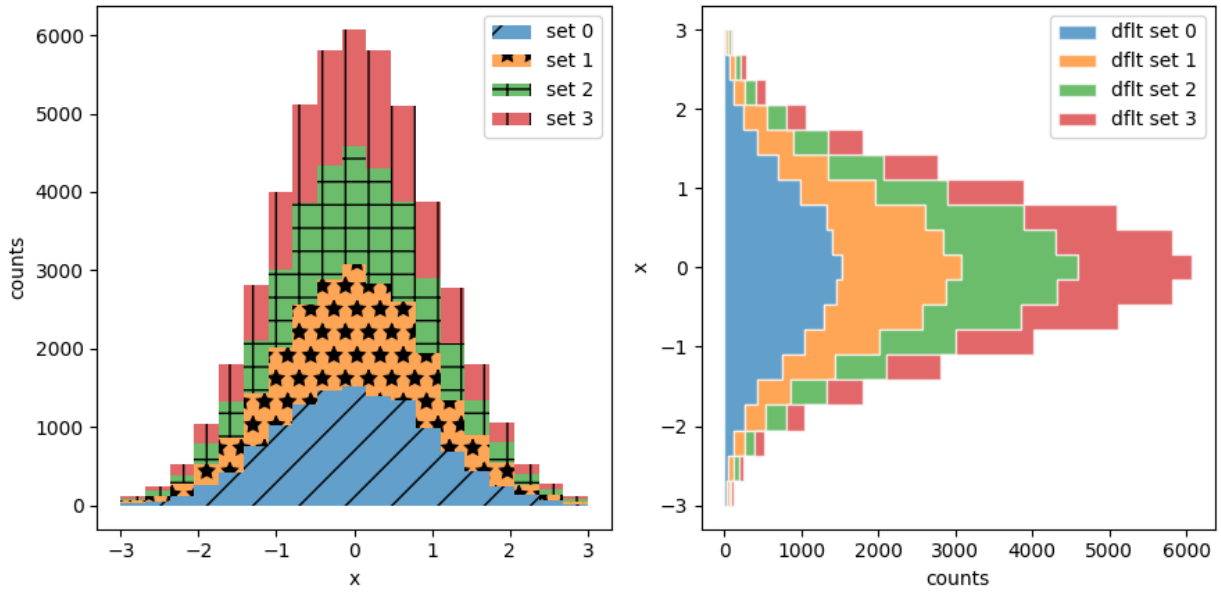
(continues on next page)

(continued from previous page)

```
    for j, (data, label, sty) in loop_iter:
        if label is None:
            label = f'dflt set {j}'
        label = sty.pop('label', label)
        vals, edges = hist_func(data)
        if bottoms is None:
            bottoms = np.zeros_like(vals)
        top = bottoms + vals
        print(sty)
        sty.update(plot_kwargs)
        print(sty)
        ret = plot_func(ax, edges, top, bottoms=bottoms,
                        label=label, **sty)
        bottoms = top
        arts[label] = ret
    ax.legend(fontsize=10)
    return arts


# set up histogram function to fixed bins
edges = np.linspace(-3, 3, 20, endpoint=True)
hist_func = partial(np.histogram, bins=edges)

# set up style cycles
color_cycle = cycler(facecolor=plt.rcParams['axes.prop_cycle'][:4])
label_cycle = cycler(label=[f'set {n}' for n in range(4)])
hatch_cycle = cycler(hatch=['/', '*', '+', '|'])

# Fixing random state for reproducibility
np.random.seed(19680801)

stack_data = np.random.randn(4, 12250)
dict_data = dict(zip((c['label'] for c in label_cycle), stack_data))
```

Work with plain arrays

```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(9, 4.5), tight_layout=True)
arts = stack_hist(ax1, stack_data, color_cycle + label_cycle + hatch_cycle,
                  hist_func=hist_func)

arts = stack_hist(ax2, stack_data, color_cycle,
                  hist_func=hist_func,
                  plot_kwargs=dict(edgecolor='w', orientation='h'))
ax1.set_ylabel('counts')
ax1.set_xlabel('x')
ax2.set_xlabel('counts')
ax2.set_ylabel('x')
```




```
{}
{'facecolor': '#1f77b4', 'hatch': '/'}
{'facecolor': '#1f77b4', 'hatch': '/'}
v
{'facecolor': '#ff7f0e', 'hatch': '*'}
{'facecolor': '#ff7f0e', 'hatch': '*'}
v
{'facecolor': '#2ca02c', 'hatch': '+'}
{'facecolor': '#2ca02c', 'hatch': '+'}
v
{'facecolor': '#d62728', 'hatch': '|'}
{'facecolor': '#d62728', 'hatch': '|'}
v
{'edgecolor': 'w', 'orientation': 'h'}
{'facecolor': '#1f77b4'}
{'facecolor': '#1f77b4', 'edgecolor': 'w', 'orientation': 'h'}
h
{'facecolor': '#ff7f0e'}
{'facecolor': '#ff7f0e', 'edgecolor': 'w', 'orientation': 'h'}
h
{'facecolor': '#2ca02c'}
{'facecolor': '#2ca02c', 'edgecolor': 'w', 'orientation': 'h'}
h
{'facecolor': '#d62728'}
{'facecolor': '#d62728', 'edgecolor': 'w', 'orientation': 'h'}
h
```

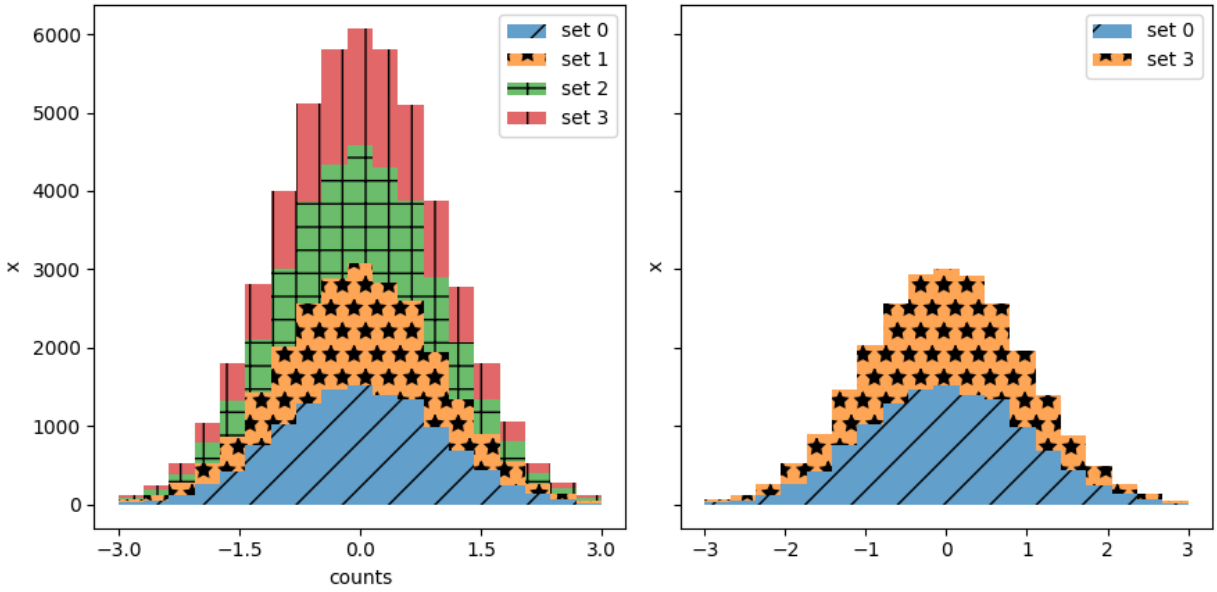
Work with labeled data

```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(9, 4.5),
                               tight_layout=True, sharey=True)
```

(continues on next page)

(continued from previous page)

```
arts = stack_hist(ax1, dict_data, color_cycle + hatch_cycle,
                  hist_func=hist_func)

arts = stack_hist(ax2, dict_data, color_cycle + hatch_cycle,
                  hist_func=hist_func, labels=['set 0', 'set 3'])
ax1.xaxis.set_major_locator(mticker.MaxNLocator(5))
ax1.set_xlabel('counts')
ax1.set_ylabel('x')
ax2.set_ylabel('x')

plt.show()
```




```
{}
{'facecolor': '#1f77b4', 'hatch': '/'}
{'facecolor': '#1f77b4', 'hatch': '/'}
v
{'facecolor': '#ff7f0e', 'hatch': '*'}
{'facecolor': '#ff7f0e', 'hatch': '*'}
v
{'facecolor': '#2ca02c', 'hatch': '+'}
{'facecolor': '#2ca02c', 'hatch': '+'}
v
{'facecolor': '#d62728', 'hatch': '|'}
{'facecolor': '#d62728', 'hatch': '|'}
v
{}
{'facecolor': '#1f77b4', 'hatch': '/'}
{'facecolor': '#1f77b4', 'hatch': '/'}
v
{'facecolor': '#ff7f0e', 'hatch': '*'}
{'facecolor': '#ff7f0e', 'hatch': '*'}
```

(continues on next page)

(continued from previous page)

```
v
```

**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.fill_betweenx* / *matplotlib.pyplot.fill_betweenx*
- *matplotlib.axes.Axes.fill_between* / *matplotlib.pyplot.fill_between*
- *matplotlib.axis.Axis.set_major_locator*

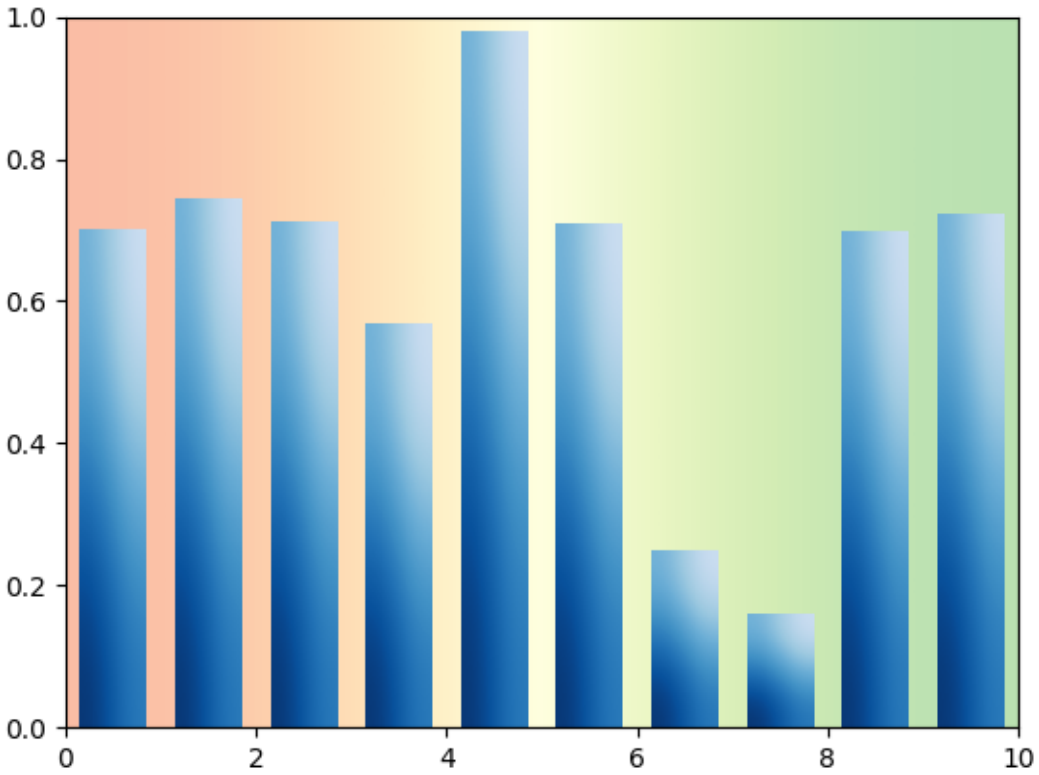
**Total running time of the script:** (0 minutes 1.101 seconds)

### Bar chart with gradients

Matplotlib does not natively support gradients. However, we can emulate a gradient-filled rectangle by an *AxesImage* of the right size and coloring.

In particular, we use a colormap to generate the actual colors. It is then sufficient to define the underlying values on the corners of the image and let bicubic interpolation fill out the area. We define the gradient direction by a unit vector $v$. The values at the corners are then obtained by the lengths of the projections of the corner vectors on $v$.

A similar approach can be used to create a gradient background for an Axes. In that case, it is helpful to use Axes coordinates (`extent=(0, 1, 0, 1), transform=ax.transAxes`) to be independent of the data coordinates.




```
import matplotlib.pyplot as plt
import numpy as np

np.random.seed(19680801)


def gradient_image(ax, direction=0.3, cmap_range=(0, 1), **kwargs):
    """
    Draw a gradient image based on a colormap.

    Parameters
    ----------
    ax : Axes
        The Axes to draw on.
    direction : float
        The direction of the gradient. This is a number in
        range 0 (=vertical) to 1 (=horizontal).
    cmap_range : float, float
        The fraction (cmin, cmax) of the colormap that should be
        used for the gradient, where the complete colormap is (0, 1).
    **kwargs
        Other parameters are passed on to `.Axes.imshow()`.
        In particular, *cmap*, *extent*, and *transform* may be useful.
```

(continues on next page)

(continued from previous page)

```
    """
    phi = direction * np.pi / 2
    v = np.array([np.cos(phi), np.sin(phi)])
    X = np.array([[v @ [1, 0], v @ [1, 1]],
                  [v @ [0, 0], v @ [0, 1]]])
    a, b = cmap_range
    X = a + (b - a) / X.max() * X
    im = ax.imshow(X, interpolation='bicubic', clim=(0, 1),
                   aspect='auto', **kwargs)
    return im


def gradient_bar(ax, x, y, width=0.5, bottom=0):
    for left, top in zip(x, y):
        right = left + width
        gradient_image(ax, extent=(left, right, bottom, top),
                       cmap=plt.cm.Blues_r, cmap_range=(0, 0.8))


fig, ax = plt.subplots()
ax.set(xlim=(0, 10), ylim=(0, 1))

# background image
gradient_image(ax, direction=1, extent=(0, 1, 0, 1), transform=ax.transAxes,
               cmap=plt.cm.RdYlGn, cmap_range=(0.2, 0.8), alpha=0.5)

N = 10
x = np.arange(N) + 0.15
y = np.random.rand(N)
gradient_bar(ax, x, y, width=0.7)
plt.show()
```

### Hat graph

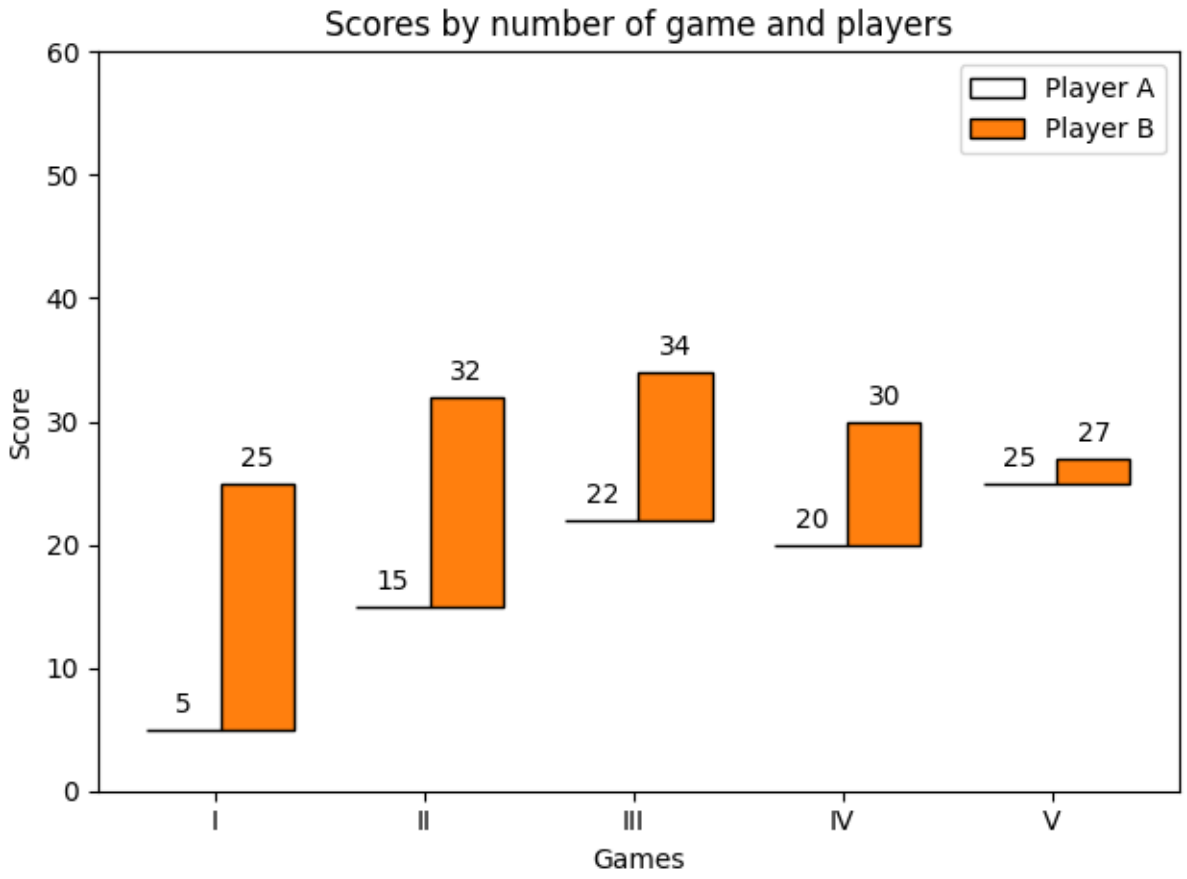
This example shows how to create a hat graph and how to annotate it with labels.

```
import matplotlib.pyplot as plt
import numpy as np


def hat_graph(ax, xlabels, values, group_labels):
    """
    Create a hat graph.

    Parameters
    ----------
    ax : matplotlib.axes.Axes
        The Axes to plot into.
    xlabels : list of str
        The category names to be displayed on the x-axis.
```

(continues on next page)

(continued from previous page)

```
    values : (M, N) array-like
        The data values.
        Rows are the groups (len(group_labels) == M).
        Columns are the categories (len(xlabels) == N).
    group_labels : list of str
        The group labels displayed in the legend.
    """

    def label_bars(heights, rects):
        """Attach a text label on top of each bar."""
        for height, rect in zip(heights, rects):
            ax.annotate(f'{height}',
                        xy=(rect.get_x() + rect.get_width() / 2, height),
                        xytext=(0, 4),  # 4 points vertical offset.
                        textcoords='offset points',
                        ha='center', va='bottom')

    values = np.asarray(values)
    x = np.arange(values.shape[1])
    ax.set_xticks(x, labels=xlabels)
    spacing = 0.3  # spacing between hat groups
    width = (1 - spacing) / values.shape[0]
    heights0 = values[0]
    for i, (heights, group_label) in enumerate(zip(values, group_labels)):
        style = {'fill': False} if i == 0 else {'edgecolor': 'black'}
        rects = ax.bar(x - spacing/2 + i * width, heights - heights0,
                       width, bottom=heights0, label=group_label, **style)
        label_bars(heights, rects)


# initialise labels and a numpy array make sure you have
# N labels of N number of values in the array
xlabels = ['I', 'II', 'III', 'IV', 'V']
playerA = np.array([5, 15, 22, 20, 25])
playerB = np.array([25, 32, 34, 30, 27])

fig, ax = plt.subplots()
hat_graph(ax, xlabels, [playerA, playerB], ['Player A', 'Player B'])

# Add some text for labels, title and custom x-axis tick labels, etc.
ax.set_xlabel('Games')
ax.set_ylabel('Score')
ax.set_ylim(0, 60)
ax.set_title('Scores by number of game and players')
ax.legend()

fig.tight_layout()
plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.bar* / *matplotlib.pyplot.bar*
- *matplotlib.axes.Axes.annotate* / *matplotlib.pyplot.annotate*

### Discrete distribution as horizontal bar chart

Stacked bar charts can be used to visualize discrete distributions.

This example visualizes the result of a survey in which people could rate their agreement to questions on a five-element scale.

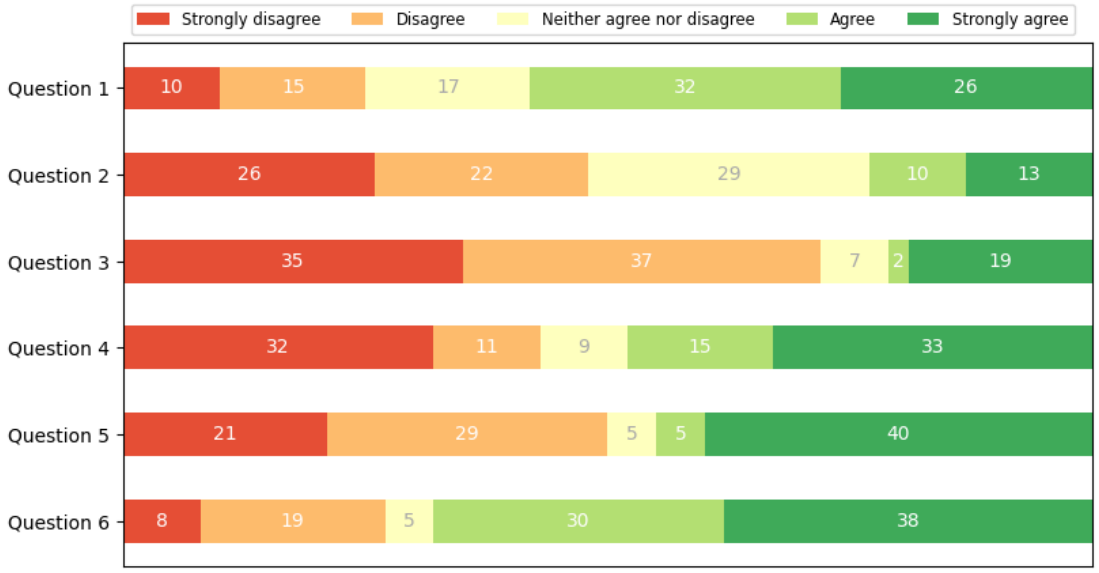
The horizontal stacking is achieved by calling *barh()* for each category and passing the starting point as the cumulative sum of the already drawn bars via the parameter `left`.

```
import matplotlib.pyplot as plt
import numpy as np
```

(continues on next page)

(continued from previous page)

```
category_names = ['Strongly disagree', 'Disagree',
                  'Neither agree nor disagree', 'Agree', 'Strongly agree']
results = {
    'Question 1': [10, 15, 17, 32, 26],
    'Question 2': [26, 22, 29, 10, 13],
    'Question 3': [35, 37, 7, 2, 19],
    'Question 4': [32, 11, 9, 15, 33],
    'Question 5': [21, 29, 5, 5, 40],
    'Question 6': [8, 19, 5, 30, 38]
}


def survey(results, category_names):
    """
    Parameters
    ----------
    results : dict
        A mapping from question labels to a list of answers per category.
        It is assumed all lists contain the same number of entries and that
        it matches the length of *category_names*.
    category_names : list of str
        The category labels.
    """
    labels = list(results.keys())
    data = np.array(list(results.values()))
    data_cum = data.cumsum(axis=1)
    category_colors = plt.colormaps['RdYlGn'](
        np.linspace(0.15, 0.85, data.shape[1]))

    fig, ax = plt.subplots(figsize=(9.2, 5))
    ax.invert_yaxis()
    ax.xaxis.set_visible(False)
    ax.set_xlim(0, np.sum(data, axis=1).max())

    for i, (colname, color) in enumerate(zip(category_names, category_
↪colors)):
        widths = data[:, i]
        starts = data_cum[:, i] - widths
        rects = ax.barh(labels, widths, left=starts, height=0.5,
                        label=colname, color=color)

        r, g, b, _ = color
        text_color = 'white' if r * g * b < 0.5 else 'darkgrey'
        ax.bar_label(rects, label_type='center', color=text_color)
    ax.legend(ncols=len(category_names), bbox_to_anchor=(0, 1),
              loc='lower left', fontsize='small')

    return fig, ax


survey(results, category_names)
plt.show()
```



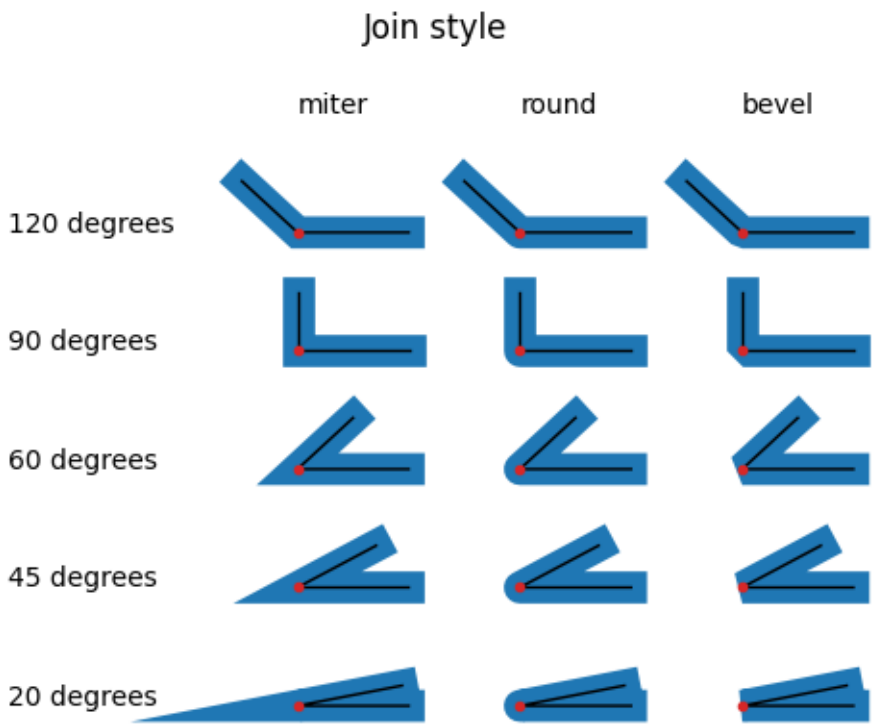

**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.barh* / *matplotlib.pyplot.barh*
- *matplotlib.axes.Axes.bar_label* / *matplotlib.pyplot.bar_label*
- *matplotlib.axes.Axes.legend* / *matplotlib.pyplot.legend*

## JoinStyle

The *matplotlib._enums.JoinStyle* controls how Matplotlib draws the corners where two different line segments meet. For more details, see the *JoinStyle* docs.




```
import matplotlib.pyplot as plt

from matplotlib._enums import JoinStyle

JoinStyle.demo()
plt.show()
```

### Customizing dashed line styles

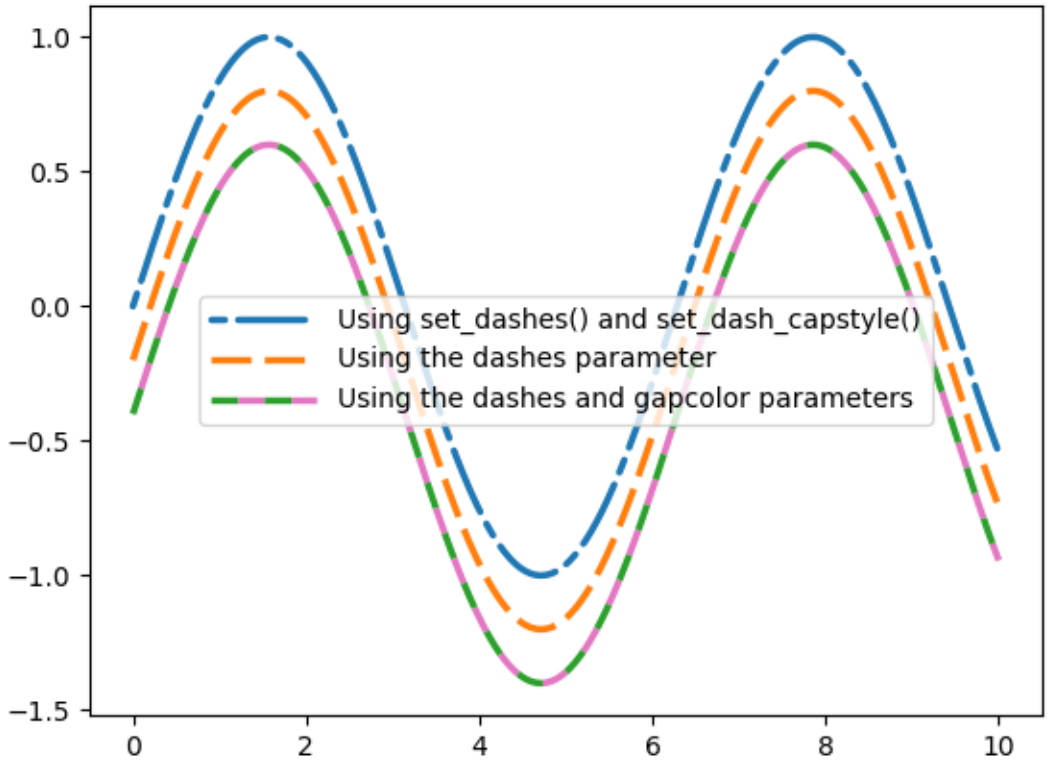
The dashing of a line is controlled via a dash sequence. It can be modified using *Line2D.set_dashes*.

The dash sequence is a series of on/off lengths in points, e.g. `[3, 1]` would be 3pt long lines separated by 1pt spaces.

Some functions like *Axes.plot* support passing Line properties as keyword arguments. In such a case, you can already set the dashing when creating the line.

*Note*: The dash style can also be configured via a *property_cycle* by passing a list of dash sequences using the keyword *dashes* to the cycler. This is not shown within this example.

Other attributes of the dash may also be set either with the relevant method (*set_dash_capstyle*, *set_dash_joinstyle*, *set_gapcolor*) or by passing the property through a plotting function.




```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 10, 500)
y = np.sin(x)

plt.rc('lines', linewidth=2.5)
fig, ax = plt.subplots()

# Using set_dashes() and set_capstyle() to modify dashing of an existing line.
line1, = ax.plot(x, y, label='Using set_dashes() and set_dash_capstyle()')
line1.set_dashes([2, 2, 10, 2])  # 2pt line, 2pt break, 10pt line, 2pt break.
line1.set_dash_capstyle('round')

# Using plot(..., dashes=...) to set the dashing when creating a line.
line2, = ax.plot(x, y - 0.2, dashes=[6, 2], label='Using the dashes parameter
 ↪')

# Using plot(..., dashes=..., gapcolor=...) to set the dashing and
# alternating color when creating a line.
line3, = ax.plot(x, y - 0.4, dashes=[4, 4], gapcolor='tab:pink',
                 label='Using the dashes and gapcolor parameters')
```

(continues on next page)

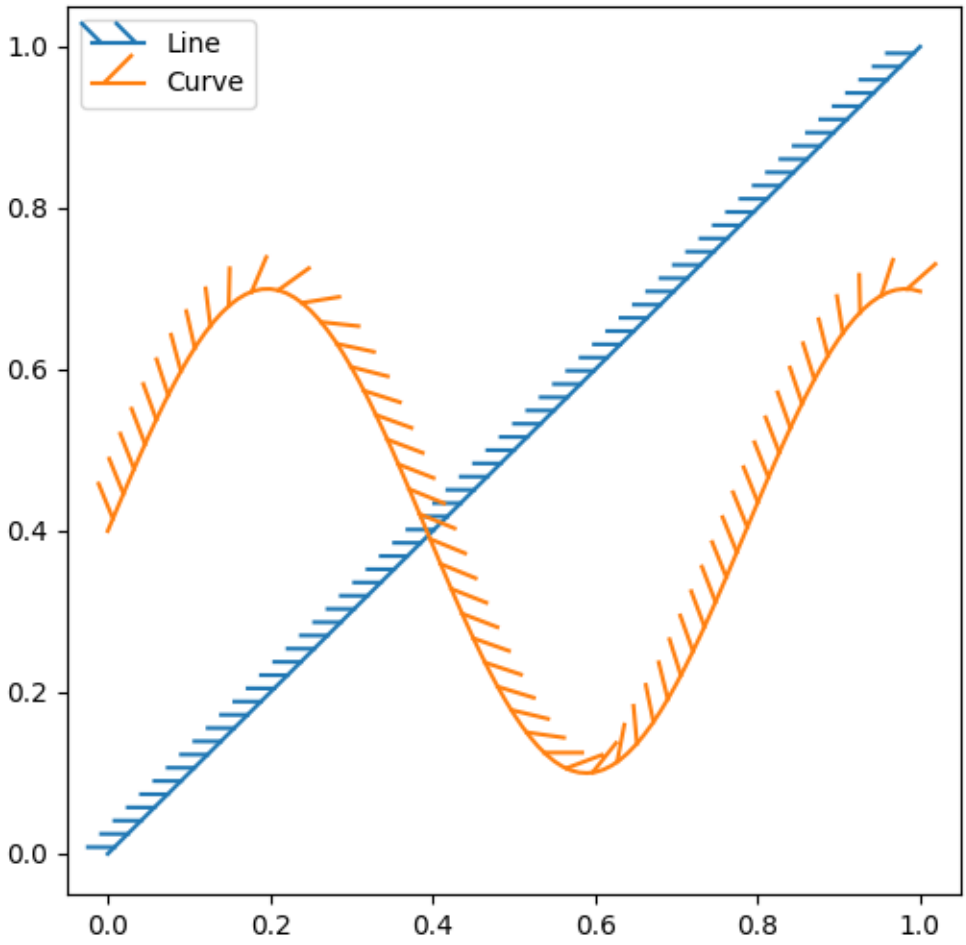
(continued from previous page)

```
ax.legend(handlelength=4)
plt.show()
```

### Lines with a ticked patheffect

Ticks can be added along a line to mark one side as a barrier using `TickedStroke`. You can control the angle, spacing, and length of the ticks.

The ticks will also appear appropriately in the legend.




```
import matplotlib.pyplot as plt
import numpy as np
```

(continues on next page)

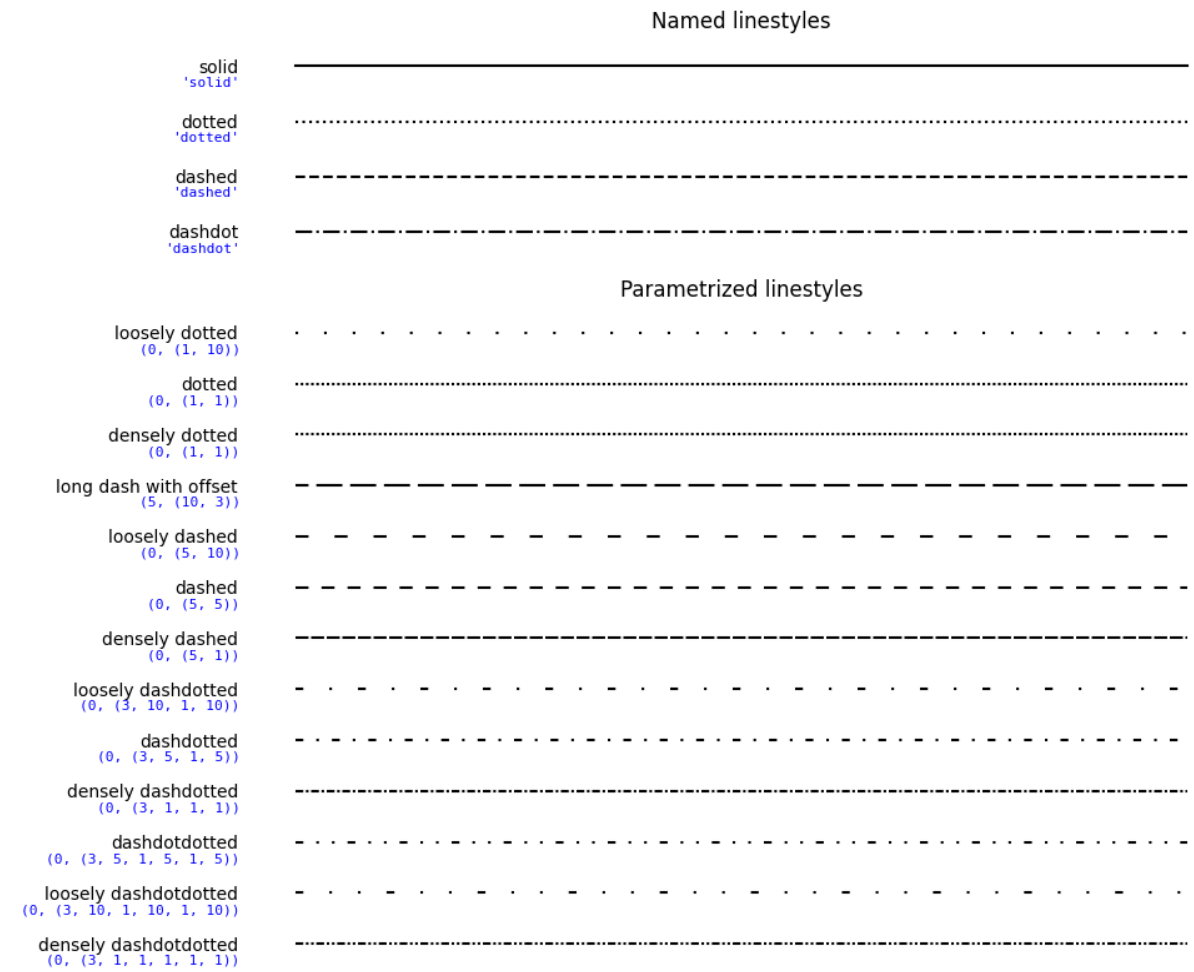
(continued from previous page)

```
from matplotlib import patheffects

# Plot a straight diagonal line with ticked style path
fig, ax = plt.subplots(figsize=(6, 6))
ax.plot([0, 1], [0, 1], label="Line",
        path_effects=[patheffects.withTickedStroke(spacing=7, angle=135)])

# Plot a curved line with ticked style path
nx = 101
x = np.linspace(0.0, 1.0, nx)
y = 0.3*np.sin(x*8) + 0.4
ax.plot(x, y, label="Curve", path_effects=[patheffects.withTickedStroke()])

ax.legend()

plt.show()
```

### Linestyles

Simple linestyles can be defined using the strings "solid", "dotted", "dashed" or "dashdot". More refined control can be achieved by providing a dash tuple `(offset, (on_off_seq))`. For example, `(0, (3, 10, 1, 15))` means (3pt line, 10pt space, 1pt line, 15pt space) with no offset, while `(5, (10, 3))`, means (10pt line, 3pt space), but skip the first 5pt line. See also *`Line2D.set_linestyle`*.

*Note*: The dash style can also be configured via *`Line2D.set_dashes`* as shown in *Customizing dashed line styles* and passing a list of dash sequences using the keyword *dashes* to the cycler in *property_cycle*.




```
import matplotlib.pyplot as plt
import numpy as np

linestyle_str = [
     ('solid', 'solid'),      # Same as (0, ()) or '-'
     ('dotted', 'dotted'),    # Same as (0, (1, 1)) or ':'
     ('dashed', 'dashed'),    # Same as '--'
     ('dashdot', 'dashdot')]  # Same as '-.'

linestyle_tuple = [
     ('loosely dotted',         (0, (1, 10))),
     ('dotted',                 (0, (1, 1))),
     ('densely dotted',         (0, (1, 1))),
     ('long dash with offset', (5, (10, 3))),
     ('loosely dashed',         (0, (5, 10))),
     ('dashed',                 (0, (5, 5))),
     ('densely dashed',         (0, (5, 1))),

     ('loosely dashdotted',     (0, (3, 10, 1, 10))),
     ('dashdotted',             (0, (3, 5, 1, 5))),
     ('densely dashdotted',     (0, (3, 1, 1, 1))),
```

(continues on next page)

(continued from previous page)

```
     ('dashdotdotted',          (0, (3, 5, 1, 5, 1, 5))),
     ('loosely dashdotdotted', (0, (3, 10, 1, 10, 1, 10))),
     ('densely dashdotdotted', (0, (3, 1, 1, 1, 1, 1)))]


def plot_linestyles(ax, linestyles, title):
    X, Y = np.linspace(0, 100, 10), np.zeros(10)
    yticklabels = []

    for i, (name, linestyle) in enumerate(linestyles):
        ax.plot(X, Y+i, linestyle=linestyle, linewidth=1.5, color='black')
        yticklabels.append(name)

    ax.set_title(title)
    ax.set(ylim=(-0.5, len(linestyles)-0.5),
           yticks=np.arange(len(linestyles)),
           yticklabels=yticklabels)
    ax.tick_params(left=False, bottom=False, labelbottom=False)
    ax.spines[:].set_visible(False)

    # For each line style, add a text annotation with a small offset from
    # the reference point (0 in Axes coords, y tick value in Data coords).
    for i, (name, linestyle) in enumerate(linestyles):
        ax.annotate(repr(linestyle),
                    xy=(0.0, i), xycoords=ax.get_yaxis_transform(),
                    xytext=(-6, -12), textcoords='offset points',
                    color="blue", fontsize=8, ha="right", family="monospace")


fig, (ax0, ax1) = plt.subplots(2, 1, figsize=(10, 8), height_ratios=[1, 3])

plot_linestyles(ax0, linestyle_str[::-1], title='Named linestyles')
plot_linestyles(ax1, linestyle_tuple[::-1], title='Parametrized linestyles')

plt.tight_layout()
plt.show()
```

### Marker reference

Matplotlib supports multiple categories of markers which are selected using the `marker` parameter of plot commands:

- *Unfilled markers*
- *Filled markers*
- *Markers created from TeX symbols*
- *Markers created from Paths*

For a list of all markers see also the *`matplotlib.markers`* documentation.

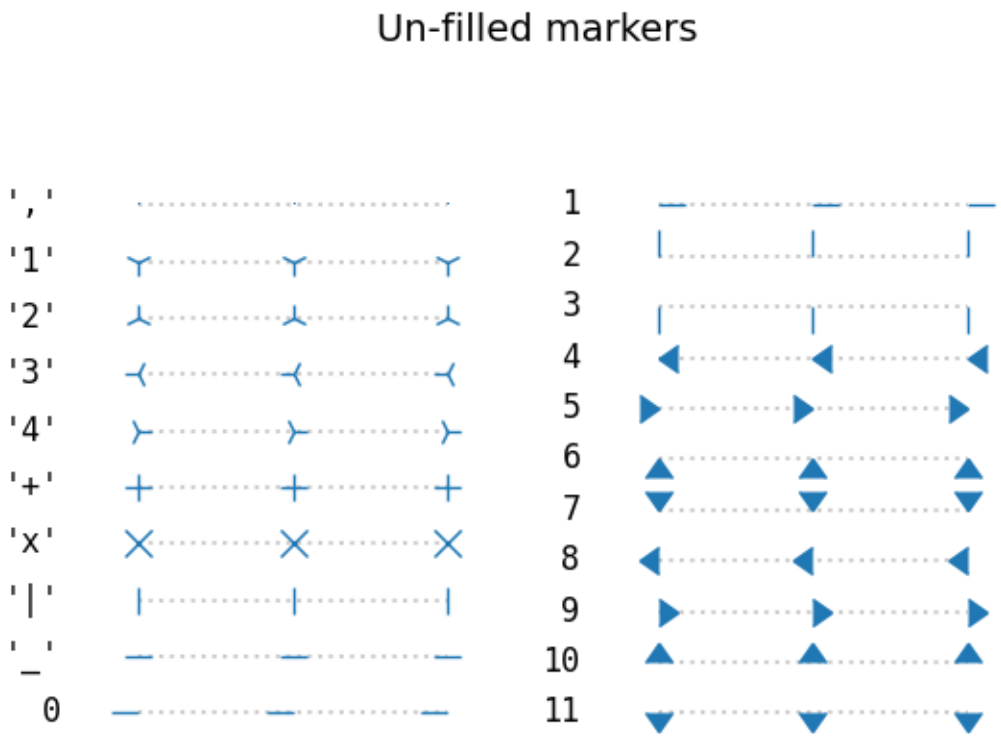
For example usages see *Marker examples*.

```
import matplotlib.pyplot as plt

from matplotlib.lines import Line2D
from matplotlib.markers import MarkerStyle
from matplotlib.transforms import Affine2D

text_style = dict(horizontalalignment='right', verticalalignment='center',
                  fontsize=12, fontfamily='monospace')
marker_style = dict(linestyle=':', color='0.8', markersize=10,
                    markerfacecolor="tab:blue", markeredgecolor="tab:blue")


def format_axes(ax):
    ax.margins(0.2)
    ax.set_axis_off()
    ax.invert_yaxis()


def split_list(a_list):
    i_half = len(a_list) // 2
    return a_list[:i_half], a_list[i_half:]
```

### Unfilled markers

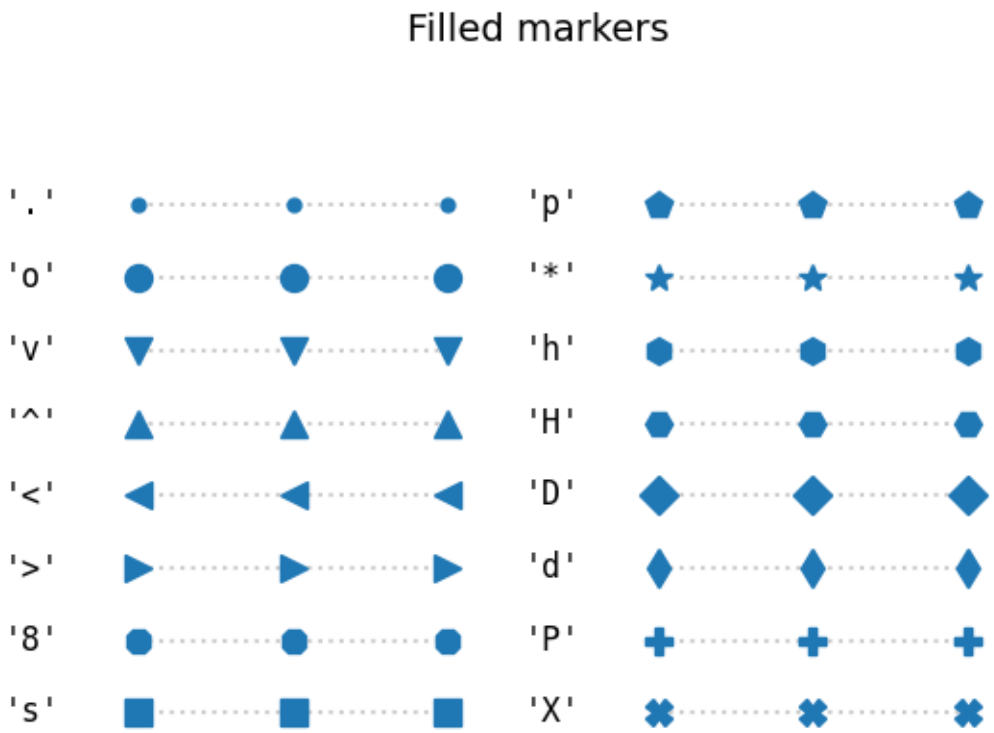
Unfilled markers are single-colored.

```
fig, axs = plt.subplots(ncols=2)
fig.suptitle('Un-filled markers', fontsize=14)

# Filter out filled markers and marker settings that do nothing.
unfilled_markers = [m for m, func in Line2D.markers.items()
                    if func != 'nothing' and m not in Line2D.filled_markers]

for ax, markers in zip(axs, split_list(unfilled_markers)):
    for y, marker in enumerate(markers):
        ax.text(-0.5, y, repr(marker), **text_style)
        ax.plot([y] * 3, marker=marker, **marker_style)
    format_axes(ax)
```




### Filled markers

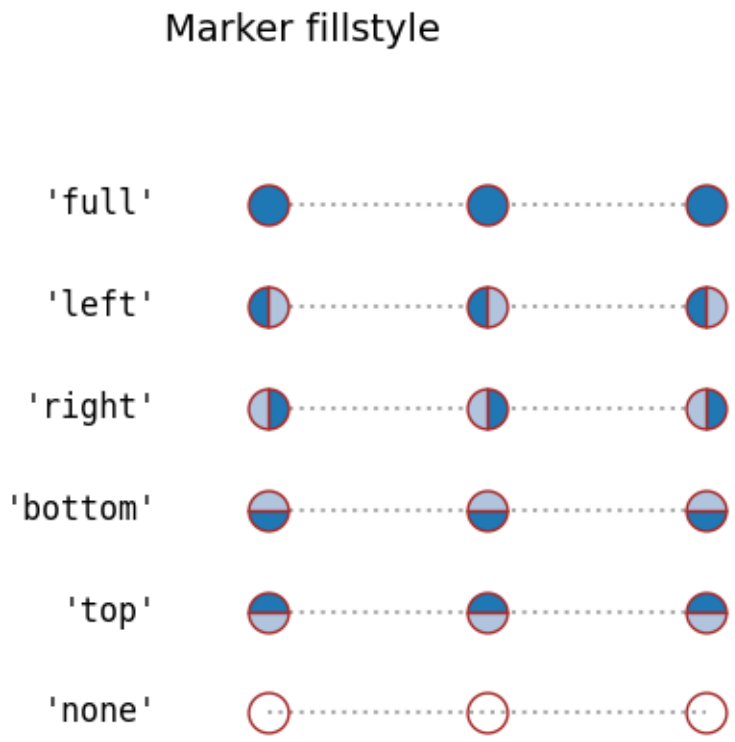
```
fig, axs = plt.subplots(ncols=2)
fig.suptitle('Filled markers', fontsize=14)
for ax, markers in zip(axs, split_list(Line2D.filled_markers)):
    for y, marker in enumerate(markers):
        ax.text(-0.5, y, repr(marker), **text_style)
        ax.plot([y] * 3, marker=marker, **marker_style)
    format_axes(ax)
```




### Marker fill styles

The edge color and fill color of filled markers can be specified separately. Additionally, the `fillstyle` can be configured to be unfilled, fully filled, or half-filled in various directions. The half-filled styles use `markerfacecoloralt` as secondary fill color.

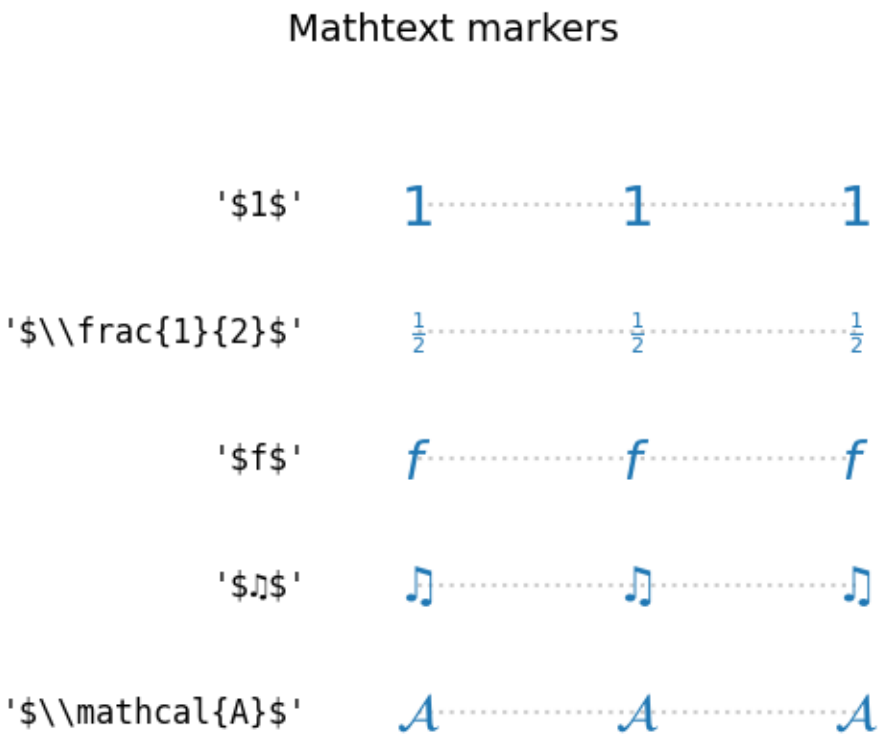
```
fig, ax = plt.subplots()
fig.suptitle('Marker fillstyle', fontsize=14)
fig.subplots_adjust(left=0.4)

filled_marker_style = dict(marker='o', linestyle=':', markersize=15,
                           color='darkgrey',
                           markerfacecolor='tab:blue',
                           markerfacecoloralt='lightsteelblue',
                           markeredgecolor='brown')

for y, fill_style in enumerate(Line2D.fillStyles):
    ax.text(-0.5, y, repr(fill_style), **text_style)
    ax.plot([y] * 3, fillstyle=fill_style, **filled_marker_style)
format_axes(ax)
```




### Markers created from TeX symbols

Use *MathText*, to use custom marker symbols, like e.g. `"$\u266B$"`. For an overview over the STIX font symbols refer to the STIX font table. Also see the *STIX Fonts*.

```
fig, ax = plt.subplots()
fig.suptitle('Mathtext markers', fontsize=14)
fig.subplots_adjust(left=0.4)

marker_style.update(markeredgecolor="none", markersize=15)
markers = ["$1$", r"$\frac{1}{2}$", "$f$", "$\u266B$", r"$\mathcal{A}$"]

for y, marker in enumerate(markers):
    # Escape dollars so that the text is written "as is", not as mathtext.
    ax.text(-0.5, y, repr(marker).replace("$", r"\$"), **text_style)
    ax.plot([y] * 3, marker=marker, **marker_style)
format_axes(ax)
```




### Markers created from Paths

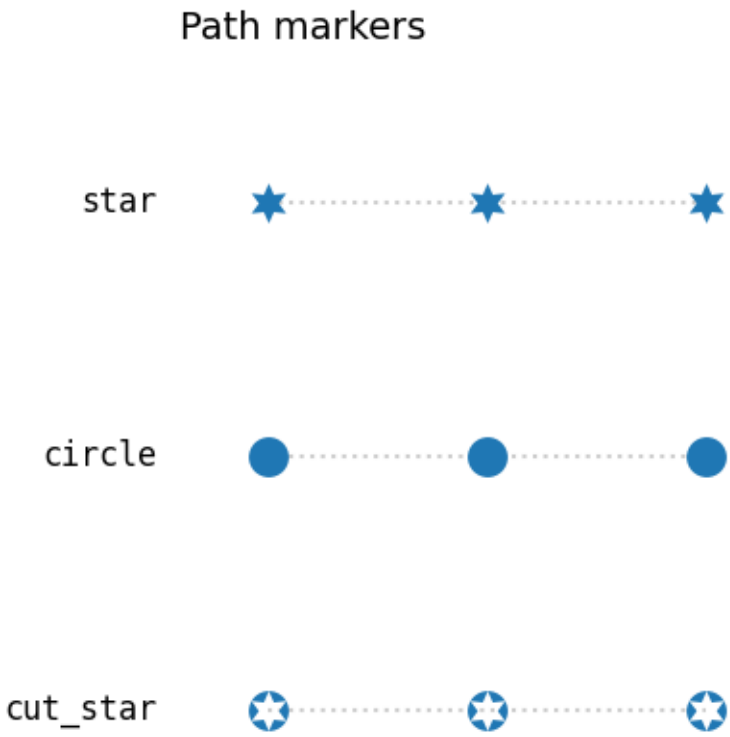
Any `Path` can be used as a marker. The following example shows two simple paths *star* and *circle*, and a more elaborate path of a circle with a cut-out star.

```
import numpy as np

import matplotlib.path as mpath

star = mpath.Path.unit_regular_star(6)
circle = mpath.Path.unit_circle()
# concatenate the circle with an internal cutout of the star
cut_star = mpath.Path(
    vertices=np.concatenate([circle.vertices, star.vertices[::-1, ...]]),
    codes=np.concatenate([circle.codes, star.codes]))

fig, ax = plt.subplots()
fig.suptitle('Path markers', fontsize=14)
fig.subplots_adjust(left=0.4)

markers = {'star': star, 'circle': circle, 'cut_star': cut_star}
```

(continues on next page)

(continued from previous page)

```
for y, (name, marker) in enumerate(markers.items()):
    ax.text(-0.5, y, name, **text_style)
    ax.plot([y] * 3, marker=marker, **marker_style)
format_axes(ax)
```




### Advanced marker modifications with transform

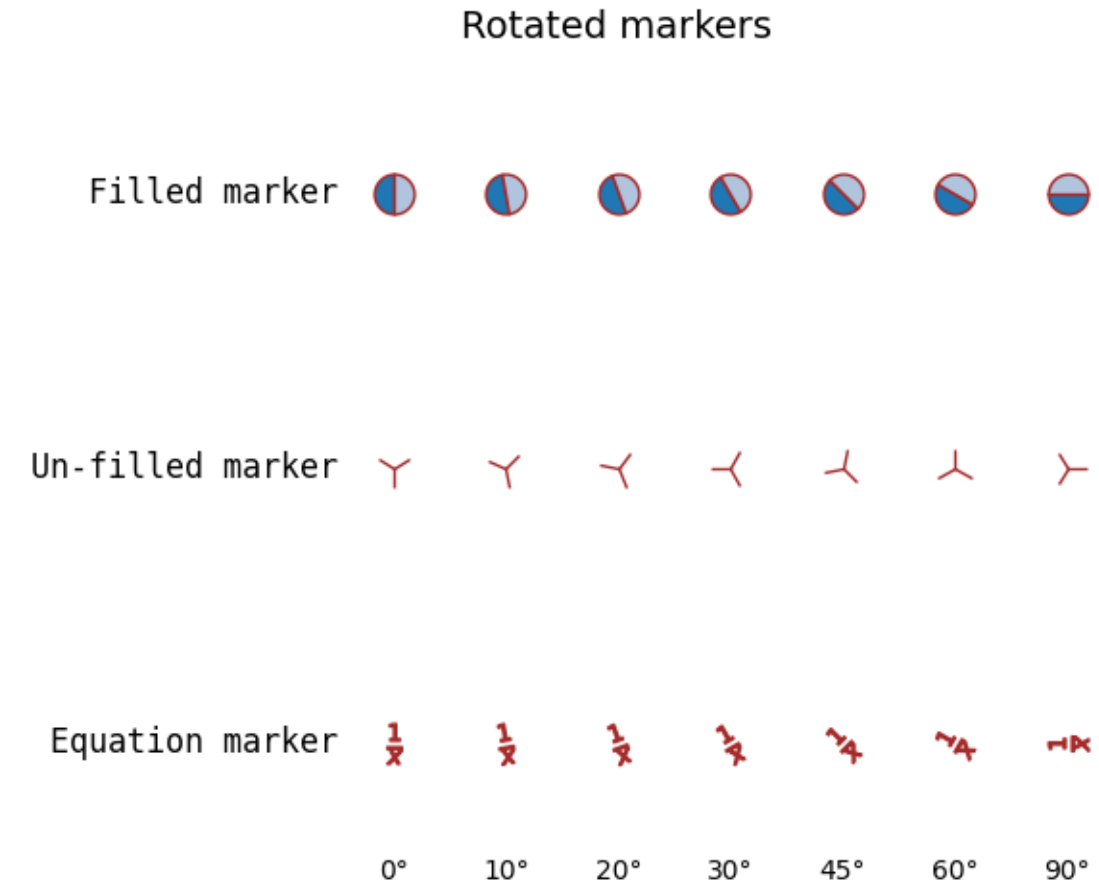
Markers can be modified by passing a transform to the MarkerStyle constructor. Following example shows how a supplied rotation is applied to several marker shapes.

```
common_style = {k: v for k, v in filled_marker_style.items() if k != 'marker'}
angles = [0, 10, 20, 30, 45, 60, 90]

fig, ax = plt.subplots()
fig.suptitle('Rotated markers', fontsize=14)

ax.text(-0.5, 0, 'Filled marker', **text_style)
for x, theta in enumerate(angles):
    t = Affine2D().rotate_deg(theta)
```

(continues on next page)

(continued from previous page)

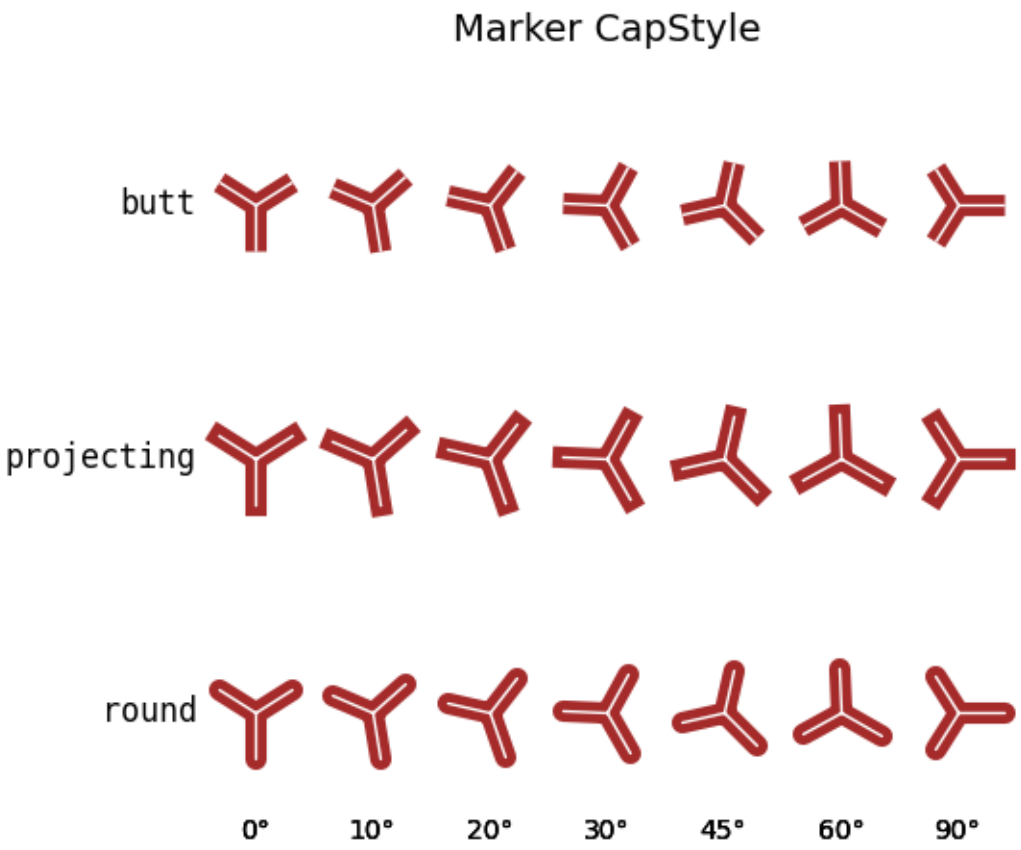
```
    ax.plot(x, 0, marker=MarkerStyle('o', 'left', t), **common_style)

ax.text(-0.5, 1, 'Un-filled marker', **text_style)
for x, theta in enumerate(angles):
    t = Affine2D().rotate_deg(theta)
    ax.plot(x, 1, marker=MarkerStyle('1', 'left', t), **common_style)

ax.text(-0.5, 2, 'Equation marker', **text_style)
for x, theta in enumerate(angles):
    t = Affine2D().rotate_deg(theta)
    eq = r'$\frac{1}{x}$'
    ax.plot(x, 2, marker=MarkerStyle(eq, 'left', t), **common_style)

for x, theta in enumerate(angles):
    ax.text(x, 2.5, f"{theta}°", horizontalalignment="center")
format_axes(ax)

fig.tight_layout()
```



### Setting marker cap style and join style

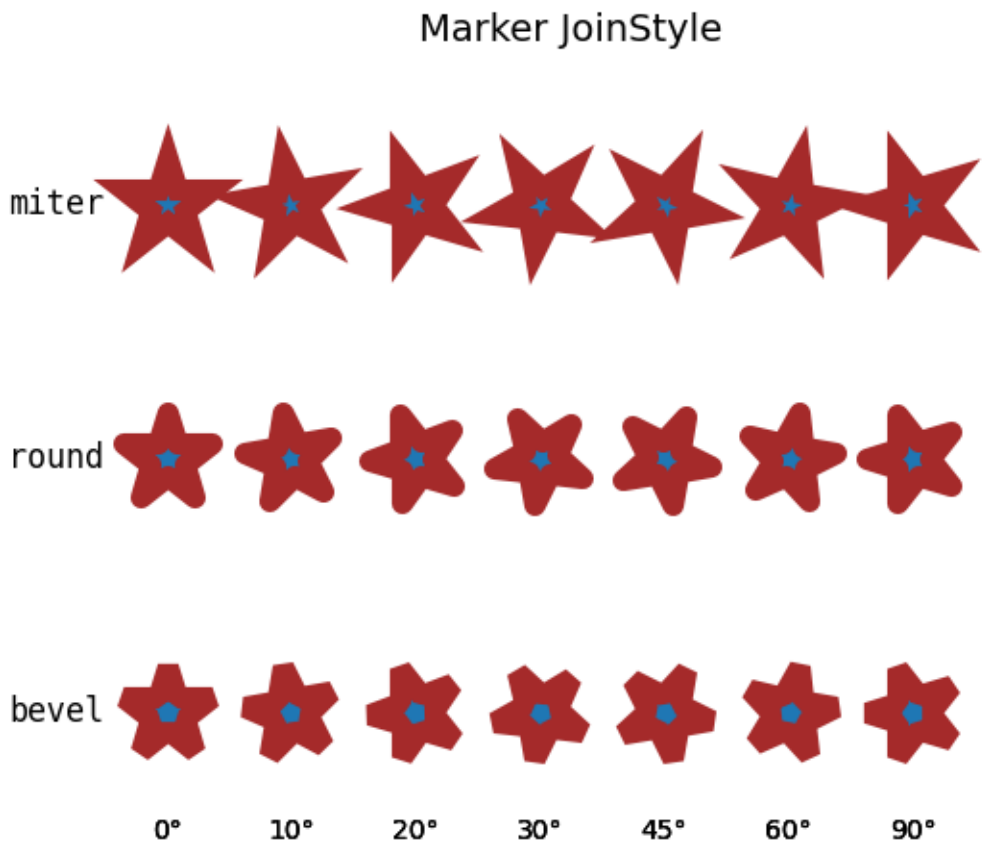
Markers have default cap and join styles, but these can be customized when creating a MarkerStyle.

```
from matplotlib.markers import CapStyle, JoinStyle

marker_inner = dict(markersize=35,
                    markerfacecolor='tab:blue',
                    markerfacecoloralt='lightsteelblue',
                    markeredgecolor='brown',
                    markeredgewidth=8,
                    )

marker_outer = dict(markersize=35,
                    markerfacecolor='tab:blue',
                    markerfacecoloralt='lightsteelblue',
                    markeredgecolor='white',
                    markeredgewidth=1,
                    )

fig, ax = plt.subplots()
fig.suptitle('Marker CapStyle', fontsize=14)
fig.subplots_adjust(left=0.1)

for y, cap_style in enumerate(CapStyle):
    ax.text(-0.5, y, cap_style.name, **text_style)
    for x, theta in enumerate(angles):
        t = Affine2D().rotate_deg(theta)
        m = MarkerStyle('1', transform=t, capstyle=cap_style)
        ax.plot(x, y, marker=m, **marker_inner)
        ax.plot(x, y, marker=m, **marker_outer)
        ax.text(x, len(CapStyle) - .5, f'{theta}°', ha='center')
format_axes(ax)
```




Modifying the join style:

```
fig, ax = plt.subplots()
fig.suptitle('Marker JoinStyle', fontsize=14)
fig.subplots_adjust(left=0.05)

for y, join_style in enumerate(JoinStyle):
    ax.text(-0.5, y, join_style.name, **text_style)
    for x, theta in enumerate(angles):
        t = Affine2D().rotate_deg(theta)
        m = MarkerStyle('*', transform=t, joinstyle=join_style)
        ax.plot(x, y, marker=m, **marker_inner)
        ax.text(x, len(JoinStyle) - .5, f'{theta}°', ha='center')
format_axes(ax)

plt.show()
```




**Total running time of the script:** (0 minutes 2.164 seconds)

### Markevery Demo

The `markevery` property of *Line2D* allows drawing markers at a subset of data points.

The list of possible parameters is specified at *Line2D.set_markevery*. In short:

- A single integer N draws every N-th marker.
- A tuple of integers (start, N) draws every N-th marker, starting at data index *start*.
- A list of integers draws the markers at the specified indices.
- A slice draws the markers at the sliced indices.
- A float specifies the distance between markers as a fraction of the Axes diagonal in screen space. This will lead to a visually uniform distribution of the points along the line, irrespective of scales and zooming.

```
import matplotlib.pyplot as plt
import numpy as np

# define a list of markevery cases to plot
```

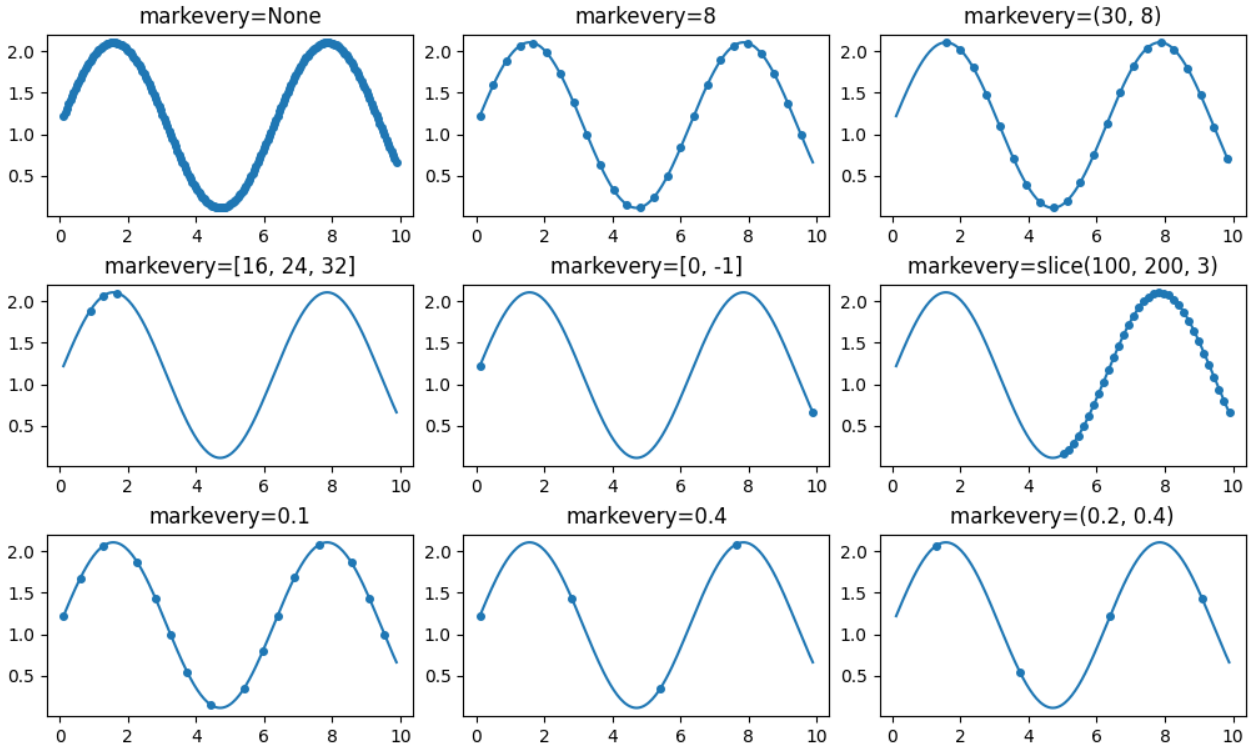
(continues on next page)

(continued from previous page)

```
cases = [
    None,
    8,
    (30, 8),
    [16, 24, 32],
    [0, -1],
    slice(100, 200, 3),
    0.1,
    0.4,
    (0.2, 0.4)
]

# data points
delta = 0.11
x = np.linspace(0, 10 - 2 * delta, 200) + delta
y = np.sin(x) + 1.0 + delta
```

### markevery with linear scales

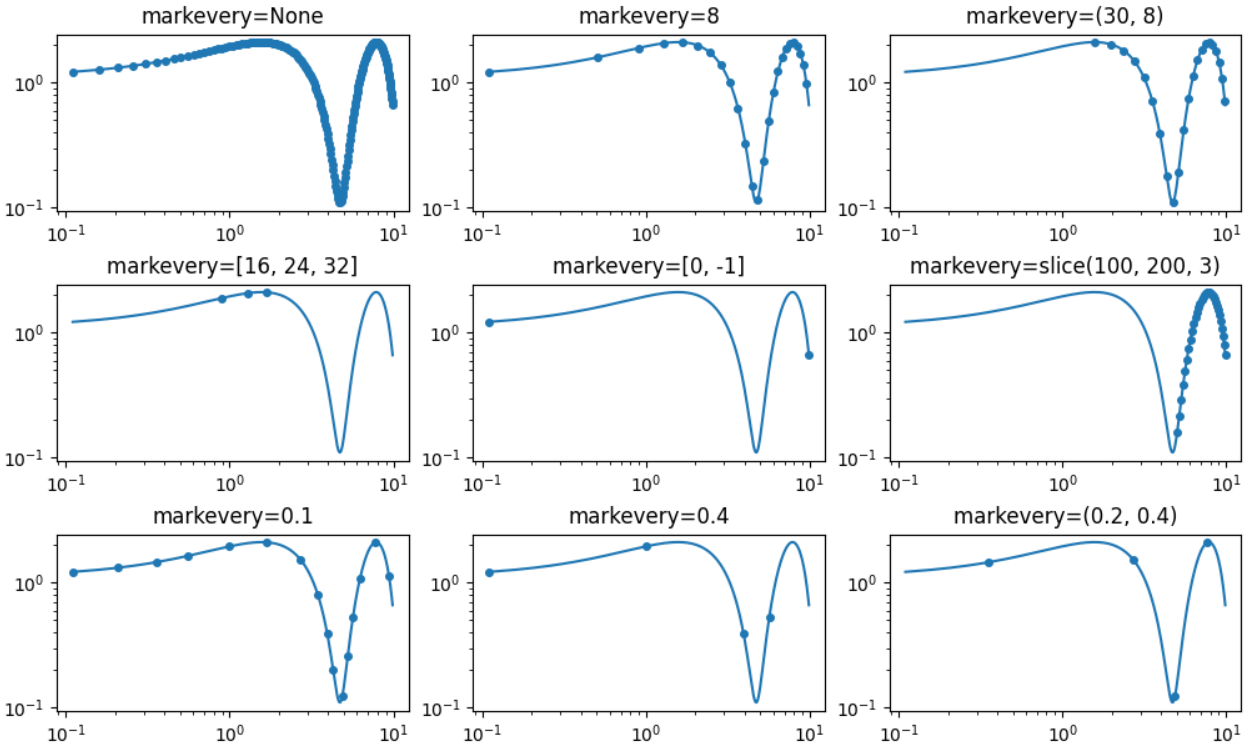
```
fig, axs = plt.subplots(3, 3, figsize=(10, 6), layout='constrained')
for ax, markevery in zip(axs.flat, cases):
    ax.set_title(f'markevery={markevery}')
    ax.plot(x, y, 'o', ls='-', ms=4, markevery=markevery)
```



### markevery with log scales

Note that the log scale causes a visual asymmetry in the marker distance for when subsampling the data using an integer. In contrast, subsampling on fraction of figure size creates even distributions, because it's based on fractions of the Axes diagonal, not on data coordinates or data indices.

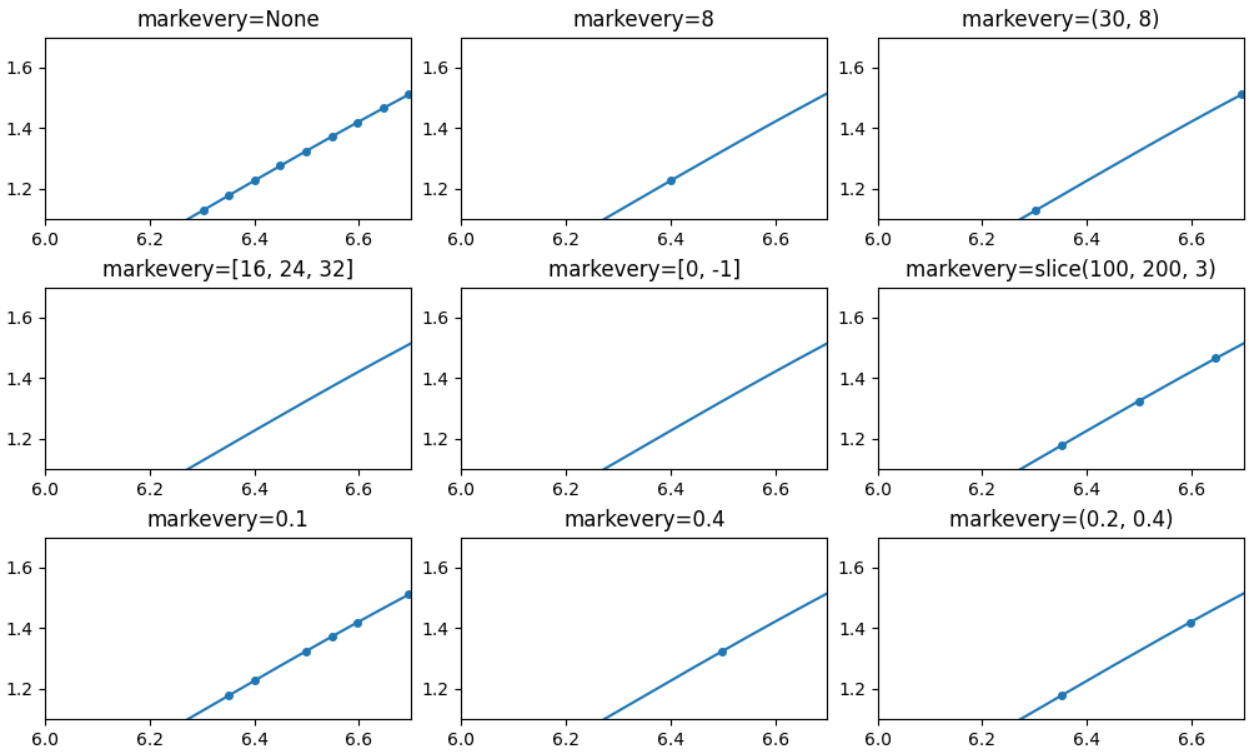
```
fig, axs = plt.subplots(3, 3, figsize=(10, 6), layout='constrained')
for ax, markevery in zip(axs.flat, cases):
    ax.set_title(f'markevery={markevery}')
    ax.set_xscale('log')
    ax.set_yscale('log')
    ax.plot(x, y, 'o', ls='-', ms=4, markevery=markevery)
```




### markevery on zoomed plots

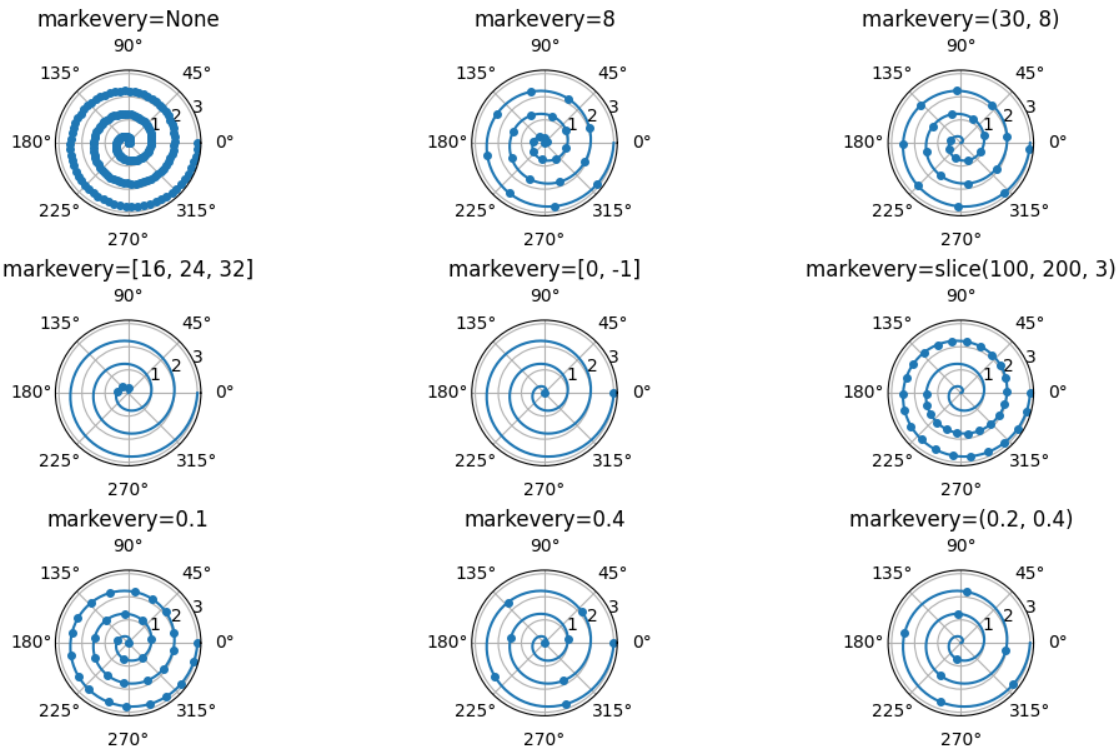
Integer-based *markevery* specifications select points from the underlying data and are independent on the view. In contrast, float-based specifications are related to the Axes diagonal. While zooming does not change the Axes diagonal, it changes the displayed data range, and more points will be displayed when zooming.

```
fig, axs = plt.subplots(3, 3, figsize=(10, 6), layout='constrained')
for ax, markevery in zip(axs.flat, cases):
    ax.set_title(f'markevery={markevery}')
    ax.plot(x, y, 'o', ls='-', ms=4, markevery=markevery)
    ax.set_xlim((6, 6.7))
    ax.set_ylim((1.1, 1.7))
```




## markevery on polar plots

```
r = np.linspace(0, 3.0, 200)
theta = 2 * np.pi * r

fig, axs = plt.subplots(3, 3, figsize=(10, 6), layout='constrained',
                        subplot_kw={'projection': 'polar'})
for ax, markevery in zip(axs.flat, cases):
    ax.set_title(f'markevery={markevery}')
    ax.plot(theta, r, 'o', ls='-', ms=4, markevery=markevery)

plt.show()
```




**Total running time of the script:** (0 minutes 7.902 seconds)

## Plotting masked and NaN values

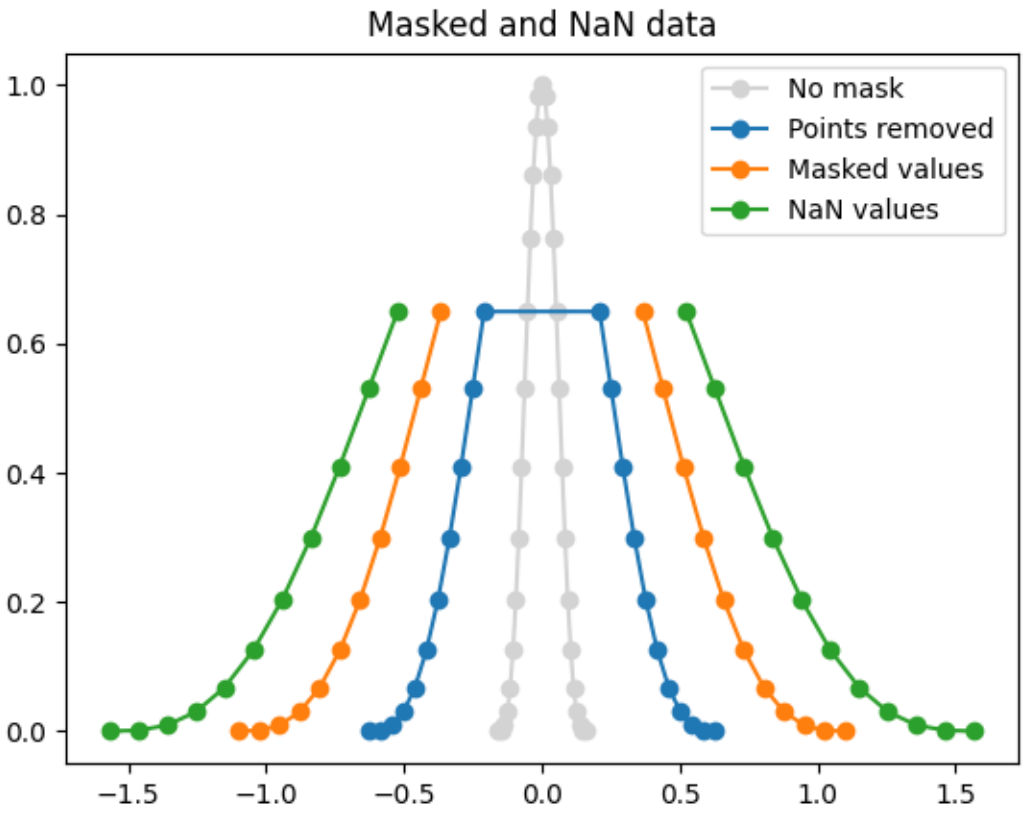
Sometimes you need to plot data with missing values.

One possibility is to simply remove undesired data points. The line plotted through the remaining data will be continuous, and not indicate where the missing data is located.

If it is useful to have gaps in the line where the data is missing, then the undesired points can be indicated using a masked array or by setting their values to NaN. No marker will be drawn where either x or y are masked and, if plotting with a line, it will be broken there.

The following example illustrates the three cases:

1) Removing points.
2) Masking points.
3) Setting to NaN.




```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(-np.pi/2, np.pi/2, 31)
y = np.cos(x)**3

# 1) remove points where y > 0.7
x2 = x[y <= 0.7]
y2 = y[y <= 0.7]

# 2) mask points where y > 0.7
y3 = np.ma.masked_where(y > 0.7, y)

# 3) set to NaN where y > 0.7
y4 = y.copy()
y4[y3 > 0.7] = np.nan

plt.plot(x*0.1, y, 'o-', color='lightgrey', label='No mask')
plt.plot(x2*0.4, y2, 'o-', label='Points removed')
plt.plot(x*0.7, y3, 'o-', label='Masked values')
plt.plot(x*1.0, y4, 'o-', label='NaN values')
plt.legend()
plt.title('Masked and NaN data')
```

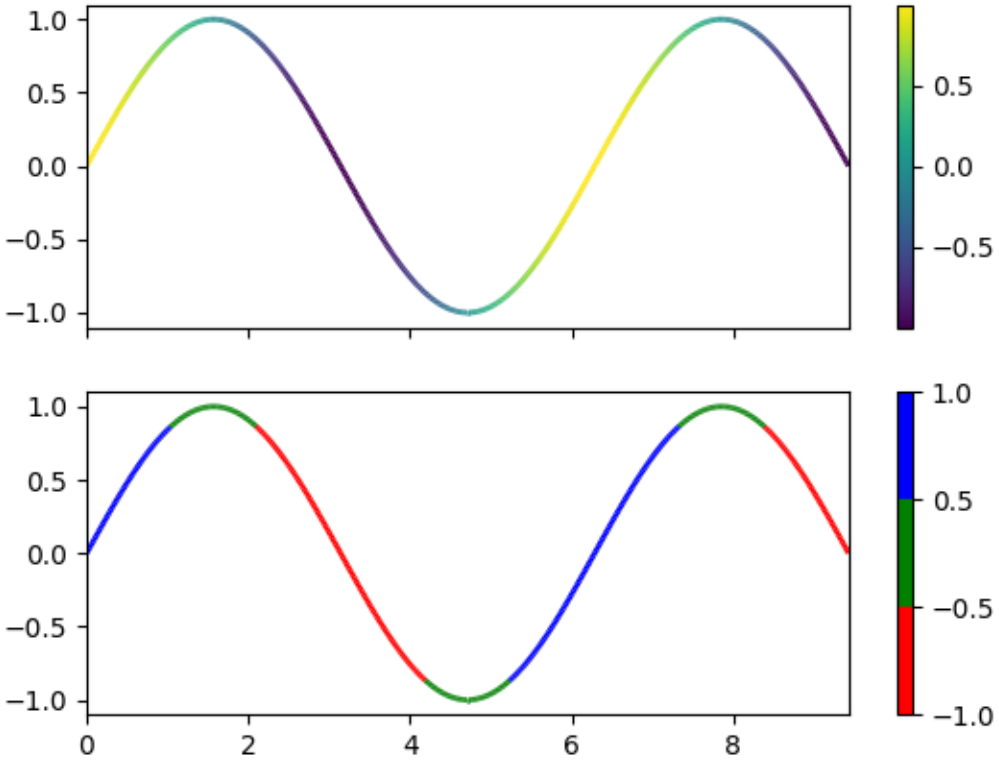
(continues on next page)

(continued from previous page)

```
plt.show()
```

### Multicolored lines

This example shows how to make a multicolored line. In this example, the line is colored based on its derivative.




```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.collections import LineCollection
from matplotlib.colors import BoundaryNorm, ListedColormap

x = np.linspace(0, 3 * np.pi, 500)
y = np.sin(x)
dydx = np.cos(0.5 * (x[:-1] + x[1:]))  # first derivative

# Create a set of line segments so that we can color them individually
# This creates the points as an N x 1 x 2 array so that we can stack points
# together easily to get the segments. The segments array for line collection
```

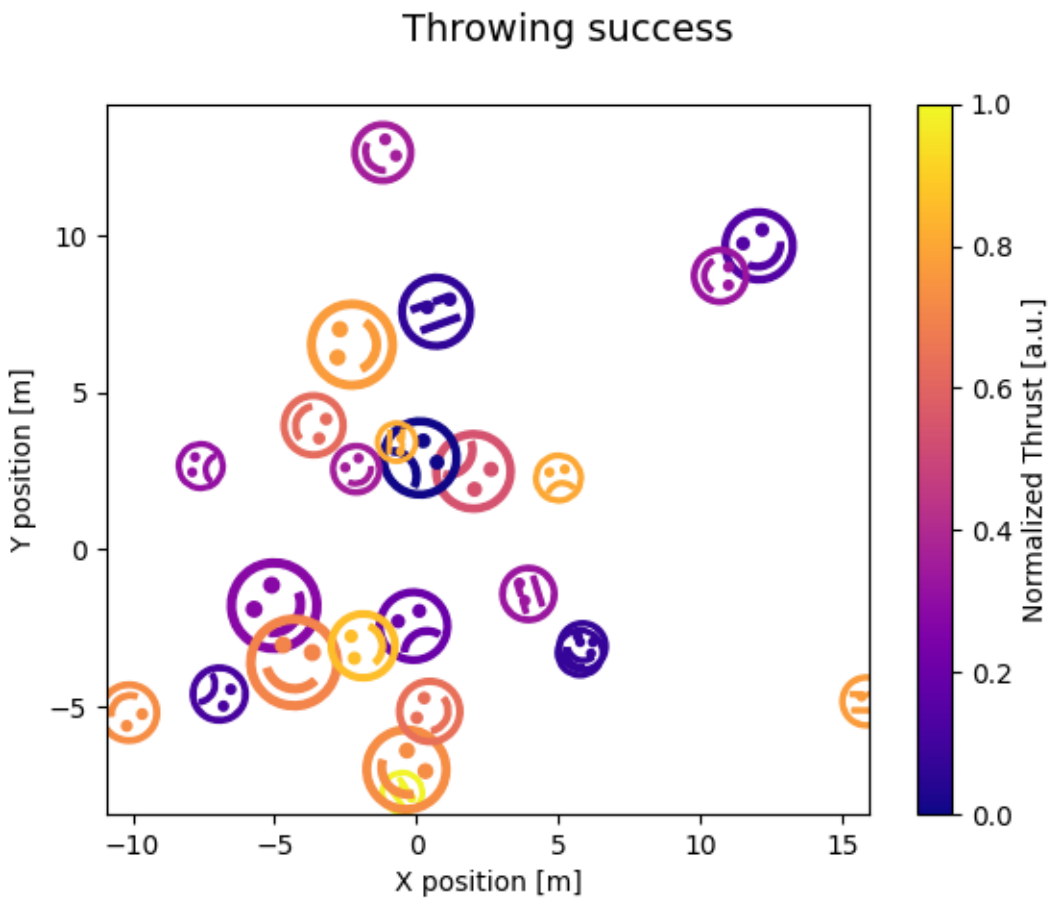
(continues on next page)

(continued from previous page)

```
# needs to be (numlines) x (points per line) x 2 (for x and y)
points = np.array([x, y]).T.reshape(-1, 1, 2)
segments = np.concatenate([points[:-1], points[1:]], axis=1)

fig, axs = plt.subplots(2, 1, sharex=True, sharey=True)

# Create a continuous norm to map from data points to colors
norm = plt.Normalize(dydx.min(), dydx.max())
lc = LineCollection(segments, cmap='viridis', norm=norm)
# Set the values used for colormapping
lc.set_array(dydx)
lc.set_linewidth(2)
line = axs[0].add_collection(lc)
fig.colorbar(line, ax=axs[0])

# Use a boundary norm instead
cmap = ListedColormap(['r', 'g', 'b'])
norm = BoundaryNorm([-1, -0.5, 0.5, 1], cmap.N)
lc = LineCollection(segments, cmap=cmap, norm=norm)
lc.set_array(dydx)
lc.set_linewidth(2)
line = axs[1].add_collection(lc)
fig.colorbar(line, ax=axs[1])

axs[0].set_xlim(x.min(), x.max())
axs[0].set_ylim(-1.1, 1.1)
plt.show()
```

### Mapping marker properties to multivariate data

This example shows how to use different properties of markers to plot multivariate datasets. Here we represent a successful baseball throw as a smiley face with marker size mapped to the skill of thrower, marker rotation to the take-off angle, and thrust to the marker color.




```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.colors import Normalize
from matplotlib.markers import MarkerStyle
from matplotlib.text import TextPath
from matplotlib.transforms import Affine2D

SUCCESS_SYMBOLS = [
    TextPath((0, 0), "☹"),
    TextPath((0, 0), "😐"),
    TextPath((0, 0), "☺"),
]

N = 25
np.random.seed(42)
skills = np.random.uniform(5, 80, size=N) * 0.1 + 5
takeoff_angles = np.random.normal(0, 90, N)
thrusts = np.random.uniform(size=N)
successful = np.random.randint(0, 3, size=N)
positions = np.random.normal(size=(N, 2)) * 5
data = zip(skills, takeoff_angles, thrusts, successful, positions)
```

(continues on next page)

(continued from previous page)

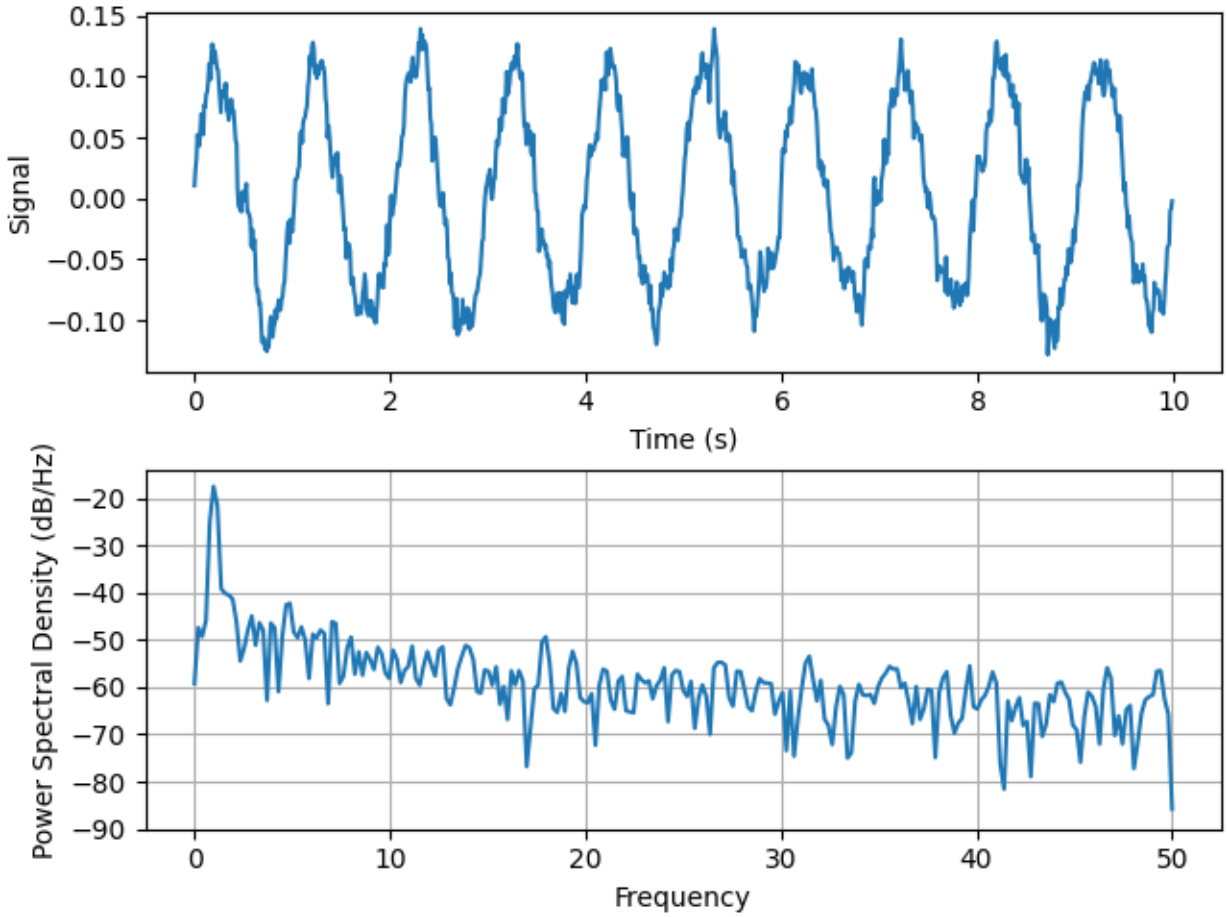
```
cmap = plt.colormaps["plasma"]
fig, ax = plt.subplots()
fig.suptitle("Throwing success", size=14)
for skill, takeoff, thrust, mood, pos in data:
    t = Affine2D().scale(skill).rotate_deg(takeoff)
    m = MarkerStyle(SUCCESS_SYMBOLS[mood], transform=t)
    ax.plot(pos[0], pos[1], marker=m, color=cmap(thrust))
fig.colorbar(plt.cm.ScalarMappable(norm=Normalize(0, 1), cmap=cmap),
             ax=ax, label="Normalized Thrust [a.u.]")
ax.set_xlabel("X position [m]")
ax.set_ylabel("Y position [m]")

plt.show()
```

### Power spectral density (PSD)

Plotting power spectral density (PSD) using *psd*.

The PSD is a common plot in the field of signal processing. NumPy has many useful libraries for computing a PSD. Below we demo a few examples of how this can be accomplished and visualized with Matplotlib.

```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib.mlab as mlab

# Fixing random state for reproducibility
np.random.seed(19680801)

dt = 0.01
t = np.arange(0, 10, dt)
nse = np.random.randn(len(t))
r = np.exp(-t / 0.05)

cnse = np.convolve(nse, r) * dt
cnse = cnse[:len(t)]
s = 0.1 * np.sin(2 * np.pi * t) + cnse

fig, (ax0, ax1) = plt.subplots(2, 1, layout='constrained')
ax0.plot(t, s)
ax0.set_xlabel('Time (s)')
ax0.set_ylabel('Signal')
ax1.psd(s, 512, 1 / dt)

plt.show()
```




Compare this with the equivalent Matlab code to accomplish the same thing:

```
dt = 0.01;
t = [0:dt:10];
nse = randn(size(t));
r = exp(-t/0.05);
cnse = conv(nse, r)*dt;
cnse = cnse(1:length(t));
s = 0.1*sin(2*pi*t) + cnse;

subplot(211)
plot(t, s)
subplot(212)
psd(s, 512, 1/dt)
```

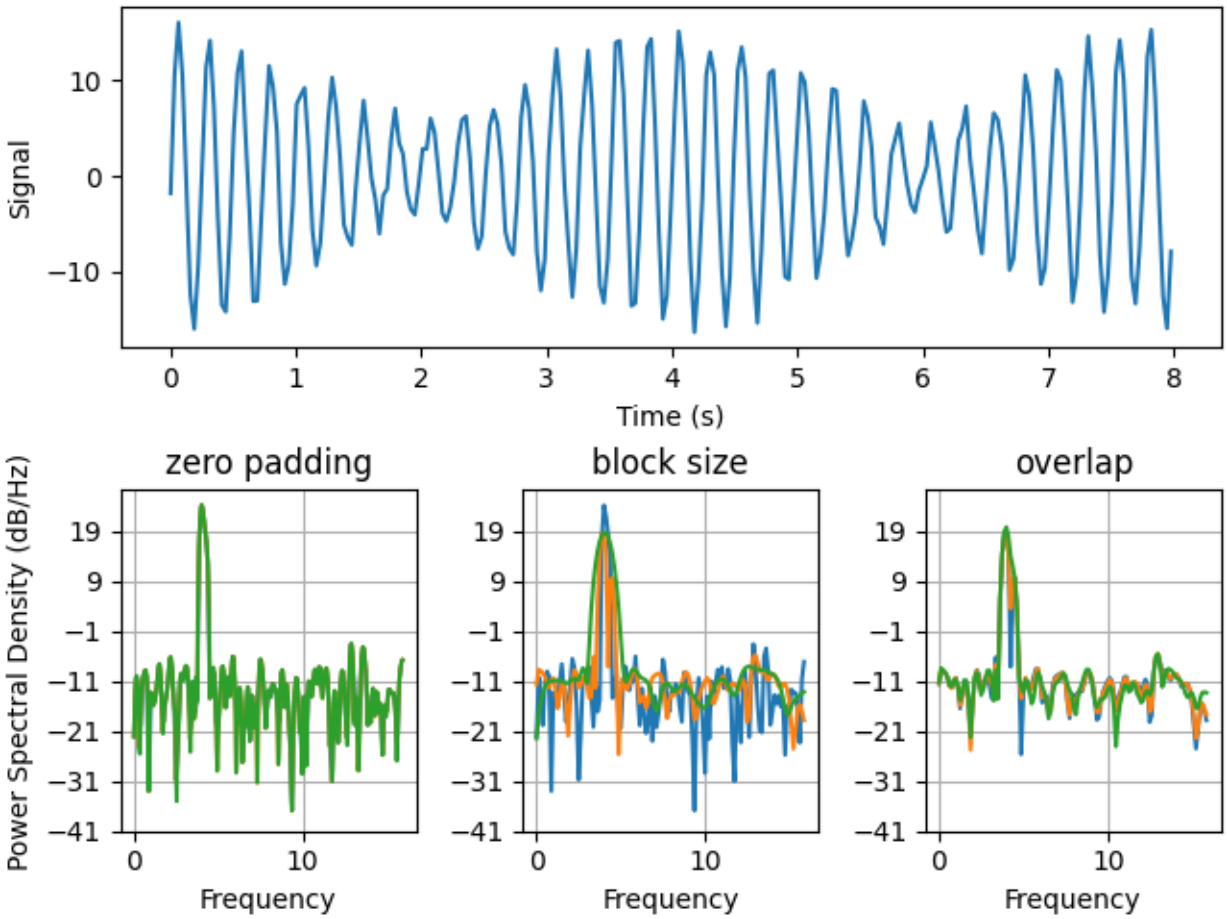
Below we'll show a slightly more complex example that demonstrates how padding affects the resulting PSD.

```
dt = np.pi / 100.
fs = 1. / dt
t = np.arange(0, 8, dt)
y = 10. * np.sin(2 * np.pi * 4 * t) + 5. * np.sin(2 * np.pi * 4.25 * t)
y = y + np.random.randn(*t.shape)
```

(continues on next page)

(continued from previous page)

```
# Plot the raw time series
fig, axs = plt.subplot_mosaic([
    ['signal', 'signal', 'signal'],
    ['zero padding', 'block size', 'overlap'],
], layout='constrained')

axs['signal'].plot(t, y)
axs['signal'].set_xlabel('Time (s)')
axs['signal'].set_ylabel('Signal')

# Plot the PSD with different amounts of zero padding. This uses the entire
# time series at once
axs['zero padding'].psd(y, NFFT=len(t), pad_to=len(t), Fs=fs)
axs['zero padding'].psd(y, NFFT=len(t), pad_to=len(t) * 2, Fs=fs)
axs['zero padding'].psd(y, NFFT=len(t), pad_to=len(t) * 4, Fs=fs)

# Plot the PSD with different block sizes, Zero pad to the length of the
# original data sequence.
axs['block size'].psd(y, NFFT=len(t), pad_to=len(t), Fs=fs)
axs['block size'].psd(y, NFFT=len(t) // 2, pad_to=len(t), Fs=fs)
axs['block size'].psd(y, NFFT=len(t) // 4, pad_to=len(t), Fs=fs)
axs['block size'].set_ylabel('')

# Plot the PSD with different amounts of overlap between blocks
axs['overlap'].psd(y, NFFT=len(t) // 2, pad_to=len(t), noverlap=0, Fs=fs)
axs['overlap'].psd(y, NFFT=len(t) // 2, pad_to=len(t),
                   noverlap=int(0.025 * len(t)), Fs=fs)
axs['overlap'].psd(y, NFFT=len(t) // 2, pad_to=len(t),
                   noverlap=int(0.1 * len(t)), Fs=fs)
axs['overlap'].set_ylabel('')
axs['overlap'].set_title('overlap')

for title, ax in axs.items():
    if title == 'signal':
        continue

    ax.set_title(title)
    ax.sharex(axs['zero padding'])
    ax.sharey(axs['zero padding'])

plt.show()
```



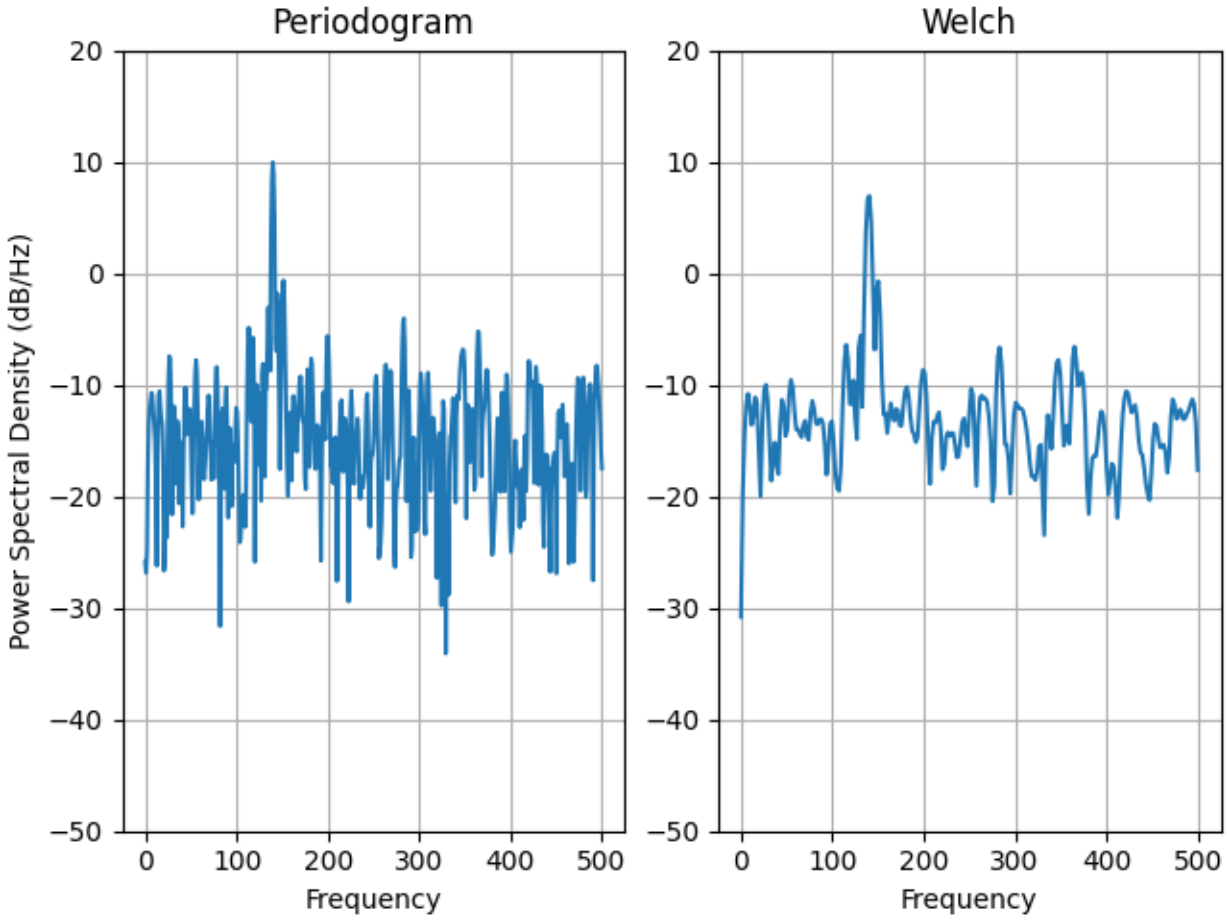

This is a ported version of a MATLAB example from the signal processing toolbox that showed some difference at one time between Matplotlib's and MATLAB's scaling of the PSD.

```
fs = 1000
t = np.linspace(0, 0.3, 301)
A = np.array([2, 8]).reshape(-1, 1)
f = np.array([150, 140]).reshape(-1, 1)
xn = (A * np.sin(2 * np.pi * f * t)).sum(axis=0)
xn += 5 * np.random.randn(*t.shape)

fig, (ax0, ax1) = plt.subplots(ncols=2, layout='constrained')

yticks = np.arange(-50, 30, 10)
yrange = (yticks[0], yticks[-1])
xticks = np.arange(0, 550, 100)

ax0.psd(xn, NFFT=301, Fs=fs, window=mlab.window_none, pad_to=1024,
        scale_by_freq=True)
ax0.set_title('Periodogram')
ax0.set_yticks(yticks)
ax0.set_xticks(xticks)
ax0.grid(True)
ax0.set_ylim(yrange)
```

(continues on next page)

(continued from previous page)

```
ax1.psd(xn, NFFT=150, Fs=fs, window=mlab.window_none, pad_to=512, noverlap=75,
        scale_by_freq=True)
ax1.set_title('Welch')
ax1.set_xticks(xticks)
ax1.set_yticks(yticks)
ax1.set_ylabel('')  # overwrite the y-label added by `psd`
ax1.grid(True)
ax1.set_ylim(yrange)

plt.show()
```




This is a ported version of a MATLAB example from the signal processing toolbox that showed some difference at one time between Matplotlib's and MATLAB's scaling of the PSD.

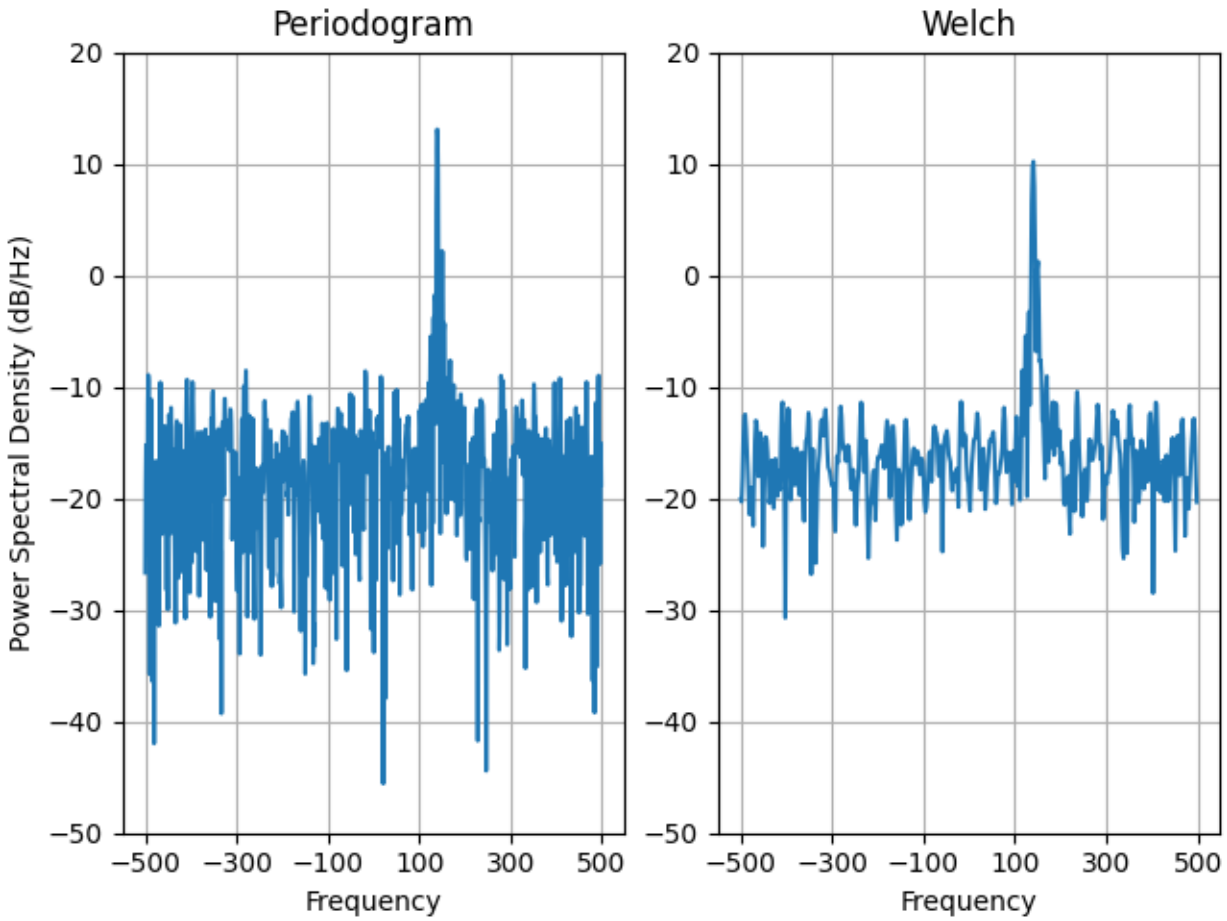
It uses a complex signal so we can see that complex PSD's work properly.

```
prng = np.random.RandomState(19680801)  # to ensure reproducibility

fs = 1000
t = np.linspace(0, 0.3, 301)
A = np.array([2, 8]).reshape(-1, 1)
```

(continues on next page)

(continued from previous page)

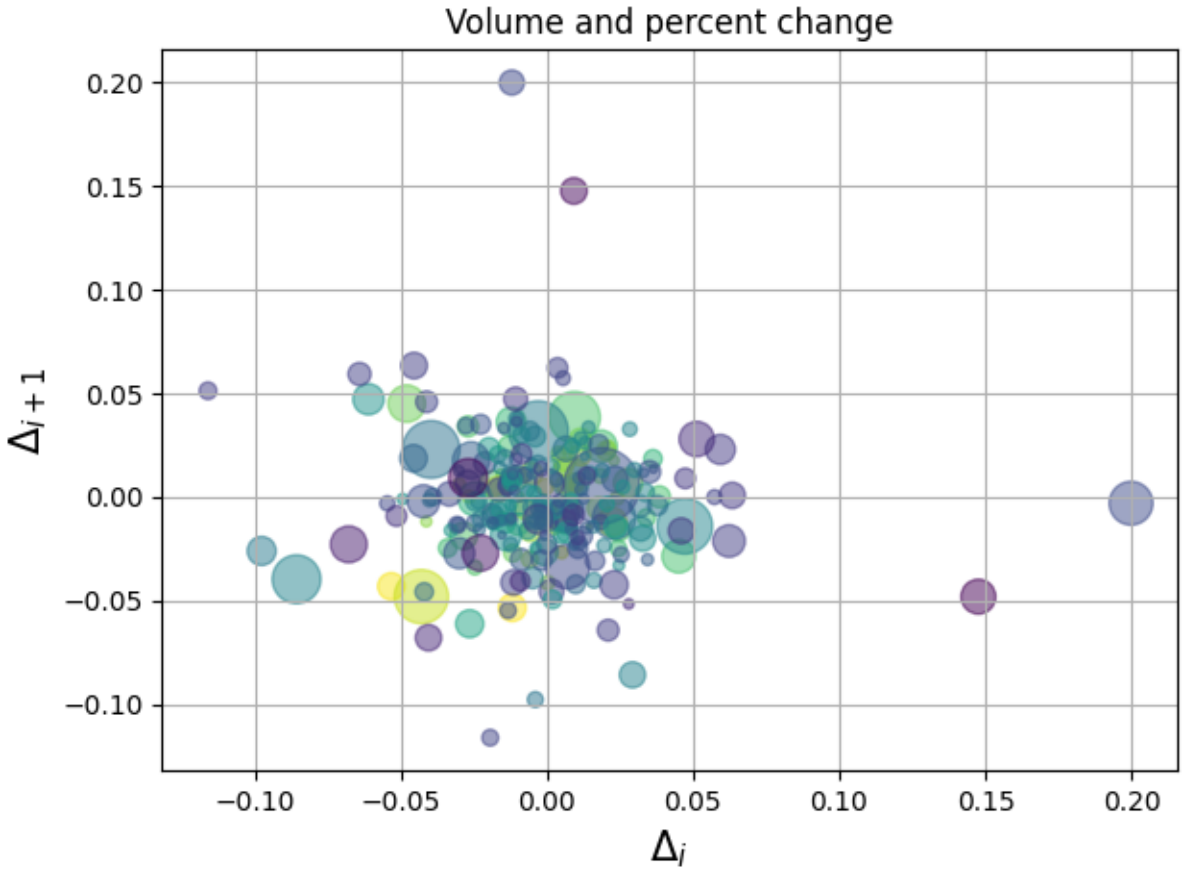
```
f = np.array([150, 140]).reshape(-1, 1)
xn = (A * np.exp(2j * np.pi * f * t)).sum(axis=0) + 5 * prng.randn(*t.shape)

fig, (ax0, ax1) = plt.subplots(ncols=2, layout='constrained')

yticks = np.arange(-50, 30, 10)
yrange = (yticks[0], yticks[-1])
xticks = np.arange(-500, 550, 200)

ax0.psd(xn, NFFT=301, Fs=fs, window=mlab.window_none, pad_to=1024,
        scale_by_freq=True)
ax0.set_title('Periodogram')
ax0.set_yticks(yticks)
ax0.set_xticks(xticks)
ax0.grid(True)
ax0.set_ylim(yrange)

ax1.psd(xn, NFFT=150, Fs=fs, window=mlab.window_none, pad_to=512, noverlap=75,
        scale_by_freq=True)
ax1.set_title('Welch')
ax1.set_xticks(xticks)
ax1.set_yticks(yticks)
ax1.set_ylabel('')  # overwrite the y-label added by `psd`
ax1.grid(True)
ax1.set_ylim(yrange)

plt.show()
```




**Total running time of the script:** (0 minutes 3.458 seconds)

### Scatter Demo2

Demo of scatter plot with varying marker colors and sizes.




```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib.cbook as cbook

# Load a numpy record array from yahoo csv data with fields date, open, high,
# low, close, volume, adj_close from the mpl-data/sample_data directory. The
# record array stores the date as an np.datetime64 with a day unit ('D') in
# the date column.
price_data = cbook.get_sample_data('goog.npz')['price_data']
price_data = price_data[-250:]  # get the most recent 250 trading days

delta1 = np.diff(price_data["adj_close"]) / price_data["adj_close"][:-1]

# Marker size in units of points^2
volume = (15 * price_data["volume"][:-2] / price_data["volume"][0])**2
close = 0.003 * price_data["close"][:-2] / 0.003 * price_data["open"][:-2]

fig, ax = plt.subplots()
ax.scatter(delta1[:-1], delta1[1:], c=close, s=volume, alpha=0.5)

ax.set_xlabel(r'$\Delta_i$', fontsize=15)
ax.set_ylabel(r'$\Delta_{i+1}$', fontsize=15)
```

(continues on next page)

(continued from previous page)

```
ax.set_title('Volume and percent change')

ax.grid(True)
fig.tight_layout()

plt.show()
```

### Scatter plot with histograms

Show the marginal distributions of a scatter plot as histograms at the sides of the plot.

For a nice alignment of the main Axes with the marginals, two options are shown below:

- *Defining the Axes positions using a gridspec*
- *Defining the Axes positions using inset_axes*

While `Axes.inset_axes` may be a bit more complex, it allows correct handling of main Axes with a fixed aspect ratio.

An alternative method to produce a similar figure using the `axes_grid1` toolkit is shown in the *Scatter Histogram (Locatable Axes)* example. Finally, it is also possible to position all Axes in absolute coordinates using `Figure.add_axes` (not shown here).

Let us first define a function that takes x and y data as input, as well as three Axes, the main Axes for the scatter, and two marginal Axes. It will then create the scatter and histograms inside the provided Axes.

```
import matplotlib.pyplot as plt
import numpy as np

# Fixing random state for reproducibility
np.random.seed(19680801)

# some random data
x = np.random.randn(1000)
y = np.random.randn(1000)


def scatter_hist(x, y, ax, ax_histx, ax_histy):
    # no labels
    ax_histx.tick_params(axis="x", labelbottom=False)
    ax_histy.tick_params(axis="y", labelleft=False)

    # the scatter plot:
    ax.scatter(x, y)

    # now determine nice limits by hand:
    binwidth = 0.25
    xymax = max(np.max(np.abs(x)), np.max(np.abs(y)))
```

(continues on next page)

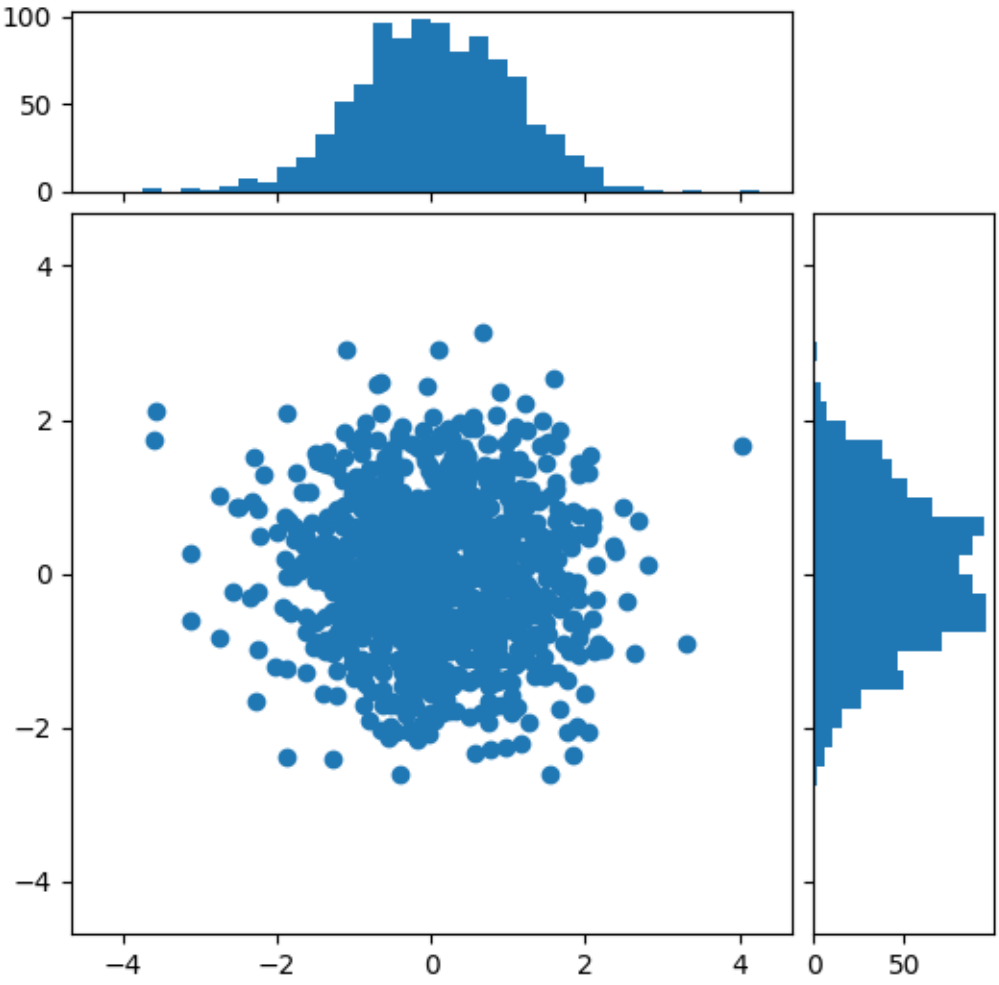
(continued from previous page)

```
    lim = (int(xymax/binwidth) + 1) * binwidth

    bins = np.arange(-lim, lim + binwidth, binwidth)
    ax_histx.hist(x, bins=bins)
    ax_histy.hist(y, bins=bins, orientation='horizontal')
```

### Defining the Axes positions using a gridspec

We define a gridspec with unequal width- and height-ratios to achieve desired layout. Also see the *Arranging multiple Axes in a Figure* tutorial.

```
# Start with a square Figure.
fig = plt.figure(figsize=(6, 6))
# Add a gridspec with two rows and two columns and a ratio of 1 to 4 between
# the size of the marginal Axes and the main Axes in both directions.
# Also adjust the subplot parameters for a square plot.
gs = fig.add_gridspec(2, 2,  width_ratios=(4, 1), height_ratios=(1, 4),
                      left=0.1, right=0.9, bottom=0.1, top=0.9,
                      wspace=0.05, hspace=0.05)
# Create the Axes.
ax = fig.add_subplot(gs[1, 0])
ax_histx = fig.add_subplot(gs[0, 0], sharex=ax)
ax_histy = fig.add_subplot(gs[1, 1], sharey=ax)
# Draw the scatter plot and marginals.
scatter_hist(x, y, ax, ax_histx, ax_histy)
```




### Defining the Axes positions using inset_axes

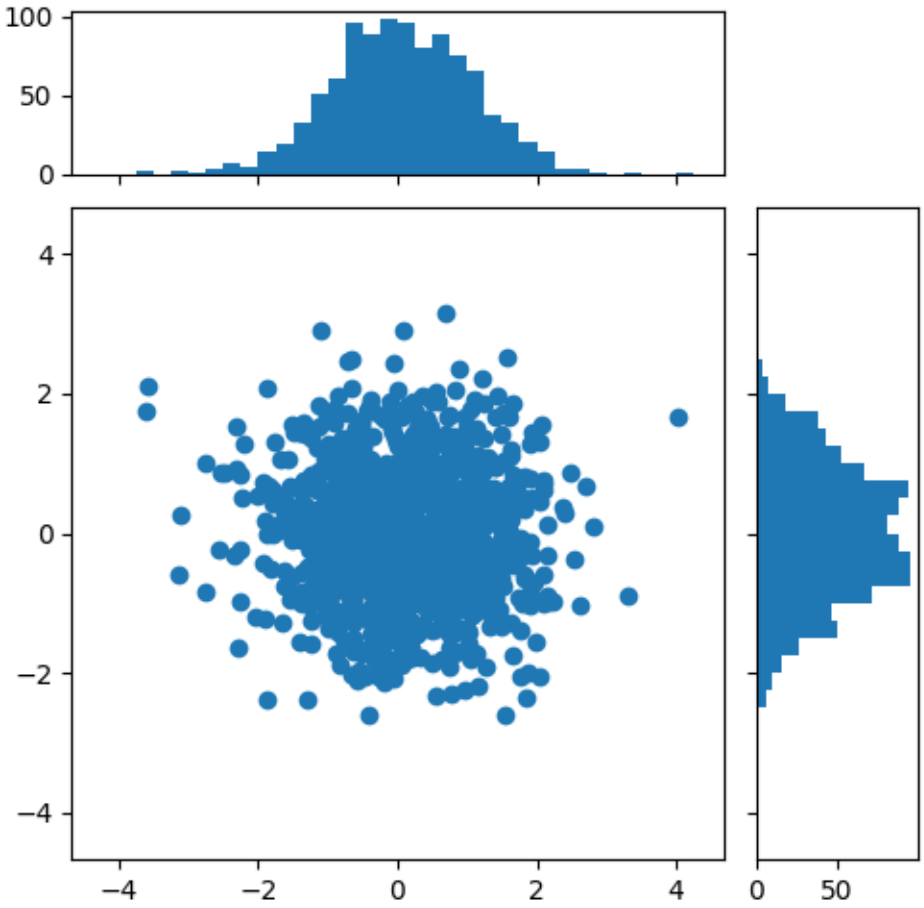
`inset_axes` can be used to position marginals *outside* the main Axes. The advantage of doing so is that the aspect ratio of the main Axes can be fixed, and the marginals will always be drawn relative to the position of the Axes.

```
# Create a Figure, which doesn't have to be square.
fig = plt.figure(layout='constrained')
# Create the main Axes, leaving 25% of the figure space at the top and on the
# right to position marginals.
ax = fig.add_gridspec(top=0.75, right=0.75).subplots()
# The main Axes' aspect can be fixed.
ax.set(aspect=1)
# Create marginal Axes, which have 25% of the size of the main Axes.  Note␣
```

(continues on next page)

(continued from previous page)

```
 ↪that
# the inset Axes are positioned *outside* (on the right and the top) of the
# main Axes, by specifying axes coordinates greater than 1.  Axes coordinates
# less than 0 would likewise specify positions on the left and the bottom of
# the main Axes.
ax_histx = ax.inset_axes([0, 1.05, 1, 0.25], sharex=ax)
ax_histy = ax.inset_axes([1.05, 0, 0.25, 1], sharey=ax)
# Draw the scatter plot and marginals.
scatter_hist(x, y, ax, ax_histx, ax_histy)

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

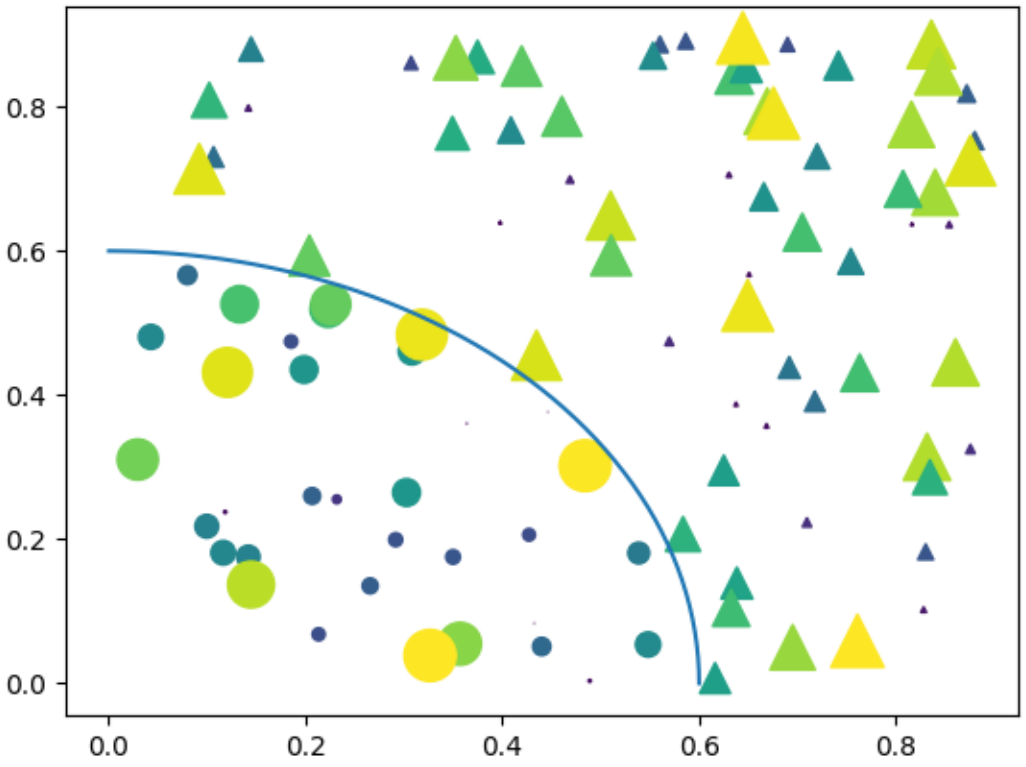
- *matplotlib.figure.Figure.add_subplot*
- *matplotlib.figure.Figure.add_gridspec*
- *matplotlib.axes.Axes.inset_axes*
- *matplotlib.axes.Axes.scatter*

- *matplotlib.axes.Axes.hist*

**Total running time of the script:** (0 minutes 1.006 seconds)

### Scatter Masked

Mask some data points and add a line demarking masked regions.




```
import matplotlib.pyplot as plt
import numpy as np

# Fixing random state for reproducibility
np.random.seed(19680801)


N = 100
r0 = 0.6
x = 0.9 * np.random.rand(N)
y = 0.9 * np.random.rand(N)
area = (20 * np.random.rand(N))**2  # 0 to 10 point radii
c = np.sqrt(area)
```

(continues on next page)

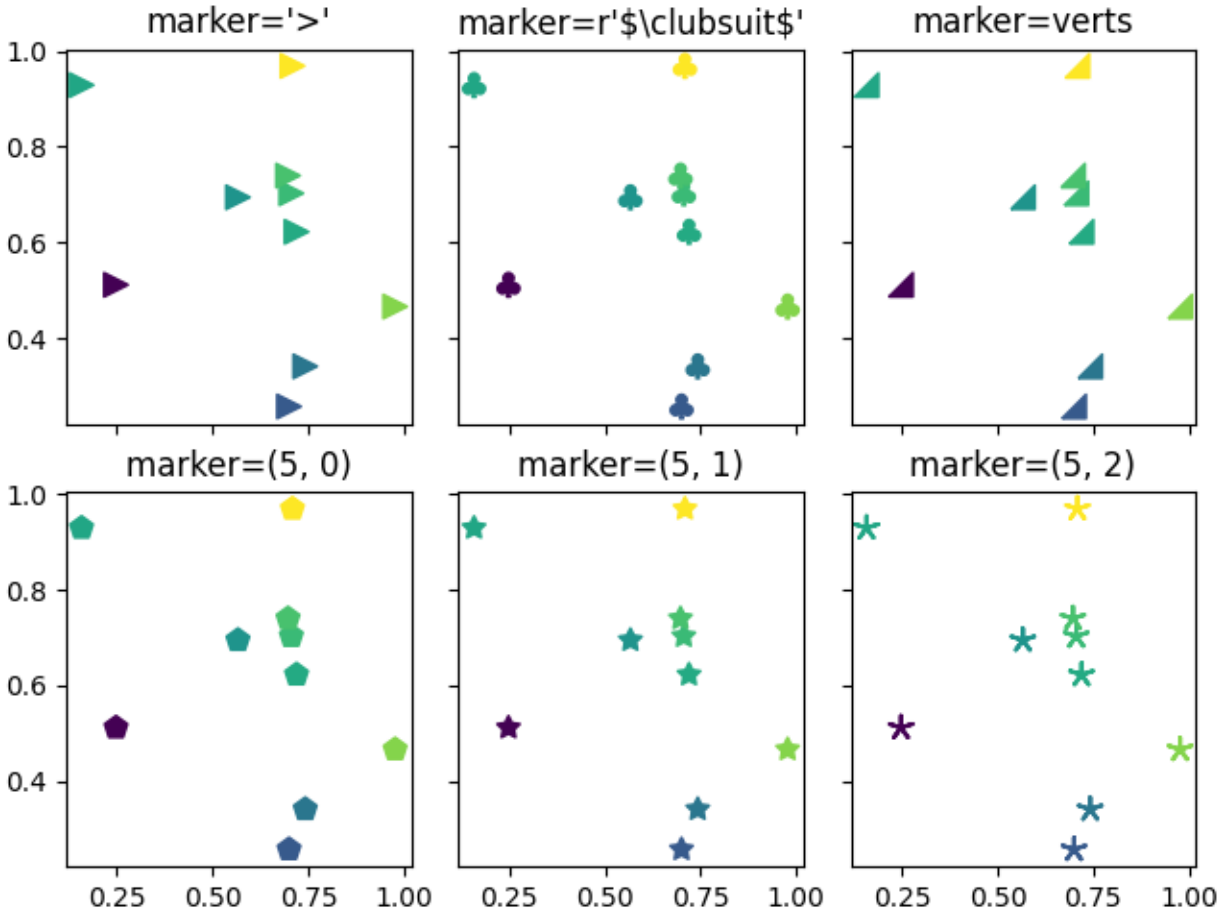
(continued from previous page)

```
r = np.sqrt(x ** 2 + y ** 2)
area1 = np.ma.masked_where(r < r0, area)
area2 = np.ma.masked_where(r >= r0, area)
plt.scatter(x, y, s=area1, marker='^', c=c)
plt.scatter(x, y, s=area2, marker='o', c=c)
# Show the boundary between the regions:
theta = np.arange(0, np.pi / 2, 0.01)
plt.plot(r0 * np.cos(theta), r0 * np.sin(theta))

plt.show()
```

### Marker examples

Example with different ways to specify markers.

See also the *matplotlib.markers* documentation for a list of all markers and *Marker reference* for more information on configuring markers.




```
import matplotlib.pyplot as plt
import numpy as np
```

(continues on next page)

(continued from previous page)

```
# Fixing random state for reproducibility
np.random.seed(19680801)

x = np.random.rand(10)
y = np.random.rand(10)
z = np.sqrt(x**2 + y**2)

fig, axs = plt.subplots(2, 3, sharex=True, sharey=True, layout="constrained")

# Matplotlib marker symbol
axs[0, 0].scatter(x, y, s=80, c=z, marker=">")
axs[0, 0].set_title("marker='>'")

# marker from TeX: passing a TeX symbol name enclosed in $-signs
axs[0, 1].scatter(x, y, s=80, c=z, marker=r"$\clubsuit$")
axs[0, 1].set_title(r"marker=r'\$\clubsuit\$'")

# marker from path: passing a custom path of N vertices as a (N, 2) array-like
verts = [[-1, -1], [1, -1], [1, 1], [-1, -1]]
axs[0, 2].scatter(x, y, s=80, c=z, marker=verts)
axs[0, 2].set_title("marker=verts")

# regular pentagon marker
axs[1, 0].scatter(x, y, s=80, c=z, marker=(5, 0))
axs[1, 0].set_title("marker=(5, 0)")

# regular 5-pointed star marker
axs[1, 1].scatter(x, y, s=80, c=z, marker=(5, 1))
axs[1, 1].set_title("marker=(5, 1)")

# regular 5-pointed asterisk marker
axs[1, 2].scatter(x, y, s=80, c=z, marker=(5, 2))
axs[1, 2].set_title("marker=(5, 2)")

plt.show()
```

### Scatter plots with a legend

To create a scatter plot with a legend one may use a loop and create one *scatter* plot per item to appear in the legend and set the `label` accordingly.

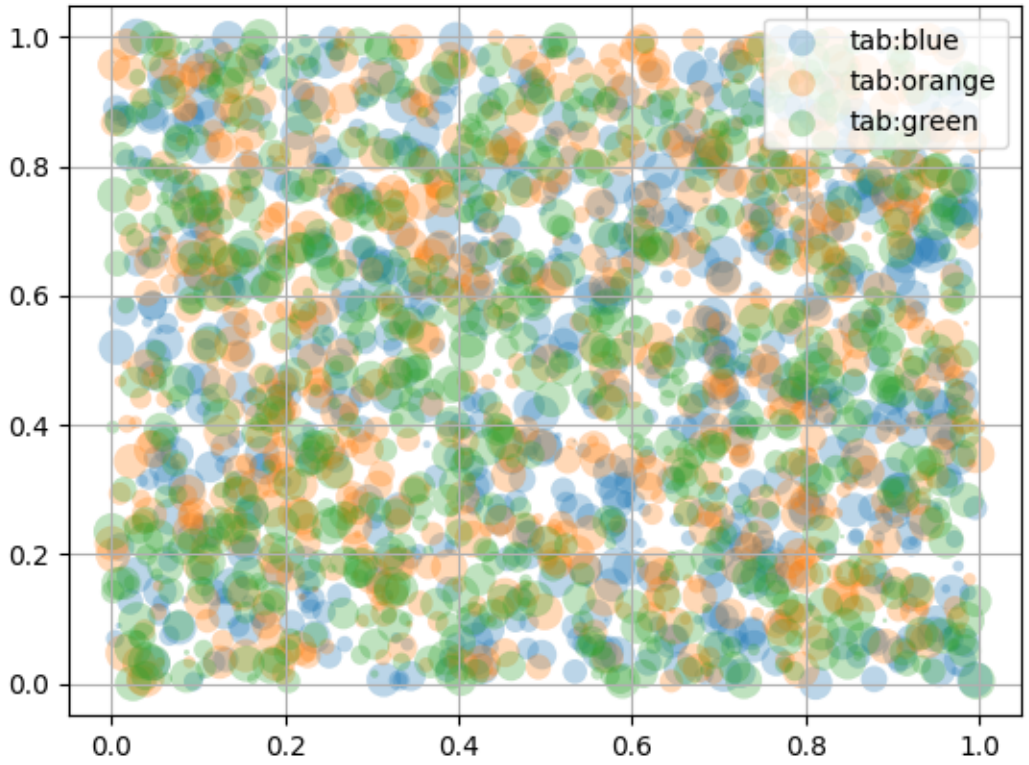
The following also demonstrates how transparency of the markers can be adjusted by giving `alpha` a value between 0 and 1.

```
import matplotlib.pyplot as plt
import numpy as np

np.random.seed(19680801)
```

(continues on next page)

(continued from previous page)

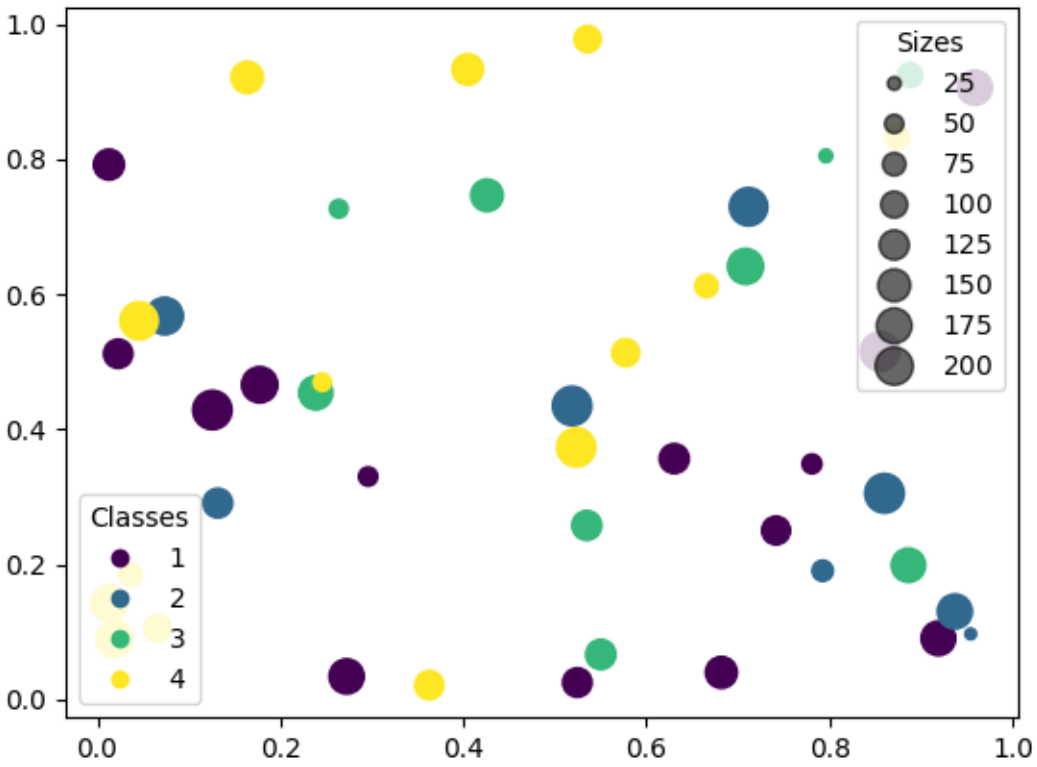
```
fig, ax = plt.subplots()
for color in ['tab:blue', 'tab:orange', 'tab:green']:
    n = 750
    x, y = np.random.rand(2, n)
    scale = 200.0 * np.random.rand(n)
    ax.scatter(x, y, c=color, s=scale, label=color,
               alpha=0.3, edgecolors='none')

ax.legend()
ax.grid(True)

plt.show()
```



### Automated legend creation

Another option for creating a legend for a scatter is to use the *PathCollection.legend_elements* method. It will automatically try to determine a useful number of legend entries to be shown and return a tuple of handles and labels. Those can be passed to the call to *legend*.

```
N = 45
x, y = np.random.rand(2, N)
c = np.random.randint(1, 5, size=N)
s = np.random.randint(10, 220, size=N)

fig, ax = plt.subplots()

scatter = ax.scatter(x, y, c=c, s=s)

# produce a legend with the unique colors from the scatter
legend1 = ax.legend(*scatter.legend_elements(),
                    loc="lower left", title="Classes")
ax.add_artist(legend1)

# produce a legend with a cross-section of sizes from the scatter
handles, labels = scatter.legend_elements(prop="sizes", alpha=0.6)
legend2 = ax.legend(handles, labels, loc="upper right", title="Sizes")

plt.show()
```



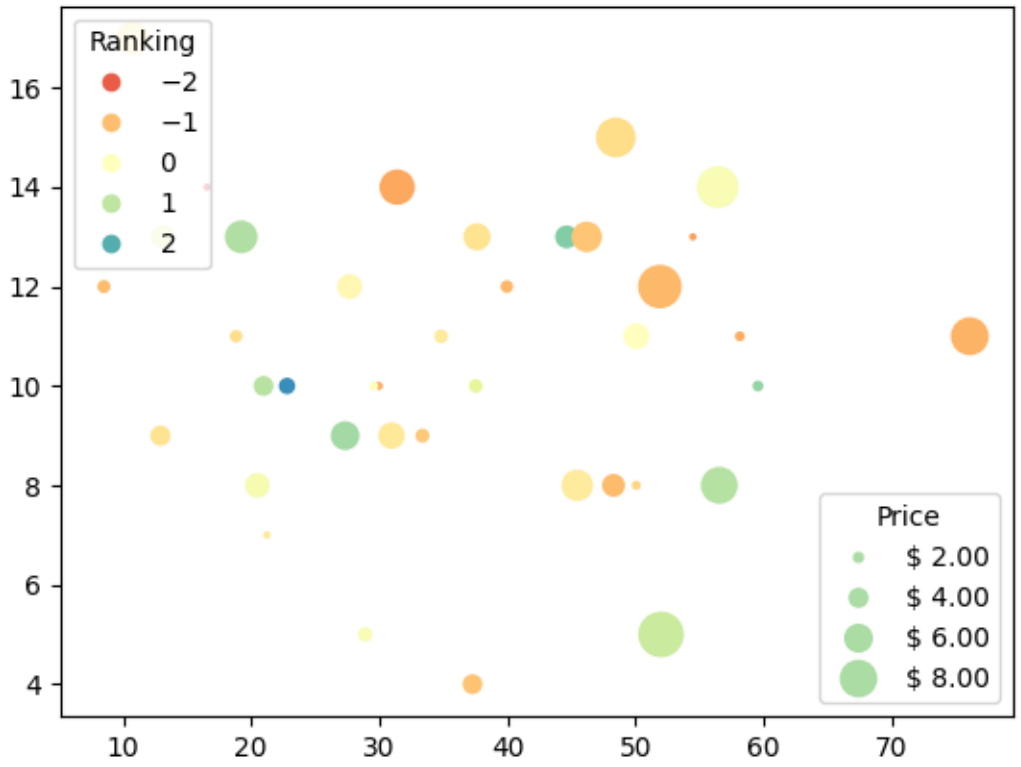

Further arguments to the *PathCollection.legend_elements* method can be used to steer how many legend entries are to be created and how they should be labeled. The following shows how to use some of them.

```
volume = np.random.rayleigh(27, size=40)
amount = np.random.poisson(10, size=40)
ranking = np.random.normal(size=40)
price = np.random.uniform(1, 10, size=40)

fig, ax = plt.subplots()

# Because the price is much too small when being provided as size for ``s``,
# we normalize it to some useful point sizes, s=0.3*(price*3)**2
scatter = ax.scatter(volume, amount, c=ranking, s=0.3*(price*3)**2,
                     vmin=-3, vmax=3, cmap="Spectral")

# Produce a legend for the ranking (colors). Even though there are 40␣
 ↪different
# rankings, we only want to show 5 of them in the legend.
legend1 = ax.legend(*scatter.legend_elements(num=5),
                    loc="upper left", title="Ranking")
ax.add_artist(legend1)
```

(continues on next page)

(continued from previous page)

```
# Produce a legend for the price (sizes). Because we want to show the prices
# in dollars, we use the *func* argument to supply the inverse of the function
# used to calculate the sizes from above. The *fmt* ensures to show the price
# in dollars. Note how we target at 5 elements here, but obtain only 4 in the
# created legend due to the automatic round prices that are chosen for us.
kw = dict(prop="sizes", num=5, color=scatter.cmap(0.7), fmt="$ {x:.2f}",
          func=lambda s: np.sqrt(s/.3)/3)
legend2 = ax.legend(*scatter.legend_elements(**kw),
                    loc="lower right", title="Price")

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

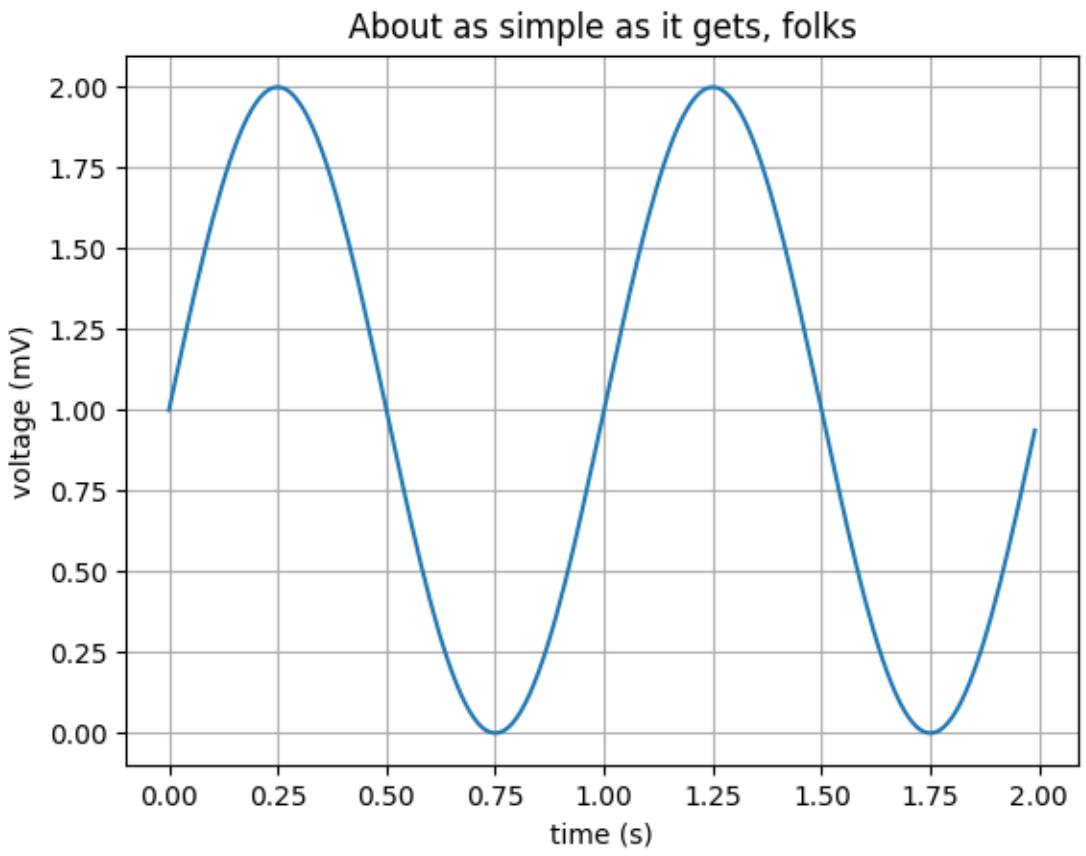
- *matplotlib.axes.Axes.scatter* / *matplotlib.pyplot.scatter*
- *matplotlib.axes.Axes.legend* / *matplotlib.pyplot.legend*
- *matplotlib.collections.PathCollection.legend_elements*

**Total running time of the script:** (0 minutes 1.454 seconds)

### Simple Plot

Create a simple plot.

```
import matplotlib.pyplot as plt
import numpy as np

# Data for plotting
t = np.arange(0.0, 2.0, 0.01)
s = 1 + np.sin(2 * np.pi * t)

fig, ax = plt.subplots()
ax.plot(t, s)

ax.set(xlabel='time (s)', ylabel='voltage (mV)',
       title='About as simple as it gets, folks')
ax.grid()

fig.savefig("test.png")
plt.show()
```



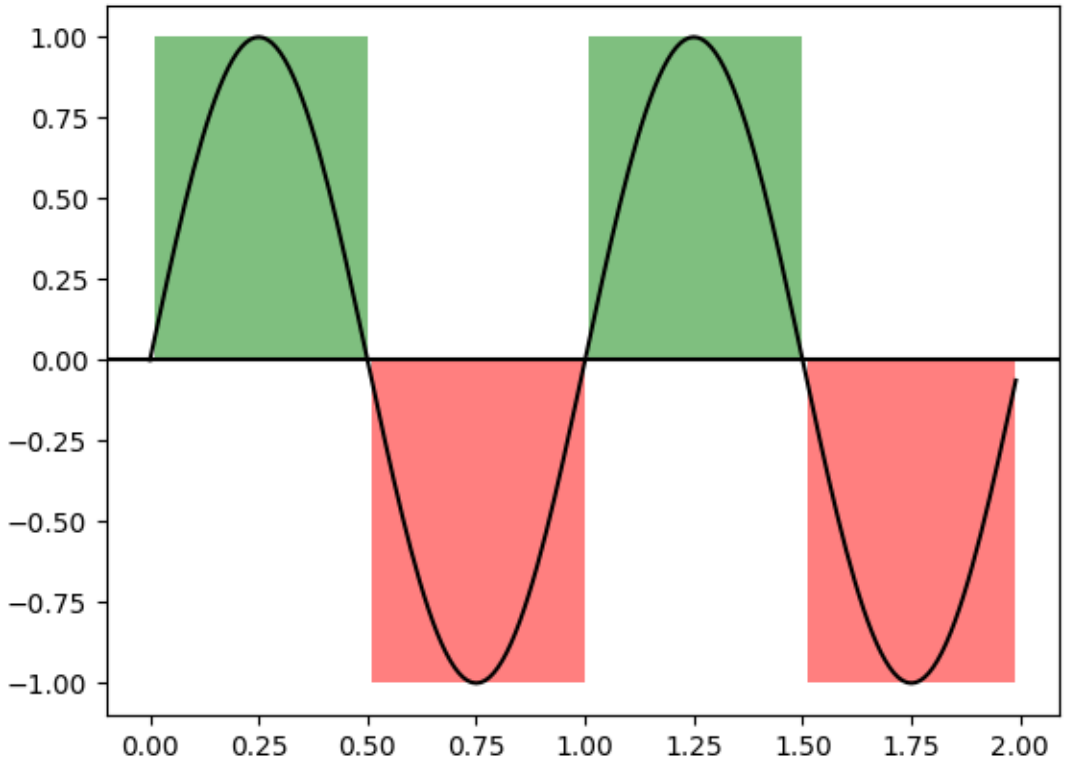
**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.plot* / *matplotlib.pyplot.plot*
- *matplotlib.pyplot.subplots*
- *matplotlib.figure.Figure.savefig*

### Shade regions defined by a logical mask using fill_between

```
import matplotlib.pyplot as plt
import numpy as np

t = np.arange(0.0, 2, 0.01)
s = np.sin(2*np.pi*t)

fig, ax = plt.subplots()

ax.plot(t, s, color='black')
ax.axhline(0, color='black')

ax.fill_between(t, 1, where=s > 0, facecolor='green', alpha=.5)
ax.fill_between(t, -1, where=s < 0, facecolor='red', alpha=.5)

plt.show()
```



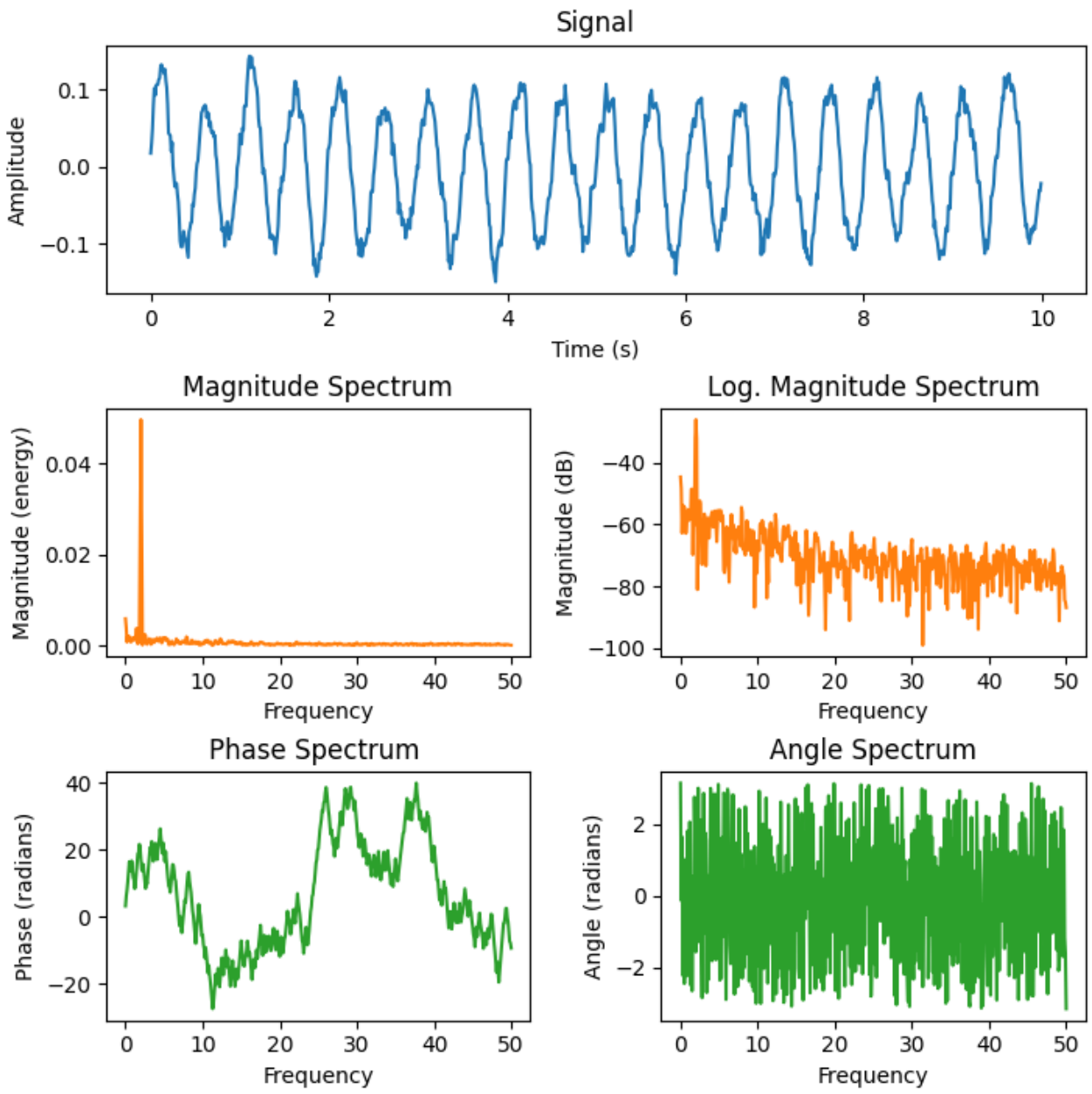

**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.fill_between*

## Spectrum representations

The plots show different spectrum representations of a sine signal with additive noise. A (frequency) spectrum of a discrete-time signal is calculated by utilizing the fast Fourier transform (FFT).




```
import matplotlib.pyplot as plt
import numpy as np

np.random.seed(0)

dt = 0.01  # sampling interval
Fs = 1 / dt  # sampling frequency
t = np.arange(0, 10, dt)

# generate noise:
nse = np.random.randn(len(t))
r = np.exp(-t / 0.05)
cnse = np.convolve(nse, r) * dt
```

(continues on next page)

(continued from previous page)

```
cnse = cnse[:len(t)]

s = 0.1 * np.sin(4 * np.pi * t) + cnse  # the signal

fig = plt.figure(figsize=(7, 7), layout='constrained')
axs = fig.subplot_mosaic([["signal", "signal"],
                          ["magnitude", "log_magnitude"],
                          ["phase", "angle"]])

# plot time signal:
axs["signal"].set_title("Signal")
axs["signal"].plot(t, s, color='C0')
axs["signal"].set_xlabel("Time (s)")
axs["signal"].set_ylabel("Amplitude")

# plot different spectrum types:
axs["magnitude"].set_title("Magnitude Spectrum")
axs["magnitude"].magnitude_spectrum(s, Fs=Fs, color='C1')

axs["log_magnitude"].set_title("Log. Magnitude Spectrum")
axs["log_magnitude"].magnitude_spectrum(s, Fs=Fs, scale='dB', color='C1')

axs["phase"].set_title("Phase Spectrum ")
axs["phase"].phase_spectrum(s, Fs=Fs, color='C2')

axs["angle"].set_title("Angle Spectrum")
axs["angle"].angle_spectrum(s, Fs=Fs, color='C2')

plt.show()
```

**Total running time of the script:** (0 minutes 1.188 seconds)

## Stackplots and streamgraphs

### Stackplots

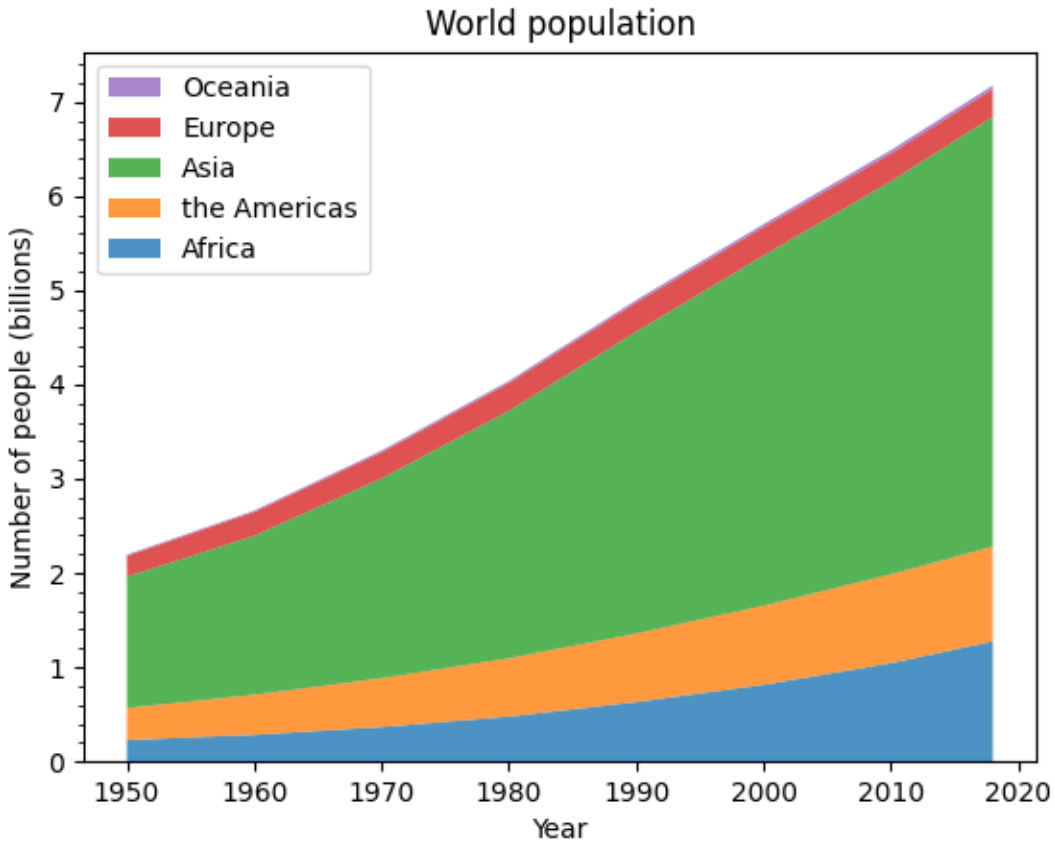
Stackplots draw multiple datasets as vertically stacked areas. This is useful when the individual data values and additionally their cumulative value are of interest.

```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib.ticker as mticker

# data from United Nations World Population Prospects (Revision 2019)
# https://population.un.org/wpp/, license: CC BY 3.0 IGO
year = [1950, 1960, 1970, 1980, 1990, 2000, 2010, 2018]
population_by_continent = {
    'Africa': [.228, .284, .365, .477, .631, .814, 1.044, 1.275],
    'the Americas': [.340, .425, .519, .619, .727, .840, .943, 1.006],
```

(continues on next page)

(continued from previous page)

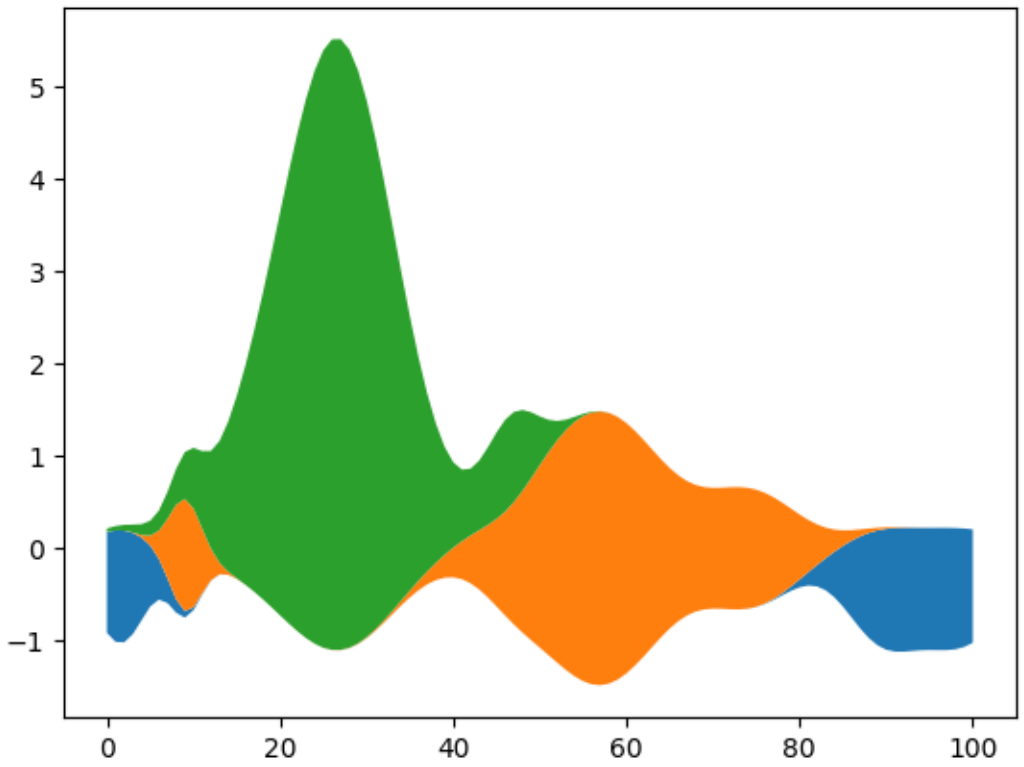
```
    'Asia': [1.394, 1.686, 2.120, 2.625, 3.202, 3.714, 4.169, 4.560],
    'Europe': [.220, .253, .276, .295, .310, .303, .294, .293],
    'Oceania': [.012, .015, .019, .022, .026, .031, .036, .039],
}

fig, ax = plt.subplots()
ax.stackplot(year, population_by_continent.values(),
             labels=population_by_continent.keys(), alpha=0.8)
ax.legend(loc='upper left', reverse=True)
ax.set_title('World population')
ax.set_xlabel('Year')
ax.set_ylabel('Number of people (billions)')
# add tick at every 200 million people
ax.yaxis.set_minor_locator(mticker.MultipleLocator(.2))

plt.show()
```



### Streamgraphs

Using the *baseline* parameter, you can turn an ordinary stacked area plot with baseline 0 into a stream graph.

```
# Fixing random state for reproducibility
np.random.seed(19680801)


def gaussian_mixture(x, n=5):
    """Return a random mixture of *n* Gaussians, evaluated at positions *x*."""
    def add_random_gaussian(a):
        amplitude = 1 / (.1 + np.random.random())
        dx = x[-1] - x[0]
        x0 = (2 * np.random.random() - .5) * dx
        z = 10 / (.1 + np.random.random()) / dx
        a += amplitude * np.exp(-(z * (x - x0))**2)
    a = np.zeros_like(x)
    for j in range(n):
        add_random_gaussian(a)
    return a


x = np.linspace(0, 100, 101)
ys = [gaussian_mixture(x) for _ in range(3)]

fig, ax = plt.subplots()
ax.stackplot(x, ys, baseline='wiggle')
plt.show()
```




### Stairs Demo

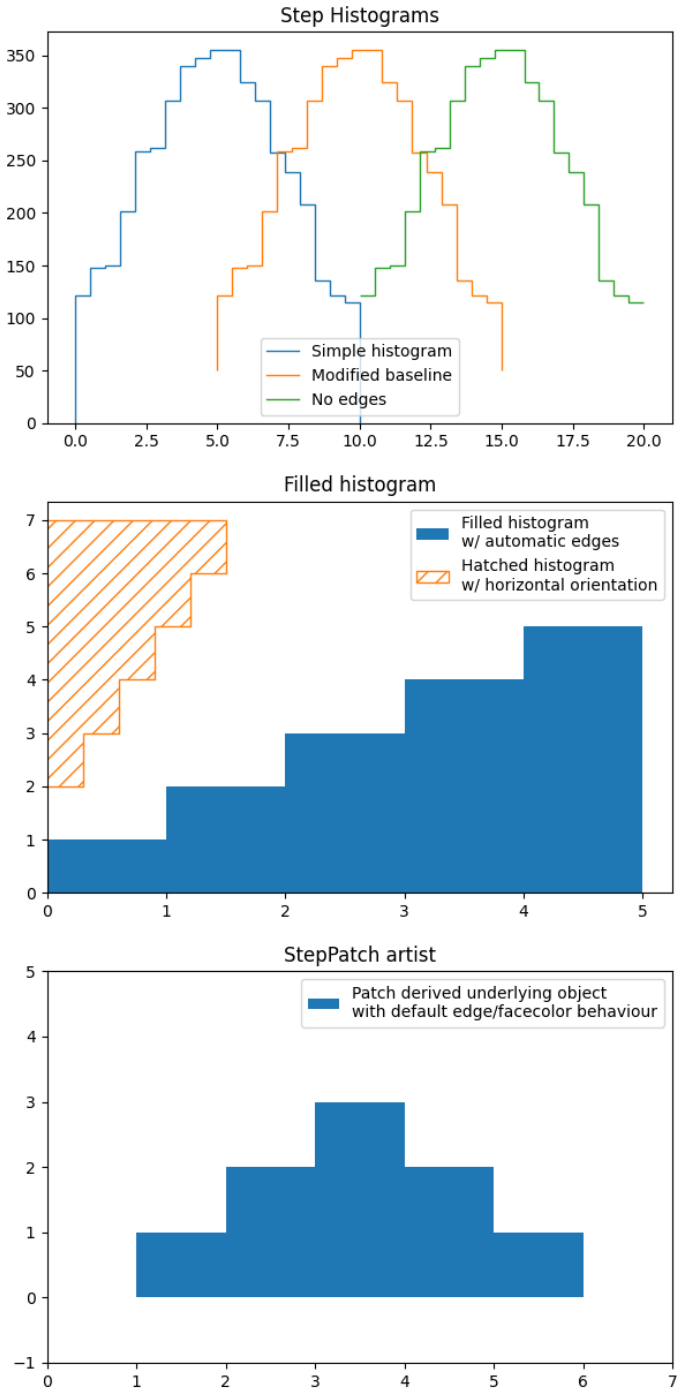
This example demonstrates the use of *stairs* for stepwise constant functions. A common use case is histogram and histogram-like data visualization.

```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.patches import StepPatch

np.random.seed(0)
h, edges = np.histogram(np.random.normal(5, 3, 5000),
                        bins=np.linspace(0, 10, 20))

fig, axs = plt.subplots(3, 1, figsize=(7, 15))
axs[0].stairs(h, edges, label='Simple histogram')
axs[0].stairs(h, edges + 5, baseline=50, label='Modified baseline')
axs[0].stairs(h, edges + 10, baseline=None, label='No edges')
axs[0].set_title("Step Histograms")

axs[1].stairs(np.arange(1, 6, 1), fill=True,
              label='Filled histogram\nw/ automatic edges')
```

(continues on next page)

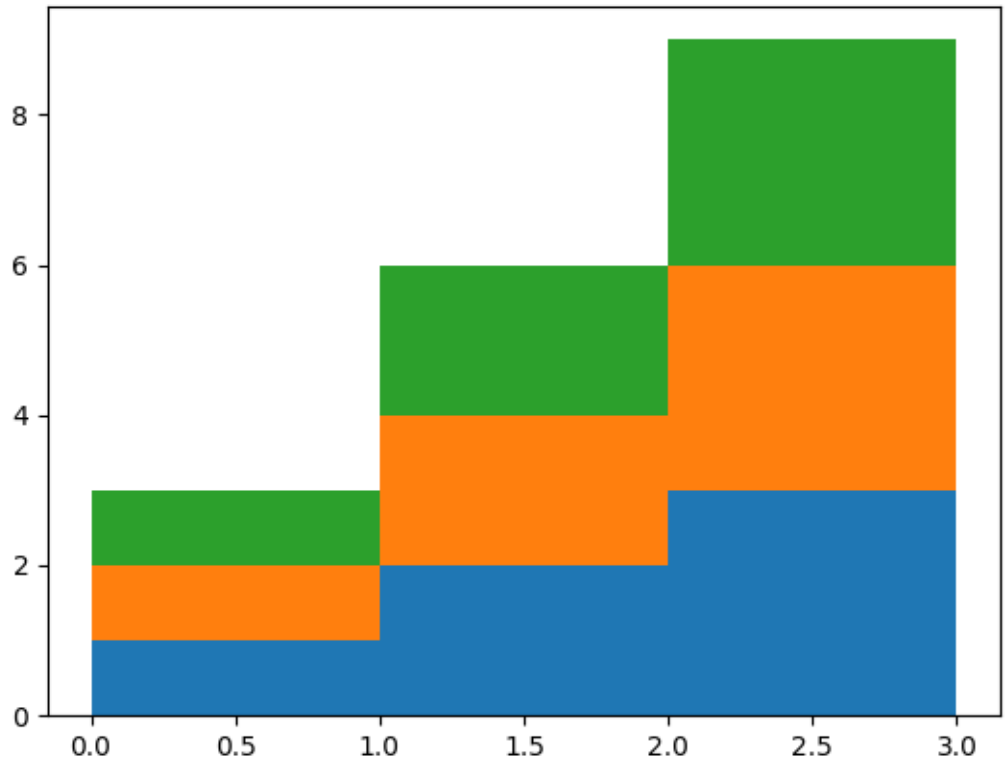
(continued from previous page)

```
axs[1].stairs(np.arange(1, 6, 1)*0.3, np.arange(2, 8, 1),
              orientation='horizontal', hatch='//',
              label='Hatched histogram\nw/ horizontal orientation')
axs[1].set_title("Filled histogram")

patch = StepPatch(values=[1, 2, 3, 2, 1],
                  edges=range(1, 7),
                  label=('Patch derived underlying object\n'
                         'with default edge/facecolor behaviour'))
axs[2].add_patch(patch)
axs[2].set_xlim(0, 7)
axs[2].set_ylim(-1, 5)
axs[2].set_title("StepPatch artist")

for ax in axs:
    ax.legend()
plt.show()
```


Step Histograms
Simple histogram
Modified baseline
No edges
Filled histogram
Filled histogram
w/ automatic edges
Hatched histogram
w/ horizontal orientation
StepPatch artist
Patch derived underlying object
with default edge/facecolor behaviour

*baseline* can take an array to allow for stacked histogram plots

```
A = [[0, 0, 0],
     [1, 2, 3],
     [2, 4, 6],
     [3, 6, 9]]

for i in range(len(A) - 1):
    plt.stairs(A[i+1], baseline=A[i], fill=True)
```




### Comparison of `pyplot.step` and `pyplot.stairs`

*pyplot.step* defines the positions of the steps as single values. The steps extend left/right/both ways from these reference values depending on the parameter *where*. The number of *x* and *y* values is the same.

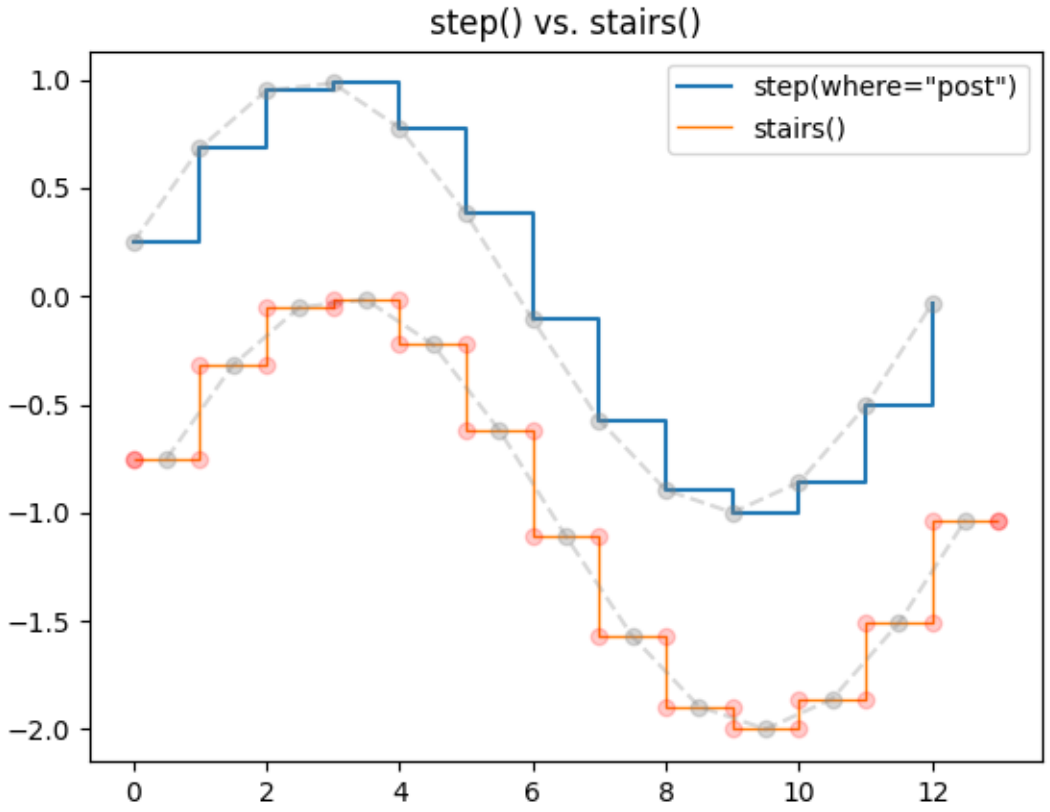
In contrast, *pyplot.stairs* defines the positions of the steps via their bounds *edges*, which is one element longer than the step values.

```
bins = np.arange(14)
centers = bins[:-1] + np.diff(bins) / 2
y = np.sin(centers / 2)
```

(continues on next page)

(continued from previous page)

```
plt.step(bins[:-1], y, where='post', label='step(where="post")')
plt.plot(bins[:-1], y, 'o--', color='grey', alpha=0.3)

plt.stairs(y - 1, bins, baseline=None, label='stairs()')
plt.plot(centers, y - 1, 'o--', color='grey', alpha=0.3)
plt.plot(np.repeat(bins, 2), np.hstack([y[0], np.repeat(y, 2), y[-1]]) - 1,
         'o', color='red', alpha=0.2)

plt.legend()
plt.title('step() vs. stairs()')
plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

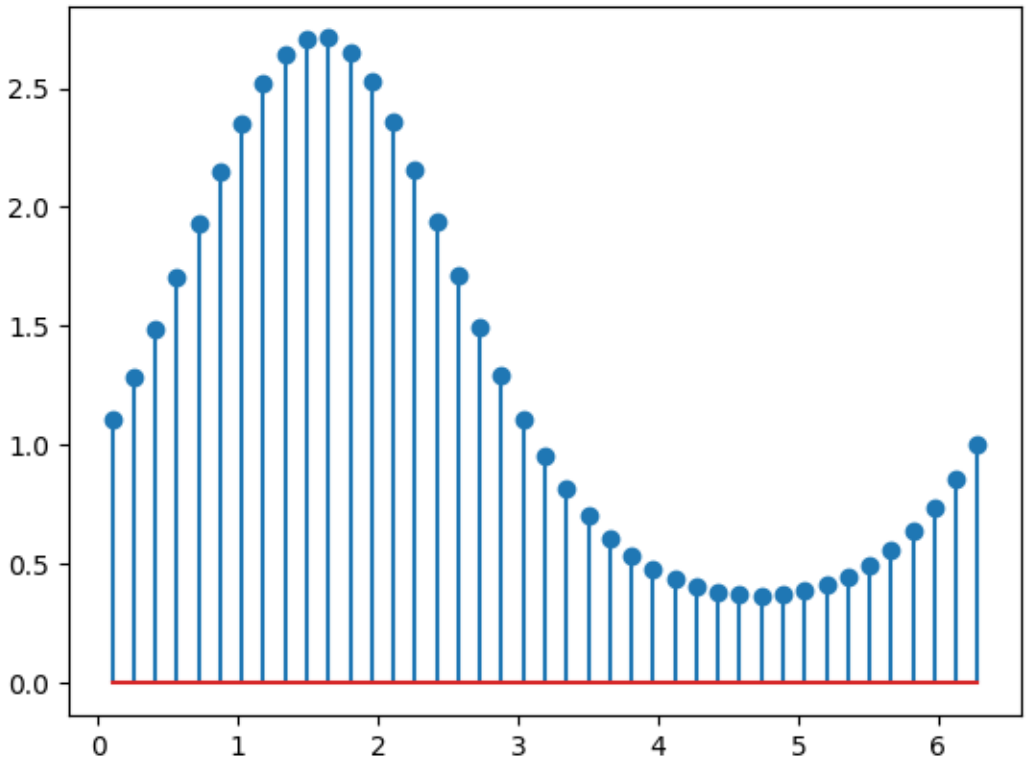
- *matplotlib.axes.Axes.stairs* / *matplotlib.pyplot.stairs*
- *matplotlib.patches.StepPatch*

**Total running time of the script:** (0 minutes 1.259 seconds)

### Stem Plot

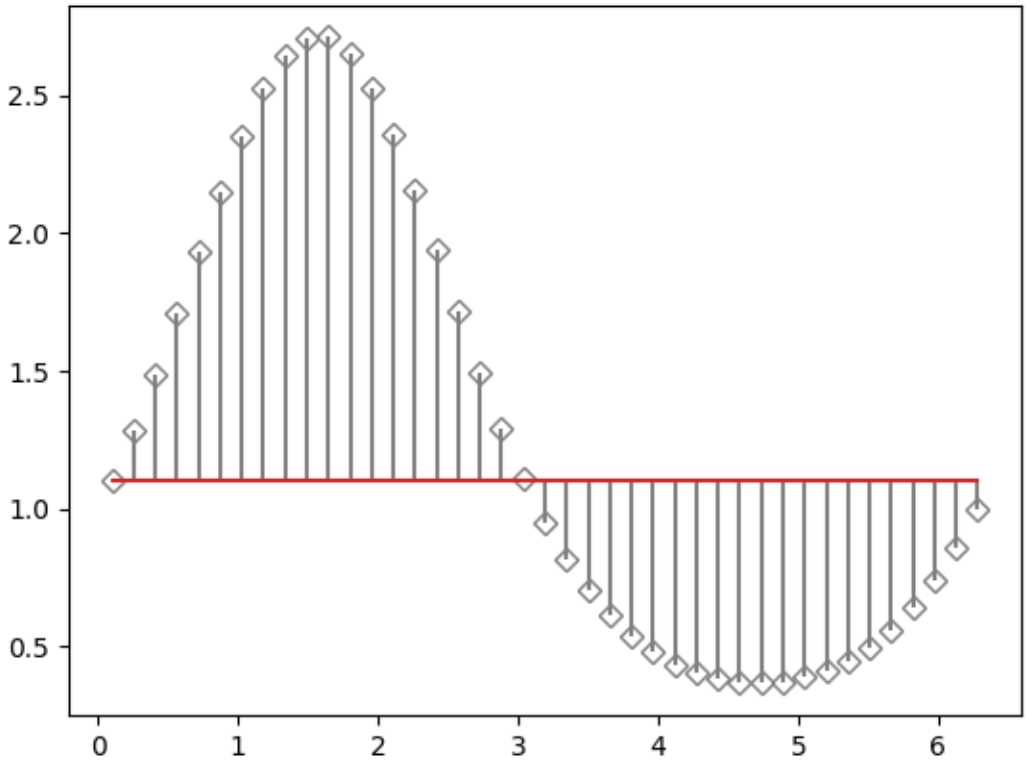
*stem* plots vertical lines from a baseline to the y-coordinate and places a marker at the tip.

```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0.1, 2 * np.pi, 41)
y = np.exp(np.sin(x))

plt.stem(x, y)
plt.show()
```




The position of the baseline can be adapted using *bottom*. The parameters *linefmt*, *markerfmt*, and *basefmt* control basic format properties of the plot. However, in contrast to *plot* not all properties are configurable via keyword arguments. For more advanced control adapt the line objects returned by *pyplot*.

```
markerline, stemlines, baseline = plt.stem(
    x, y, linefmt='grey', markerfmt='D', bottom=1.1)
markerline.set_markerfacecolor('none')
plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.stem* / *matplotlib.pyplot.stem*

## Step Demo

This example demonstrates the use of *pyplot.step* for piece-wise constant curves. In particular, it illustrates the effect of the parameter *where* on the step position.

**Note:** For the common case that you know the edge positions, use *pyplot.stairs* instead.

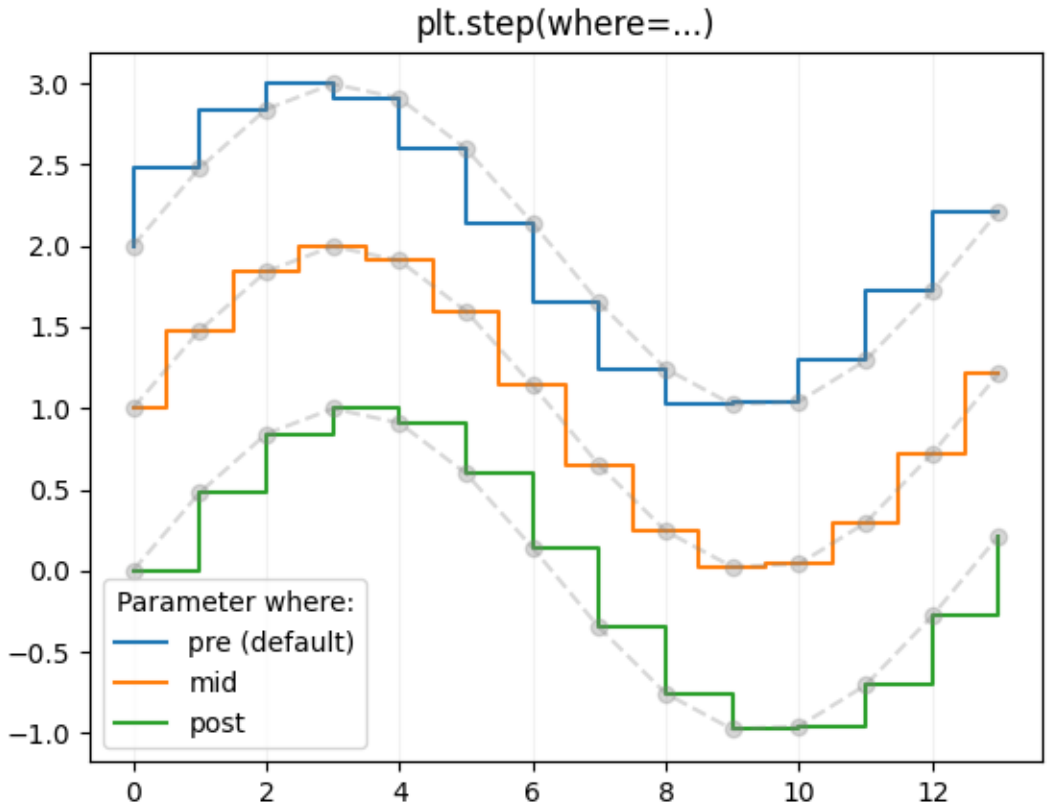
The circular markers created with *pyplot.plot* show the actual data positions so that it's easier to see the effect of *where*.

```
import matplotlib.pyplot as plt
import numpy as np
```

(continues on next page)

(continued from previous page)

```
x = np.arange(14)
y = np.sin(x / 2)

plt.step(x, y + 2, label='pre (default)')
plt.plot(x, y + 2, 'o--', color='grey', alpha=0.3)

plt.step(x, y + 1, where='mid', label='mid')
plt.plot(x, y + 1, 'o--', color='grey', alpha=0.3)

plt.step(x, y, where='post', label='post')
plt.plot(x, y, 'o--', color='grey', alpha=0.3)

plt.grid(axis='x', color='0.95')
plt.legend(title='Parameter where:')
plt.title('plt.step(where=...)')
plt.show()
```



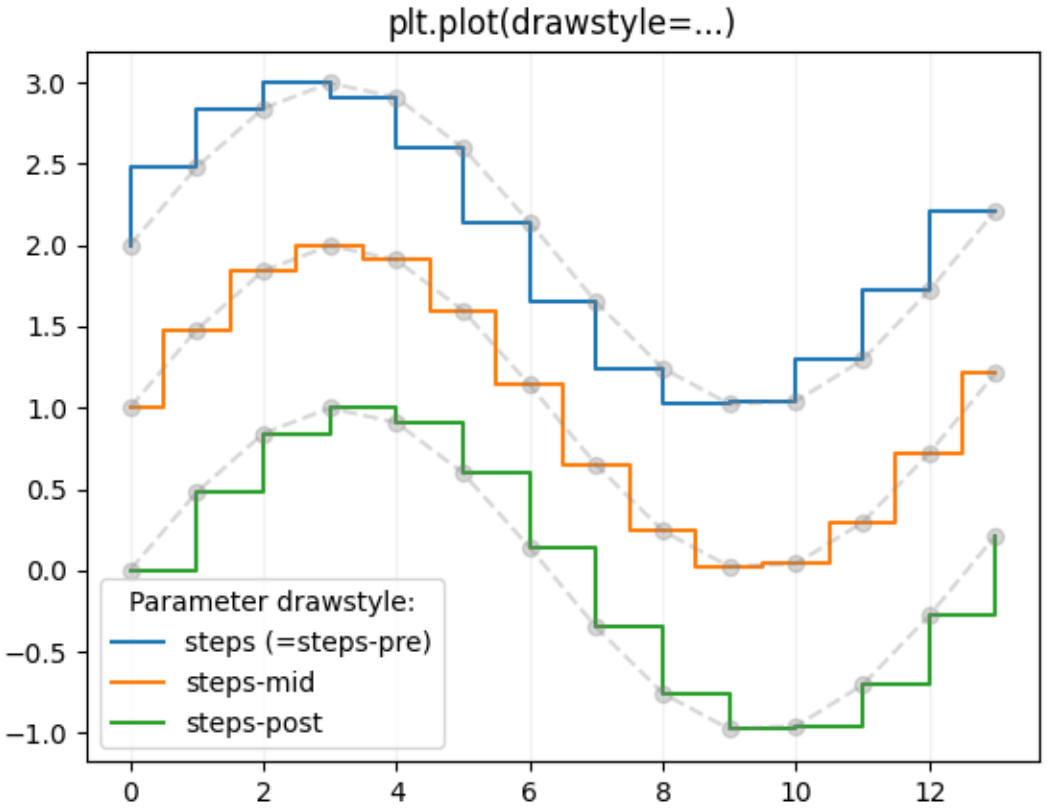

The same behavior can be achieved by using the `drawstyle` parameter of *`pyplot.plot`*.

```
plt.plot(x, y + 2, drawstyle='steps', label='steps (=steps-pre)')
plt.plot(x, y + 2, 'o--', color='grey', alpha=0.3)
```

(continues on next page)

(continued from previous page)

```
plt.plot(x, y + 1, drawstyle='steps-mid', label='steps-mid')
plt.plot(x, y + 1, 'o--', color='grey', alpha=0.3)

plt.plot(x, y, drawstyle='steps-post', label='steps-post')
plt.plot(x, y, 'o--', color='grey', alpha=0.3)

plt.grid(axis='x', color='0.95')
plt.legend(title='Parameter drawstyle:')
plt.title('plt.plot(drawstyle=...)')
plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.step* / *matplotlib.pyplot.step*
- *matplotlib.axes.Axes.plot* / *matplotlib.pyplot.plot*

### Creating a timeline with lines, dates, and text

How to create a simple timeline using Matplotlib release dates.

Timelines can be created with a collection of dates and text. In this example, we show how to create a simple timeline using the dates for recent releases of Matplotlib. First, we'll pull the data from GitHub.

```
from datetime import datetime

import matplotlib.pyplot as plt
import numpy as np

import matplotlib.dates as mdates

try:
    # Try to fetch a list of Matplotlib releases and their dates
    # from https://api.github.com/repos/matplotlib/matplotlib/releases
    import json
    import urllib.request

    url = 'https://api.github.com/repos/matplotlib/matplotlib/releases'
    url += '?per_page=100'
    data = json.loads(urllib.request.urlopen(url, timeout=1).read().decode())

    dates = []
    releases = []
    for item in data:
        if 'rc' not in item['tag_name'] and 'b' not in item['tag_name']:
            dates.append(item['published_at'].split("T")[0])
            releases.append(item['tag_name'].lstrip("v"))

except Exception:
    # In case the above fails, e.g. because of missing internet connection
    # use the following lists as fallback.
    releases = ['2.2.4', '3.0.3', '3.0.2', '3.0.1', '3.0.0', '2.2.3',
                '2.2.2', '2.2.1', '2.2.0', '2.1.2', '2.1.1', '2.1.0',
                '2.0.2', '2.0.1', '2.0.0', '1.5.3', '1.5.2', '1.5.1',
                '1.5.0', '1.4.3', '1.4.2', '1.4.1', '1.4.0']
    dates = ['2019-02-26', '2019-02-26', '2018-11-10', '2018-11-10',
             '2018-09-18', '2018-08-10', '2018-03-17', '2018-03-16',
             '2018-03-06', '2018-01-18', '2017-12-10', '2017-10-07',
             '2017-05-10', '2017-05-02', '2017-01-17', '2016-09-09',
             '2016-07-03', '2016-01-10', '2015-10-29', '2015-02-16',
             '2014-10-26', '2014-10-18', '2014-08-26']

dates = [datetime.strptime(d, "%Y-%m-%d") for d in dates]  # Convert strs to␣
 ↪dates.
dates, releases = zip(*sorted(zip(dates, releases)))  # Sort by increasing␣
 ↪date.
```

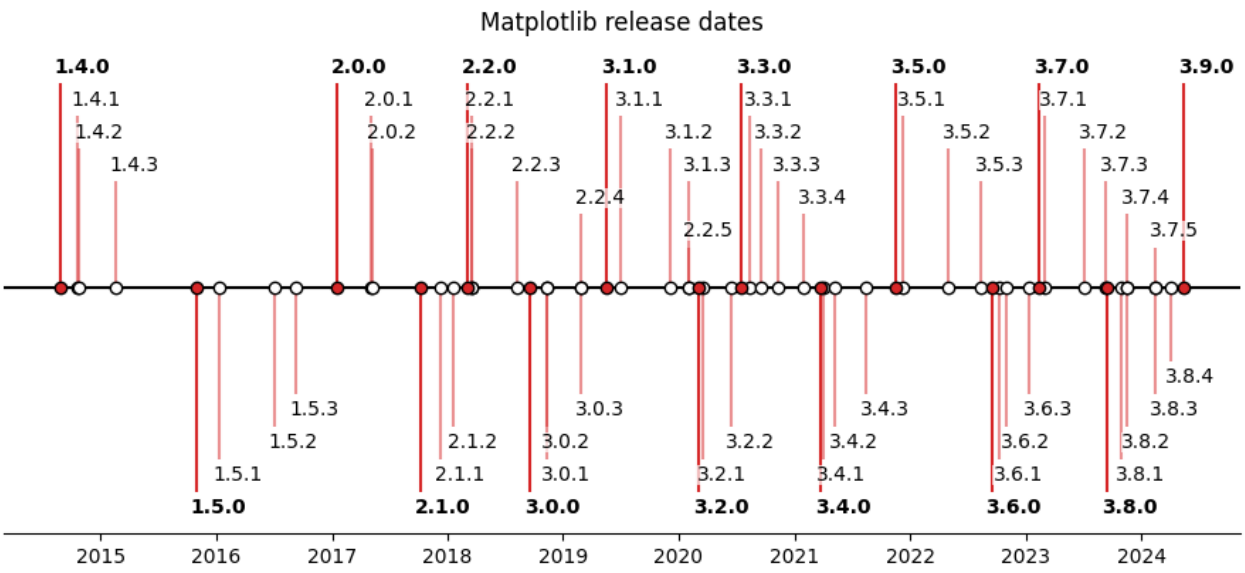
Next, we'll create a stem plot with some variation in levels as to distinguish even close-by events. We add markers on the baseline for visual emphasis on the one-dimensional nature of the timeline.

For each event, we add a text label via *annotate*, which is offset in units of points from the tip of the event

line.

Note that Matplotlib will automatically plot datetime inputs.

```
# Choose some nice levels: alternate minor releases between top and bottom,␣
 ↪and
# progressievly shorten the stems for bugfix releases.
levels = []
major_minor_releases = sorted({release[:3] for release in releases})
for release in releases:
    major_minor = release[:3]
    bugfix = int(release[4])
    h = 1 + 0.8 * (5 - bugfix)
    level = h if major_minor_releases.index(major_minor) % 2 == 0 else -h
    levels.append(level)

# The figure and the axes.
fig, ax = plt.subplots(figsize=(8.8, 4), layout="constrained")
ax.set(title="Matplotlib release dates")

# The vertical stems.
ax.vlines(dates, 0, levels,
          color=[("tab:red", 1 if release.endswith(".0") else .5)
                 for release in releases])
# The baseline.
ax.axhline(0, c="black")
# The markers on the baseline.
minor_dates = [date for date, release in zip(dates, releases) if release[-1]␣
 ↪== '0']
bugfix_dates = [date for date, release in zip(dates, releases) if release[-1]␣
 ↪!= '0']
ax.plot(bugfix_dates, np.zeros_like(bugfix_dates), "ko", mfc="white")
ax.plot(minor_dates, np.zeros_like(minor_dates), "ko", mfc="tab:red")

# Annotate the lines.
for date, level, release in zip(dates, levels, releases):
    ax.annotate(release, xy=(date, level),
                xytext=(-3, np.sign(level)*3), textcoords="offset points",
                verticalalignment="bottom" if level > 0 else "top",
                weight="bold" if release.endswith(".0") else "normal",
                bbox=dict(boxstyle='square', pad=0, lw=0, fc=(1, 1, 1, 0.7)))

ax.yaxis.set(major_locator=mdates.YearLocator(),
             major_formatter=mdates.DateFormatter("%Y"))

# Remove the y-axis and some spines.
ax.yaxis.set_visible(False)
ax.spines[["left", "top", "right"]].set_visible(False)

ax.margins(y=0.1)
plt.show()
```




**References**

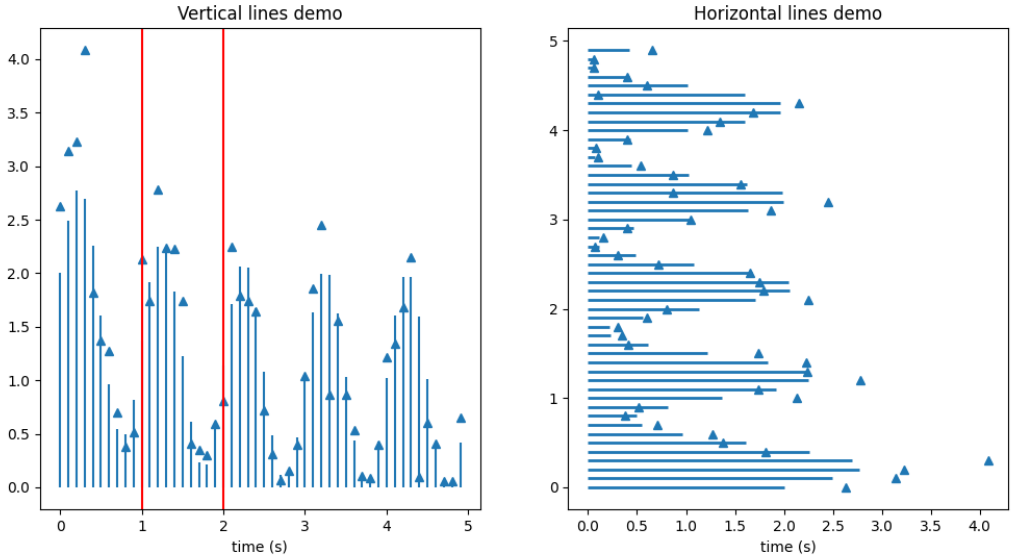
The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.annotate*
- *matplotlib.axes.Axes.vlines*
- *matplotlib.axis.Axis.set_major_locator*
- *matplotlib.axis.Axis.set_major_formatter*
- *matplotlib.dates.MonthLocator*
- *matplotlib.dates.DateFormatter*

**Total running time of the script:** (0 minutes 1.524 seconds)

### hlines and vlines

This example showcases the functions hlines and vlines.

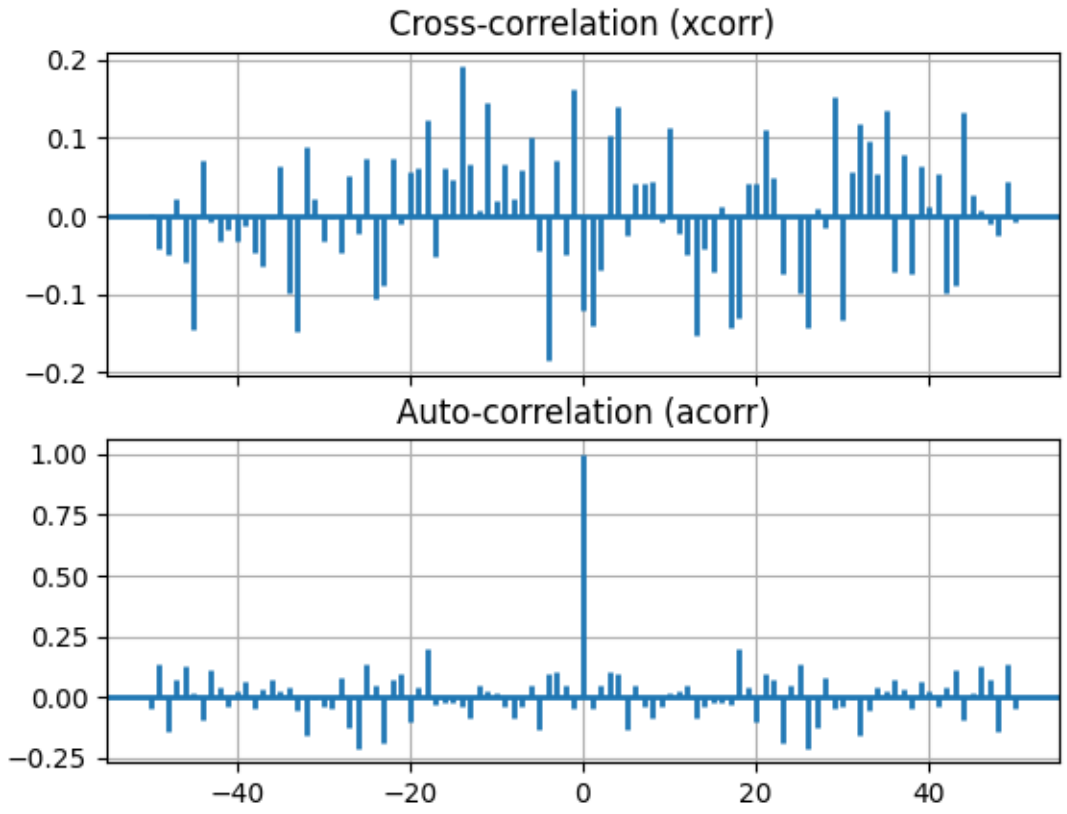



```
import matplotlib.pyplot as plt
import numpy as np

# Fixing random state for reproducibility
np.random.seed(19680801)

t = np.arange(0.0, 5.0, 0.1)
s = np.exp(-t) + np.sin(2 * np.pi * t) + 1
nse = np.random.normal(0.0, 0.3, t.shape) * s

fig, (vax, hax) = plt.subplots(1, 2, figsize=(12, 6))

vax.plot(t, s + nse, '^')
vax.vlines(t, [0], s)
# By using ``transform=vax.get_xaxis_transform()`` the y coordinates are␣
 ↪scaled
# such that 0 maps to the bottom of the Axes and 1 to the top.
vax.vlines([1, 2], 0, 1, transform=vax.get_xaxis_transform(), colors='r')
vax.set_xlabel('time (s)')
vax.set_title('Vertical lines demo')

hax.plot(s + nse, t, '^')
hax.hlines(t, [0], s, lw=2)
hax.set_xlabel('time (s)')
hax.set_title('Horizontal lines demo')

plt.show()
```

### Cross- and auto-correlation

Example use of cross-correlation (*xcorr*) and auto-correlation (*acorr*) plots.

```
import matplotlib.pyplot as plt
import numpy as np

# Fixing random state for reproducibility
np.random.seed(19680801)


x, y = np.random.randn(2, 100)
fig, [ax1, ax2] = plt.subplots(2, 1, sharex=True)
ax1.xcorr(x, y, usevlines=True, maxlags=50, normed=True, lw=2)
ax1.grid(True)
ax1.set_title('Cross-correlation (xcorr)')

ax2.acorr(x, usevlines=True, normed=True, maxlags=50, lw=2)
ax2.grid(True)
ax2.set_title('Auto-correlation (acorr)')

plt.show()
```



**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.acorr* / *matplotlib.pyplot.acorr*
- *matplotlib.axes.Axes.xcorr* / *matplotlib.pyplot.xcorr*

### 6.25.2 Images, contours and fields

#### Affine transform of an image

Prepending an affine transformation (*Affine2D*) to the *data transform* of an image allows to manipulate the image's shape and orientation. This is an example of the concept of *transform chaining*.

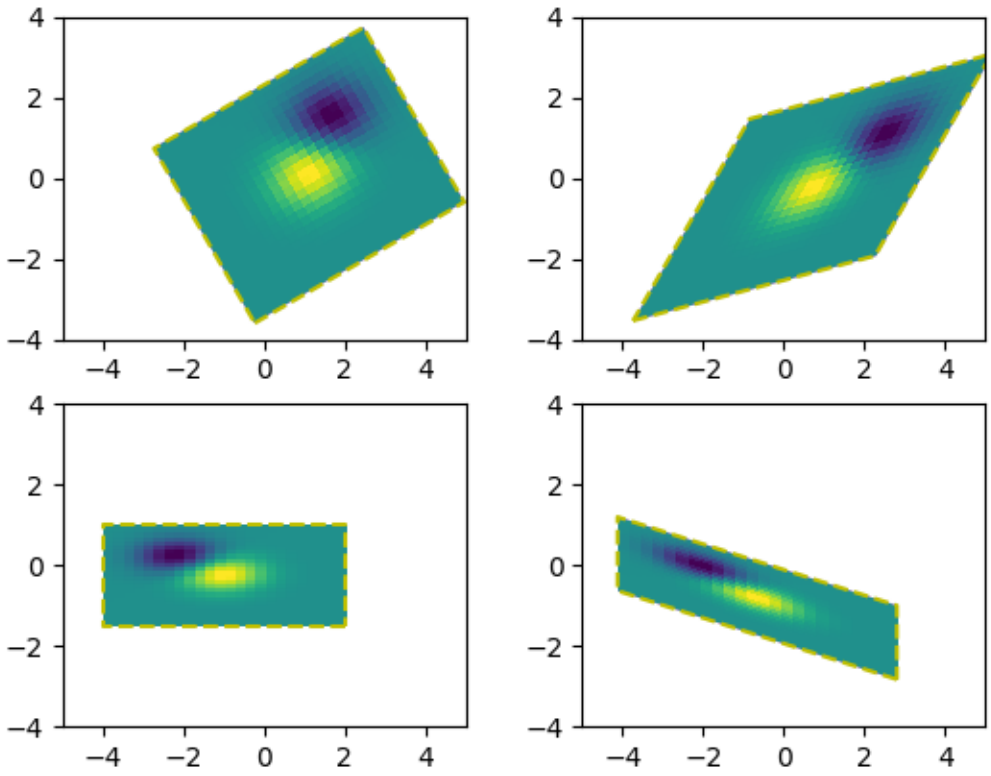
The image of the output should have its boundary match the dashed yellow rectangle.

```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib.transforms as mtransforms


def get_image():
    delta = 0.25
    x = y = np.arange(-3.0, 3.0, delta)
    X, Y = np.meshgrid(x, y)
    Z1 = np.exp(-X**2 - Y**2)
    Z2 = np.exp(-(X - 1)**2 - (Y - 1)**2)
    Z = (Z1 - Z2)
    return Z


def do_plot(ax, Z, transform):
    im = ax.imshow(Z, interpolation='none',
                   origin='lower',
                   extent=[-2, 4, -3, 2], clip_on=True)

    trans_data = transform + ax.transData
    im.set_transform(trans_data)

    # display intended extent of the image
    x1, x2, y1, y2 = im.get_extent()
    ax.plot([x1, x2, x2, x1, x1], [y1, y1, y2, y2, y1], "y--",
            transform=trans_data)
    ax.set_xlim(-5, 5)
    ax.set_ylim(-4, 4)


# prepare image and figure
```

(continues on next page)

(continued from previous page)

```
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2)
Z = get_image()

# image rotation
do_plot(ax1, Z, mtransforms.Affine2D().rotate_deg(30))

# image skew
do_plot(ax2, Z, mtransforms.Affine2D().skew_deg(30, 15))

# scale and reflection
do_plot(ax3, Z, mtransforms.Affine2D().scale(-1, .5))

# everything and a translation
do_plot(ax4, Z, mtransforms.Affine2D().
        rotate_deg(30).skew_deg(30, 15).scale(-1, .5).translate(.5, -1))

plt.show()
```




**References**

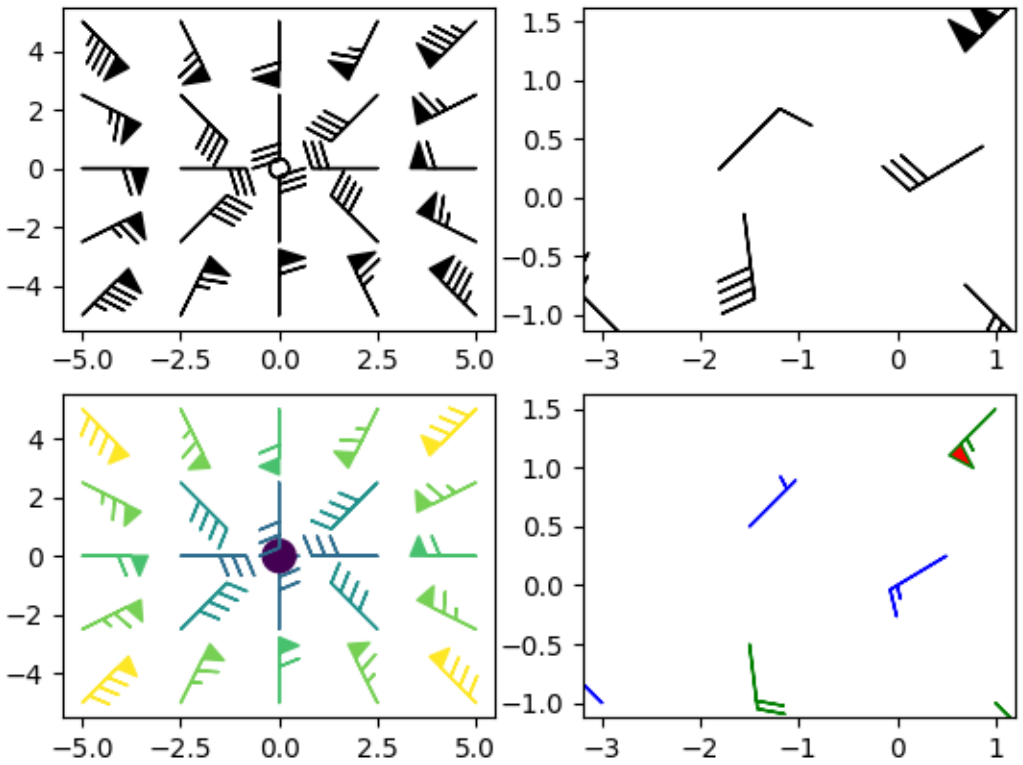
The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.imshow* / *matplotlib.pyplot.imshow*
- *matplotlib.transforms.Affine2D*

### Wind Barbs

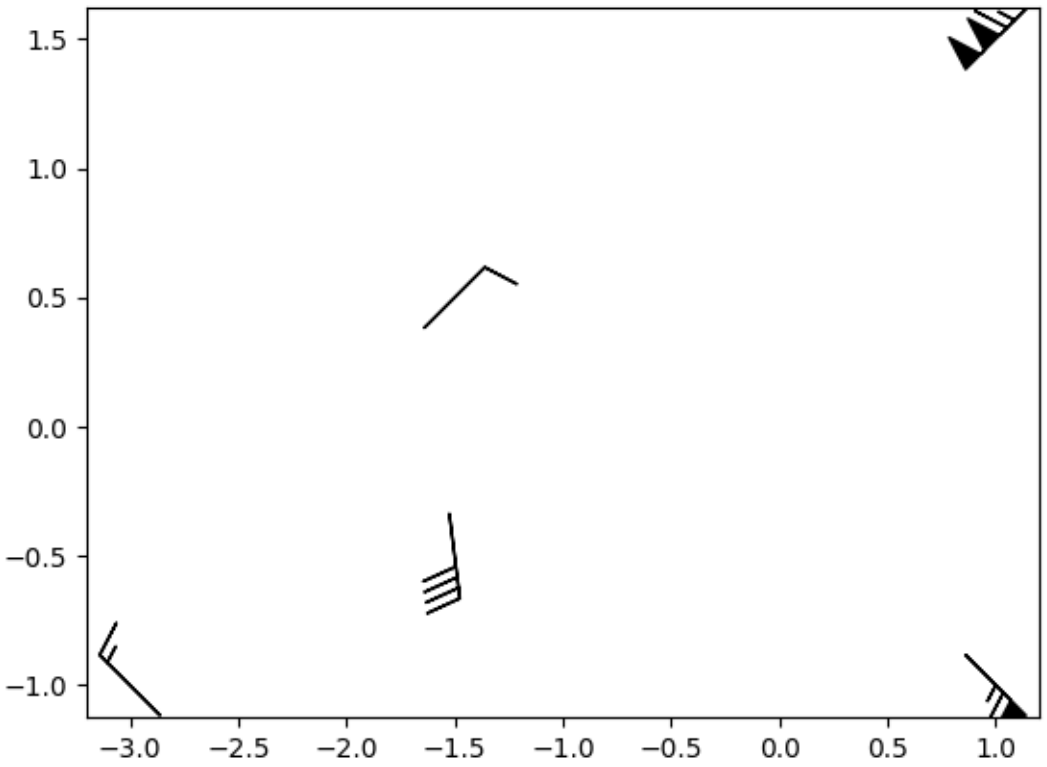
Demonstration of wind barb plots.

```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(-5, 5, 5)
X, Y = np.meshgrid(x, x)
U, V = 12 * X, 12 * Y

data = [(-1.5, .5, -6, -6),
        (1, -1, -46, 46),
        (-3, -1, 11, -11),
        (1, 1.5, 80, 80),
        (0.5, 0.25, 25, 15),
        (-1.5, -0.5, -5, 40)]

data = np.array(data, dtype=[('x', np.float32), ('y', np.float32),
                             ('u', np.float32), ('v', np.float32)])

fig1, axs1 = plt.subplots(nrows=2, ncols=2)
# Default parameters, uniform grid
axs1[0, 0].barbs(X, Y, U, V)

# Arbitrary set of vectors, make them longer and change the pivot point
# (point around which they're rotated) to be the middle
axs1[0, 1].barbs(
    data['x'], data['y'], data['u'], data['v'], length=8, pivot='middle')

# Showing colormapping with uniform grid.  Fill the circle for an empty barb,
# don't round the values, and change some of the size parameters
axs1[1, 0].barbs(
    X, Y, U, V, np.sqrt(U ** 2 + V ** 2), fill_empty=True, rounding=False,
    sizes=dict(emptybarb=0.25, spacing=0.2, height=0.3))

# Change colors as well as the increments for parts of the barbs
axs1[1, 1].barbs(data['x'], data['y'], data['u'], data['v'], flagcolor='r',
                 barbcolor=['b', 'g'], flip_barb=True,
                 barb_increments=dict(half=10, full=20, flag=100))

# Masked arrays are also supported
masked_u = np.ma.masked_array(data['u'])
masked_u[4] = 1000  # Bad value that should not be plotted when masked
masked_u[4] = np.ma.masked
```




Identical plot to panel 2 in the first figure, but with the point at (0.5, 0.25) missing (masked)

```
fig2, ax2 = plt.subplots()
ax2.barbs(data['x'], data['y'], masked_u, data['v'], length=8, pivot='middle')

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.barbs* / *matplotlib.pyplot.barbs*

## Barcode

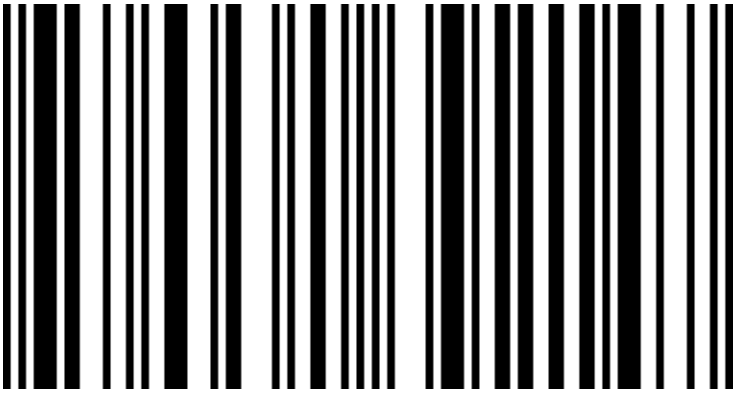
This demo shows how to produce a bar code.

The figure size is calculated so that the width in pixels is a multiple of the number of data points to prevent interpolation artifacts. Additionally, the `Axes` is defined to span the whole figure and all `Axis` are turned off.

The data itself is rendered with *imshow* using

- `code.reshape(1, -1)` to turn the data into a 2D array with one row.
- `imshow(..., aspect='auto')` to allow for non-square pixels.
- `imshow(..., interpolation='nearest')` to prevent blurred edges. This should not happen anyway because we fine-tuned the figure width in pixels, but just to be safe.

```
import matplotlib.pyplot as plt
import numpy as np

code = np.array([
    1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1,
    0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0,
    1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1,
    1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1])

pixel_per_bar = 4
dpi = 100

fig = plt.figure(figsize=(len(code) * pixel_per_bar / dpi, 2), dpi=dpi)
ax = fig.add_axes([0, 0, 1, 1])  # span the whole figure
ax.set_axis_off()
ax.imshow(code.reshape(1, -1), cmap='binary', aspect='auto',
          interpolation='nearest')
plt.show()
```



**References**

The use of the following functions, methods, classes and modules is shown in this example:

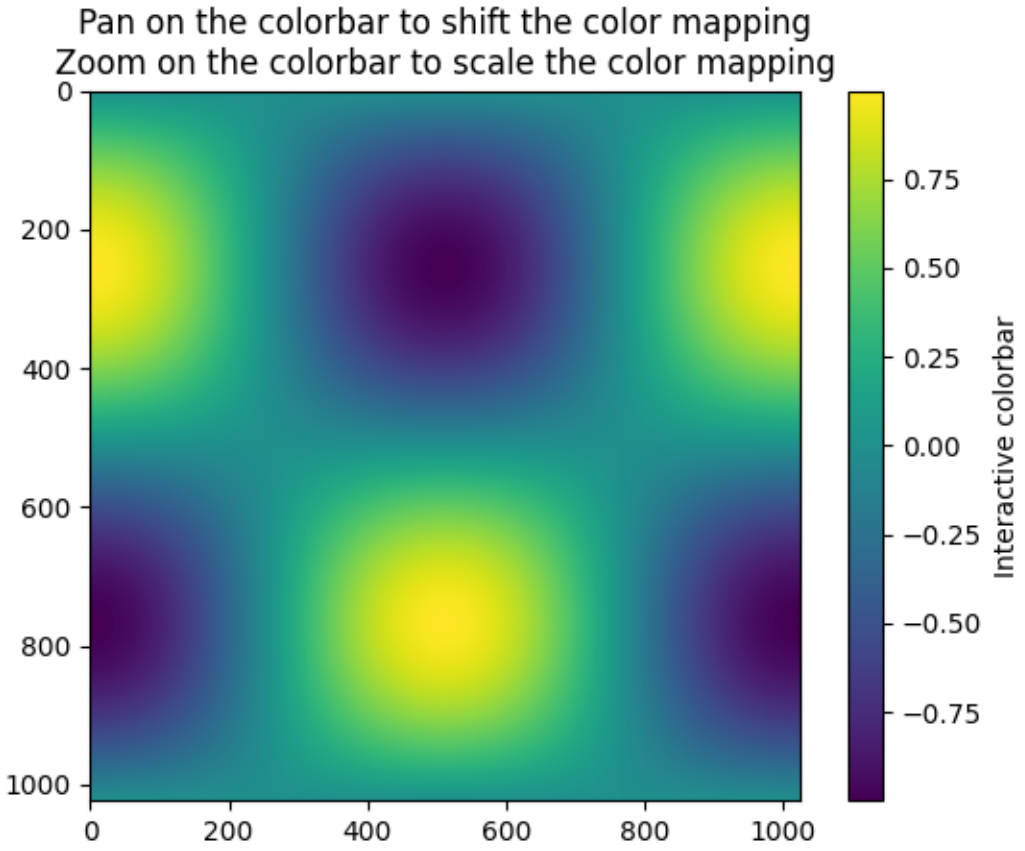
- *matplotlib.axes.Axes.imshow* / *matplotlib.pyplot.imshow*
- *matplotlib.figure.Figure.add_axes*

### Interactive Adjustment of Colormap Range

Demonstration of how a colorbar can be used to interactively adjust the range of colormapping on an image. To use the interactive feature, you must be in either zoom mode (magnifying glass toolbar button) or pan mode (4-way arrow toolbar button) and click inside the colorbar.

When zooming, the bounding box of the zoom region defines the new vmin and vmax of the norm. Zooming using the right mouse button will expand the vmin and vmax proportionally to the selected region, in the same manner that one can zoom out on an axis. When panning, the vmin and vmax of the norm are both

shifted according to the direction of movement. The Home/Back/Forward buttons can also be used to get back to a previous state.




```
import matplotlib.pyplot as plt
import numpy as np

t = np.linspace(0, 2 * np.pi, 1024)
data2d = np.sin(t)[:, np.newaxis] * np.cos(t)[np.newaxis, :]

fig, ax = plt.subplots()
im = ax.imshow(data2d)
ax.set_title('Pan on the colorbar to shift the color mapping\n'
             'Zoom on the colorbar to scale the color mapping')

fig.colorbar(im, ax=ax, label='Interactive colorbar')

plt.show()
```

### Colormap normalizations

Demonstration of using norm to map colormaps onto data in non-linear ways.

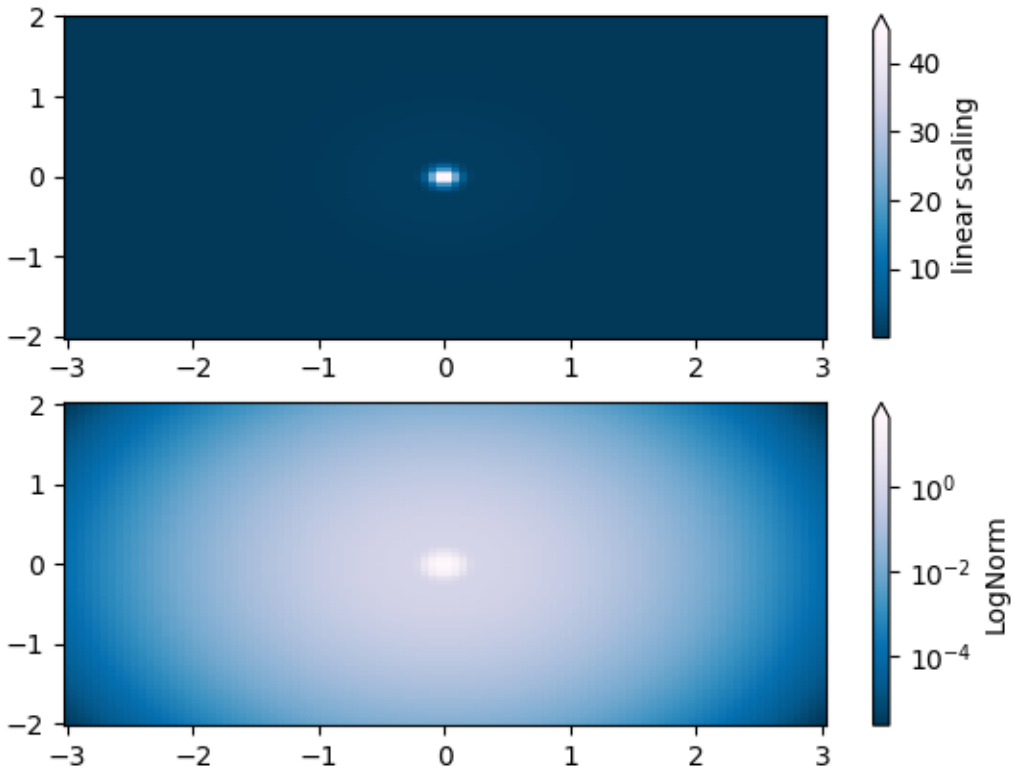
```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib.colors as colors

N = 100
```

### LogNorm

This example data has a low hump with a spike coming out of its center. If plotted using a linear colour scale, then only the spike will be visible. To see both hump and spike, this requires the z/colour axis on a log scale.

Instead of transforming the data with `pcolor(log10(Z))`, the color mapping can be made logarithmic using a *LogNorm*.

```
X, Y = np.mgrid[-3:3:complex(0, N), -2:2:complex(0, N)]
Z1 = np.exp(-X**2 - Y**2)
Z2 = np.exp(-(X * 10)**2 - (Y * 10)**2)
Z = Z1 + 50 * Z2

fig, ax = plt.subplots(2, 1)

pcm = ax[0].pcolor(X, Y, Z, cmap='PuBu_r', shading='nearest')
fig.colorbar(pcm, ax=ax[0], extend='max', label='linear scaling')

pcm = ax[1].pcolor(X, Y, Z, cmap='PuBu_r', shading='nearest',
                   norm=colors.LogNorm(vmin=Z.min(), vmax=Z.max()))
fig.colorbar(pcm, ax=ax[1], extend='max', label='LogNorm')
```

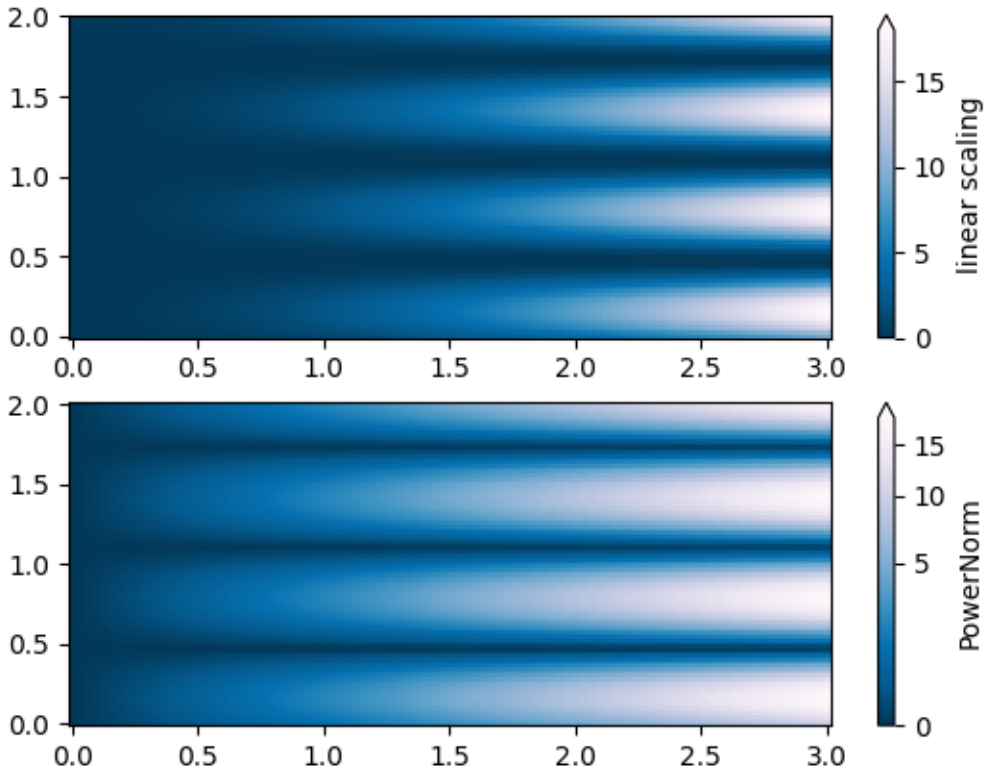



### PowerNorm

This example data mixes a power-law trend in X with a rectified sine wave in Y. If plotted using a linear colour scale, then the power-law trend in X partially obscures the sine wave in Y.

The power law can be removed using a *PowerNorm*.

```
X, Y = np.mgrid[0:3:complex(0, N), 0:2:complex(0, N)]
Z = (1 + np.sin(Y * 10)) * X**2

fig, ax = plt.subplots(2, 1)

pcm = ax[0].pcolormesh(X, Y, Z, cmap='PuBu_r', shading='nearest')
fig.colorbar(pcm, ax=ax[0], extend='max', label='linear scaling')

pcm = ax[1].pcolormesh(X, Y, Z, cmap='PuBu_r', shading='nearest',
                       norm=colors.PowerNorm(gamma=0.5))
fig.colorbar(pcm, ax=ax[1], extend='max', label='PowerNorm')
```




### SymLogNorm

This example data has two humps, one negative and one positive, The positive hump has 5 times the amplitude of the negative. If plotted with a linear colour scale, then the detail in the negative hump is obscured.

Here we logarithmically scale the positive and negative data separately with *SymLogNorm*.

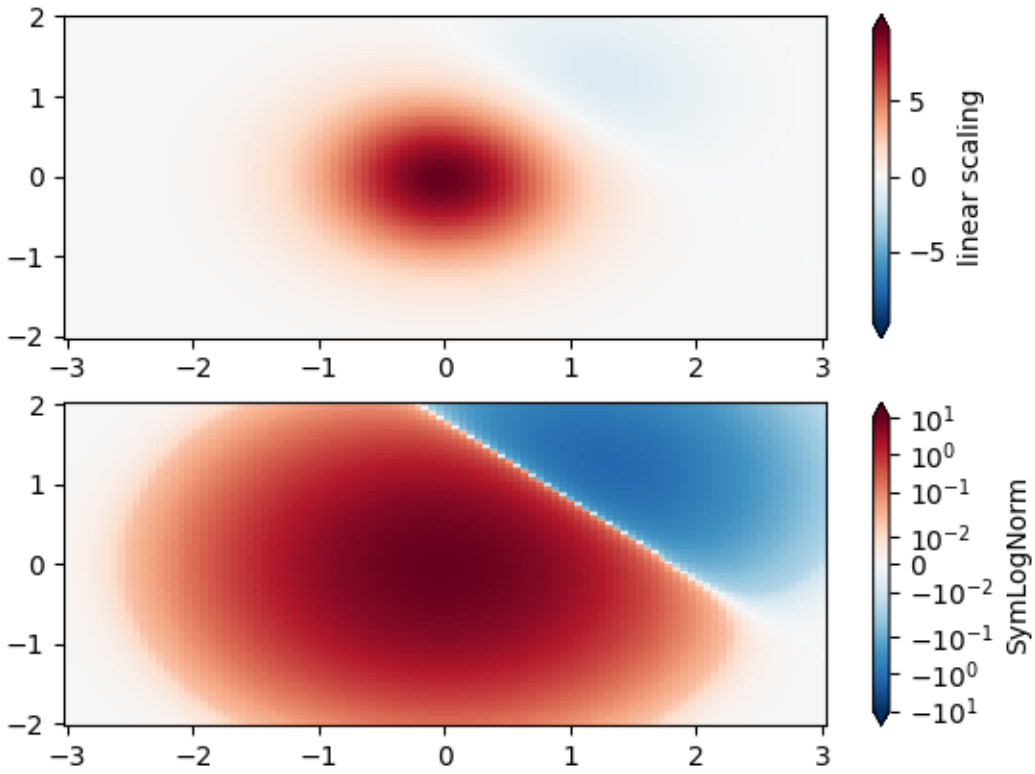
Note that colorbar labels do not come out looking very good.

```
X, Y = np.mgrid[-3:3:complex(0, N), -2:2:complex(0, N)]
Z1 = np.exp(-X**2 - Y**2)
Z2 = np.exp(-(X - 1)**2 - (Y - 1)**2)
Z = (5 * Z1 - Z2) * 2

fig, ax = plt.subplots(2, 1)

pcm = ax[0].pcolormesh(X, Y, Z, cmap='RdBu_r', shading='nearest',
                       vmin=-np.max(Z))
fig.colorbar(pcm, ax=ax[0], extend='both', label='linear scaling')

pcm = ax[1].pcolormesh(X, Y, Z, cmap='RdBu_r', shading='nearest',
                       norm=colors.SymLogNorm(linthresh=0.015,
```

(continues on next page)

(continued from previous page)

```
                                                vmin=-10.0, vmax=10.0, base=10))
fig.colorbar(pcm, ax=ax[1], extend='both', label='SymLogNorm')
```

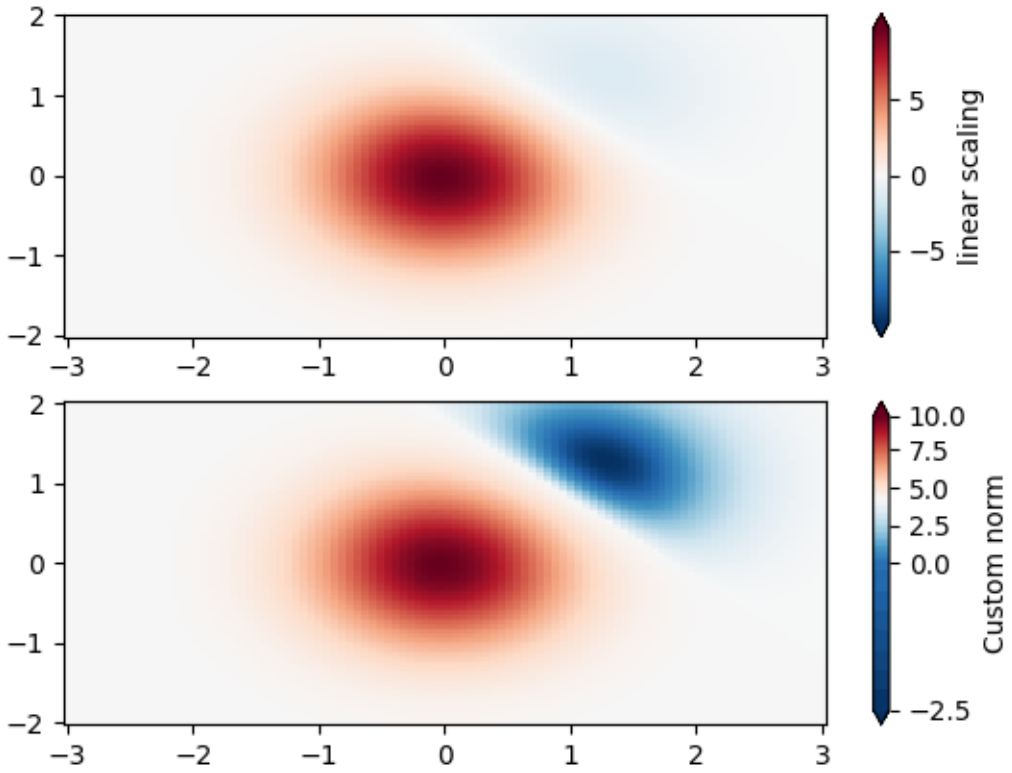



## Custom Norm

Alternatively, the above example data can be scaled with a customized normalization. This one normalizes the negative data differently from the positive.

```
# Example of making your own norm.  Also see matplotlib.colors.
# From Joe Kington: This one gives two different linear ramps:
class MidpointNormalize(colors.Normalize):
    def __init__(self, vmin=None, vmax=None, midpoint=None, clip=False):
        self.midpoint = midpoint
        super().__init__(vmin, vmax, clip)

    def __call__(self, value, clip=None):
        # I'm ignoring masked values and all kinds of edge cases to make a
        # simple example...
        x, y = [self.vmin, self.midpoint, self.vmax], [0, 0.5, 1]
        return np.ma.masked_array(np.interp(value, x, y))
```

```
fig, ax = plt.subplots(2, 1)

pcm = ax[0].pcolormesh(X, Y, Z, cmap='RdBu_r', shading='nearest',
                       vmin=-np.max(Z))
fig.colorbar(pcm, ax=ax[0], extend='both', label='linear scaling')

pcm = ax[1].pcolormesh(X, Y, Z, cmap='RdBu_r', shading='nearest',
                       norm=MidpointNormalize(midpoint=0))
fig.colorbar(pcm, ax=ax[1], extend='both', label='Custom norm')
```




### BoundaryNorm

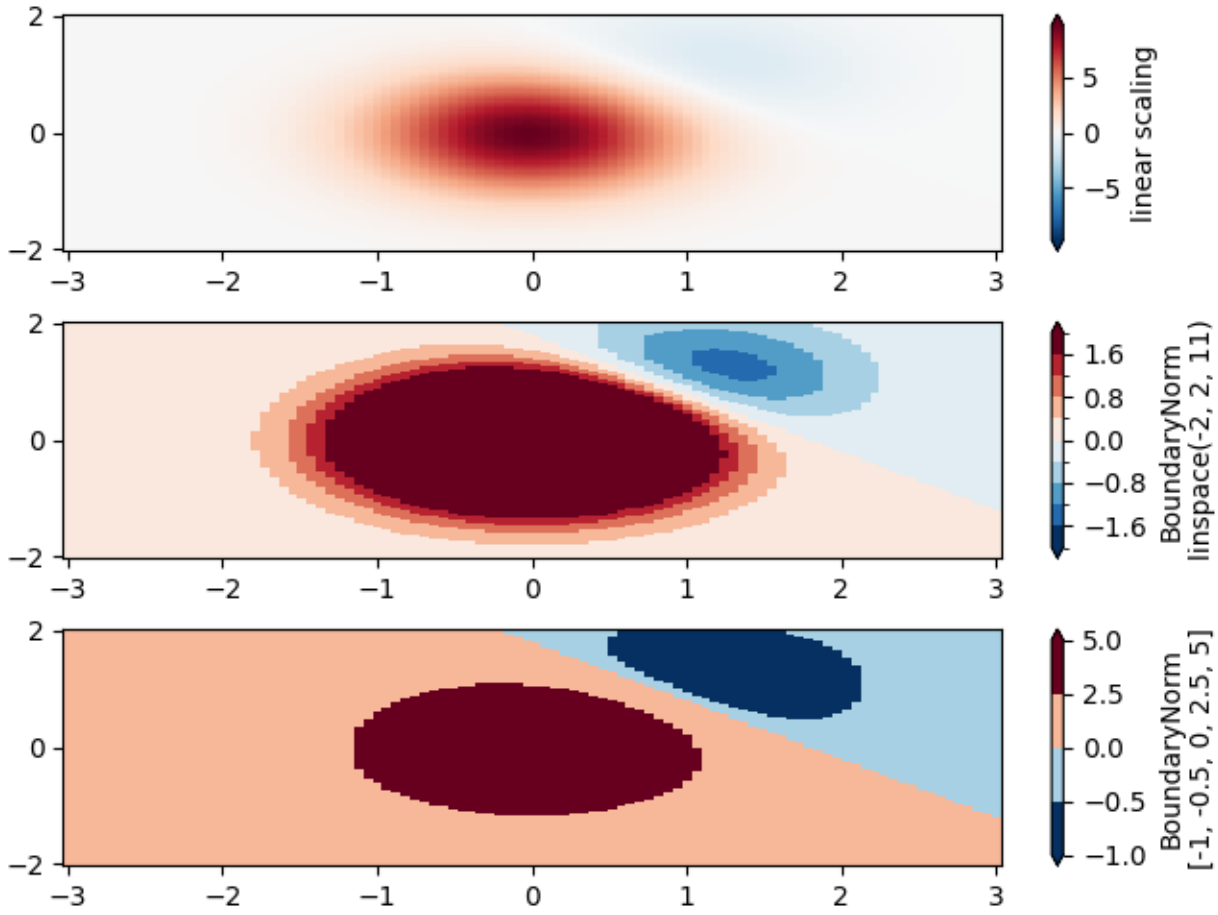
For arbitrarily dividing the color scale, the *BoundaryNorm* may be used; by providing the boundaries for colors, this norm puts the first color in between the first pair, the second color between the second pair, etc.

```
fig, ax = plt.subplots(3, 1, layout='constrained')

pcm = ax[0].pcolormesh(X, Y, Z, cmap='RdBu_r', shading='nearest',
                       vmin=-np.max(Z))
fig.colorbar(pcm, ax=ax[0], extend='both', orientation='vertical',
             label='linear scaling')
```

(continues on next page)

(continued from previous page)

```
# Evenly-spaced bounds gives a contour-like effect.
bounds = np.linspace(-2, 2, 11)
norm = colors.BoundaryNorm(boundaries=bounds, ncolors=256)
pcm = ax[1].pcolormesh(X, Y, Z, cmap='RdBu_r', shading='nearest',
                       norm=norm)
fig.colorbar(pcm, ax=ax[1], extend='both', orientation='vertical',
             label='BoundaryNorm\nlinspace(-2, 2, 11)')

# Unevenly-spaced bounds changes the colormapping.
bounds = np.array([-1, -0.5, 0, 2.5, 5])
norm = colors.BoundaryNorm(boundaries=bounds, ncolors=256)
pcm = ax[2].pcolormesh(X, Y, Z, cmap='RdBu_r', shading='nearest',
                       norm=norm)
fig.colorbar(pcm, ax=ax[2], extend='both', orientation='vertical',
             label='BoundaryNorm\n[-1, -0.5, 0, 2.5, 5]')

plt.show()
```




**Total running time of the script:** (0 minutes 2.755 seconds)

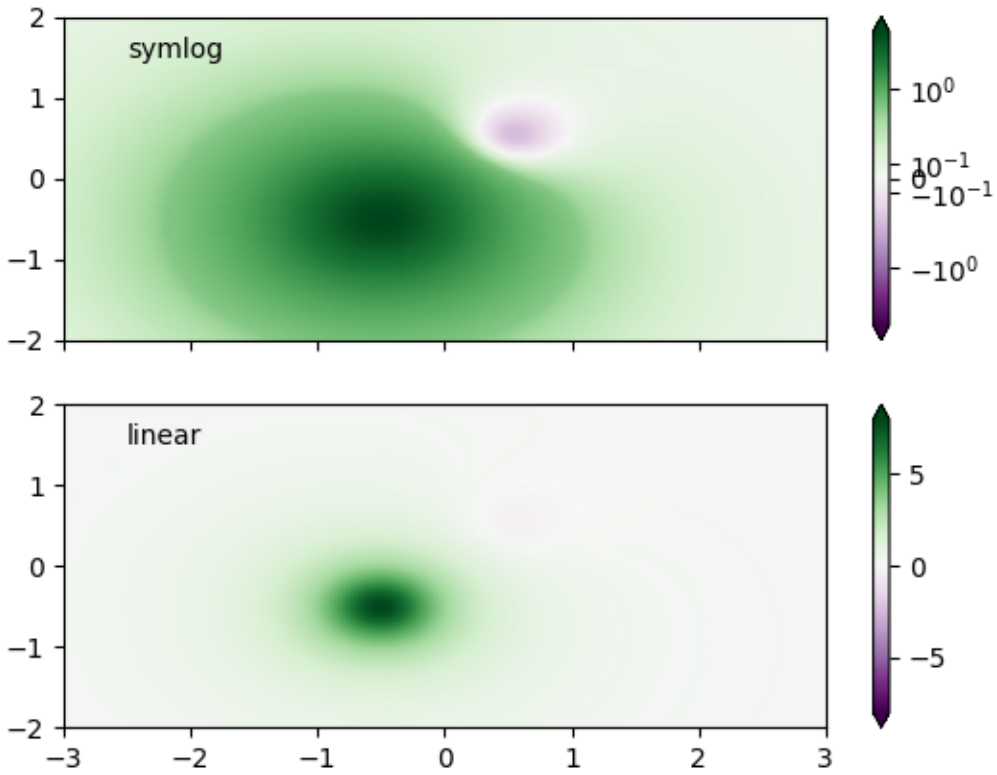
### Colormap normalizations SymLogNorm

Demonstration of using norm to map colormaps onto data in non-linear ways.

Synthetic dataset consisting of two humps, one negative and one positive, the positive with 8-times the amplitude. Linearly, the negative hump is almost invisible, and it is very difficult to see any detail of its profile. With the logarithmic scaling applied to both positive and negative values, it is much easier to see the shape of each hump.

See *SymLogNorm*.

```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib.colors as colors


def rbf(x, y):
    return 1.0 / (1 + 5 * ((x ** 2) + (y ** 2)))

N = 200
gain = 8
X, Y = np.mgrid[-3:3:complex(0, N), -2:2:complex(0, N)]
Z1 = rbf(X + 0.5, Y + 0.5)
Z2 = rbf(X - 0.5, Y - 0.5)
Z = gain * Z1 - Z2

shadeopts = {'cmap': 'PRGn', 'shading': 'gouraud'}
colormap = 'PRGn'
lnrwidth = 0.5

fig, ax = plt.subplots(2, 1, sharex=True, sharey=True)

pcm = ax[0].pcolormesh(X, Y, Z,
                       norm=colors.SymLogNorm(linthresh=lnrwidth, linscale=1,
                                              vmin=-gain, vmax=gain, base=10),
                       **shadeopts)
fig.colorbar(pcm, ax=ax[0], extend='both')
ax[0].text(-2.5, 1.5, 'symlog')

pcm = ax[1].pcolormesh(X, Y, Z, vmin=-gain, vmax=gain,
                       **shadeopts)
fig.colorbar(pcm, ax=ax[1], extend='both')
ax[1].text(-2.5, 1.5, 'linear')
```



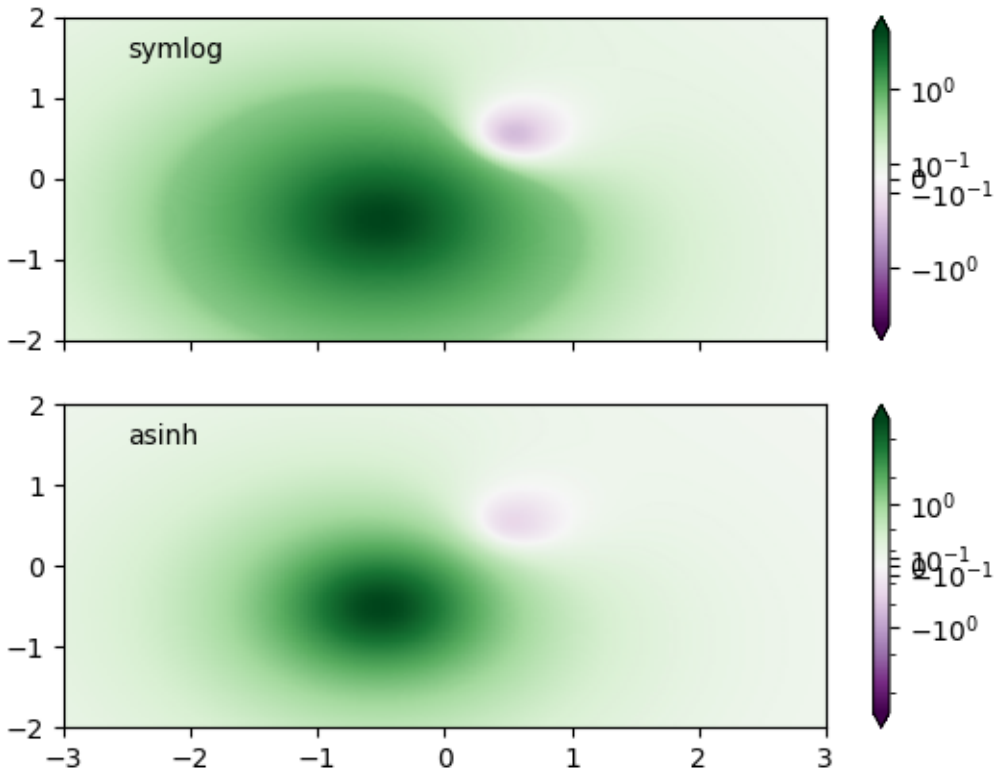

In order to find the best visualization for any particular dataset, it may be necessary to experiment with multiple different color scales. As well as the *SymLogNorm* scaling, there is also the option of using *AsinhNorm* (experimental), which has a smoother transition between the linear and logarithmic regions of the transformation applied to the data values, "Z". In the plots below, it may be possible to see contour-like artifacts around each hump despite there being no sharp features in the dataset itself. The `asinh` scaling shows a smoother shading of each hump.

```
fig, ax = plt.subplots(2, 1, sharex=True, sharey=True)

pcm = ax[0].pcolormesh(X, Y, Z,
                       norm=colors.SymLogNorm(linthresh=lnrwidth, linscale=1,
                                              vmin=-gain, vmax=gain, base=10),
                       **shadeopts)
fig.colorbar(pcm, ax=ax[0], extend='both')
ax[0].text(-2.5, 1.5, 'symlog')

pcm = ax[1].pcolormesh(X, Y, Z,
                       norm=colors.AsinhNorm(linear_width=lnrwidth,
                                             vmin=-gain, vmax=gain),
                       **shadeopts)
fig.colorbar(pcm, ax=ax[1], extend='both')
ax[1].text(-2.5, 1.5, 'asinh')
```

(continues on next page)

(continued from previous page)

```
plt.show()
```




**Total running time of the script:** (0 minutes 2.617 seconds)

## Contour Corner Mask

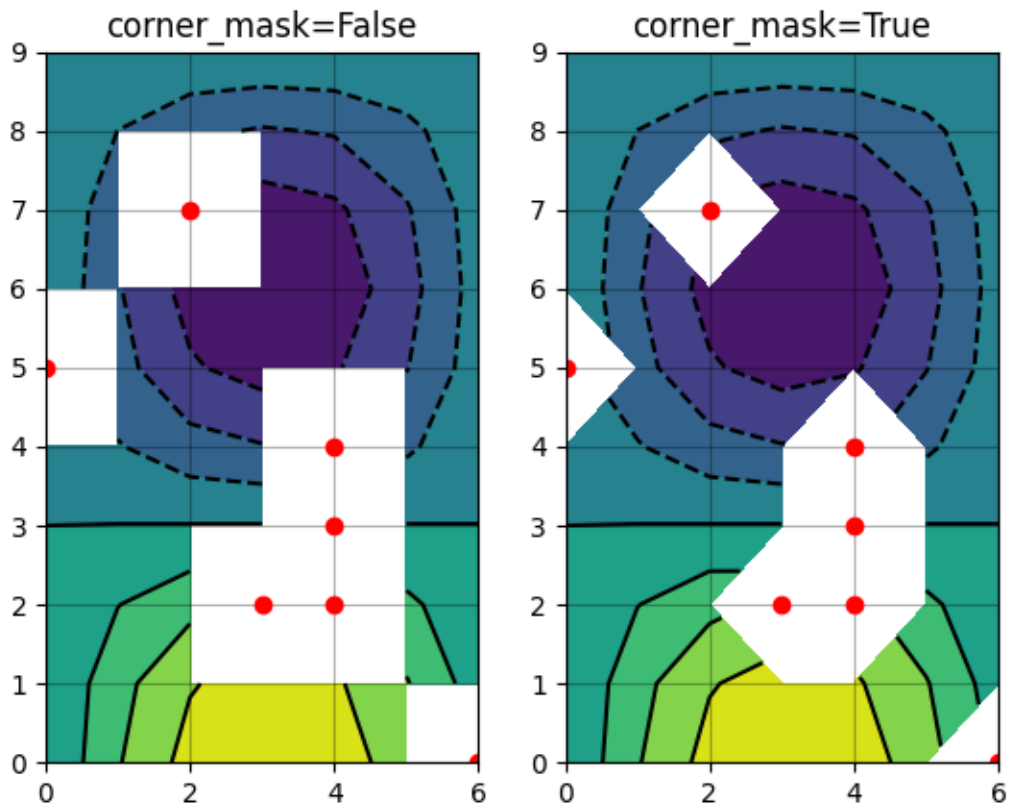
Illustrate the difference between `corner_mask=False` and `corner_mask=True` for masked contour plots. The default is controlled by *`rcParams["contour.corner_mask"]`* (default: `True`).

```
import matplotlib.pyplot as plt
import numpy as np

# Data to plot.
x, y = np.meshgrid(np.arange(7), np.arange(10))
z = np.sin(0.5 * x) * np.cos(0.52 * y)

# Mask various z values.
mask = np.zeros_like(z, dtype=bool)
mask[2, 3:5] = True
mask[3:5, 4] = True
```

(continues on next page)

(continued from previous page)

```
mask[7, 2] = True
mask[5, 0] = True
mask[0, 6] = True
z = np.ma.array(z, mask=mask)

corner_masks = [False, True]
fig, axs = plt.subplots(ncols=2)
for ax, corner_mask in zip(axs, corner_masks):
    cs = ax.contourf(x, y, z, corner_mask=corner_mask)
    ax.contour(cs, colors='k')
    ax.set_title(f'{corner_mask=}')

    # Plot grid.
    ax.grid(c='k', ls='-', alpha=0.3)

    # Indicate masked points with red circles.
    ax.plot(np.ma.array(x, mask=~mask), y, 'ro')

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.contour* / *matplotlib.pyplot.contour*
- *matplotlib.axes.Axes.contourf* / *matplotlib.pyplot.contourf*

### Contour Demo

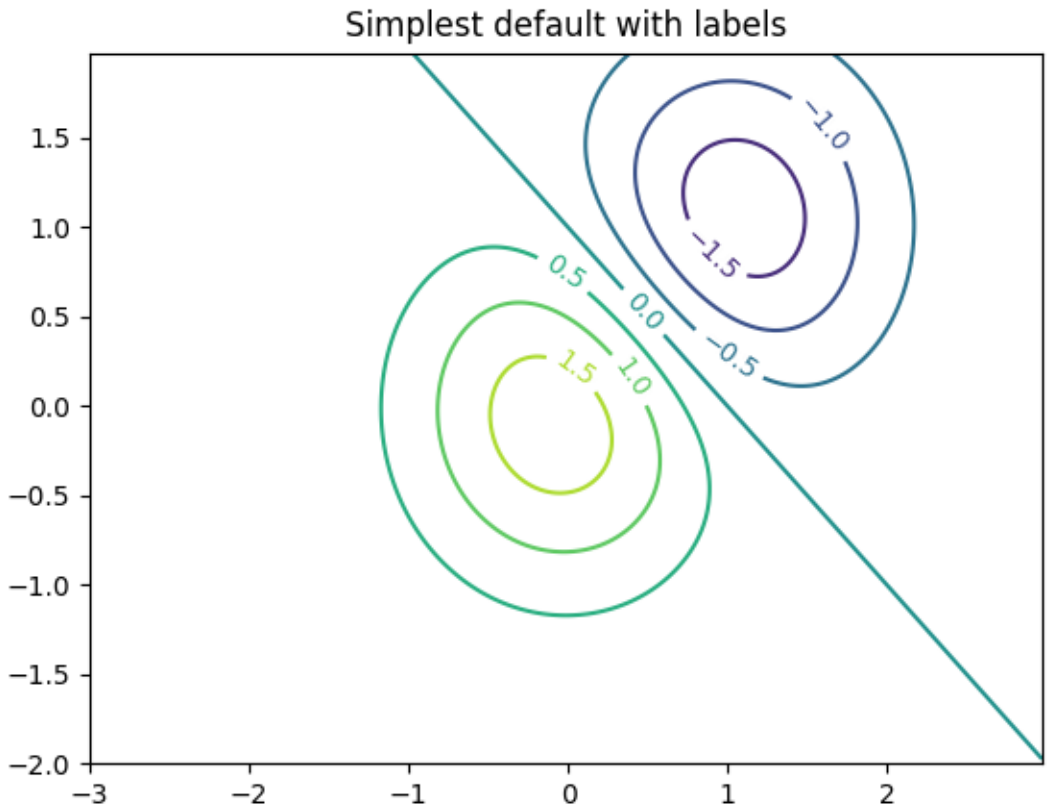
Illustrate simple contour plotting, contours on an image with a colorbar for the contours, and labelled contours.

See also the *contour image example*.

```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib.cm as cm

delta = 0.025
x = np.arange(-3.0, 3.0, delta)
y = np.arange(-2.0, 2.0, delta)
X, Y = np.meshgrid(x, y)
Z1 = np.exp(-X**2 - Y**2)
Z2 = np.exp(-(X - 1)**2 - (Y - 1)**2)
Z = (Z1 - Z2) * 2
```

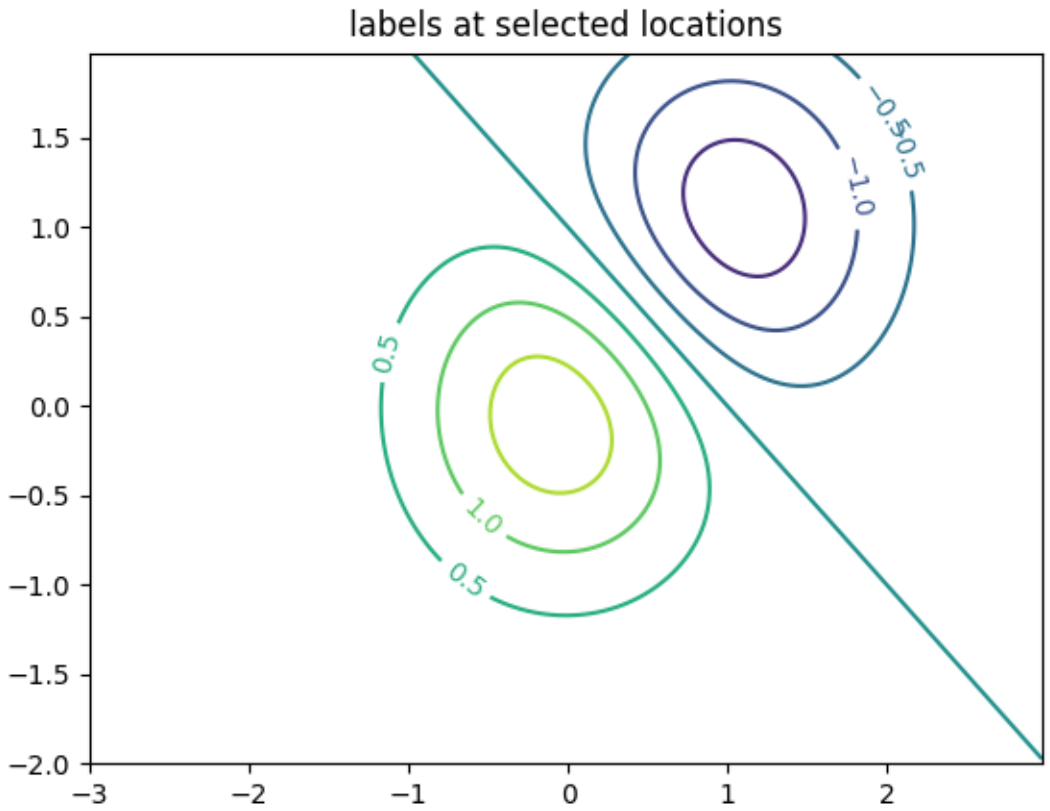
Create a simple contour plot with labels using default colors. The inline argument to clabel will control whether the labels are draw over the line segments of the contour, removing the lines beneath the label.

```
fig, ax = plt.subplots()
CS = ax.contour(X, Y, Z)
ax.clabel(CS, inline=True, fontsize=10)
ax.set_title('Simplest default with labels')
```



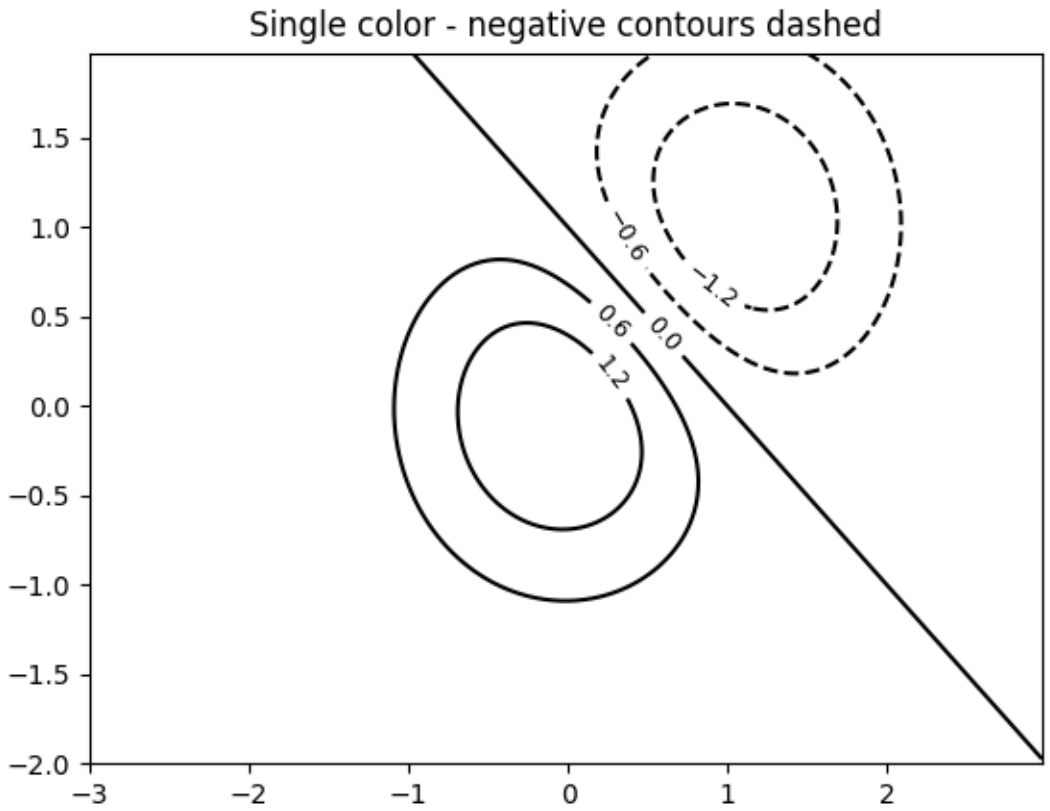

Contour labels can be placed manually by providing list of positions (in data coordinate). See *Interactive functions* for interactive placement.

```
fig, ax = plt.subplots()
CS = ax.contour(X, Y, Z)
manual_locations = [
    (-1, -1.4), (-0.62, -0.7), (-2, 0.5), (1.7, 1.2), (2.0, 1.4), (2.4, 1.7)]
ax.clabel(CS, inline=True, fontsize=10, manual=manual_locations)
ax.set_title('labels at selected locations')
```



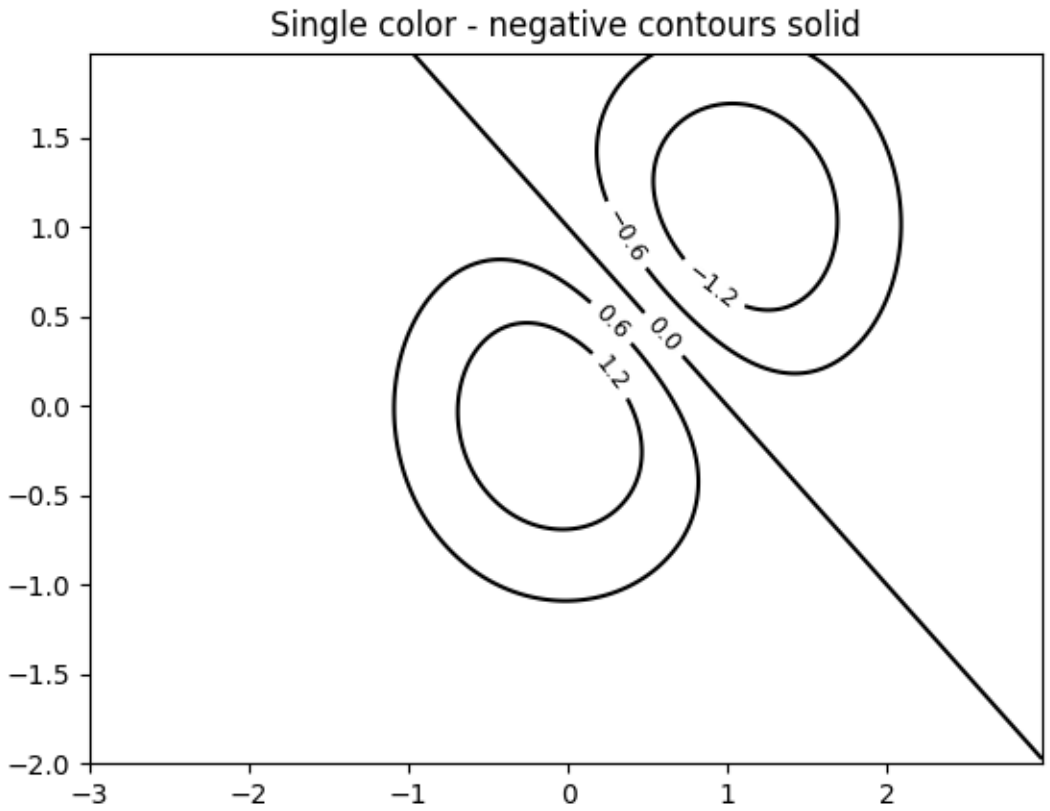

You can force all the contours to be the same color.

```
fig, ax = plt.subplots()
CS = ax.contour(X, Y, Z, 6, colors='k')  # Negative contours default to␣
 ↪dashed.
ax.clabel(CS, fontsize=9, inline=True)
ax.set_title('Single color - negative contours dashed')
```



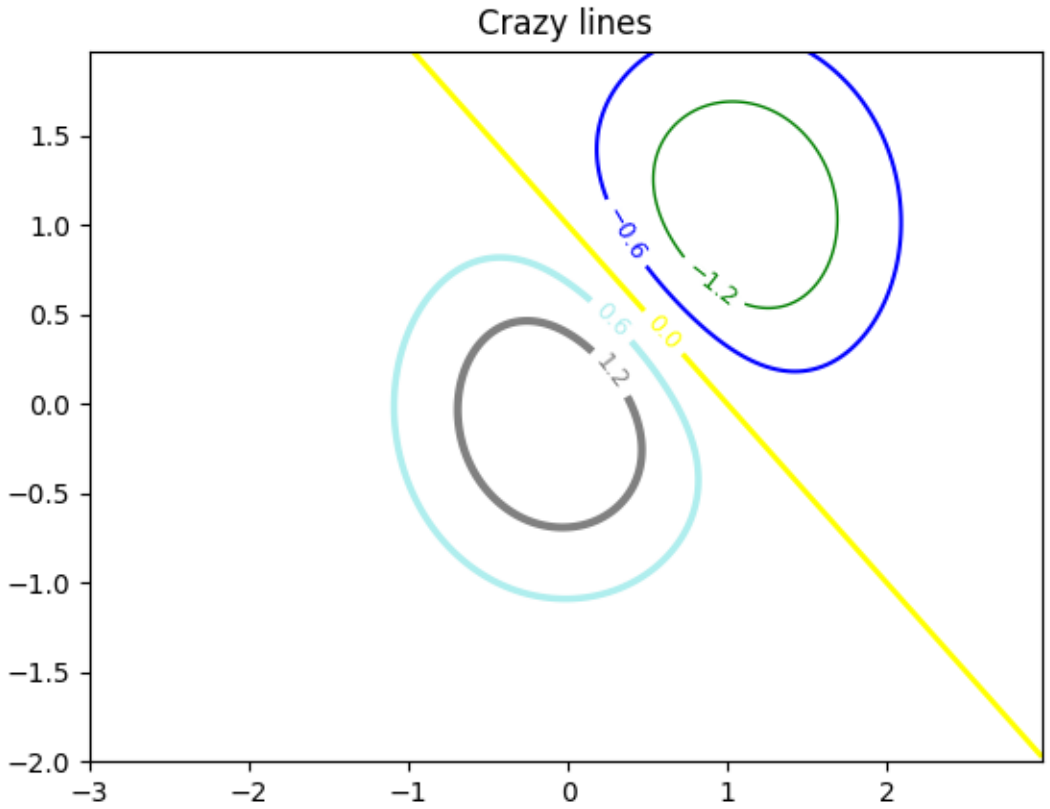

You can set negative contours to be solid instead of dashed:

```
plt.rcParams['contour.negative_linestyle'] = 'solid'
fig, ax = plt.subplots()
CS = ax.contour(X, Y, Z, 6, colors='k')  # Negative contours default to␣
↪dashed.
ax.clabel(CS, fontsize=9, inline=True)
ax.set_title('Single color - negative contours solid')
```




And you can manually specify the colors of the contour

```
fig, ax = plt.subplots()
CS = ax.contour(X, Y, Z, 6,
                linewidths=np.arange(.5, 4, .5),
                colors=('r', 'green', 'blue', (1, 1, 0), '#afeeee', '0.5'),
                )
ax.clabel(CS, fontsize=9, inline=True)
ax.set_title('Crazy lines')
```



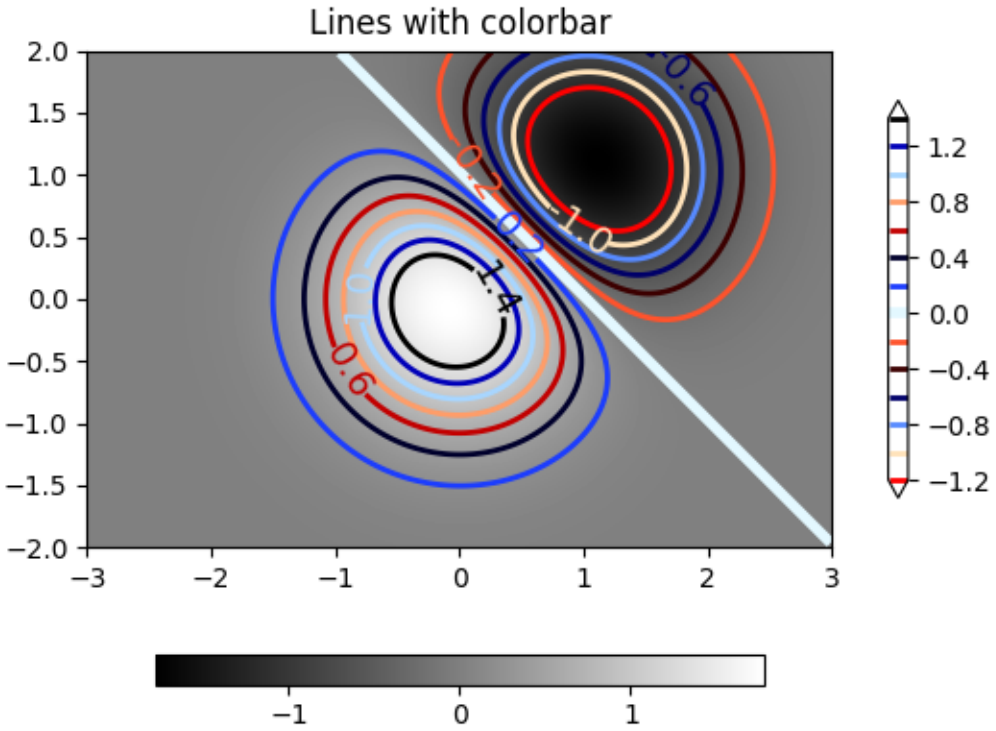

Or you can use a colormap to specify the colors; the default colormap will be used for the contour lines

```
fig, ax = plt.subplots()
im = ax.imshow(Z, interpolation='bilinear', origin='lower',
               cmap=cm.gray, extent=(-3, 3, -2, 2))
levels = np.arange(-1.2, 1.6, 0.2)
CS = ax.contour(Z, levels, origin='lower', cmap='flag', extend='both',
                linewidths=2, extent=(-3, 3, -2, 2))

# Thicken the zero contour.
lws = np.resize(CS.get_linewidth(), len(levels))
lws[6] = 4
CS.set_linewidth(lws)

ax.clabel(CS, levels[1::2],  # label every second level
          inline=True, fmt='%1.1f', fontsize=14)

# make a colorbar for the contour lines
CB = fig.colorbar(CS, shrink=0.8)

ax.set_title('Lines with colorbar')

# We can still add a colorbar for the image, too.
```

(continues on next page)

(continued from previous page)

```
CBI = fig.colorbar(im, orientation='horizontal', shrink=0.8)

# This makes the original colorbar look a bit out of place,
# so let's improve its position.

l, b, w, h = ax.get_position().bounds
ll, bb, ww, hh = CB.ax.get_position().bounds
CB.ax.set_position([ll, b + 0.1*h, ww, h*0.8])

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.contour* / *matplotlib.pyplot.contour*
- *matplotlib.figure.Figure.colorbar* / *matplotlib.pyplot.colorbar*
- *matplotlib.axes.Axes.clabel* / *matplotlib.pyplot.clabel*
- *matplotlib.axes.Axes.get_position*

- *matplotlib.axes.Axes.set_position*

**Total running time of the script:** (0 minutes 2.155 seconds)

## Contour Image

Test combinations of contouring, filled contouring, and image plotting. For contour labelling, see also the *contour demo example*.

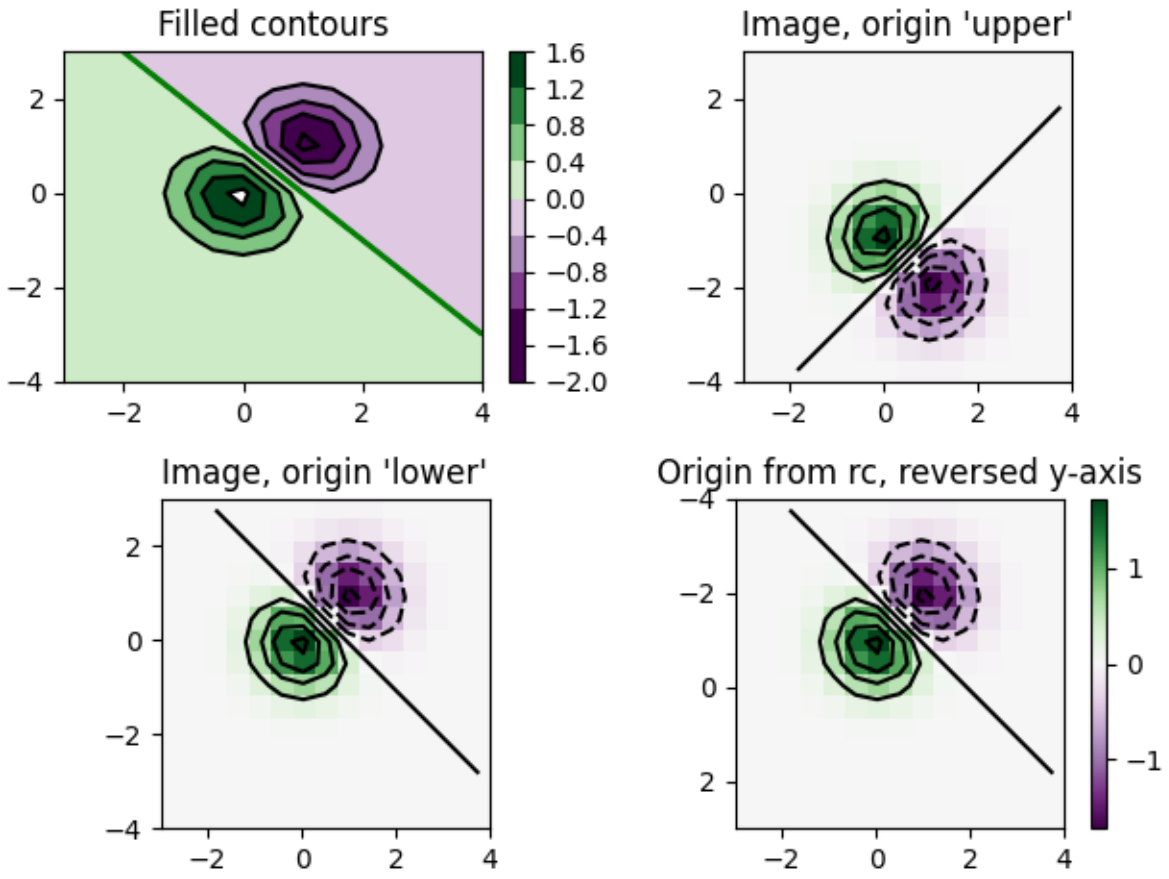
The emphasis in this demo is on showing how to make contours register correctly on images, and on how to get both of them oriented as desired. In particular, note the usage of the *"origin" and "extent"* keyword arguments to imshow and contour.

```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib import cm

# Default delta is large because that makes it fast, and it illustrates
# the correct registration between image and contours.
delta = 0.5

extent = (-3, 4, -4, 3)

x = np.arange(-3.0, 4.001, delta)
y = np.arange(-4.0, 3.001, delta)
X, Y = np.meshgrid(x, y)
Z1 = np.exp(-X**2 - Y**2)
Z2 = np.exp(-(X - 1)**2 - (Y - 1)**2)
Z = (Z1 - Z2) * 2

# Boost the upper limit to avoid truncation errors.
levels = np.arange(-2.0, 1.601, 0.4)

norm = cm.colors.Normalize(vmax=abs(Z).max(), vmin=-abs(Z).max())
cmap = cm.PRGn

fig, _axs = plt.subplots(nrows=2, ncols=2)
fig.subplots_adjust(hspace=0.3)
axs = _axs.flatten()

cset1 = axs[0].contourf(X, Y, Z, levels, norm=norm,
                        cmap=cmap.resampled(len(levels) - 1))
# It is not necessary, but for the colormap, we need only the
# number of levels minus 1.  To avoid discretization error, use
# either this number or a large number such as the default (256).

# If we want lines as well as filled regions, we need to call
# contour separately; don't try to change the edgecolor or edgewidth
# of the polygons in the collections returned by contourf.
# Use levels output from previous call to guarantee they are the same.
```

(continues on next page)

(continued from previous page)

```
cset2 = axs[0].contour(X, Y, Z, cset1.levels, colors='k')

# We don't really need dashed contour lines to indicate negative
# regions, so let's turn them off.
cset2.set_linestyle('solid')

# It is easier here to make a separate call to contour than
# to set up an array of colors and linewidths.
# We are making a thick green line as a zero contour.
# Specify the zero level as a tuple with only 0 in it.

cset3 = axs[0].contour(X, Y, Z, (0,), colors='g', linewidths=2)
axs[0].set_title('Filled contours')
fig.colorbar(cset1, ax=axs[0])


axs[1].imshow(Z, extent=extent, cmap=cmap, norm=norm)
axs[1].contour(Z, levels, colors='k', origin='upper', extent=extent)
axs[1].set_title("Image, origin 'upper'")

axs[2].imshow(Z, origin='lower', extent=extent, cmap=cmap, norm=norm)
axs[2].contour(Z, levels, colors='k', origin='lower', extent=extent)
axs[2].set_title("Image, origin 'lower'")

# We will use the interpolation "nearest" here to show the actual
# image pixels.
# Note that the contour lines don't extend to the edge of the box.
# This is intentional. The Z values are defined at the center of each
# image pixel (each color block on the following subplot), so the
# domain that is contoured does not extend beyond these pixel centers.
im = axs[3].imshow(Z, interpolation='nearest', extent=extent,
                   cmap=cmap, norm=norm)
axs[3].contour(Z, levels, colors='k', origin='image', extent=extent)
ylim = axs[3].get_ylim()
axs[3].set_ylim(ylim[::-1])
axs[3].set_title("Origin from rc, reversed y-axis")
fig.colorbar(im, ax=axs[3])

fig.tight_layout()
plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.contour* / *matplotlib.pyplot.contour*
- *matplotlib.axes.Axes.imshow* / *matplotlib.pyplot.imshow*
- *matplotlib.figure.Figure.colorbar* / *matplotlib.pyplot.colorbar*
- *matplotlib.colors.Normalize*

## Contour Label Demo

Illustrate some of the more advanced things that one can do with contour labels.

See also the *contour demo example*.

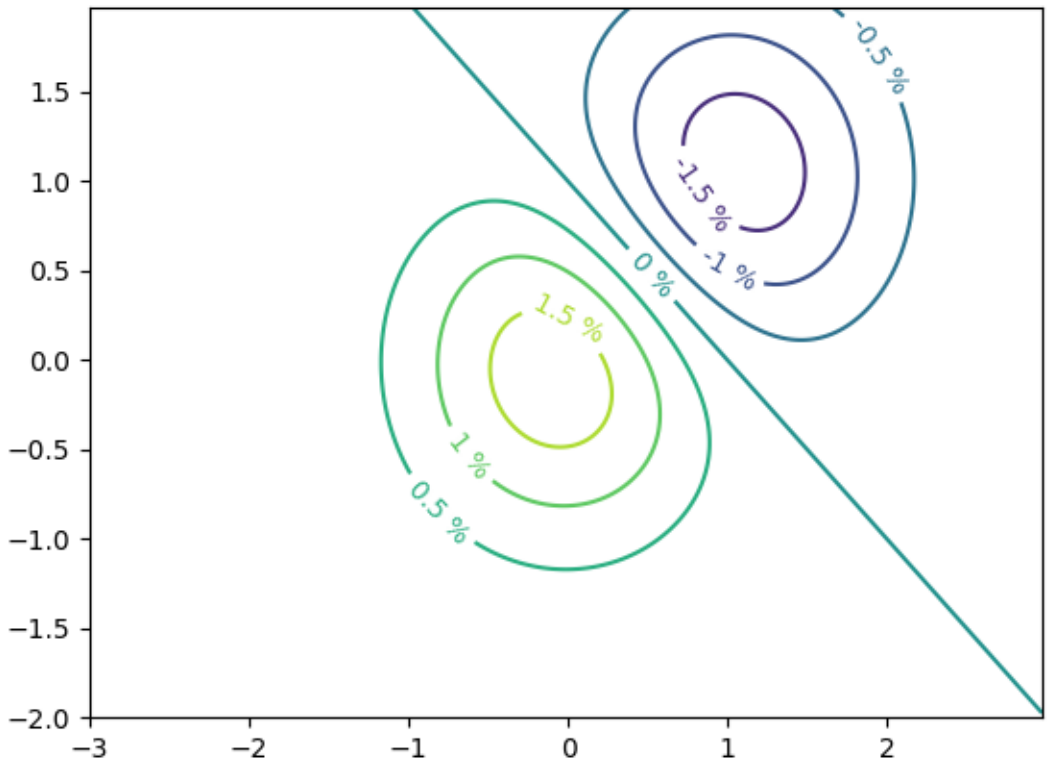
```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib.ticker as ticker
```

Define our surface

```
delta = 0.025
x = np.arange(-3.0, 3.0, delta)
y = np.arange(-2.0, 2.0, delta)
X, Y = np.meshgrid(x, y)
Z1 = np.exp(-X**2 - Y**2)
Z2 = np.exp(-(X - 1)**2 - (Y - 1)**2)
Z = (Z1 - Z2) * 2
```

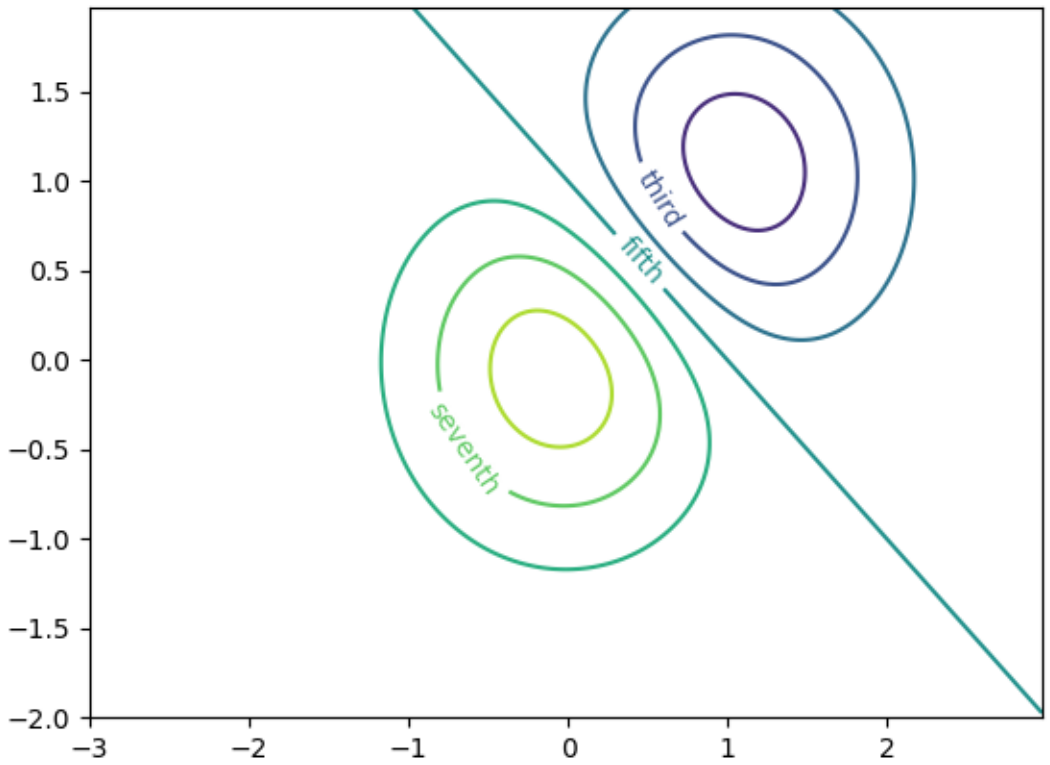
Make contour labels with custom level formatters

```
# This custom formatter removes trailing zeros, e.g. "1.0" becomes "1", and
# then adds a percent sign.
def fmt(x):
    s = f"{x:.1f}"
    if s.endswith("0"):
        s = f"{x:.0f}"
    return rf"{s} \%" if plt.rcParams["text.usetex"] else f"{s} %"


# Basic contour plot
fig, ax = plt.subplots()
CS = ax.contour(X, Y, Z)

ax.clabel(CS, CS.levels, inline=True, fmt=fmt, fontsize=10)
```



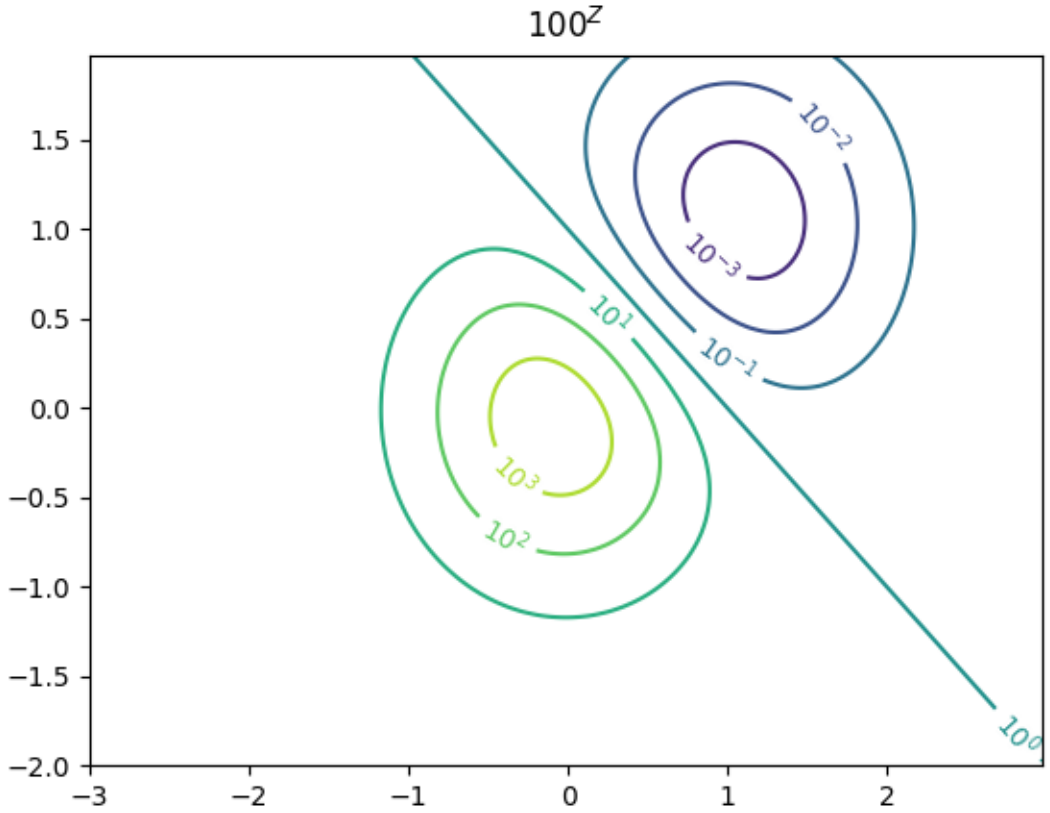

Label contours with arbitrary strings using a dictionary

```
fig1, ax1 = plt.subplots()

# Basic contour plot
CS1 = ax1.contour(X, Y, Z)

fmt = {}
strs = ['first', 'second', 'third', 'fourth', 'fifth', 'sixth', 'seventh']
for l, s in zip(CS1.levels, strs):
    fmt[l] = s

# Label every other level using strings
ax1.clabel(CS1, CS1.levels[::2], inline=True, fmt=fmt, fontsize=10)
```




Use a Formatter

```
fig2, ax2 = plt.subplots()

CS2 = ax2.contour(X, Y, 100**Z, locator=plt.LogLocator())
fmt = ticker.LogFormatterMathtext()
fmt.create_dummy_axis()
ax2.clabel(CS2, CS2.levels, fmt=fmt)
ax2.set_title("$100^Z$")

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.contour* / *matplotlib.pyplot.contour*
- *matplotlib.axes.Axes.clabel* / *matplotlib.pyplot.clabel*
- *matplotlib.ticker.LogFormatterMathtext*
- *matplotlib.ticker.TickHelper.create_dummy_axis*

**Total running time of the script:** (0 minutes 1.172 seconds)

### Contourf demo

How to use the *axes.Axes.contourf* method to create filled contour plots.

```
import matplotlib.pyplot as plt
import numpy as np

delta = 0.025

x = y = np.arange(-3.0, 3.01, delta)
X, Y = np.meshgrid(x, y)
Z1 = np.exp(-X**2 - Y**2)
Z2 = np.exp(-(X - 1)**2 - (Y - 1)**2)
Z = (Z1 - Z2) * 2

nr, nc = Z.shape

# put NaNs in one corner:
Z[-nr // 6:, -nc // 6:] = np.nan
# contourf will convert these to masked


Z = np.ma.array(Z)
# mask another corner:
Z[:nr // 6, :nc // 6] = np.ma.masked

# mask a circle in the middle:
interior = np.sqrt(X**2 + Y**2) < 0.5
Z[interior] = np.ma.masked
```

### Automatic contour levels

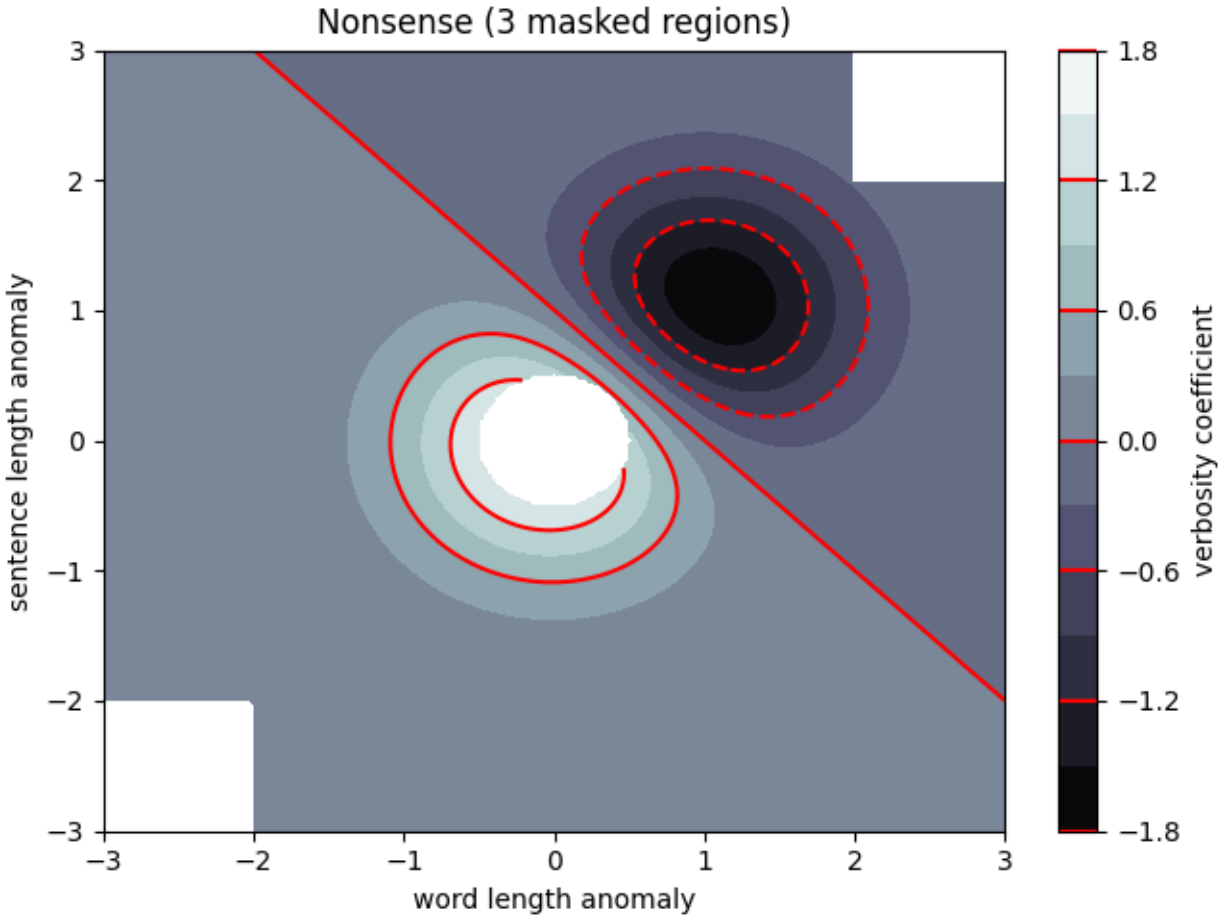
We are using automatic selection of contour levels; this is usually not such a good idea, because they don't occur on nice boundaries, but we do it here for purposes of illustration.

```
fig1, ax2 = plt.subplots(layout='constrained')
CS = ax2.contourf(X, Y, Z, 10, cmap=plt.cm.bone)

# Note that in the following, we explicitly pass in a subset of the contour
# levels used for the filled contours.  Alternatively, we could pass in
# additional levels to provide extra resolution, or leave out the *levels*
# keyword argument to use all of the original levels.

CS2 = ax2.contour(CS, levels=CS.levels[::2], colors='r')

ax2.set_title('Nonsense (3 masked regions)')
ax2.set_xlabel('word length anomaly')
ax2.set_ylabel('sentence length anomaly')

# Make a colorbar for the ContourSet returned by the contourf call.
cbar = fig1.colorbar(CS)
```

(continues on next page)

(continued from previous page)

```
cbar.ax.set_ylabel('verbosity coefficient')
# Add the contour line levels to the colorbar
cbar.add_lines(CS2)
```




## Explicit contour levels

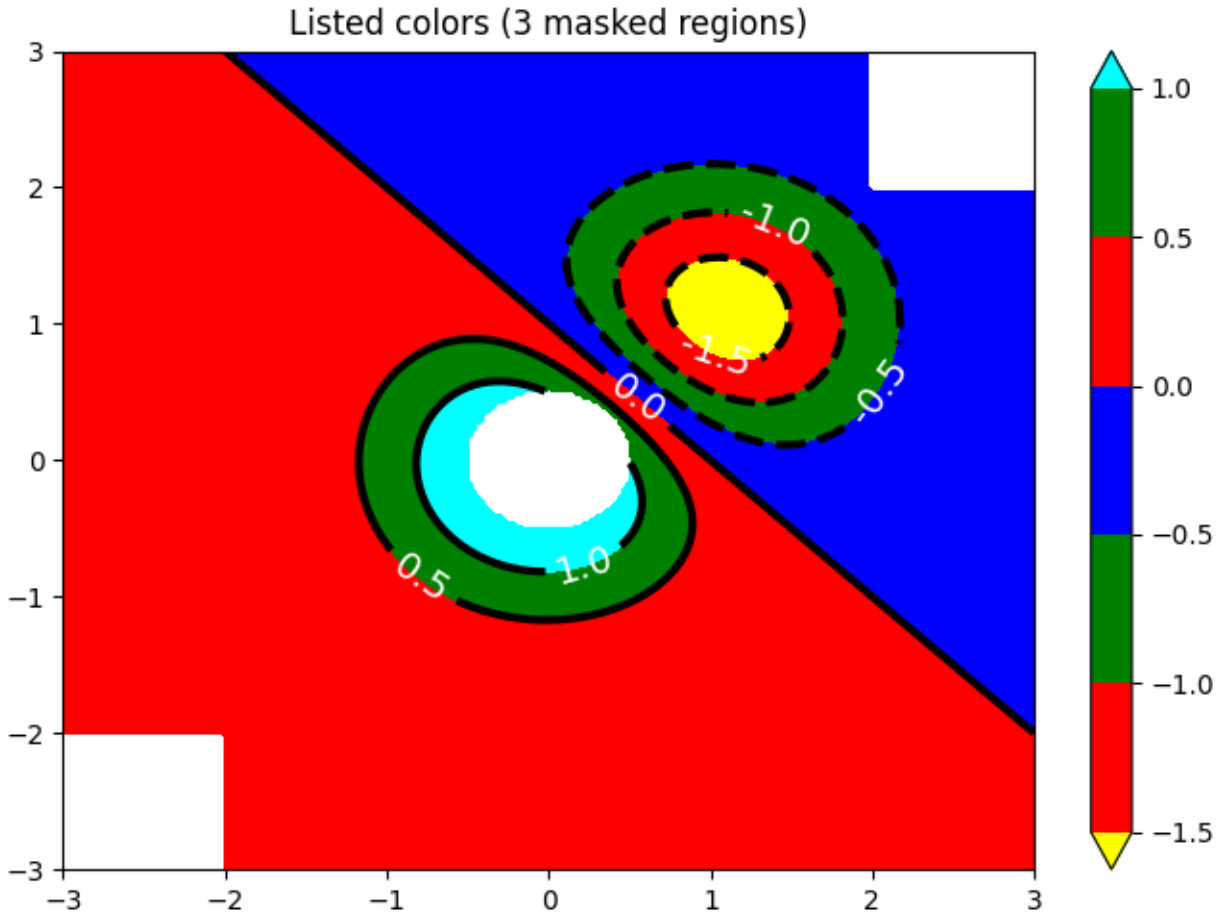
Now make a contour plot with the levels specified, and with the colormap generated automatically from a list of colors.

```
fig2, ax2 = plt.subplots(layout='constrained')
levels = [-1.5, -1, -0.5, 0, 0.5, 1]
CS3 = ax2.contourf(X, Y, Z, levels, colors=('r', 'g', 'b'), extend='both')
# Our data range extends outside the range of levels; make
# data below the lowest contour level yellow, and above the
# highest level cyan:
CS3.cmap.set_under('yellow')
CS3.cmap.set_over('cyan')

CS4 = ax2.contour(X, Y, Z, levels, colors=('k',), linewidths=(3,))
ax2.set_title('Listed colors (3 masked regions)')
```

(continues on next page)

(continued from previous page)

```
ax2.clabel(CS4, fmt='%2.1f', colors='w', fontsize=14)

# Notice that the colorbar gets all the information it
# needs from the ContourSet object, CS3.
fig2.colorbar(CS3)
```




## Extension settings

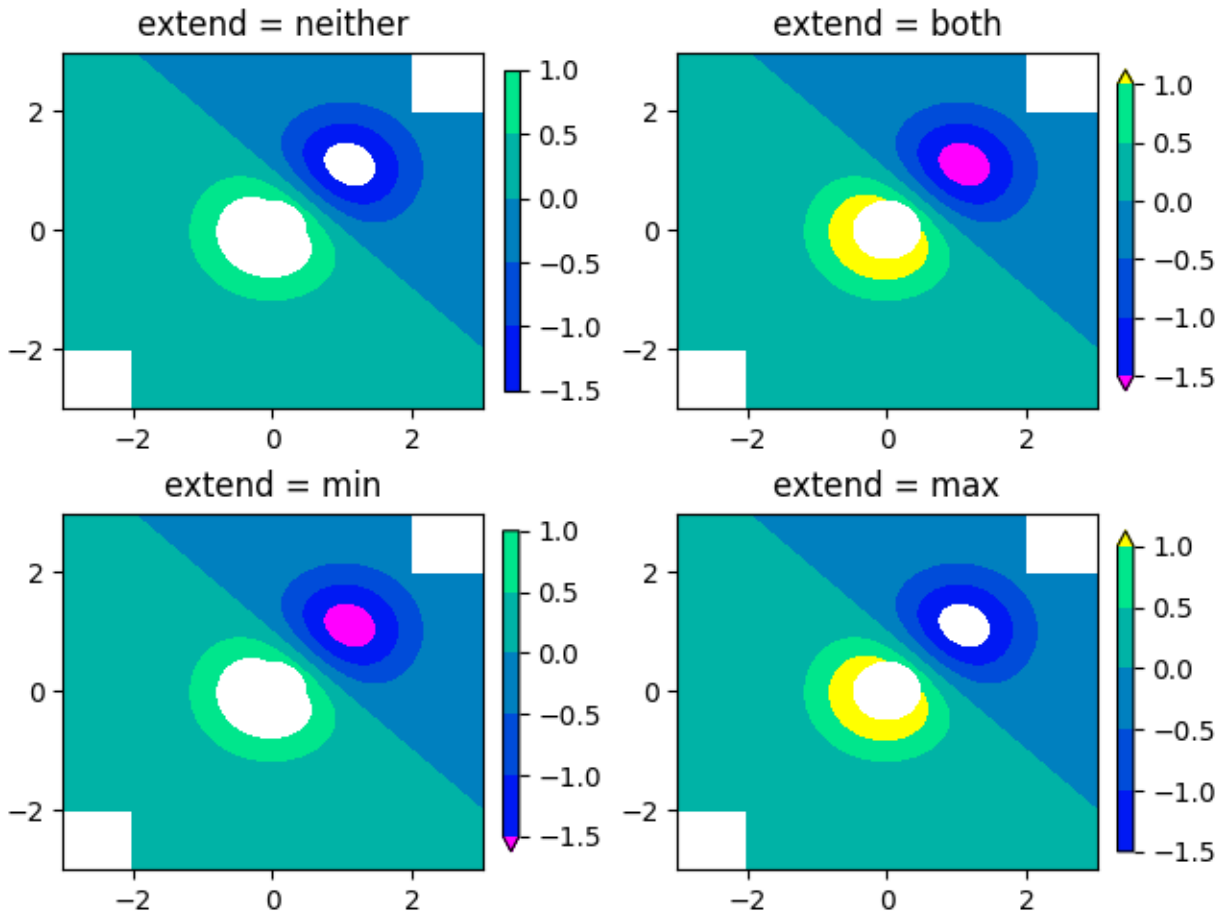
Illustrate all 4 possible "extend" settings:

```
extends = ["neither", "both", "min", "max"]
cmap = plt.colormaps["winter"].with_extremes(under="magenta", over="yellow")
# Note: contouring simply excludes masked or nan regions, so
# instead of using the "bad" colormap value for them, it draws
# nothing at all in them.  Therefore, the following would have
# no effect:
# cmap.set_bad("red")

fig, axs = plt.subplots(2, 2, layout="constrained")
```

(continues on next page)

(continued from previous page)

```
for ax, extend in zip(axs.flat, extends):
    cs = ax.contourf(X, Y, Z, levels, cmap=cmap, extend=extend)
    fig.colorbar(cs, ax=ax, shrink=0.9)
    ax.set_title("extend = %s" % extend)
    ax.locator_params(nbins=4)

plt.show()
```




## Orient contour plots using the origin keyword

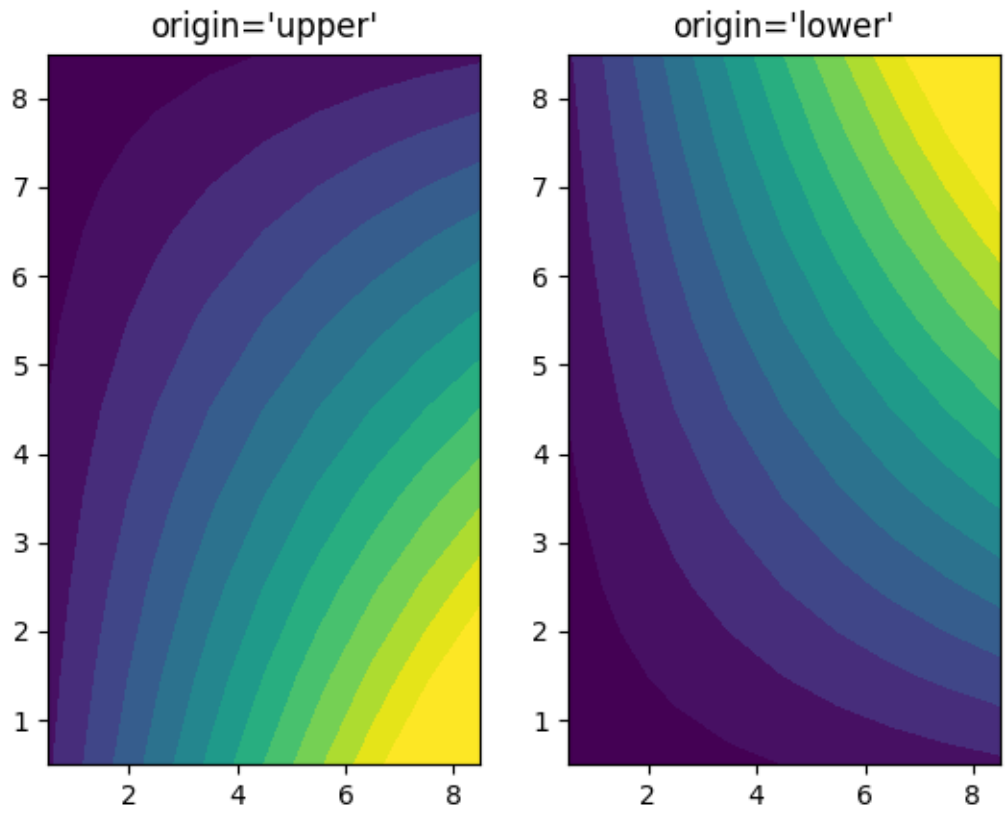
This code demonstrates orienting contour plot data using the "origin" keyword

```
x = np.arange(1, 10)
y = x.reshape(-1, 1)
h = x * y

fig, (ax1, ax2) = plt.subplots(ncols=2)

ax1.set_title("origin='upper'")
ax2.set_title("origin='lower'")
```

(continues on next page)

(continued from previous page)

```
ax1.contourf(h, levels=np.arange(5, 70, 5), extend='both', origin="upper")
ax2.contourf(h, levels=np.arange(5, 70, 5), extend='both', origin="lower")

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.contour* / *matplotlib.pyplot.contour*
- *matplotlib.axes.Axes.contourf* / *matplotlib.pyplot.contourf*
- *matplotlib.axes.Axes.clabel* / *matplotlib.pyplot.clabel*
- *matplotlib.figure.Figure.colorbar* / *matplotlib.pyplot.colorbar*
- *matplotlib.colors.Colormap*
- *matplotlib.colors.Colormap.set_bad*
- *matplotlib.colors.Colormap.set_under*
- *matplotlib.colors.Colormap.set_over*

**Total running time of the script:** (0 minutes 2.121 seconds)

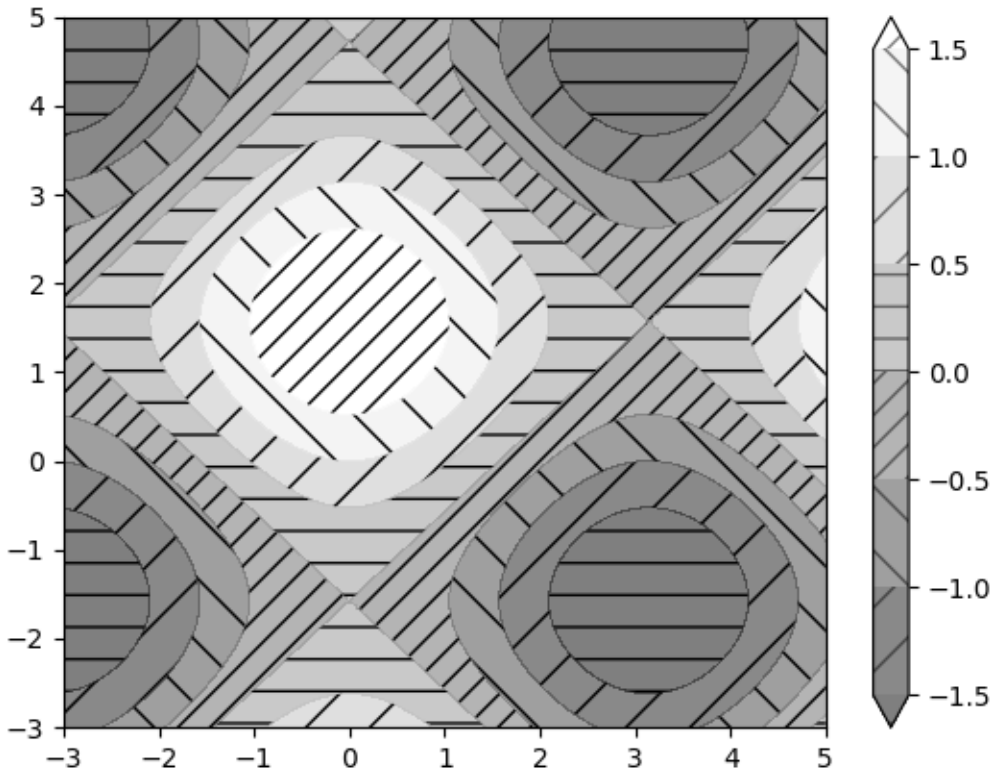
### Contourf Hatching

Demo filled contour plots with hatched patterns.

```
import matplotlib.pyplot as plt
import numpy as np

# invent some numbers, turning the x and y arrays into simple
# 2d arrays, which make combining them together easier.
x = np.linspace(-3, 5, 150).reshape(1, -1)
y = np.linspace(-3, 5, 120).reshape(-1, 1)
z = np.cos(x) + np.sin(y)

# we no longer need x and y to be 2 dimensional, so flatten them.
x, y = x.flatten(), y.flatten()
```

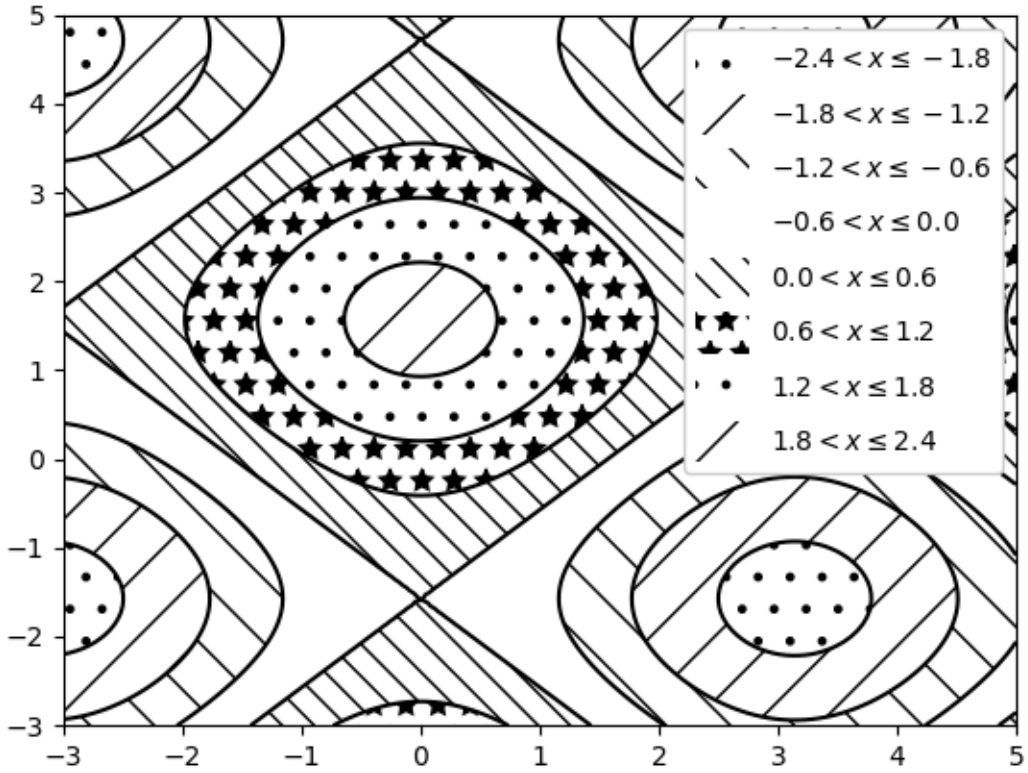
Plot 1: the simplest hatched plot with a colorbar

```
fig1, ax1 = plt.subplots()
cs = ax1.contourf(x, y, z, hatches=['-', '/', '\\', '//'],
                  cmap='gray', extend='both', alpha=0.5)
fig1.colorbar(cs)
```




Plot 2: a plot of hatches without color with a legend

```
fig2, ax2 = plt.subplots()
n_levels = 6
ax2.contour(x, y, z, n_levels, colors='black', linestyles='-')
cs = ax2.contourf(x, y, z, n_levels, colors='none',
                  hatches=['.', '/', '\\', None, '\\\\', '*'],
                  extend='lower')

# create a legend for the contour set
artists, labels = cs.legend_elements(str_format='{:2.1f}'.format)
ax2.legend(artists, labels, handleheight=2, framealpha=1)
plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

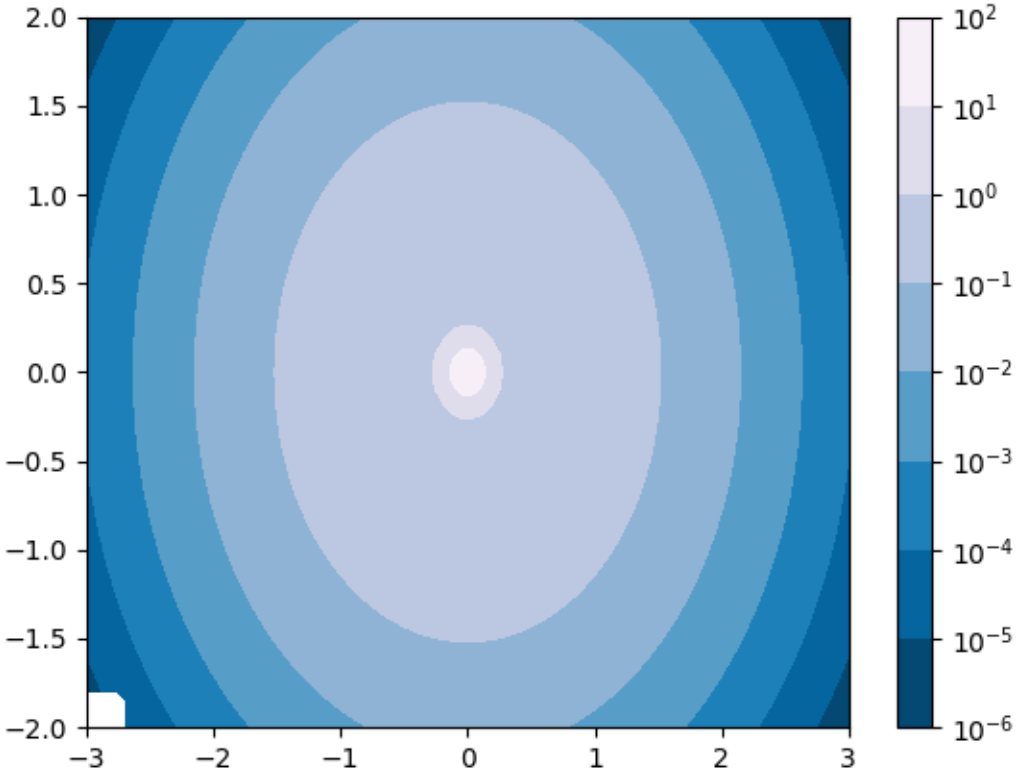
- *matplotlib.axes.Axes.contour* / *matplotlib.pyplot.contour*
- *matplotlib.axes.Axes.contourf* / *matplotlib.pyplot.contourf*
- *matplotlib.figure.Figure.colorbar* / *matplotlib.pyplot.colorbar*
- *matplotlib.axes.Axes.legend* / *matplotlib.pyplot.legend*
- *matplotlib.contour.ContourSet*
- *matplotlib.contour.ContourSet.legend_elements*

**Total running time of the script:** (0 minutes 1.594 seconds)

### Contourf and log color scale

Demonstrate use of a log color scale in contourf

```
import matplotlib.pyplot as plt
import numpy as np
from numpy import ma

from matplotlib import cm, ticker

N = 100
x = np.linspace(-3.0, 3.0, N)
y = np.linspace(-2.0, 2.0, N)

X, Y = np.meshgrid(x, y)

# A low hump with a spike coming out.
# Needs to have z/colour axis on a log scale, so we see both hump and spike.
# A linear scale only shows the spike.
Z1 = np.exp(-X**2 - Y**2)
Z2 = np.exp(-(X * 10)**2 - (Y * 10)**2)
z = Z1 + 50 * Z2

# Put in some negative values (lower left corner) to cause trouble with logs:
z[:5, :5] = -1

# The following is not strictly essential, but it will eliminate
# a warning.  Comment it out to see the warning.
z = ma.masked_where(z <= 0, z)


# Automatic selection of levels works; setting the
# log locator tells contourf to use a log scale:
fig, ax = plt.subplots()
cs = ax.contourf(X, Y, z, locator=ticker.LogLocator(), cmap=cm.PuBu_r)

# Alternatively, you can manually set the levels
# and the norm:
# lev_exp = np.arange(np.floor(np.log10(z.min())-1),
#                     np.ceil(np.log10(z.max())+1))
# levs = np.power(10, lev_exp)
# cs = ax.contourf(X, Y, z, levs, norm=colors.LogNorm())

cbar = fig.colorbar(cs)

plt.show()
```




**References**

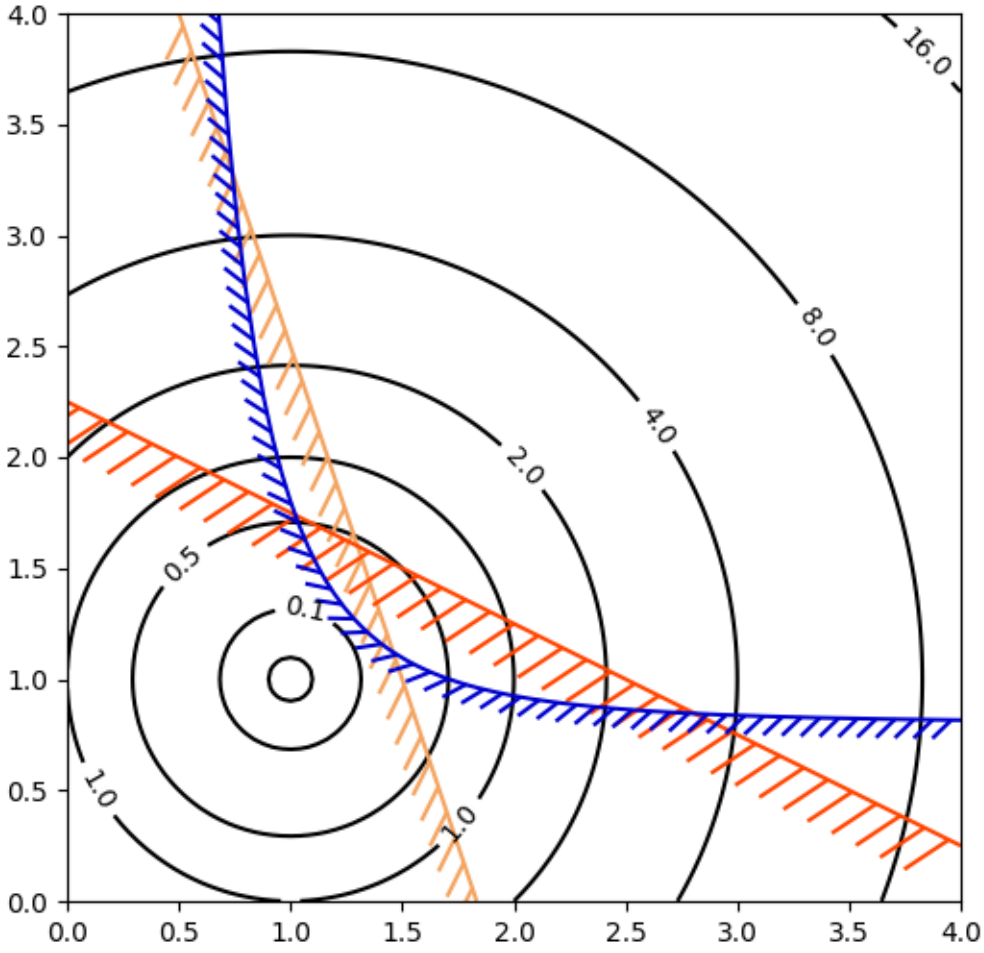
The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.contourf* / *matplotlib.pyplot.contourf*
- *matplotlib.figure.Figure.colorbar* / *matplotlib.pyplot.colorbar*
- *matplotlib.axes.Axes.legend* / *matplotlib.pyplot.legend*
- *matplotlib.ticker.LogLocator*

## Contouring the solution space of optimizations

Contour plotting is particularly handy when illustrating the solution space of optimization problems. Not only can *axes.Axes.contour* be used to represent the topography of the objective function, it can be used to generate boundary curves of the constraint functions. The constraint lines can be drawn with *TickedStroke* to distinguish the valid and invalid sides of the constraint boundaries.

*axes.Axes.contour* generates curves with larger values to the left of the contour. The angle parameter is measured zero ahead with increasing values to the left. Consequently, when using *TickedStroke* to illustrate a constraint in a typical optimization problem, the angle should be set between zero and 180 degrees.




```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib import patheffects

fig, ax = plt.subplots(figsize=(6, 6))

nx = 101
ny = 105

# Set up survey vectors
xvec = np.linspace(0.001, 4.0, nx)
yvec = np.linspace(0.001, 4.0, ny)

# Set up survey matrices.  Design disk loading and gear ratio.
x1, x2 = np.meshgrid(xvec, yvec)
```

(continues on next page)

(continued from previous page)

```
# Evaluate some stuff to plot
obj = x1**2 + x2**2 - 2*x1 - 2*x2 + 2
g1 = -(3*x1 + x2 - 5.5)
g2 = -(x1 + 2*x2 - 4.5)
g3 = 0.8 + x1**-3 - x2

cntr = ax.contour(x1, x2, obj, [0.01, 0.1, 0.5, 1, 2, 4, 8, 16],
                  colors='black')
ax.clabel(cntr, fmt="%2.1f", use_clabeltext=True)

cg1 = ax.contour(x1, x2, g1, [0], colors='sandybrown')
cg1.set(path_effects=[patheffects.withTickedStroke(angle=135)])

cg2 = ax.contour(x1, x2, g2, [0], colors='orangered')
cg2.set(path_effects=[patheffects.withTickedStroke(angle=60, length=2)])

cg3 = ax.contour(x1, x2, g3, [0], colors='mediumblue')
cg3.set(path_effects=[patheffects.withTickedStroke(spacing=7)])

ax.set_xlim(0, 4)
ax.set_ylim(0, 4)

plt.show()
```

### BboxImage Demo

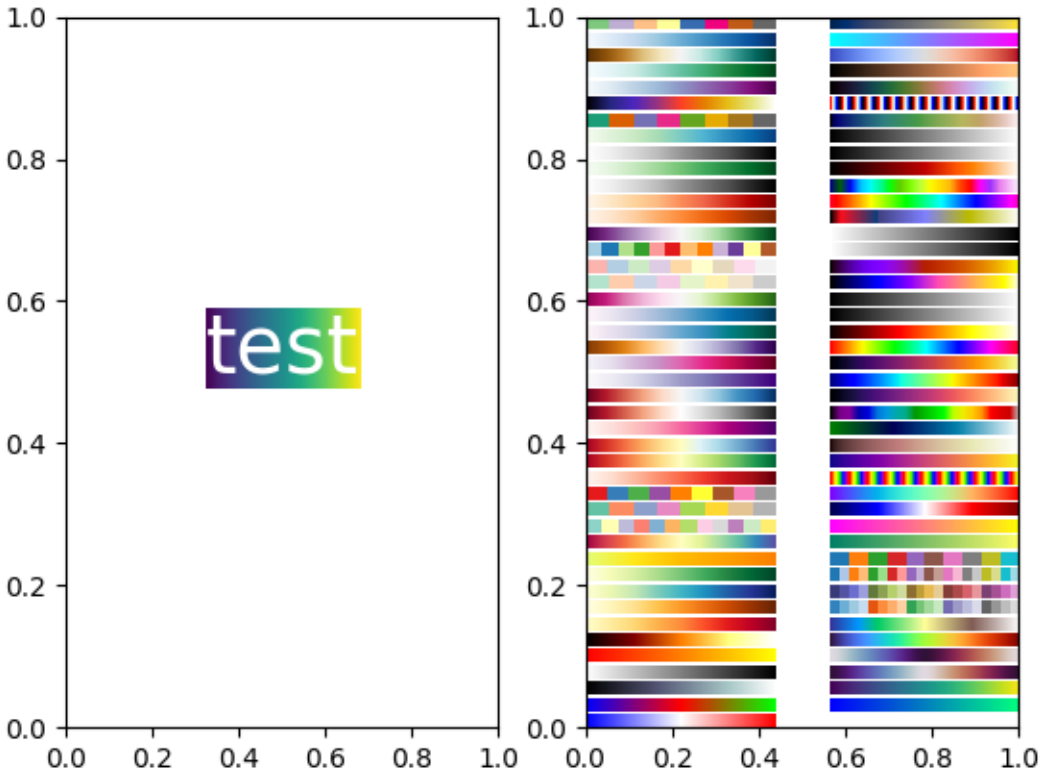
A *BboxImage* can be used to position an image according to a bounding box. This demo shows how to show an image inside a *text.Text*'s bounding box as well as how to manually create a bounding box for the image.

```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.image import BboxImage
from matplotlib.transforms import Bbox, TransformedBbox

fig, (ax1, ax2) = plt.subplots(ncols=2)

# ----------------------------
# Create a BboxImage with Text
# ----------------------------
txt = ax1.text(0.5, 0.5, "test", size=30, ha="center", color="w")
ax1.add_artist(
    BboxImage(txt.get_window_extent, data=np.arange(256).reshape((1, -1))))

# ------------------------------------
# Create a BboxImage for each colormap
# ------------------------------------
# List of all colormaps; skip reversed colormaps.
```

(continues on next page)

(continued from previous page)

```
cmap_names = sorted(m for m in plt.colormaps if not m.endswith("_r"))

ncol = 2
nrow = len(cmap_names) // ncol + 1

xpad_fraction = 0.3
dx = 1 / (ncol + xpad_fraction * (ncol - 1))

ypad_fraction = 0.3
dy = 1 / (nrow + ypad_fraction * (nrow - 1))

for i, cmap_name in enumerate(cmap_names):
    ix, iy = divmod(i, nrow)
    bbox0 = Bbox.from_bounds(ix*dx*(1+xpad_fraction),
                             1 - iy*dy*(1+ypad_fraction) - dy,
                             dx, dy)
    bbox = TransformedBbox(bbox0, ax2.transAxes)
    ax2.add_artist(
        BboxImage(bbox, cmap=cmap_name, data=np.arange(256).reshape((1, -1))))

plt.show()
```



**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.image.BboxImage*
- *matplotlib.transforms.Bbox*
- *matplotlib.transforms.TransformedBbox*
- *matplotlib.text.Text*

## Figimage Demo

This illustrates placing images directly in the figure, with no Axes objects.

```
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure()
Z = np.arange(10000).reshape((100, 100))
Z[:, 50:] = 1

im1 = fig.figimage(Z, xo=50, yo=0, origin='lower')
im2 = fig.figimage(Z, xo=100, yo=100, alpha=.8, origin='lower')

plt.show()
```

**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.figure.Figure*
- *matplotlib.figure.Figure.figimage* / *matplotlib.pyplot.figimage*

## Creating annotated heatmaps

It is often desirable to show data which depends on two independent variables as a color coded image plot. This is often referred to as a heatmap. If the data is categorical, this would be called a categorical heatmap.

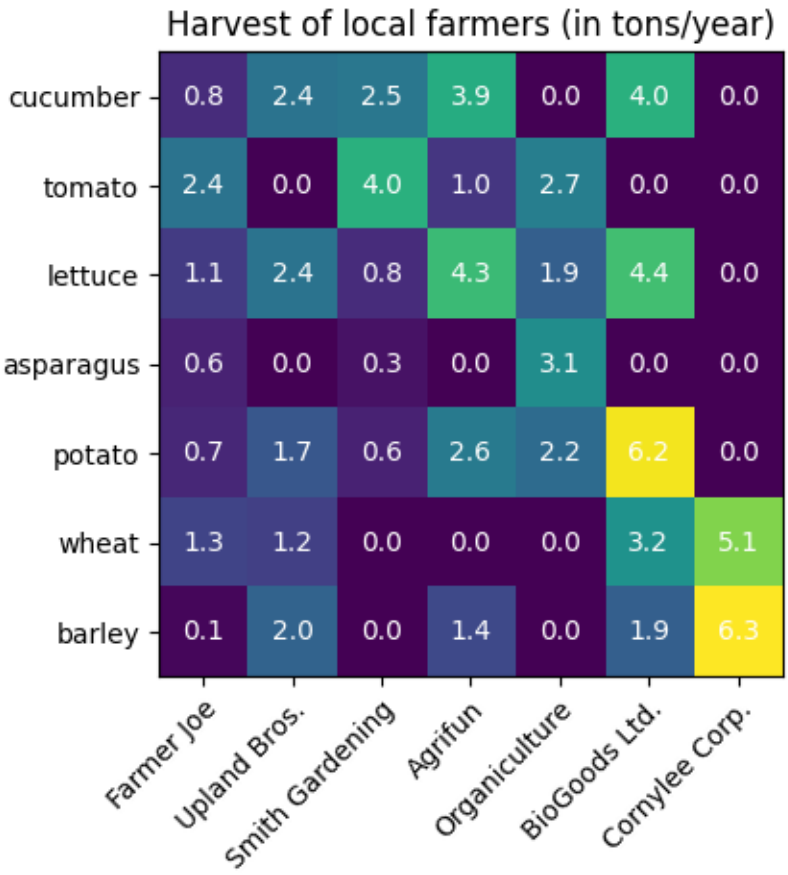
Matplotlib's *imshow* function makes production of such plots particularly easy.

The following examples show how to create a heatmap with annotations. We will start with an easy example and expand it to be usable as a universal function.

### A simple categorical heatmap

We may start by defining some data. What we need is a 2D list or array which defines the data to color code. We then also need two lists or arrays of categories; of course the number of elements in those lists need to match the data along the respective axes. The heatmap itself is an *imshow* plot with the labels set to the categories we have. Note that it is important to set both, the tick locations (*set_xticks*) as well as the tick labels (*set_xticklabels*), otherwise they would become out of sync. The locations are just the ascending integer numbers, while the ticklabels are the labels to show. Finally, we can label the data itself by creating a *Text* within each cell showing the value of that cell.

```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib
import matplotlib as mpl


vegetables = ["cucumber", "tomato", "lettuce", "asparagus",
              "potato", "wheat", "barley"]
farmers = ["Farmer Joe", "Upland Bros.", "Smith Gardening",
           "Agrifun", "Organiculture", "BioGoods Ltd.", "Cornylee Corp."]

harvest = np.array([[0.8, 2.4, 2.5, 3.9, 0.0, 4.0, 0.0],
                    [2.4, 0.0, 4.0, 1.0, 2.7, 0.0, 0.0],
                    [1.1, 2.4, 0.8, 4.3, 1.9, 4.4, 0.0],
                    [0.6, 0.0, 0.3, 0.0, 3.1, 0.0, 0.0],
                    [0.7, 1.7, 0.6, 2.6, 2.2, 6.2, 0.0],
                    [1.3, 1.2, 0.0, 0.0, 0.0, 3.2, 5.1],
                    [0.1, 2.0, 0.0, 1.4, 0.0, 1.9, 6.3]])


fig, ax = plt.subplots()
im = ax.imshow(harvest)

# Show all ticks and label them with the respective list entries
ax.set_xticks(np.arange(len(farmers)), labels=farmers)
ax.set_yticks(np.arange(len(vegetables)), labels=vegetables)

# Rotate the tick labels and set their alignment.
plt.setp(ax.get_xticklabels(), rotation=45, ha="right",
         rotation_mode="anchor")

# Loop over data dimensions and create text annotations.
for i in range(len(vegetables)):
    for j in range(len(farmers)):
        text = ax.text(j, i, harvest[i, j],
                       ha="center", va="center", color="w")

ax.set_title("Harvest of local farmers (in tons/year)")
fig.tight_layout()
plt.show()
```




### Using the helper function code style

As discussed in the *Coding styles* one might want to reuse such code to create some kind of heatmap for different input data and/or on different axes. We create a function that takes the data and the row and column labels as input, and allows arguments that are used to customize the plot

Here, in addition to the above we also want to create a colorbar and position the labels above of the heatmap instead of below it. The annotations shall get different colors depending on a threshold for better contrast against the pixel color. Finally, we turn the surrounding axes spines off and create a grid of white lines to separate the cells.

```
def heatmap(data, row_labels, col_labels, ax=None,
            cbar_kw=None, cbarlabel="", **kwargs):
    """
    Create a heatmap from a numpy array and two lists of labels.

    Parameters
    ----------
    data
        A 2D numpy array of shape (M, N).
    row_labels
```

(continues on next page)

(continued from previous page)

```
        A list or array of length M with the labels for the rows.
    col_labels
        A list or array of length N with the labels for the columns.
    ax
        A `matplotlib.axes.Axes` instance to which the heatmap is plotted.  If
        not provided, use current Axes or create a new one.  Optional.
    cbar_kw
        A dictionary with arguments to `matplotlib.Figure.colorbar`. ␣
↪Optional.
    cbarlabel
        The label for the colorbar.  Optional.
    **kwargs
        All other arguments are forwarded to `imshow`.
    """

    if ax is None:
        ax = plt.gca()

    if cbar_kw is None:
        cbar_kw = {}

    # Plot the heatmap
    im = ax.imshow(data, **kwargs)

    # Create colorbar
    cbar = ax.figure.colorbar(im, ax=ax, **cbar_kw)
    cbar.ax.set_ylabel(cbarlabel, rotation=-90, va="bottom")

    # Show all ticks and label them with the respective list entries.
    ax.set_xticks(np.arange(data.shape[1]), labels=col_labels)
    ax.set_yticks(np.arange(data.shape[0]), labels=row_labels)

    # Let the horizontal axes labeling appear on top.
    ax.tick_params(top=True, bottom=False,
                   labeltop=True, labelbottom=False)

    # Rotate the tick labels and set their alignment.
    plt.setp(ax.get_xticklabels(), rotation=-30, ha="right",
             rotation_mode="anchor")

    # Turn spines off and create white grid.
    ax.spines[:].set_visible(False)

    ax.set_xticks(np.arange(data.shape[1]+1)-.5, minor=True)
    ax.set_yticks(np.arange(data.shape[0]+1)-.5, minor=True)
    ax.grid(which="minor", color="w", linestyle='-', linewidth=3)
    ax.tick_params(which="minor", bottom=False, left=False)

    return im, cbar


def annotate_heatmap(im, data=None, valfmt="{x:.2f}",
```

(continues on next page)

(continued from previous page)

```
                    textcolors=("black", "white"),
                    threshold=None, **textkw):
    """
    A function to annotate a heatmap.

    Parameters
    ----------
    im
        The AxesImage to be labeled.
    data
        Data used to annotate.  If None, the image's data is used.  Optional.
    valfmt
        The format of the annotations inside the heatmap.  This should either
        use the string format method, e.g. "$ {x:.2f}", or be a
        `matplotlib.ticker.Formatter`.  Optional.
    textcolors
        A pair of colors.  The first is used for values below a threshold,
        the second for those above.  Optional.
    threshold
        Value in data units according to which the colors from textcolors are
        applied.  If None (the default) uses the middle of the colormap as
        separation.  Optional.
    **kwargs
        All other arguments are forwarded to each call to `text` used to␣
↪create
        the text labels.
    """

    if not isinstance(data, (list, np.ndarray)):
        data = im.get_array()

    # Normalize the threshold to the images color range.
    if threshold is not None:
        threshold = im.norm(threshold)
    else:
        threshold = im.norm(data.max())/2.

    # Set default alignment to center, but allow it to be
    # overwritten by textkw.
    kw = dict(horizontalalignment="center",
              verticalalignment="center")
    kw.update(textkw)

    # Get the formatter in case a string is supplied
    if isinstance(valfmt, str):
        valfmt = matplotlib.ticker.StrMethodFormatter(valfmt)

    # Loop over the data and create a `Text` for each "pixel".
    # Change the text's color depending on the data.
    texts = []
    for i in range(data.shape[0]):
        for j in range(data.shape[1]):
```

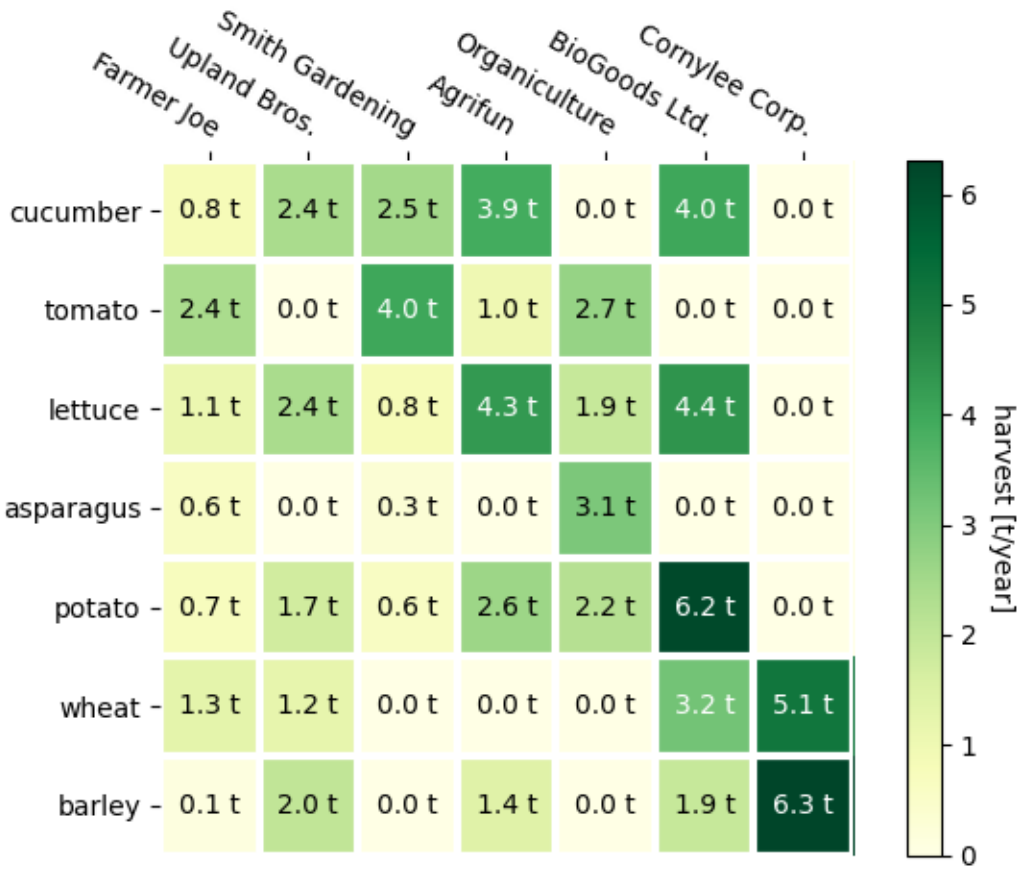
(continues on next page)

(continued from previous page)

```
            kw.update(color=textcolors[int(im.norm(data[i, j]) > threshold)])
            text = im.axes.text(j, i, valfmt(data[i, j], None), **kw)
            texts.append(text)

    return texts
```

The above now allows us to keep the actual plot creation pretty compact.

```
fig, ax = plt.subplots()

im, cbar = heatmap(harvest, vegetables, farmers, ax=ax,
                   cmap="YlGn", cbarlabel="harvest [t/year]")
texts = annotate_heatmap(im, valfmt="{x:.1f} t")

fig.tight_layout()
plt.show()
```



### Some more complex heatmap examples

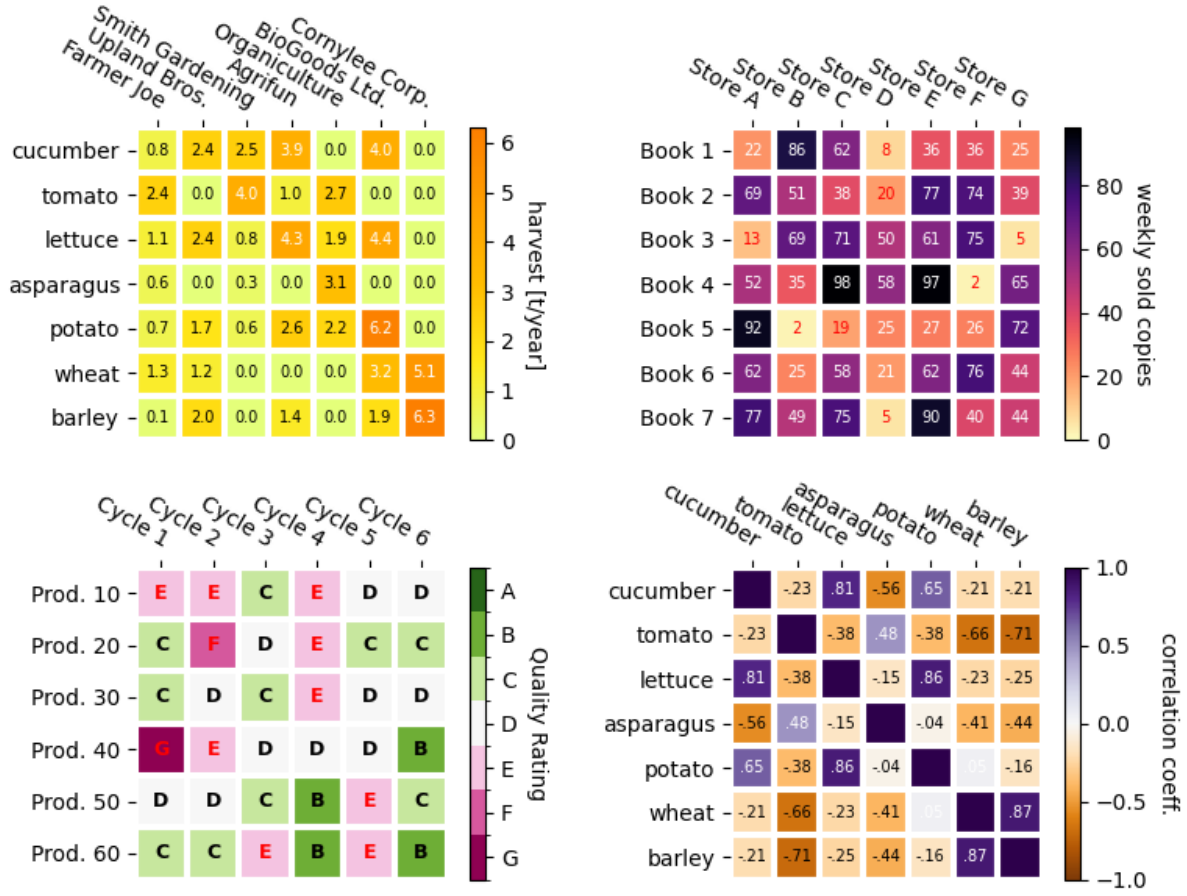
In the following we show the versatility of the previously created functions by applying it in different cases and using different arguments.

```
np.random.seed(19680801)

fig, ((ax, ax2), (ax3, ax4)) = plt.subplots(2, 2, figsize=(8, 6))

# Replicate the above example with a different font size and colormap.

im, _ = heatmap(harvest, vegetables, farmers, ax=ax,
                cmap="Wistia", cbarlabel="harvest [t/year]")
annotate_heatmap(im, valfmt="{x:.1f}", size=7)

# Create some new data, give further arguments to imshow (vmin),
# use an integer format on the annotations and provide some colors.

data = np.random.randint(2, 100, size=(7, 7))
y = [f"Book {i}" for i in range(1, 8)]
x = [f"Store {i}" for i in list("ABCDEFG")]
im, _ = heatmap(data, y, x, ax=ax2, vmin=0,
                cmap="magma_r", cbarlabel="weekly sold copies")
annotate_heatmap(im, valfmt="{x:d}", size=7, threshold=20,
                 textcolors=("red", "white"))

# Sometimes even the data itself is categorical. Here we use a
# `matplotlib.colors.BoundaryNorm` to get the data into classes
# and use this to colorize the plot, but also to obtain the class
# labels from an array of classes.

data = np.random.randn(6, 6)
y = [f"Prod. {i}" for i in range(10, 70, 10)]
x = [f"Cycle {i}" for i in range(1, 7)]

qrates = list("ABCDEFG")
norm = matplotlib.colors.BoundaryNorm(np.linspace(-3.5, 3.5, 8), 7)
fmt = matplotlib.ticker.FuncFormatter(lambda x, pos: qrates[::-1][norm(x)])

im, _ = heatmap(data, y, x, ax=ax3,
                cmap=mpl.colormaps["PiYG"].resampled(7), norm=norm,
                cbar_kw=dict(ticks=np.arange(-3, 4), format=fmt),
                cbarlabel="Quality Rating")

annotate_heatmap(im, valfmt=fmt, size=9, fontweight="bold", threshold=-1,
                 textcolors=("red", "black"))

# We can nicely plot a correlation matrix. Since this is bound by -1 and 1,
# we use those as vmin and vmax. We may also remove leading zeros and hide
# the diagonal elements (which are all 1) by using a
# `matplotlib.ticker.FuncFormatter`.
```

(continues on next page)

(continued from previous page)

```
corr_matrix = np.corrcoef(harvest)
im, _ = heatmap(corr_matrix, vegetables, vegetables, ax=ax4,
                cmap="PuOr", vmin=-1, vmax=1,
                cbarlabel="correlation coeff.")


def func(x, pos):
    return f"{x:.2f}".replace("0.", ".").replace("1.00", "")

annotate_heatmap(im, valfmt=matplotlib.ticker.FuncFormatter(func), size=7)


plt.tight_layout()
plt.show()
```




## References

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.imshow* / *matplotlib.pyplot.imshow*
- *matplotlib.figure.Figure.colorbar* / *matplotlib.pyplot.colorbar*

**Total running time of the script:** (0 minutes 2.384 seconds)

### Image antialiasing

Images are represented by discrete pixels, either on the screen or in an image file. When data that makes up the image has a different resolution than its representation on the screen we will see aliasing effects. How noticeable these are depends on how much down-sampling takes place in the change of resolution (if any).

When subsampling data, aliasing is reduced by smoothing first and then subsampling the smoothed data. In Matplotlib, we can do that smoothing before mapping the data to colors, or we can do the smoothing on the RGB(A) data in the final image. The differences between these are shown below, and controlled with the *interpolation_stage* keyword argument.

The default image interpolation in Matplotlib is 'antialiased', and it is applied to the data. This uses a hanning interpolation on the data provided by the user for reduced aliasing in most situations. Only when there is upsampling by a factor of 1, 2 or >=3 is 'nearest' neighbor interpolation used.

Other anti-aliasing filters can be specified in `Axes.imshow` using the *interpolation* keyword argument.

```
import matplotlib.pyplot as plt
import numpy as np
```

First we generate a 450x450 pixel image with varying frequency content:

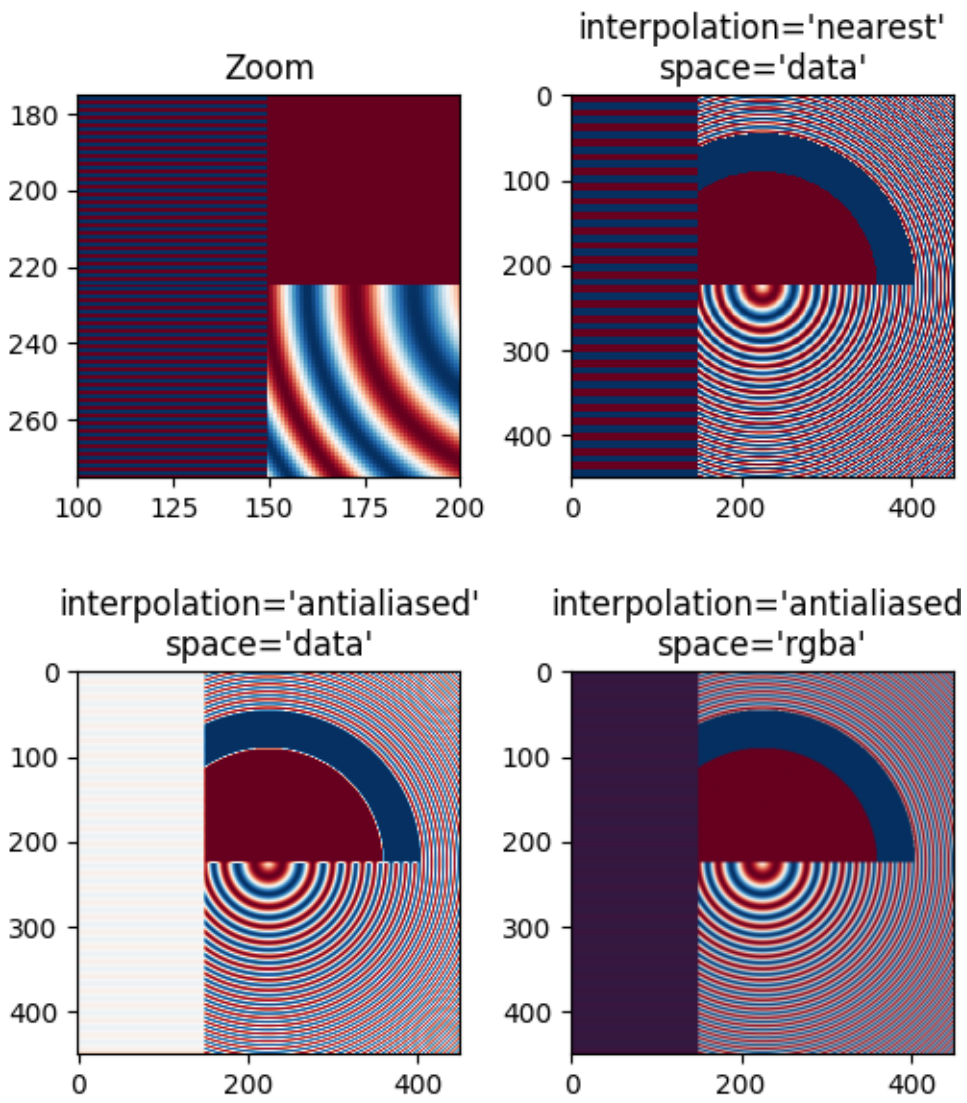
```
N = 450
x = np.arange(N) / N - 0.5
y = np.arange(N) / N - 0.5
aa = np.ones((N, N))
aa[::2, :] = -1

X, Y = np.meshgrid(x, y)
R = np.sqrt(X**2 + Y**2)
f0 = 5
k = 100
a = np.sin(np.pi * 2 * (f0 * R + k * R**2 / 2))
# make the left hand side of this
a[:int(N / 2), :][R[:int(N / 2), :] < 0.4] = -1
a[:int(N / 2), :][R[:int(N / 2), :] < 0.3] = 1
aa[:, int(N / 3):] = a[:, int(N / 3):]
a = aa
```

The following images are subsampled from 450 data pixels to either 125 pixels or 250 pixels (depending on your display). The Moiré patterns in the 'nearest' interpolation are caused by the high-frequency data being subsampled. The 'antialiased' imaged still has some Moiré patterns as well, but they are greatly reduced.

There are substantial differences between the 'data' interpolation and the 'rgba' interpolation. The alternating bands of red and blue on the left third of the image are subsampled. By interpolating in 'data' space (the default) the antialiasing filter makes the stripes close to white, because the average of -1 and +1 is zero, and zero is white in this colormap.

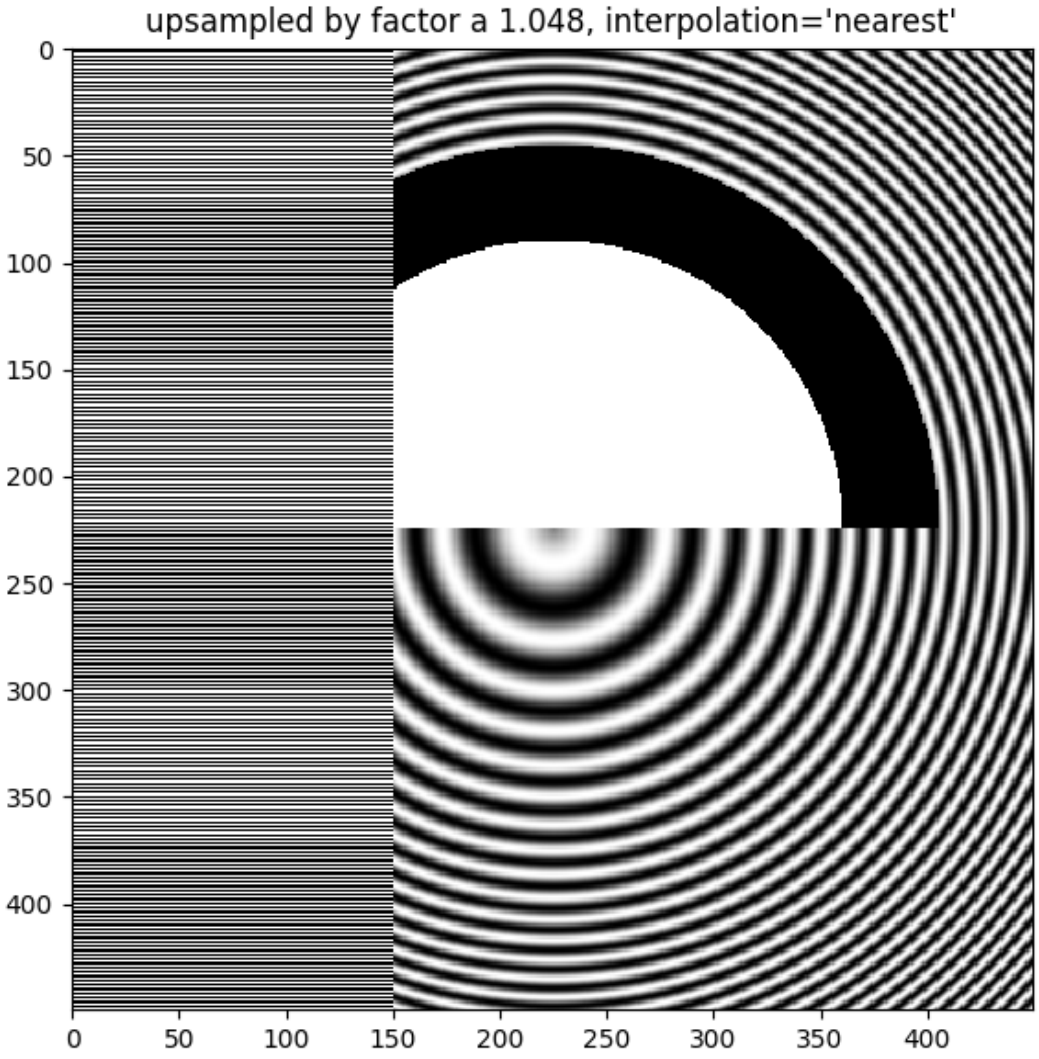
Conversely, when the anti-aliasing occurs in 'rgba' space, the red and blue are combined visually to make purple. This behaviour is more like a typical image processing package, but note that purple is not in the original colormap, so it is no longer possible to invert individual pixels back to their data value.

```
fig, axs = plt.subplots(2, 2, figsize=(5, 6), layout='constrained')
axs[0, 0].imshow(a, interpolation='nearest', cmap='RdBu_r')
axs[0, 0].set_xlim(100, 200)
axs[0, 0].set_ylim(275, 175)
axs[0, 0].set_title('Zoom')

for ax, interp, space in zip(axs.flat[1:],
                             ['nearest', 'antialiased', 'antialiased'],
                             ['data', 'data', 'rgba']):
    ax.imshow(a, interpolation=interp, interpolation_stage=space,
              cmap='RdBu_r')
    ax.set_title(f"interpolation='{interp}'\nspace='{space}'")
plt.show()
```



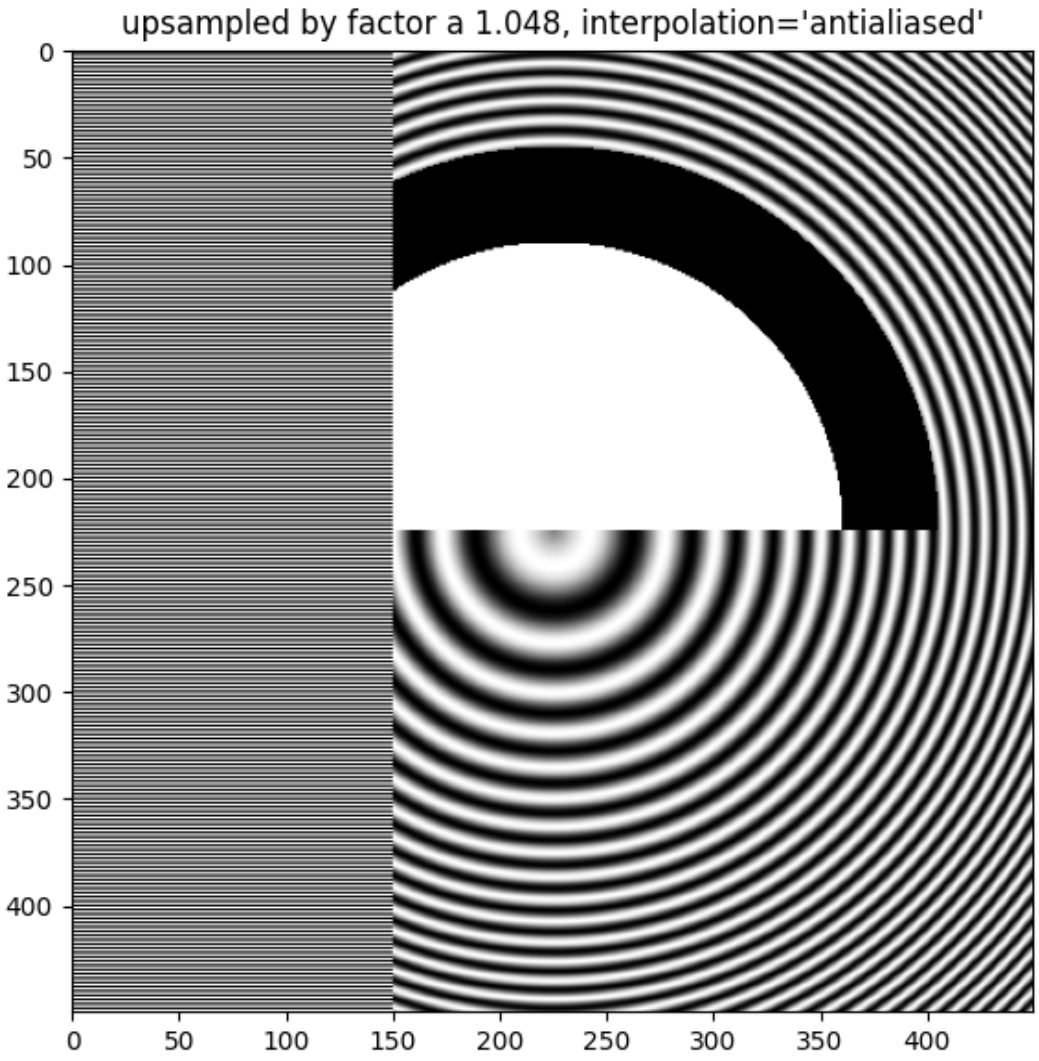

Even up-sampling an image with 'nearest' interpolation will lead to Moiré patterns when the upsampling factor is not integer. The following image upsamples 500 data pixels to 530 rendered pixels. You may note a grid of 30 line-like artifacts which stem from the 524 - 500 = 24 extra pixels that had to be made up. Since interpolation is 'nearest' they are the same as a neighboring line of pixels and thus stretch the image locally so that it looks distorted.

```
fig, ax = plt.subplots(figsize=(6.8, 6.8))
ax.imshow(a, interpolation='nearest', cmap='gray')
ax.set_title("upsampled by factor a 1.048, interpolation='nearest'")
plt.show()
```



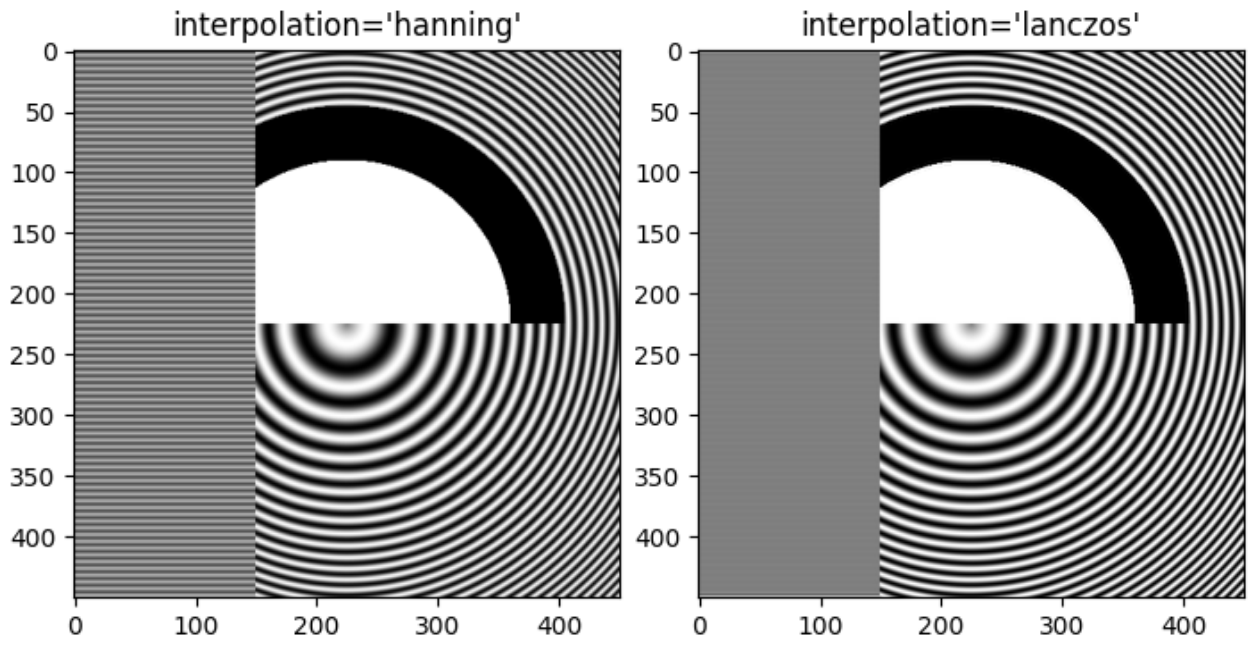

Better antialiasing algorithms can reduce this effect:

```
fig, ax = plt.subplots(figsize=(6.8, 6.8))
ax.imshow(a, interpolation='antialiased', cmap='gray')
ax.set_title("upsampled by factor a 1.048, interpolation='antialiased'")
plt.show()
```




Apart from the default 'hanning' antialiasing, *imshow* supports a number of different interpolation algorithms, which may work better or worse depending on the pattern.

```
fig, axs = plt.subplots(1, 2, figsize=(7, 4), layout='constrained')
for ax, interp in zip(axs, ['hanning', 'lanczos']):
    ax.imshow(a, interpolation=interp, cmap='gray')
    ax.set_title(f"interpolation='{interp}'")
plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.imshow*

**Total running time of the script:** (0 minutes 2.001 seconds)

## Clipping images with patches

Demo of image that's been clipped by a circular patch.

```
import matplotlib.pyplot as plt

import matplotlib.cbook as cbook
import matplotlib.patches as patches

with cbook.get_sample_data('grace_hopper.jpg') as image_file:
    image = plt.imread(image_file)

fig, ax = plt.subplots()
im = ax.imshow(image)
patch = patches.Circle((260, 200), radius=200, transform=ax.transData)
im.set_clip_path(patch)

ax.axis('off')
plt.show()
```

**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.imshow* / *matplotlib.pyplot.imshow*
- *matplotlib.artist.Artist.set_clip_path*

## Many ways to plot images

The most common way to plot images in Matplotlib is with *imshow*. The following examples demonstrate much of the functionality of imshow and the many images you can create.

```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib.cbook as cbook
import matplotlib.cm as cm
from matplotlib.patches import PathPatch
from matplotlib.path import Path
```

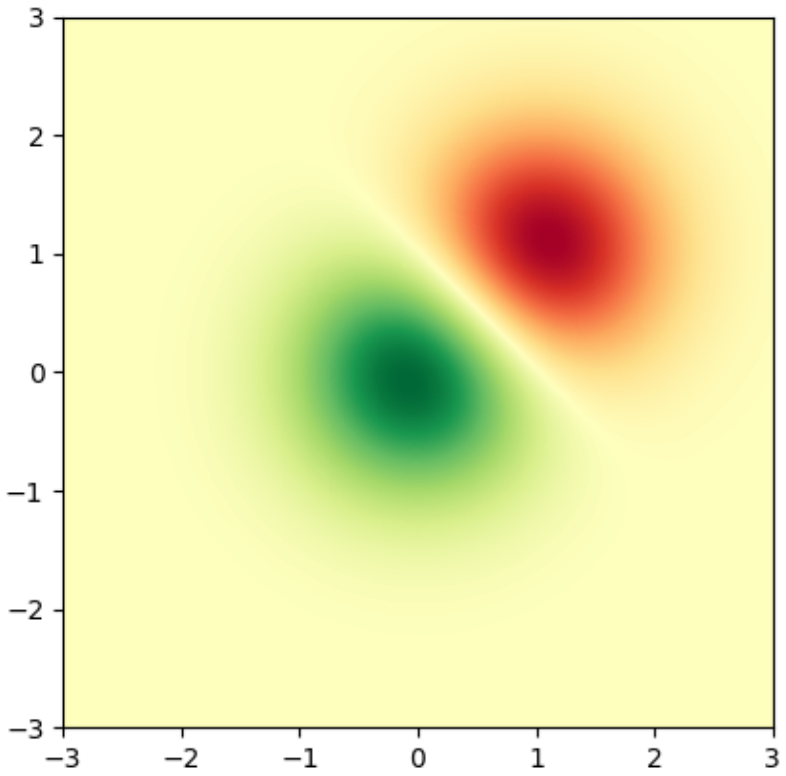
(continues on next page)

(continued from previous page)

```
# Fixing random state for reproducibility
np.random.seed(19680801)
```

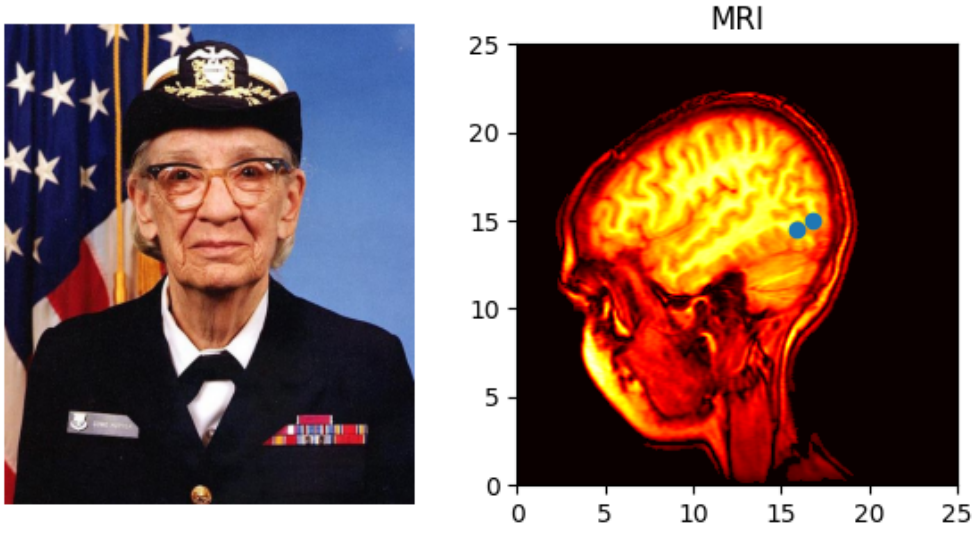
First we'll generate a simple bivariate normal distribution.

```
delta = 0.025
x = y = np.arange(-3.0, 3.0, delta)
X, Y = np.meshgrid(x, y)
Z1 = np.exp(-X**2 - Y**2)
Z2 = np.exp(-(X - 1)**2 - (Y - 1)**2)
Z = (Z1 - Z2) * 2

fig, ax = plt.subplots()
im = ax.imshow(Z, interpolation='bilinear', cmap=cm.RdYlGn,
               origin='lower', extent=[-3, 3, -3, 3],
               vmax=abs(Z).max(), vmin=-abs(Z).max())

plt.show()
```




It is also possible to show images of pictures.

```
# A sample image
with cbook.get_sample_data('grace_hopper.jpg') as image_file:
    image = plt.imread(image_file)

# And another image, using 256x256 16-bit integers.
w, h = 256, 256
with cbook.get_sample_data('s1045.ima.gz') as datafile:
    s = datafile.read()
A = np.frombuffer(s, np.uint16).astype(float).reshape((w, h))
extent = (0, 25, 0, 25)

fig, ax = plt.subplot_mosaic([
    ['hopper', 'mri']
], figsize=(7, 3.5))

ax['hopper'].imshow(image)
ax['hopper'].axis('off')  # clear x-axis and y-axis

im = ax['mri'].imshow(A, cmap=plt.cm.hot, origin='upper', extent=extent)

markers = [(15.9, 14.5), (16.8, 15)]
x, y = zip(*markers)
ax['mri'].plot(x, y, 'o')

ax['mri'].set_title('MRI')

plt.show()
```



### Interpolating images

It is also possible to interpolate images before displaying them. Be careful, as this may manipulate the way your data looks, but it can be helpful for achieving the look you want. Below we'll display the same (small) array, interpolated with three different interpolation methods.

The center of the pixel at A[i, j] is plotted at (i+0.5, i+0.5). If you are using interpolation='nearest', the region bounded by (i, j) and (i+1, j+1) will have the same color. If you are using interpolation, the pixel center will have the same color as it does with nearest, but other pixels will be interpolated between the neighboring pixels.

To prevent edge effects when doing interpolation, Matplotlib pads the input array with identical pixels around the edge: if you have a 5x5 array with colors a-y as below:

```
a b c d e
f g h i j
k l m n o
p q r s t
u v w x y
```

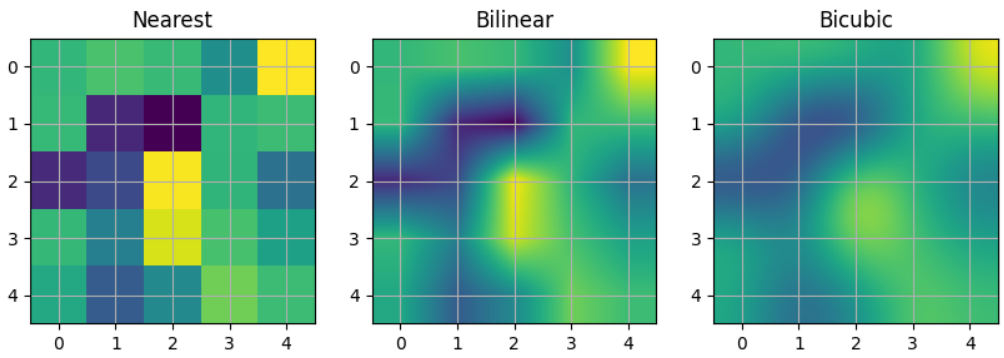
Matplotlib computes the interpolation and resizing on the padded array

```
a a b c d e e
a a b c d e e
f f g h i j j
k k l m n o o
p p q r s t t
o u v w x y y
o u v w x y y
```

and then extracts the central region of the result. (Extremely old versions of Matplotlib (<0.63) did not pad the array, but instead adjusted the view limits to hide the affected edge areas.)

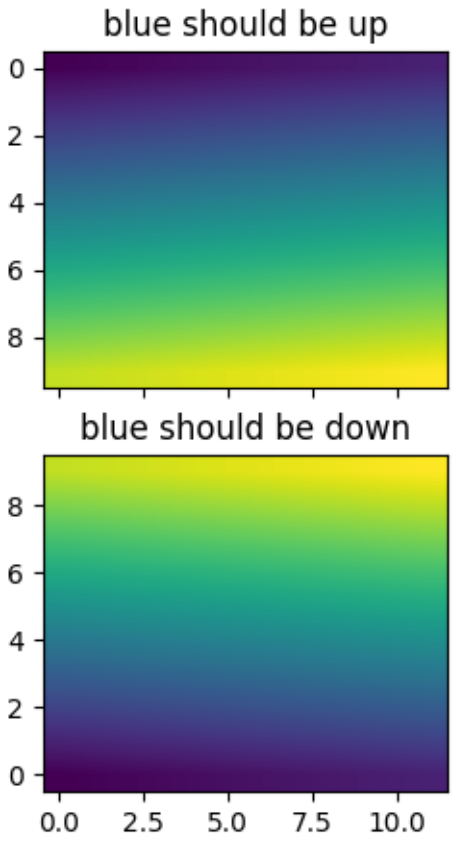
This approach allows plotting the full extent of an array without edge effects, and for example to layer multiple images of different sizes over one another with different interpolation methods -- see *Layer Images*. It also implies a performance hit, as this new temporary, padded array must be created. Sophisticated interpolation also implies a performance hit; for maximal performance or very large images, interpolation='nearest' is suggested.

```
A = np.random.rand(5, 5)

fig, axs = plt.subplots(1, 3, figsize=(10, 3))
for ax, interp in zip(axs, ['nearest', 'bilinear', 'bicubic']):
    ax.imshow(A, interpolation=interp)
    ax.set_title(interp.capitalize())
    ax.grid(True)

plt.show()
```



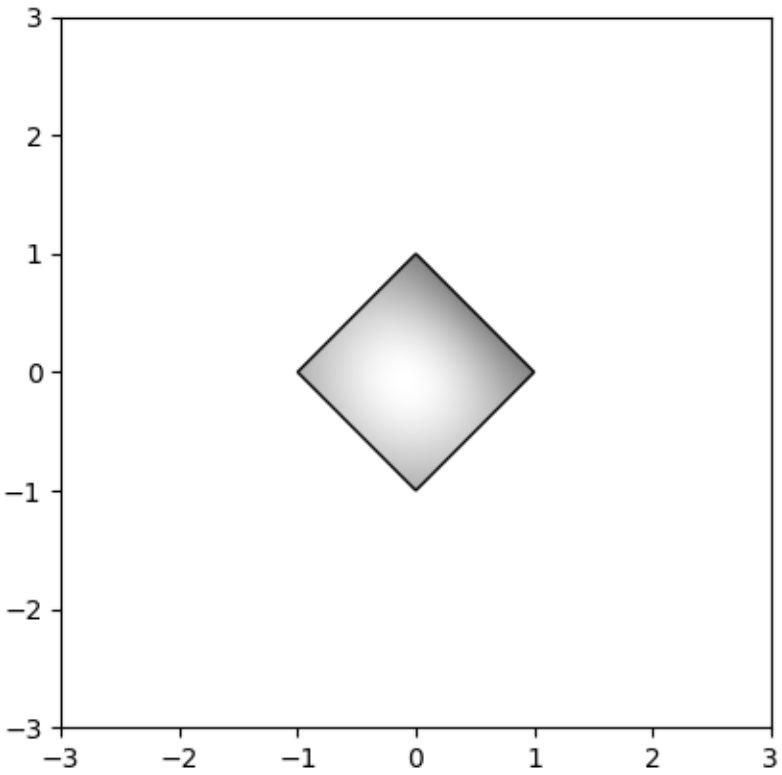

You can specify whether images should be plotted with the array origin x[0, 0] in the upper left or lower right by using the origin parameter. You can also control the default setting image.origin in your *matplotlibrc file*. For more on this topic see the *complete guide on origin and extent*.

```
x = np.arange(120).reshape((10, 12))

interp = 'bilinear'
fig, axs = plt.subplots(nrows=2, sharex=True, figsize=(3, 5))
axs[0].set_title('blue should be up')
axs[0].imshow(x, origin='upper', interpolation=interp)

axs[1].set_title('blue should be down')
axs[1].imshow(x, origin='lower', interpolation=interp)
plt.show()
```




Finally, we'll show an image using a clip path.

```
delta = 0.025
x = y = np.arange(-3.0, 3.0, delta)
X, Y = np.meshgrid(x, y)
Z1 = np.exp(-X**2 - Y**2)
Z2 = np.exp(-(X - 1)**2 - (Y - 1)**2)
Z = (Z1 - Z2) * 2

path = Path([[0, 1], [1, 0], [0, -1], [-1, 0], [0, 1]])
patch = PathPatch(path, facecolor='none')

fig, ax = plt.subplots()
ax.add_patch(patch)

im = ax.imshow(Z, interpolation='bilinear', cmap=cm.gray,
               origin='lower', extent=[-3, 3, -3, 3],
               clip_path=patch, clip_on=True)
im.set_clip_path(patch)

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.imshow* / *matplotlib.pyplot.imshow*
- *matplotlib.artist.Artist.set_clip_path*
- *matplotlib.patches.PathPatch*

**Total running time of the script:** (0 minutes 1.862 seconds)

### Image Masked

imshow with masked array input and out-of-range colors.

The second subplot illustrates the use of BoundaryNorm to get a filled contour effect.

```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib.colors as colors
```

(continues on next page)

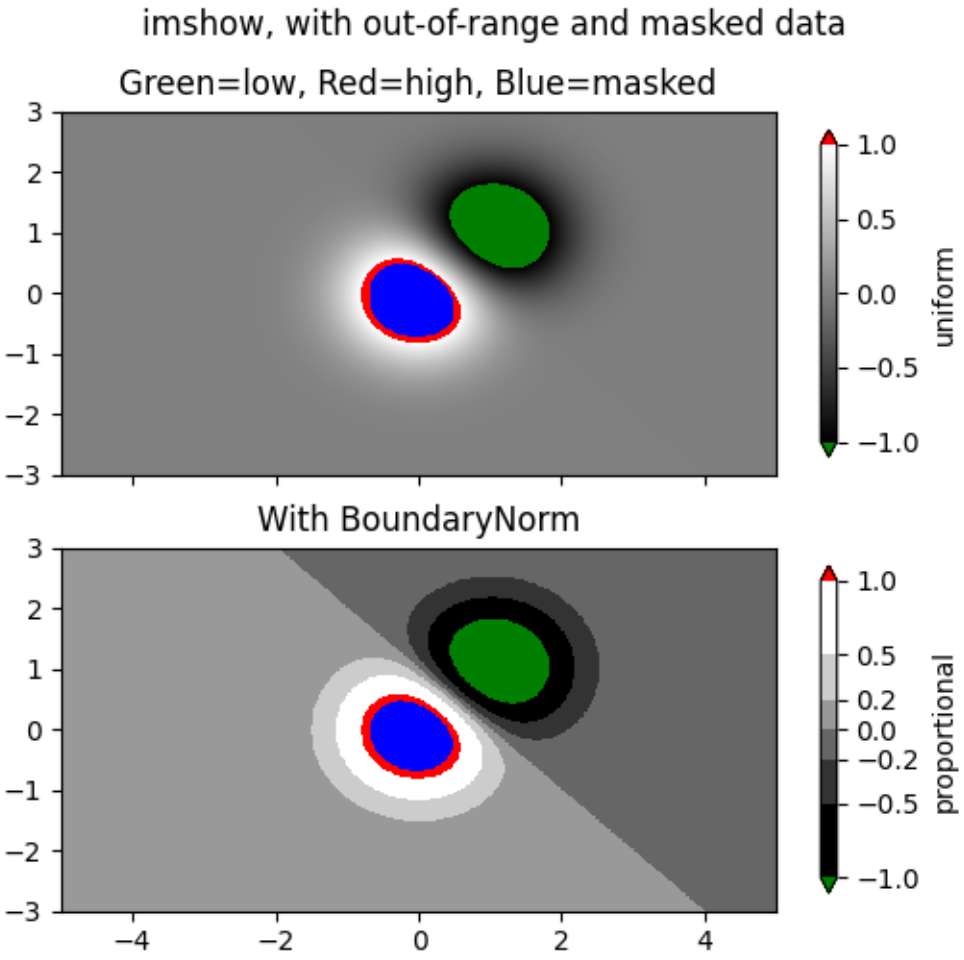
(continued from previous page)

```
# compute some interesting data
x0, x1 = -5, 5
y0, y1 = -3, 3
x = np.linspace(x0, x1, 500)
y = np.linspace(y0, y1, 500)
X, Y = np.meshgrid(x, y)
Z1 = np.exp(-X**2 - Y**2)
Z2 = np.exp(-(X - 1)**2 - (Y - 1)**2)
Z = (Z1 - Z2) * 2

# Set up a colormap:
palette = plt.cm.gray.with_extremes(over='r', under='g', bad='b')
# Alternatively, we could use
# palette.set_bad(alpha = 0.0)
# to make the bad region transparent.  This is the default.
# If you comment out all the palette.set* lines, you will see
# all the defaults; under and over will be colored with the
# first and last colors in the palette, respectively.
Zm = np.ma.masked_where(Z > 1.2, Z)

# By setting vmin and vmax in the norm, we establish the
# range to which the regular palette color scale is applied.
# Anything above that range is colored based on palette.set_over, etc.

# set up the Axes objects
fig, (ax1, ax2) = plt.subplots(nrows=2, figsize=(6, 5.4))

# plot using 'continuous' colormap
im = ax1.imshow(Zm, interpolation='bilinear',
                cmap=palette,
                norm=colors.Normalize(vmin=-1.0, vmax=1.0),
                aspect='auto',
                origin='lower',
                extent=[x0, x1, y0, y1])
ax1.set_title('Green=low, Red=high, Blue=masked')
cbar = fig.colorbar(im, extend='both', shrink=0.9, ax=ax1)
cbar.set_label('uniform')
ax1.tick_params(axis='x', labelbottom=False)

# Plot using a small number of colors, with unevenly spaced boundaries.
im = ax2.imshow(Zm, interpolation='nearest',
                cmap=palette,
                norm=colors.BoundaryNorm([-1, -0.5, -0.2, 0, 0.2, 0.5, 1],
                                         ncolors=palette.N),
                aspect='auto',
                origin='lower',
                extent=[x0, x1, y0, y1])
ax2.set_title('With BoundaryNorm')
cbar = fig.colorbar(im, extend='both', spacing='proportional',
                    shrink=0.9, ax=ax2)
cbar.set_label('proportional')
```

(continues on next page)

(continued from previous page)

```
fig.suptitle('imshow, with out-of-range and masked data')
plt.show()
```



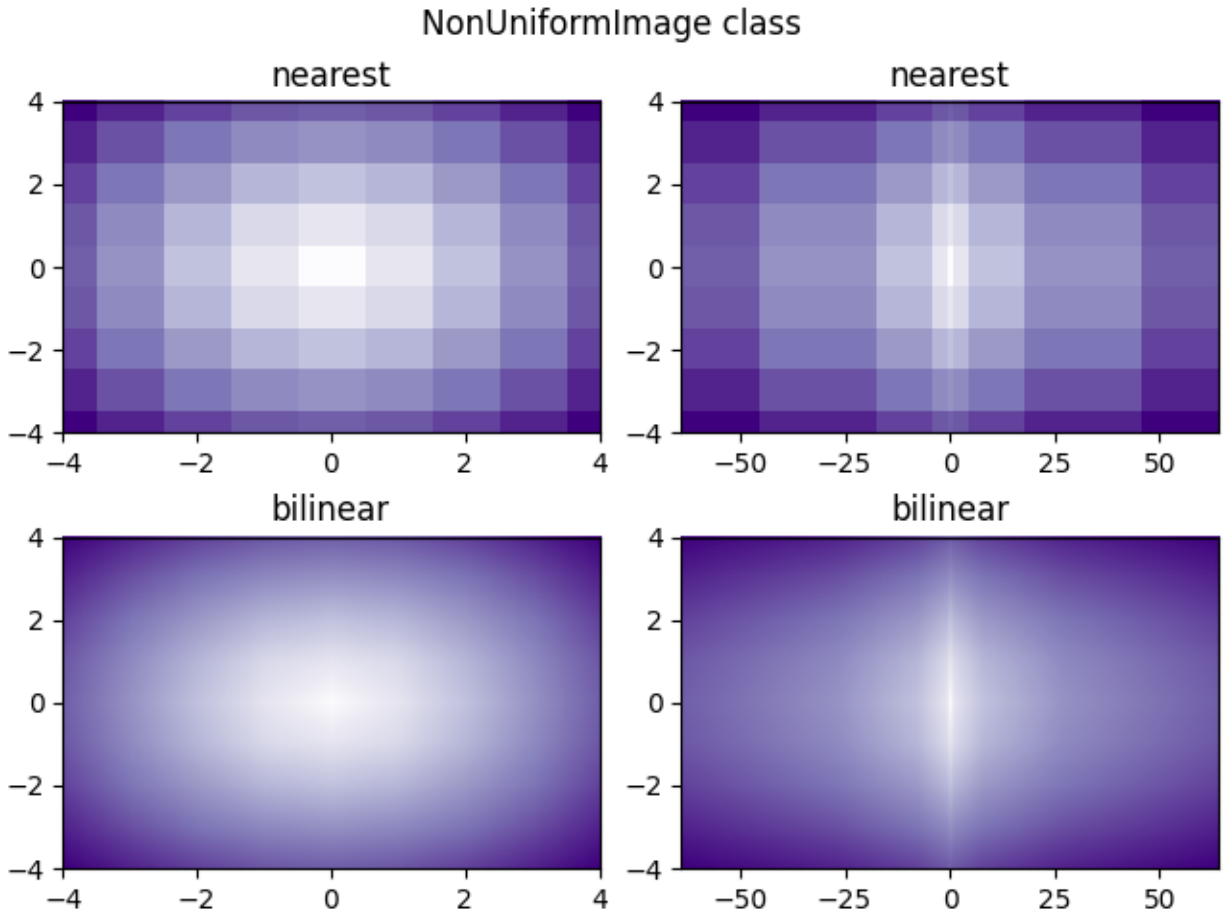

**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.imshow* / *matplotlib.pyplot.imshow*
- *matplotlib.figure.Figure.colorbar* / *matplotlib.pyplot.colorbar*
- *matplotlib.colors.BoundaryNorm*
- *matplotlib.colorbar.Colorbar.set_label*

### Image nonuniform

This illustrates the NonUniformImage class. It is not available via an Axes method, but it is easily added to an Axes instance as shown here.




```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib import cm
from matplotlib.image import NonUniformImage

interp = 'nearest'

# Linear x array for cell centers:
x = np.linspace(-4, 4, 9)

# Highly nonlinear x array:
x2 = x**3

y = np.linspace(-4, 4, 9)

z = np.sqrt(x[np.newaxis, :]**2 + y[:, np.newaxis]**2)
```

(continues on next page)

(continued from previous page)

```
fig, axs = plt.subplots(nrows=2, ncols=2, layout='constrained')
fig.suptitle('NonUniformImage class', fontsize='large')
ax = axs[0, 0]
im = NonUniformImage(ax, interpolation=interp, extent=(-4, 4, -4, 4),
                     cmap=cm.Purples)
im.set_data(x, y, z)
ax.add_image(im)
ax.set_xlim(-4, 4)
ax.set_ylim(-4, 4)
ax.set_title(interp)

ax = axs[0, 1]
im = NonUniformImage(ax, interpolation=interp, extent=(-64, 64, -4, 4),
                     cmap=cm.Purples)
im.set_data(x2, y, z)
ax.add_image(im)
ax.set_xlim(-64, 64)
ax.set_ylim(-4, 4)
ax.set_title(interp)

interp = 'bilinear'

ax = axs[1, 0]
im = NonUniformImage(ax, interpolation=interp, extent=(-4, 4, -4, 4),
                     cmap=cm.Purples)
im.set_data(x, y, z)
ax.add_image(im)
ax.set_xlim(-4, 4)
ax.set_ylim(-4, 4)
ax.set_title(interp)

ax = axs[1, 1]
im = NonUniformImage(ax, interpolation=interp, extent=(-64, 64, -4, 4),
                     cmap=cm.Purples)
im.set_data(x2, y, z)
ax.add_image(im)
ax.set_xlim(-64, 64)
ax.set_ylim(-4, 4)
ax.set_title(interp)

plt.show()
```

**Total running time of the script:** (0 minutes 1.567 seconds)

### Blend transparency with color in 2D images

Blend transparency with color to highlight parts of data with imshow.

A common use for *matplotlib.pyplot.imshow* is to plot a 2D statistical map. The function makes it easy to visualize a 2D matrix as an image and add transparency to the output. For example, one can plot a statistic (such as a t-statistic) and color the transparency of each pixel according to its p-value. This example demonstrates how you can achieve this effect.

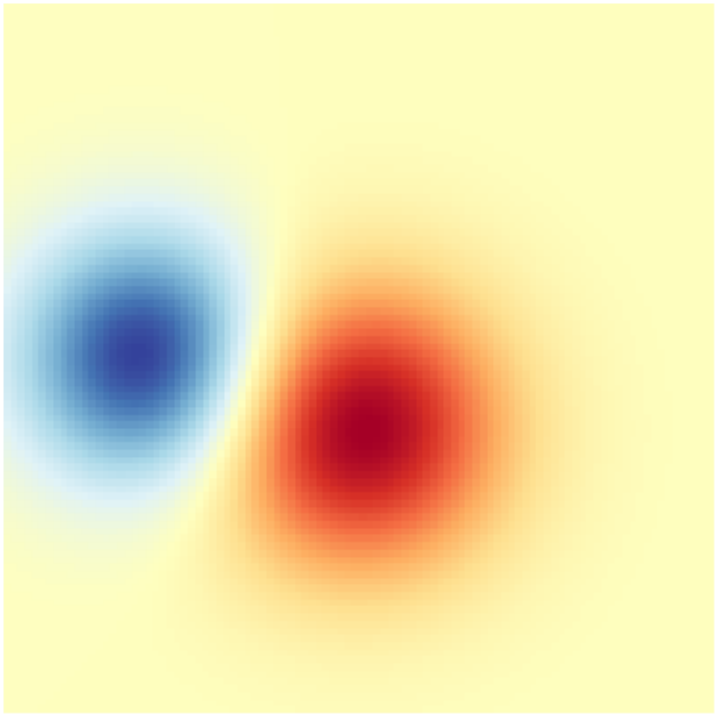
First we will generate some data, in this case, we'll create two 2D "blobs" in a 2D grid. One blob will be positive, and the other negative.

```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.colors import Normalize


def normal_pdf(x, mean, var):
    return np.exp(-(x - mean)**2 / (2*var))


# Generate the space in which the blobs will live
xmin, xmax, ymin, ymax = (0, 100, 0, 100)
n_bins = 100
xx = np.linspace(xmin, xmax, n_bins)
yy = np.linspace(ymin, ymax, n_bins)

# Generate the blobs. The range of the values is roughly -.0002 to .0002
means_high = [20, 50]
means_low = [50, 60]
var = [150, 200]

gauss_x_high = normal_pdf(xx, means_high[0], var[0])
gauss_y_high = normal_pdf(yy, means_high[1], var[0])

gauss_x_low = normal_pdf(xx, means_low[0], var[1])
gauss_y_low = normal_pdf(yy, means_low[1], var[1])

weights = (np.outer(gauss_y_high, gauss_x_high)
           - np.outer(gauss_y_low, gauss_x_low))

# We'll also create a grey background into which the pixels will fade
greys = np.full((*weights.shape, 3), 70, dtype=np.uint8)

# First we'll plot these blobs using ``imshow`` without transparency.
vmax = np.abs(weights).max()
imshow_kwargs = {
    'vmax': vmax,
    'vmin': -vmax,
    'cmap': 'RdYlBu',
    'extent': (xmin, xmax, ymin, ymax),
}
```

(continues on next page)

(continued from previous page)

```
fig, ax = plt.subplots()
ax.imshow(greys)
ax.imshow(weights, **imshow_kwargs)
ax.set_axis_off()
```



### Blending in transparency

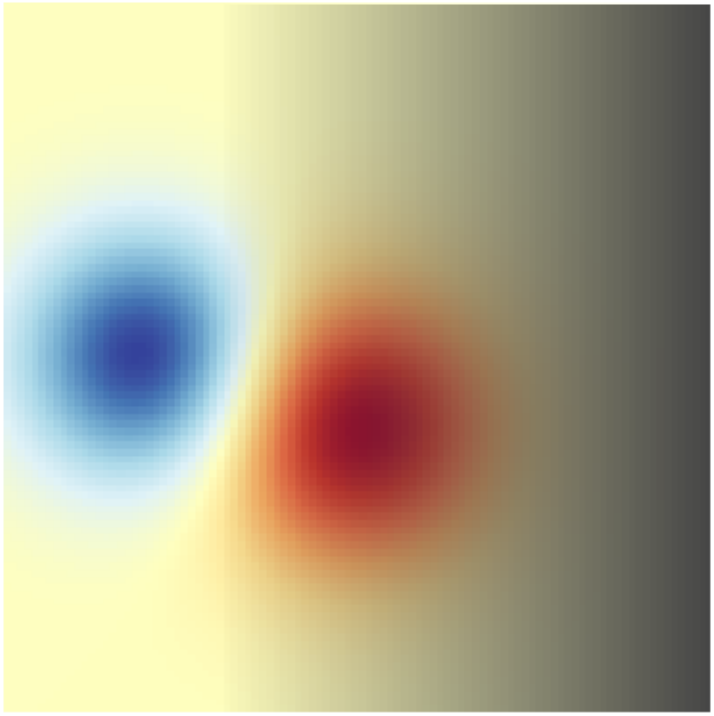
The simplest way to include transparency when plotting data with *matplotlib.pyplot.imshow* is to pass an array matching the shape of the data to the `alpha` argument. For example, we'll create a gradient moving from left to right below.

```
# Create an alpha channel of linearly increasing values moving to the right.
alphas = np.ones(weights.shape)
alphas[:, 30:] = np.linspace(1, 0, 70)

# Create the figure and image
# Note that the absolute values may be slightly different
fig, ax = plt.subplots()
ax.imshow(greys)
```

(continues on next page)

(continued from previous page)

```
ax.imshow(weights, alpha=alphas, **imshow_kwargs)
ax.set_axis_off()
```



### Using transparency to highlight values with high amplitude

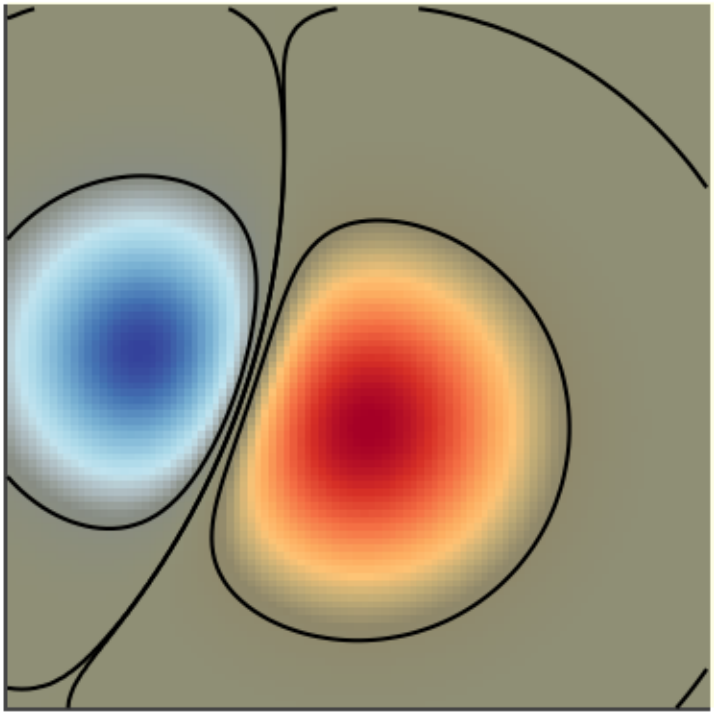
Finally, we'll recreate the same plot, but this time we'll use transparency to highlight the extreme values in the data. This is often used to highlight data points with smaller p-values. We'll also add in contour lines to highlight the image values.

```
# Create an alpha channel based on weight values
# Any value whose absolute value is > .0001 will have zero transparency
alphas = Normalize(0, .3, clip=True)(np.abs(weights))
alphas = np.clip(alphas, .4, 1)  # alpha value clipped at the bottom at .4

# Create the figure and image
# Note that the absolute values may be slightly different
fig, ax = plt.subplots()
ax.imshow(greys)
ax.imshow(weights, alpha=alphas, **imshow_kwargs)
```

(continues on next page)

(continued from previous page)

```
# Add contour lines to further highlight different levels.
ax.contour(weights[::-1], levels=[-.1, .1], colors='k', linestyles='-')
ax.set_axis_off()
plt.show()

ax.contour(weights[::-1], levels=[-.0001, .0001], colors='k', linestyles='-')
ax.set_axis_off()
plt.show()
```



**References**

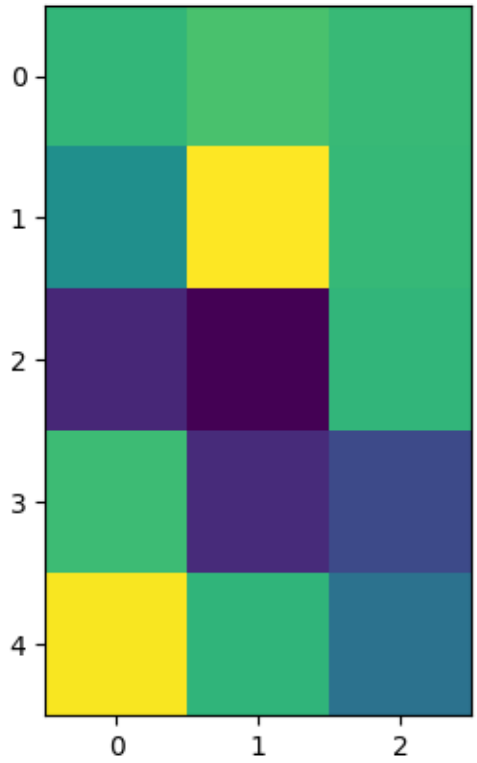
The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.imshow* / *matplotlib.pyplot.imshow*
- *matplotlib.axes.Axes.contour* / *matplotlib.pyplot.contour*
- *matplotlib.colors.Normalize*
- *matplotlib.axes.Axes.set_axis_off*

### Modifying the coordinate formatter

Modify the coordinate formatter to report the image "z" value of the nearest pixel given x and y. This functionality is built in by default; this example just showcases how to customize the *format_coord* function.

```
import matplotlib.pyplot as plt
import numpy as np

# Fixing random state for reproducibility
np.random.seed(19680801)


X = 10*np.random.rand(5, 3)

fig, ax = plt.subplots()
ax.imshow(X)


def format_coord(x, y):
    col = round(x)
    row = round(y)
    nrows, ncols = X.shape
    if 0 <= col < ncols and 0 <= row < nrows:
        z = X[row, col]
        return f'x={x:1.4f}, y={y:1.4f}, z={z:1.4f}'
    else:
        return f'x={x:1.4f}, y={y:1.4f}'


ax.format_coord = format_coord
plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.format_coord*
- *matplotlib.axes.Axes.imshow*

## Interpolations for imshow

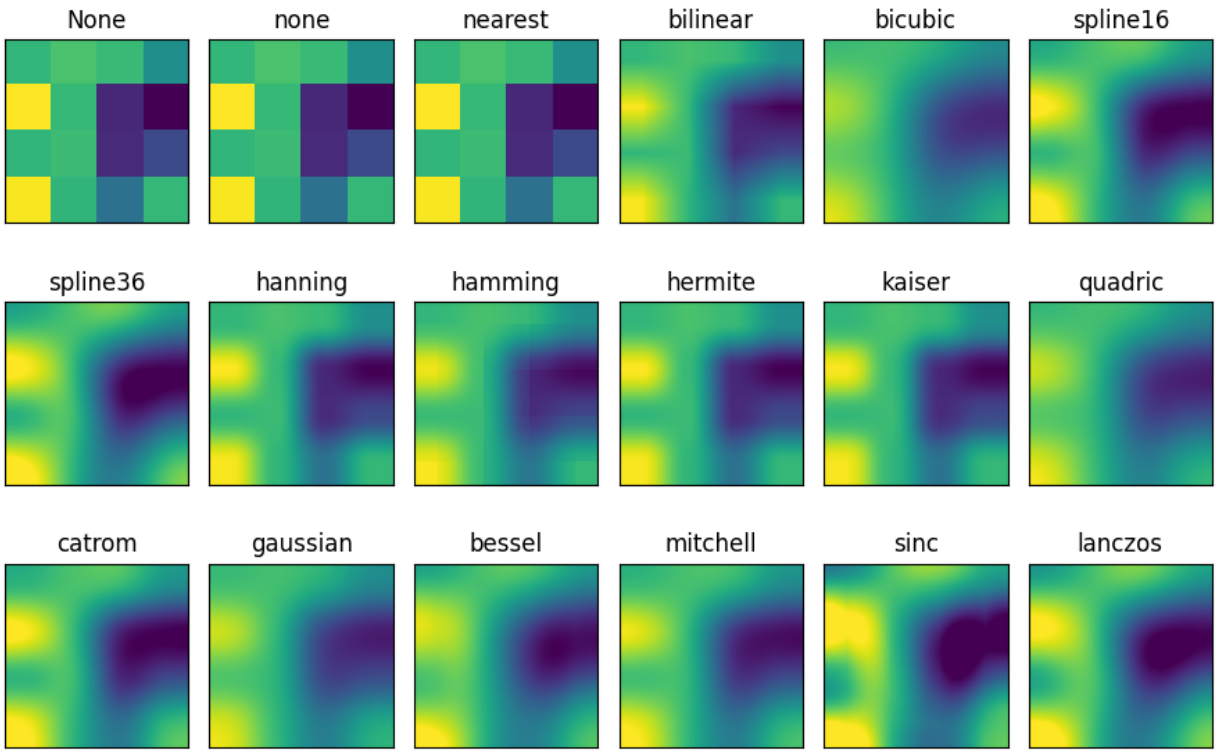
This example displays the difference between interpolation methods for *imshow*.

If *interpolation* is None, it defaults to the *rcParams["image.interpolation"]* (default: `'antialiased'`). If the interpolation is `'none'`, then no interpolation is performed for the Agg, ps and pdf backends. Other backends will default to `'antialiased'`.

For the Agg, ps and pdf backends, `interpolation='none'` works well when a big image is scaled down, while `interpolation='nearest'` works well when a small image is scaled up.

See *Image antialiasing* for a discussion on the default `interpolation='antialiased'` option.

```
import matplotlib.pyplot as plt
import numpy as np

methods = [None, 'none', 'nearest', 'bilinear', 'bicubic', 'spline16',
           'spline36', 'hanning', 'hamming', 'hermite', 'kaiser', 'quadric',
           'catrom', 'gaussian', 'bessel', 'mitchell', 'sinc', 'lanczos']

# Fixing random state for reproducibility
np.random.seed(19680801)

grid = np.random.rand(4, 4)

fig, axs = plt.subplots(nrows=3, ncols=6, figsize=(9, 6),
                        subplot_kw={'xticks': [], 'yticks': []})

for ax, interp_method in zip(axs.flat, methods):
    ax.imshow(grid, interpolation=interp_method, cmap='viridis')
    ax.set_title(str(interp_method))

plt.tight_layout()
plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.imshow* / *matplotlib.pyplot.imshow*

**Total running time of the script:** (0 minutes 1.451 seconds)

### Contour plot of irregularly spaced data

Comparison of a contour plot of irregularly spaced data interpolated on a regular grid versus a tricontour plot for an unstructured triangular grid.

Since *contour* and *contourf* expect the data to live on a regular grid, plotting a contour plot of irregularly spaced data requires different methods. The two options are:

- Interpolate the data to a regular grid first. This can be done with on-board means, e.g. via *LinearTriInterpolator* or using external functionality e.g. via `scipy.interpolate.griddata`. Then plot the interpolated data with the usual *contour*.
- Directly use *tricontour* or *tricontourf* which will perform a triangulation internally.

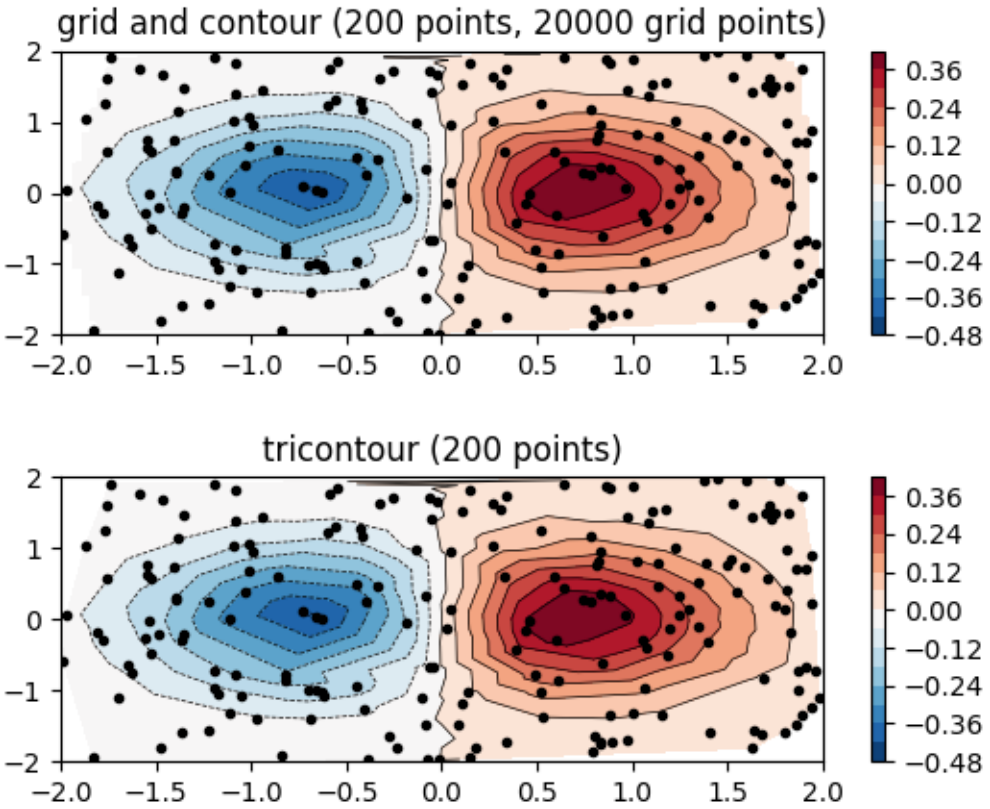
This example shows both methods in action.

```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib.tri as tri

np.random.seed(19680801)
npts = 200
ngridx = 100
ngridy = 200
x = np.random.uniform(-2, 2, npts)
y = np.random.uniform(-2, 2, npts)
z = x * np.exp(-x**2 - y**2)

fig, (ax1, ax2) = plt.subplots(nrows=2)

# ----------------------
# Interpolation on a grid
# ----------------------
# A contour plot of irregularly spaced data coordinates
# via interpolation on a grid.

# Create grid values first.
xi = np.linspace(-2.1, 2.1, ngridx)
yi = np.linspace(-2.1, 2.1, ngridy)

# Linearly interpolate the data (x, y) on a grid defined by (xi, yi).
triang = tri.Triangulation(x, y)
interpolator = tri.LinearTriInterpolator(triang, z)
Xi, Yi = np.meshgrid(xi, yi)
zi = interpolator(Xi, Yi)

# Note that scipy.interpolate provides means to interpolate data on a grid
```

(continues on next page)

(continued from previous page)

```
# as well. The following would be an alternative to the four lines above:
# from scipy.interpolate import griddata
# zi = griddata((x, y), z, (xi[None, :], yi[:, None]), method='linear')

ax1.contour(xi, yi, zi, levels=14, linewidths=0.5, colors='k')
cntr1 = ax1.contourf(xi, yi, zi, levels=14, cmap="RdBu_r")

fig.colorbar(cntr1, ax=ax1)
ax1.plot(x, y, 'ko', ms=3)
ax1.set(xlim=(-2, 2), ylim=(-2, 2))
ax1.set_title('grid and contour (%d points, %d grid points)' %
              (npts, ngridx * ngridy))

# ----------
# Tricontour
# ----------
# Directly supply the unordered, irregularly spaced coordinates
# to tricontour.

ax2.tricontour(x, y, z, levels=14, linewidths=0.5, colors='k')
cntr2 = ax2.tricontourf(x, y, z, levels=14, cmap="RdBu_r")

fig.colorbar(cntr2, ax=ax2)
ax2.plot(x, y, 'ko', ms=3)
ax2.set(xlim=(-2, 2), ylim=(-2, 2))
ax2.set_title('tricontour (%d points)' % npts)

plt.subplots_adjust(hspace=0.5)
plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.contour* / *matplotlib.pyplot.contour*
- *matplotlib.axes.Axes.contourf* / *matplotlib.pyplot.contourf*
- *matplotlib.axes.Axes.tricontour* / *matplotlib.pyplot.tricontour*
- *matplotlib.axes.Axes.tricontourf* / *matplotlib.pyplot.tricontourf*

## Layer Images

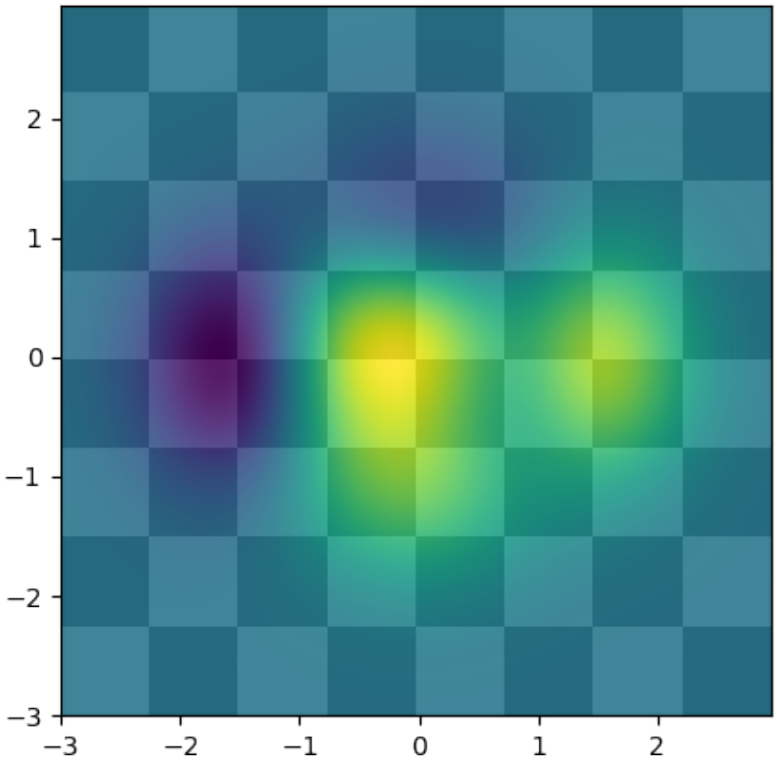
Layer images above one another using alpha blending

```
import matplotlib.pyplot as plt
import numpy as np


def func3(x, y):
```

(continues on next page)

(continued from previous page)

```
    return (1 - x / 2 + x**5 + y**3) * np.exp(-(x**2 + y**2))


# make these smaller to increase the resolution
dx, dy = 0.05, 0.05

x = np.arange(-3.0, 3.0, dx)
y = np.arange(-3.0, 3.0, dy)
X, Y = np.meshgrid(x, y)

# when layering multiple images, the images need to have the same
# extent.  This does not mean they need to have the same shape, but
# they both need to render to the same coordinate system determined by
# xmin, xmax, ymin, ymax.  Note if you use different interpolations
# for the images their apparent extent could be different due to
# interpolation edge effects

extent = np.min(x), np.max(x), np.min(y), np.max(y)
fig = plt.figure(frameon=False)

Z1 = np.add.outer(range(8), range(8)) % 2  # chessboard
im1 = plt.imshow(Z1, cmap=plt.cm.gray, interpolation='nearest',
                 extent=extent)

Z2 = func3(X, Y)

im2 = plt.imshow(Z2, cmap=plt.cm.viridis, alpha=.9, interpolation='bilinear',
                 extent=extent)

plt.show()
```




**References**

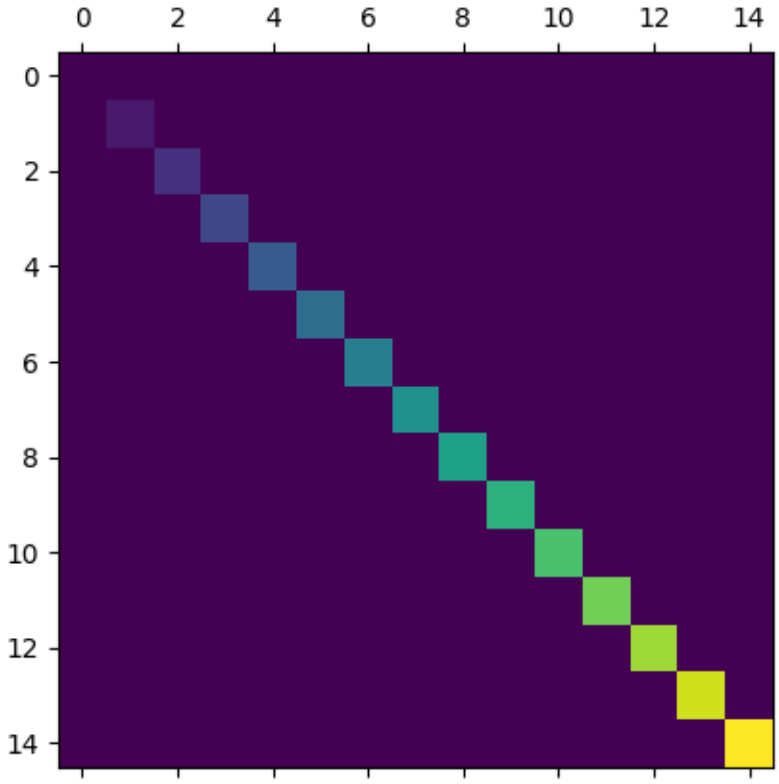
The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.imshow* / *matplotlib.pyplot.imshow*

### Visualize matrices with matshow

*matshow* visualizes a 2D matrix or array as color-coded image.

```
import matplotlib.pyplot as plt
import numpy as np

# a 2D array with linearly increasing values on the diagonal
a = np.diag(range(15))

plt.matshow(a)

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.imshow* / *matplotlib.pyplot.imshow*

## Multiple images

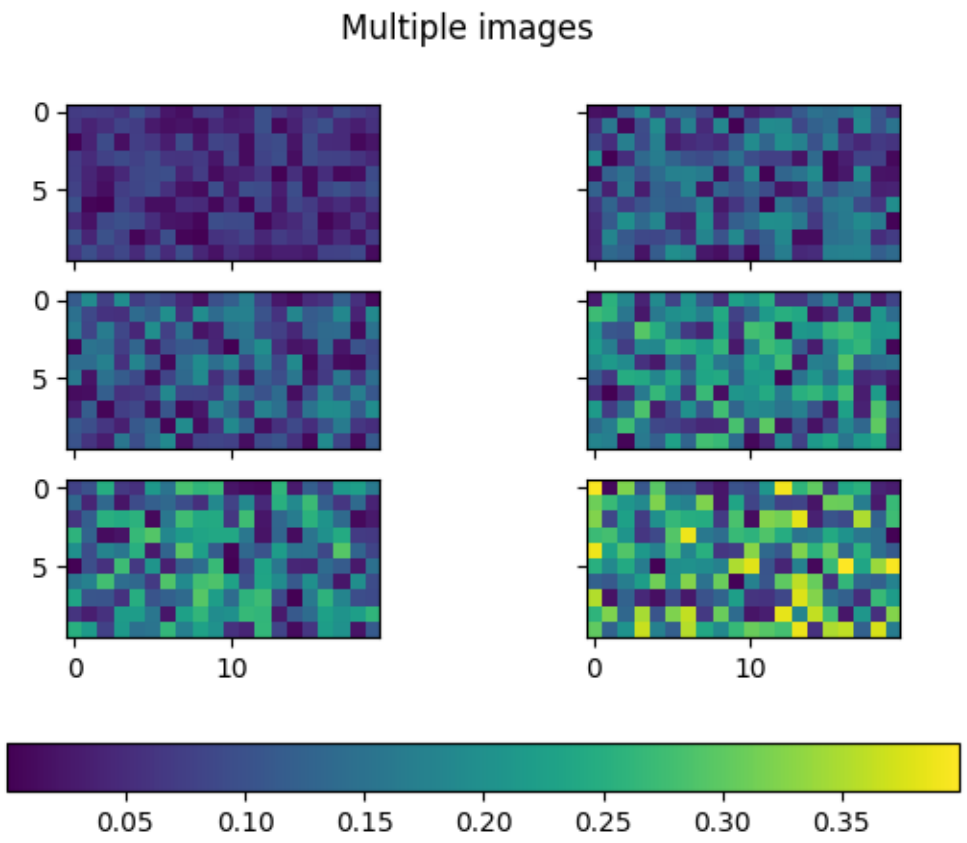
Make a set of images with a single colormap, norm, and colorbar.

```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib import colors

np.random.seed(19680801)
Nr = 3
Nc = 2

fig, axs = plt.subplots(Nr, Nc)
```

(continues on next page)

(continued from previous page)

```
fig.suptitle('Multiple images')

images = []
for i in range(Nr):
    for j in range(Nc):
        # Generate data with a range that varies from one plot to the next.
        data = ((1 + i + j) / 10) * np.random.rand(10, 20)
        images.append(axs[i, j].imshow(data))
        axs[i, j].label_outer()

# Find the min and max of all colors for use in setting the color scale.
vmin = min(image.get_array().min() for image in images)
vmax = max(image.get_array().max() for image in images)
norm = colors.Normalize(vmin=vmin, vmax=vmax)
for im in images:
    im.set_norm(norm)

fig.colorbar(images[0], ax=axs, orientation='horizontal', fraction=.1)


# Make images respond to changes in the norm of other images (e.g. via the
# "edit axis, curves and images parameters" GUI on Qt), but be careful not to
# recurse infinitely!
def update(changed_image):
    for im in images:
        if (changed_image.get_cmap() != im.get_cmap()
                or changed_image.get_clim() != im.get_clim()):
            im.set_cmap(changed_image.get_cmap())
            im.set_clim(changed_image.get_clim())


for im in images:
    im.callbacks.connect('changed', update)

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.imshow* / *matplotlib.pyplot.imshow*
- *matplotlib.figure.Figure.colorbar* / *matplotlib.pyplot.colorbar*
- *matplotlib.colors.Normalize*
- *matplotlib.cm.ScalarMappable.set_cmap*
- *matplotlib.cm.ScalarMappable.set_norm*
- *matplotlib.cm.ScalarMappable.set_clim*
- *matplotlib.cbook.CallbackRegistry.connect*

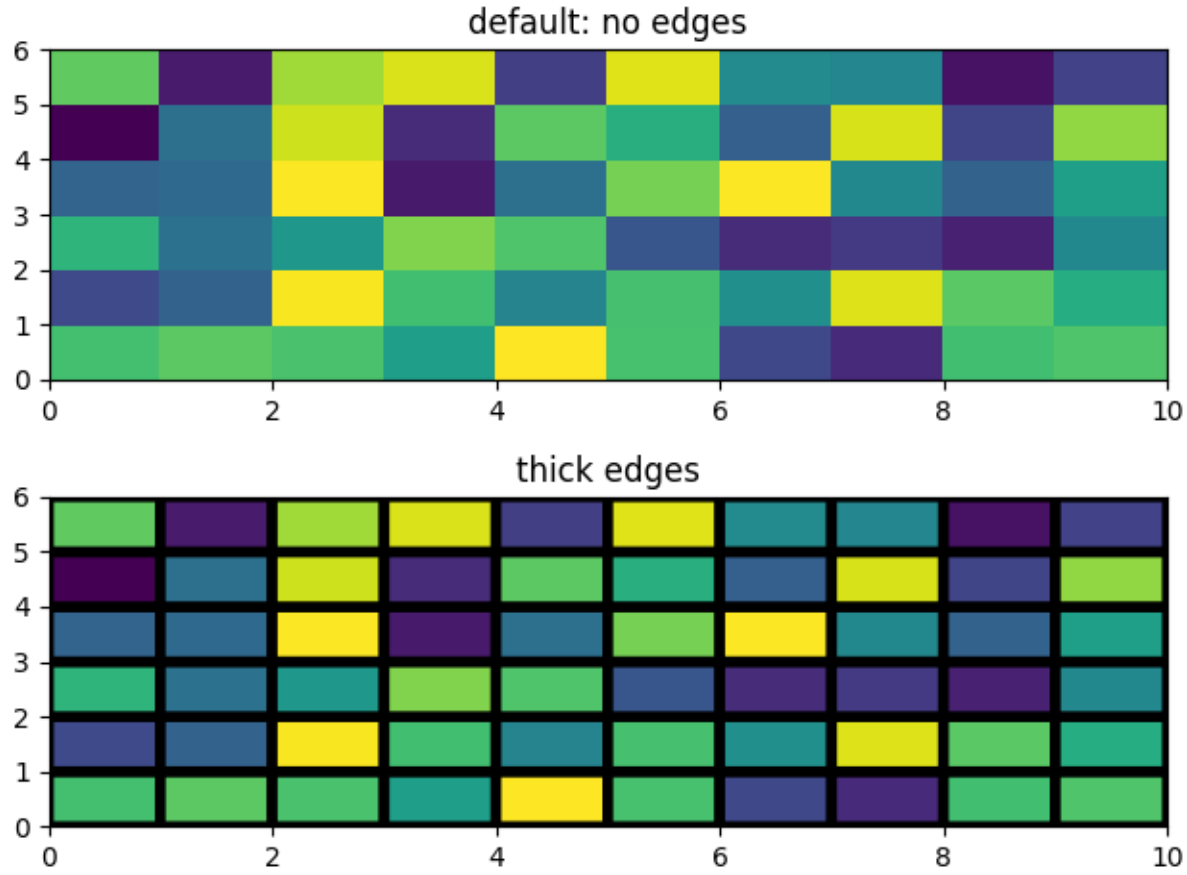
### pcolor images

*pcolor* generates 2D image-style plots, as illustrated below.

```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.colors import LogNorm

# Fixing random state for reproducibility
np.random.seed(19680801)
```

### A simple pcolor demo

```
Z = np.random.rand(6, 10)

fig, (ax0, ax1) = plt.subplots(2, 1)

c = ax0.pcolor(Z)
ax0.set_title('default: no edges')

c = ax1.pcolor(Z, edgecolors='k', linewidths=4)
ax1.set_title('thick edges')

fig.tight_layout()
plt.show()
```




### Comparing pcolor with similar functions

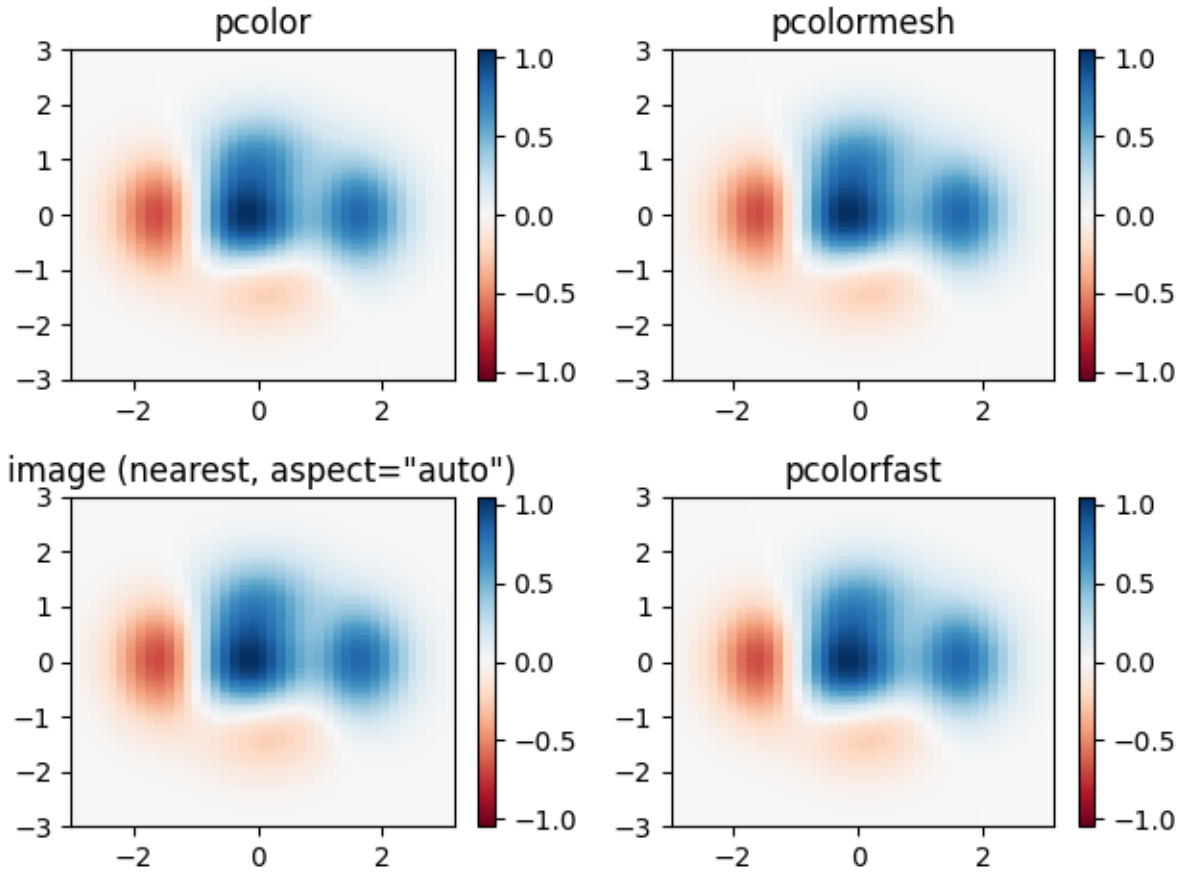
Demonstrates similarities between *pcolor*, *pcolormesh*, *imshow* and *pcolorfast* for drawing quadrilateral grids. Note that we call `imshow` with `aspect="auto"` so that it doesn't force the data pixels to be square (the default is `aspect="equal"`).

```
# make these smaller to increase the resolution
dx, dy = 0.15, 0.05

# generate 2 2d grids for the x & y bounds
y, x = np.mgrid[-3:3+dy:dy, -3:3+dx:dx]
z = (1 - x/2 + x**5 + y**3) * np.exp(-x**2 - y**2)
# x and y are bounds, so z should be the value *inside* those bounds.
# Therefore, remove the last value from the z array.
z = z[:-1, :-1]
z_min, z_max = -abs(z).max(), abs(z).max()

fig, axs = plt.subplots(2, 2)

ax = axs[0, 0]
c = ax.pcolor(x, y, z, cmap='RdBu', vmin=z_min, vmax=z_max)
```

(continues on next page)

(continued from previous page)

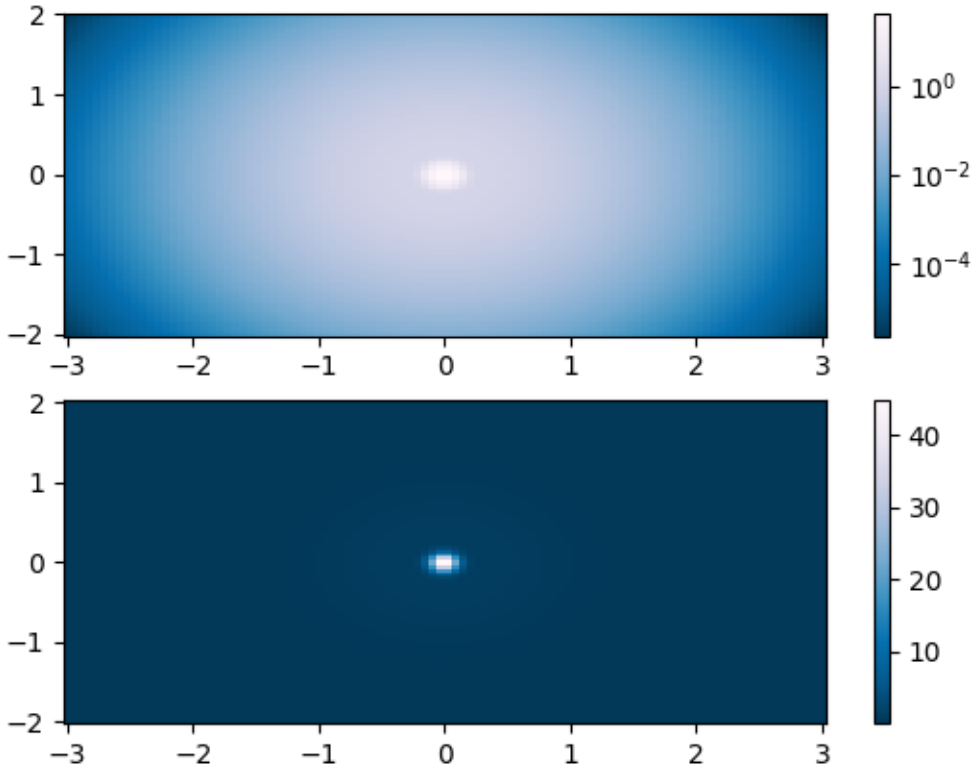
```
ax.set_title('pcolor')
fig.colorbar(c, ax=ax)

ax = axs[0, 1]
c = ax.pcolormesh(x, y, z, cmap='RdBu', vmin=z_min, vmax=z_max)
ax.set_title('pcolormesh')
fig.colorbar(c, ax=ax)

ax = axs[1, 0]
c = ax.imshow(z, cmap='RdBu', vmin=z_min, vmax=z_max,
              extent=[x.min(), x.max(), y.min(), y.max()],
              interpolation='nearest', origin='lower', aspect='auto')
ax.set_title('image (nearest, aspect="auto")')
fig.colorbar(c, ax=ax)

ax = axs[1, 1]
c = ax.pcolorfast(x, y, z, cmap='RdBu', vmin=z_min, vmax=z_max)
ax.set_title('pcolorfast')
fig.colorbar(c, ax=ax)

fig.tight_layout()
plt.show()
```



### Pcolor with a log scale

The following shows pcolor plots with a log scale.

```
N = 100
X, Y = np.meshgrid(np.linspace(-3, 3, N), np.linspace(-2, 2, N))

# A low hump with a spike coming out.
# Needs to have z/colour axis on a log scale, so we see both hump and spike.
# A linear scale only shows the spike.
Z1 = np.exp(-X**2 - Y**2)
Z2 = np.exp(-(X * 10)**2 - (Y * 10)**2)
Z = Z1 + 50 * Z2

fig, (ax0, ax1) = plt.subplots(2, 1)

c = ax0.pcolor(X, Y, Z, shading='auto',
               norm=LogNorm(vmin=Z.min(), vmax=Z.max()), cmap='PuBu_r')
fig.colorbar(c, ax=ax0)

c = ax1.pcolor(X, Y, Z, cmap='PuBu_r', shading='auto')
fig.colorbar(c, ax=ax1)

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.pcolor* / *matplotlib.pyplot.pcolor*
- *matplotlib.axes.Axes.pcolormesh* / *matplotlib.pyplot.pcolormesh*
- *matplotlib.axes.Axes.pcolorfast*
- *matplotlib.axes.Axes.imshow* / *matplotlib.pyplot.imshow*
- *matplotlib.figure.Figure.colorbar* / *matplotlib.pyplot.colorbar*
- *matplotlib.colors.LogNorm*

**Total running time of the script:** (0 minutes 1.702 seconds)

### pcolormesh grids and shading

*axes.Axes.pcolormesh* and *pcolor* have a few options for how grids are laid out and the shading between the grid points.

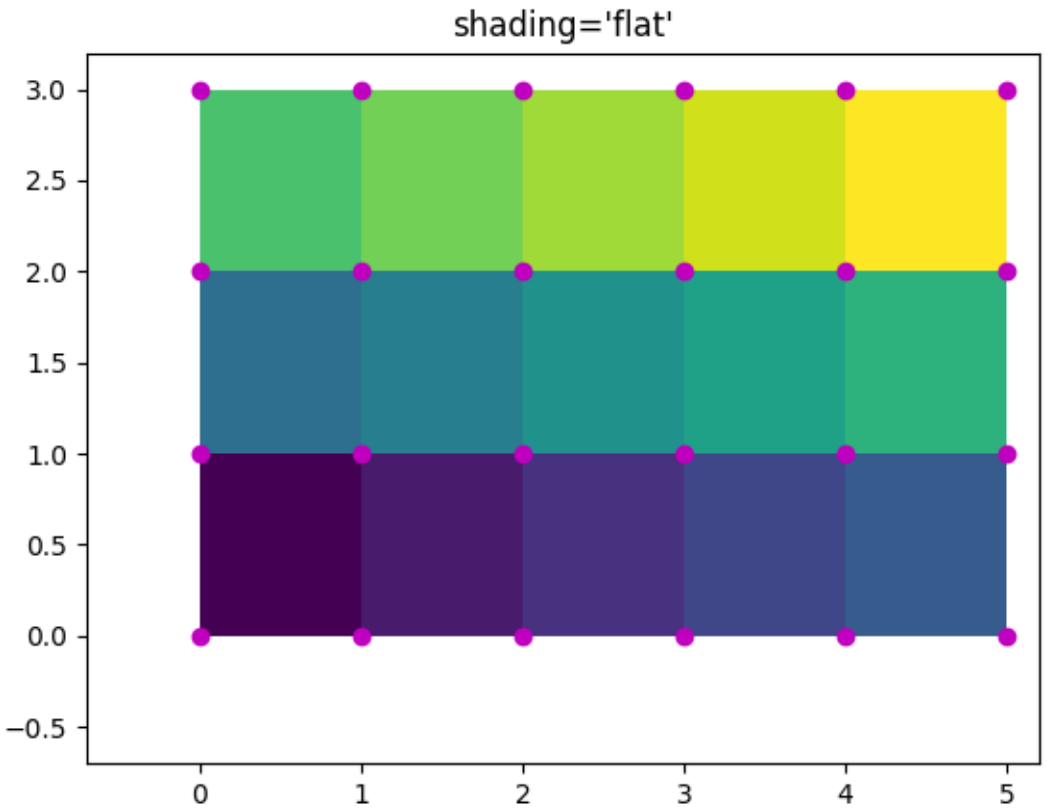
Generally, if *Z* has shape *(M, N)* then the grid *X* and *Y* can be specified with either shape *(M+1, N+1)* or *(M, N)*, depending on the argument for the `shading` keyword argument. Note that below we specify vectors *x* as either length N or N+1 and *y* as length M or M+1, and *pcolormesh* internally makes the mesh matrices *X* and *Y* from the input vectors.

```
import matplotlib.pyplot as plt
import numpy as np
```

### Flat Shading

The grid specification with the least assumptions is `shading='flat'` and if the grid is one larger than the data in each dimension, i.e. has shape *(M+1, N+1)*. In that case *X* and *Y* specify the corners of quadrilaterals that are colored with the values in *Z*. Here we specify the edges of the *(3, 5)* quadrilaterals with *X* and *Y* that are *(4, 6)*.

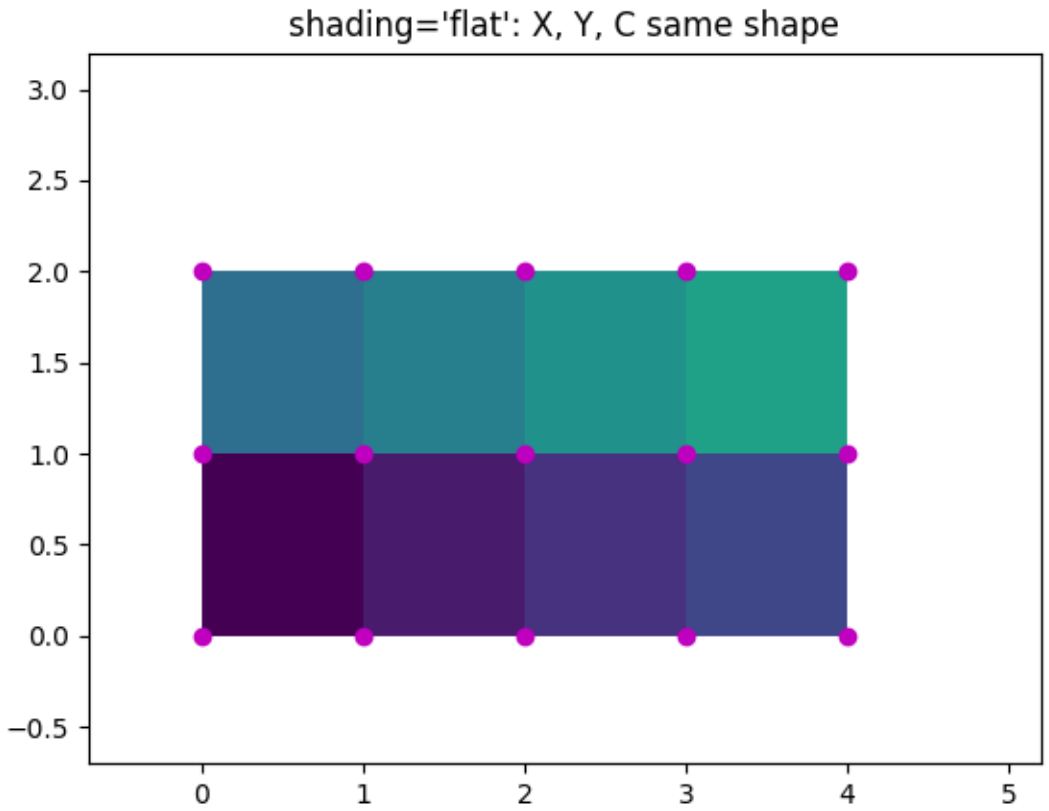
```
nrows = 3
ncols = 5
Z = np.arange(nrows * ncols).reshape(nrows, ncols)
x = np.arange(ncols + 1)
y = np.arange(nrows + 1)

fig, ax = plt.subplots()
ax.pcolormesh(x, y, Z, shading='flat', vmin=Z.min(), vmax=Z.max())


def _annotate(ax, x, y, title):
    # this all gets repeated below:
    X, Y = np.meshgrid(x, y)
    ax.plot(X.flat, Y.flat, 'o', color='m')
    ax.set_xlim(-0.7, 5.2)
    ax.set_ylim(-0.7, 3.2)
    ax.set_title(title)

_annotate(ax, x, y, "shading='flat'")
```




### Flat Shading, same shape grid

Often, however, data is provided where *X* and *Y* match the shape of *Z*. While this makes sense for other `shading` types, it is not permitted when `shading='flat'`. Historically, Matplotlib silently dropped the last row and column of *Z* in this case, to match Matlab's behavior. If this behavior is still desired, simply drop the last row and column manually:

```
x = np.arange(ncols)  # note *not* ncols + 1 as before
y = np.arange(nrows)
fig, ax = plt.subplots()
ax.pcolormesh(x, y, Z[:-1, :-1], shading='flat', vmin=Z.min(), vmax=Z.max())
_annotate(ax, x, y, "shading='flat': X, Y, C same shape")
```

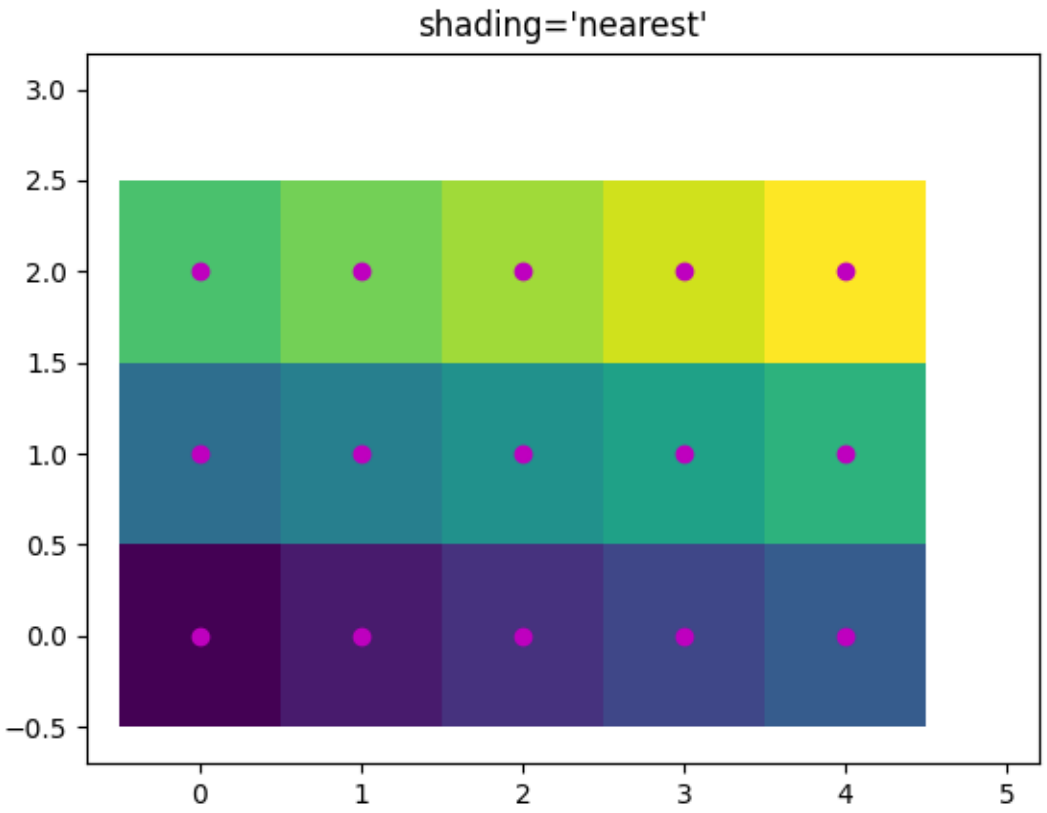



### Nearest Shading, same shape grid

Usually, dropping a row and column of data is not what the user means when they make *X*, *Y* and *Z* all the same shape. For this case, Matplotlib allows `shading='nearest'` and centers the colored quadrilaterals on the grid points.

If a grid that is not the correct shape is passed with `shading='nearest'` an error is raised.

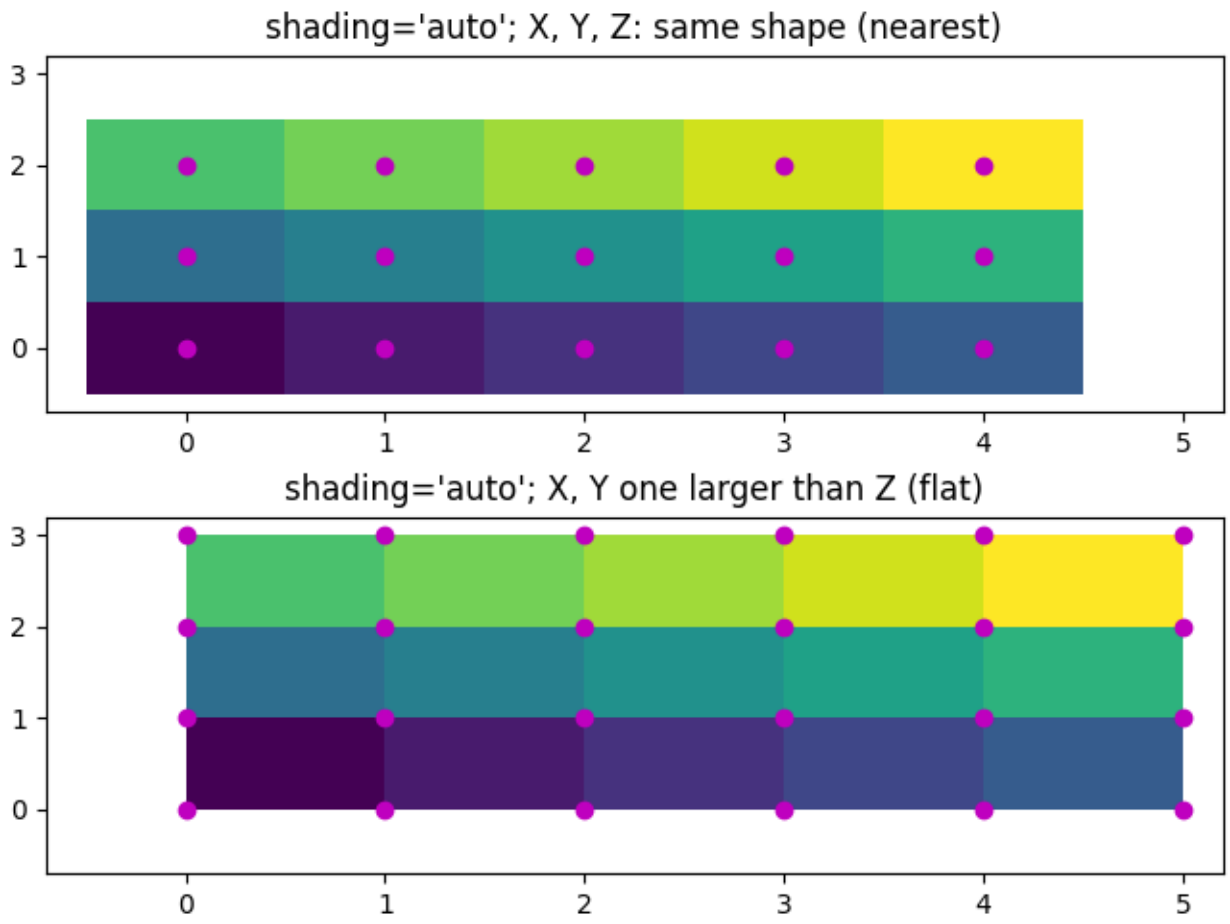
```
fig, ax = plt.subplots()
ax.pcolormesh(x, y, Z, shading='nearest', vmin=Z.min(), vmax=Z.max())
_annotate(ax, x, y, "shading='nearest'")
```




### Auto Shading

It's possible that the user would like the code to automatically choose which to use, in this case `shading='auto'` will decide whether to use 'flat' or 'nearest' shading based on the shapes of *X*, *Y* and *Z*.

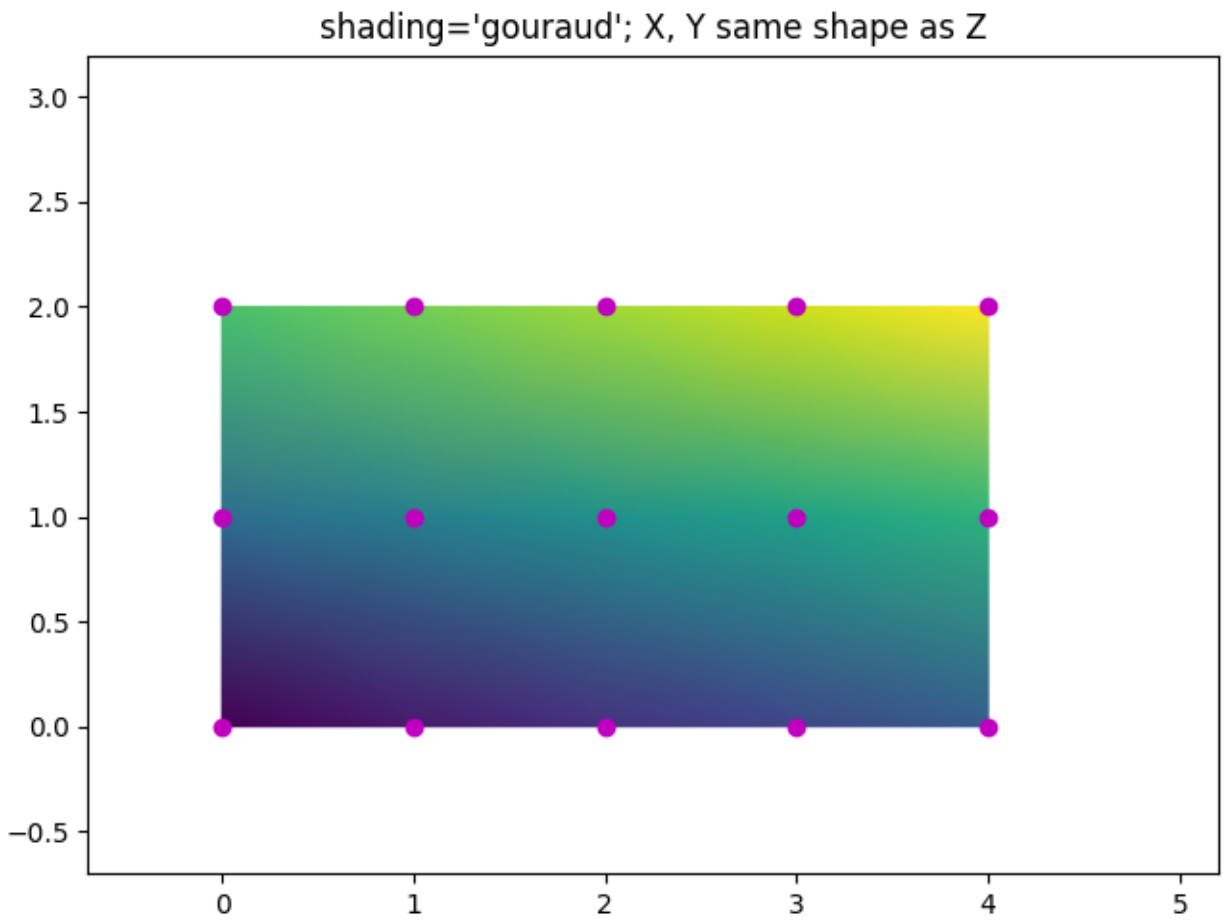
```
fig, axs = plt.subplots(2, 1, layout='constrained')
ax = axs[0]
x = np.arange(ncols)
y = np.arange(nrows)
ax.pcolormesh(x, y, Z, shading='auto', vmin=Z.min(), vmax=Z.max())
_annotate(ax, x, y, "shading='auto'; X, Y, Z: same shape (nearest)")

ax = axs[1]
x = np.arange(ncols + 1)
y = np.arange(nrows + 1)
ax.pcolormesh(x, y, Z, shading='auto', vmin=Z.min(), vmax=Z.max())
_annotate(ax, x, y, "shading='auto'; X, Y one larger than Z (flat)")
```




### Gouraud Shading

Gouraud shading can also be specified, where the color in the quadrilaterals is linearly interpolated between the grid points. The shapes of $X$, $Y$, $Z$ must be the same.

```
fig, ax = plt.subplots(layout='constrained')
x = np.arange(ncols)
y = np.arange(nrows)
ax.pcolormesh(x, y, Z, shading='gouraud', vmin=Z.min(), vmax=Z.max())
_annotate(ax, x, y, "shading='gouraud'; X, Y same shape as Z")

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.pcolormesh* / *matplotlib.pyplot.pcolormesh*

**Total running time of the script:** (0 minutes 1.894 seconds)

## pcolormesh

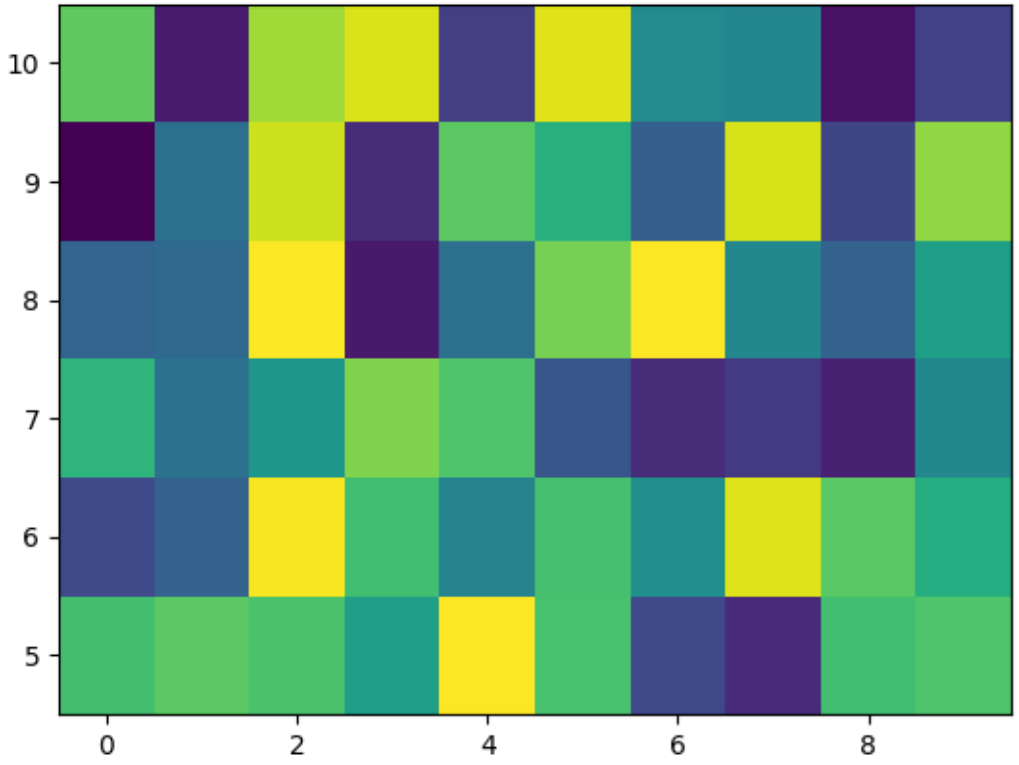
*axes.Axes.pcolormesh* allows you to generate 2D image-style plots. Note that it is faster than the similar *pcolor*.

```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.colors import BoundaryNorm
from matplotlib.ticker import MaxNLocator
```

### Basic pcolormesh

We usually specify a pcolormesh by defining the edge of quadrilaterals and the value of the quadrilateral. Note that here *x* and *y* each have one extra element than Z in the respective dimension.

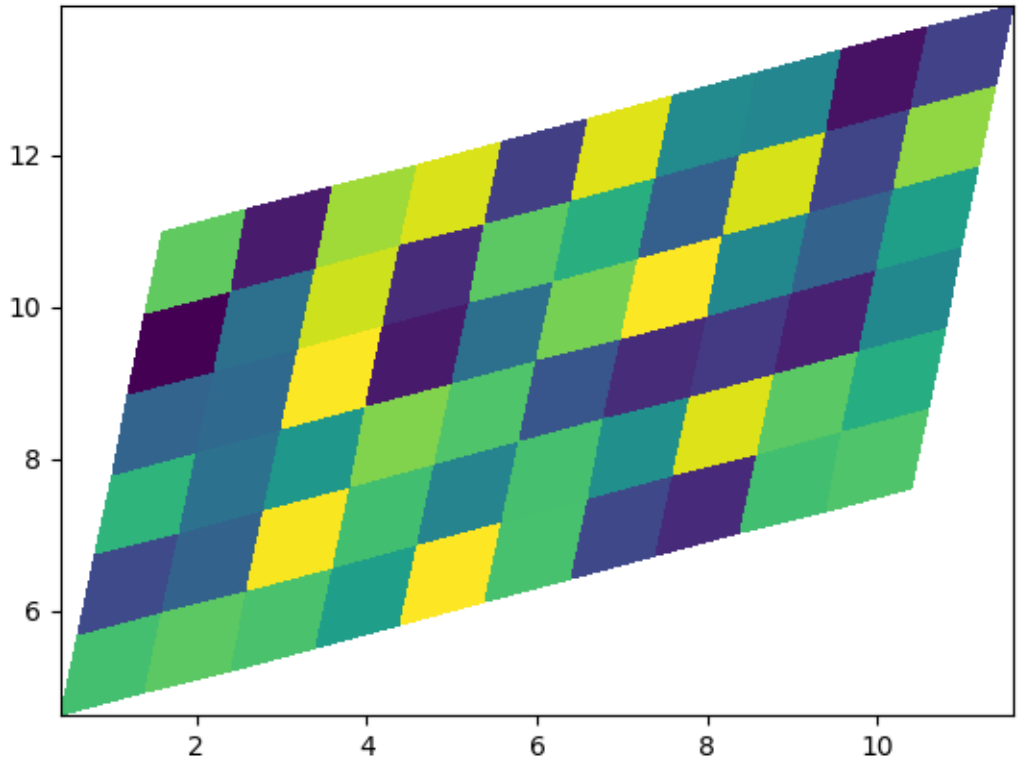
```
np.random.seed(19680801)
Z = np.random.rand(6, 10)
x = np.arange(-0.5, 10, 1)  # len = 11
y = np.arange(4.5, 11, 1)  # len = 7

fig, ax = plt.subplots()
ax.pcolormesh(x, y, Z)
```



### Non-rectilinear pcolormesh

Note that we can also specify matrices for $X$ and $Y$ and have non-rectilinear quadrilaterals.

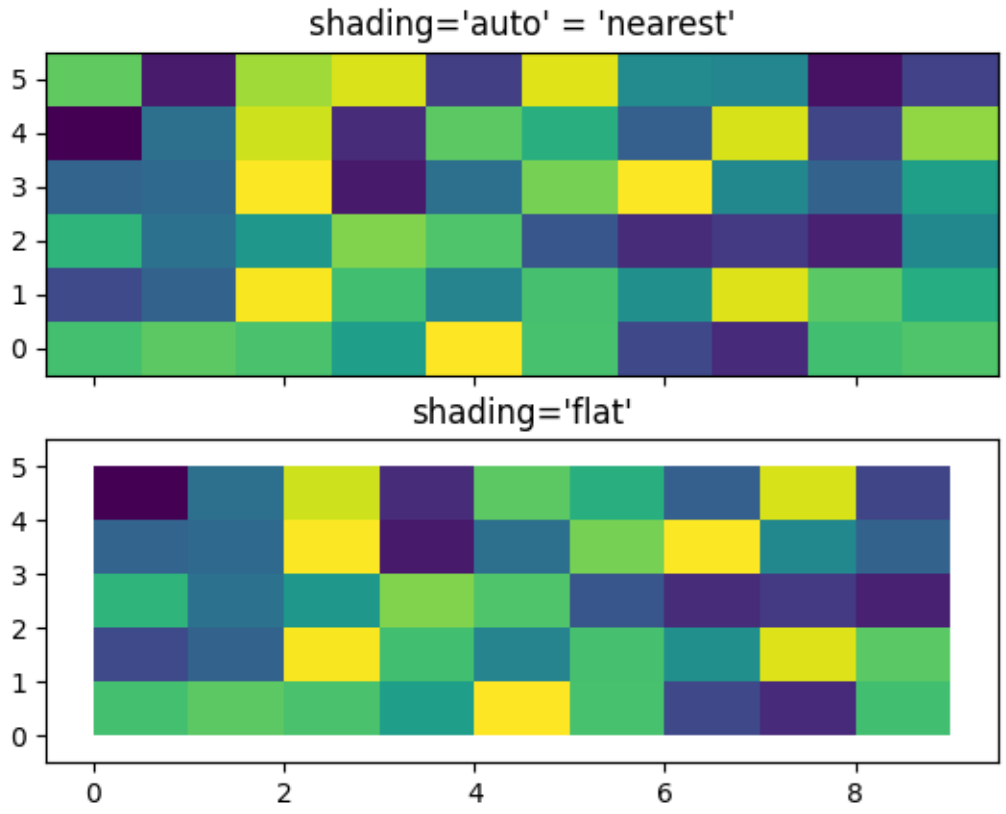
```
x = np.arange(-0.5, 10, 1)  # len = 11
y = np.arange(4.5, 11, 1)  # len = 7
X, Y = np.meshgrid(x, y)
X = X + 0.2 * Y  # tilt the coordinates.
Y = Y + 0.3 * X

fig, ax = plt.subplots()
ax.pcolormesh(X, Y, Z)
```



### Centered Coordinates

Often a user wants to pass *X* and *Y* with the same sizes as *Z* to *axes.Axes.pcolormesh*. This is also allowed if `shading='auto'` is passed (default set by *rcParams["pcolor.shading"]* (default: `'auto'`)). Pre Matplotlib 3.3, `shading='flat'` would drop the last column and row of *Z*, but now gives an error. If this is really what you want, then simply drop the last row and column of Z manually:

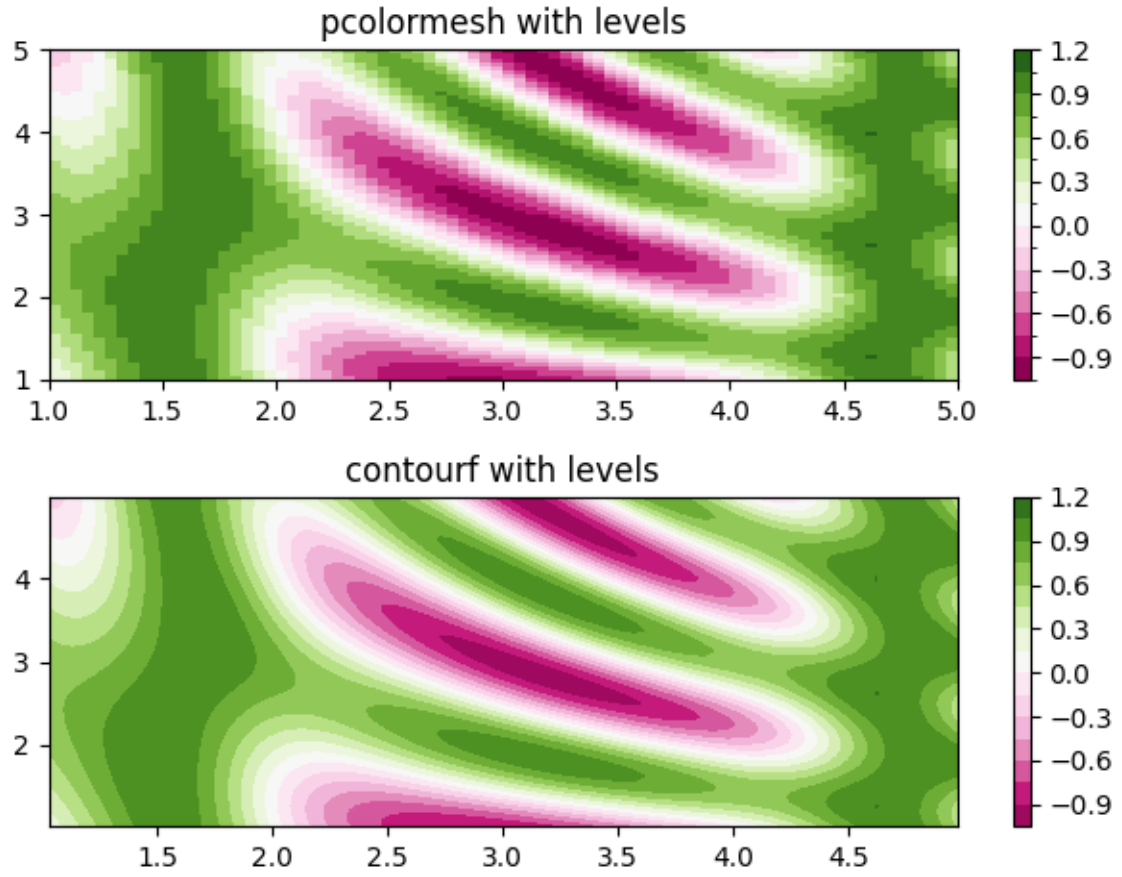
```
x = np.arange(10)  # len = 10
y = np.arange(6)  # len = 6
X, Y = np.meshgrid(x, y)

fig, axs = plt.subplots(2, 1, sharex=True, sharey=True)
axs[0].pcolormesh(X, Y, Z, vmin=np.min(Z), vmax=np.max(Z), shading='auto')
axs[0].set_title("shading='auto' = 'nearest'")
axs[1].pcolormesh(X, Y, Z[:-1, :-1], vmin=np.min(Z), vmax=np.max(Z),
                  shading='flat')
axs[1].set_title("shading='flat'")
```



### Making levels using Norms

Shows how to combine Normalization and Colormap instances to draw "levels" in *axes.Axes.pcolor*, *axes.Axes.pcolormesh* and *axes.Axes.imshow* type plots in a similar way to the levels keyword argument to contour/contourf.

```
# make these smaller to increase the resolution
dx, dy = 0.05, 0.05

# generate 2 2d grids for the x & y bounds
y, x = np.mgrid[slice(1, 5 + dy, dy),
                slice(1, 5 + dx, dx)]

z = np.sin(x)**10 + np.cos(10 + y*x) * np.cos(x)

# x and y are bounds, so z should be the value *inside* those bounds.
# Therefore, remove the last value from the z array.
z = z[:-1, :-1]
levels = MaxNLocator(nbins=15).tick_values(z.min(), z.max())


# pick the desired colormap, sensible levels, and define a normalization
# instance which takes data values and translates those into levels.
cmap = plt.colormaps['PiYG']
norm = BoundaryNorm(levels, ncolors=cmap.N, clip=True)

fig, (ax0, ax1) = plt.subplots(nrows=2)

im = ax0.pcolormesh(x, y, z, cmap=cmap, norm=norm)
fig.colorbar(im, ax=ax0)
ax0.set_title('pcolormesh with levels')


# contours are *point* based plots, so convert our bound into point
# centers
cf = ax1.contourf(x[:-1, :-1] + dx/2.,
                  y[:-1, :-1] + dy/2., z, levels=levels,
                  cmap=cmap)
fig.colorbar(cf, ax=ax1)
ax1.set_title('contourf with levels')

# adjust spacing between subplots so `ax1` title and `ax0` tick labels
# don't overlap
fig.tight_layout()

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.pcolormesh* / *matplotlib.pyplot.pcolormesh*
- *matplotlib.axes.Axes.contourf* / *matplotlib.pyplot.contourf*
- *matplotlib.figure.Figure.colorbar* / *matplotlib.pyplot.colorbar*
- *matplotlib.colors.BoundaryNorm*
- *matplotlib.ticker.MaxNLocator*

**Total running time of the script:** (0 minutes 1.204 seconds)

### Streamplot

A stream plot, or streamline plot, is used to display 2D vector fields. This example shows a few features of the *streamplot* function:

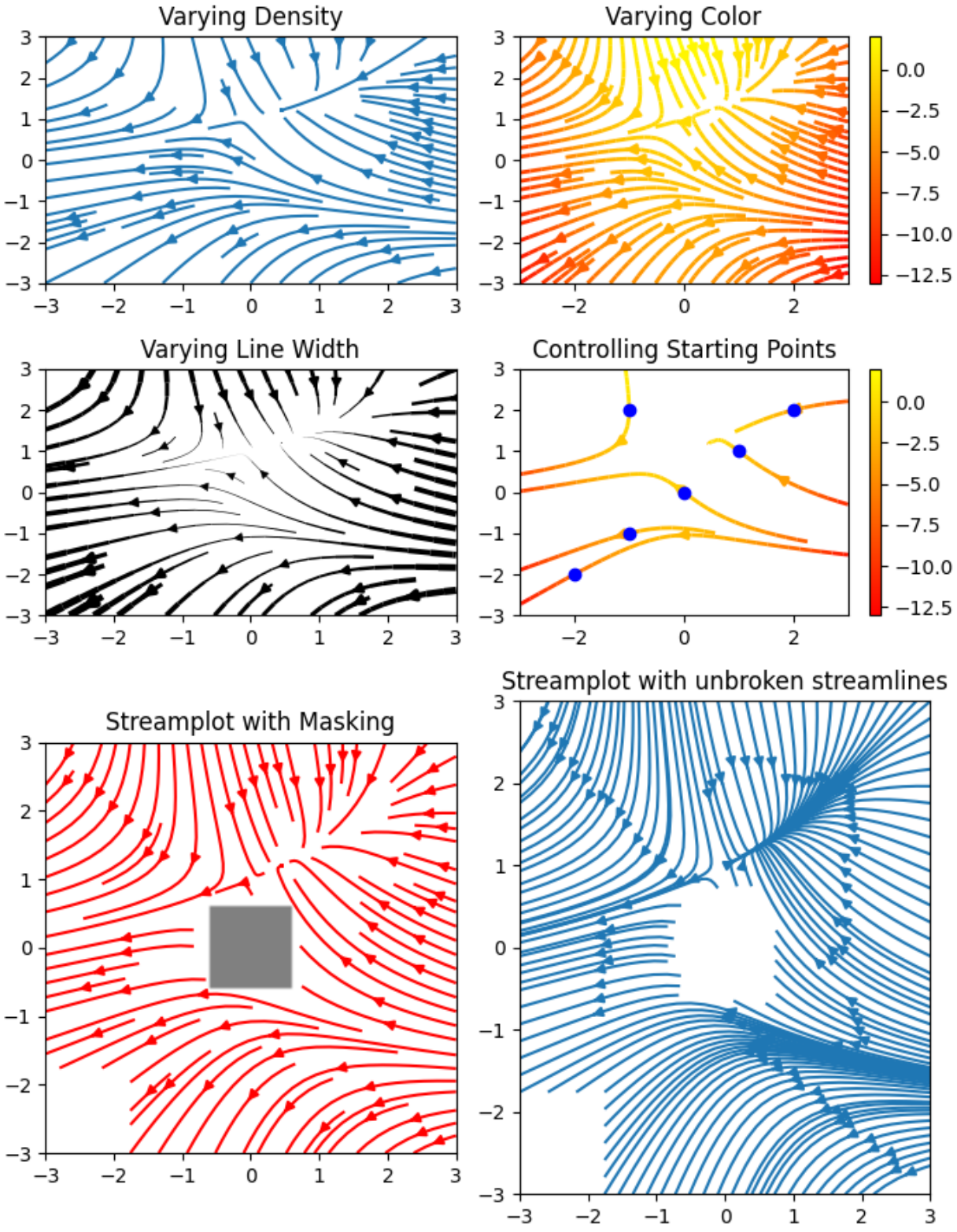
- Varying the color along a streamline.
- Varying the density of streamlines.
- Varying the line width along a streamline.
- Controlling the starting points of streamlines.
- Streamlines skipping masked regions and NaN values.
- Unbroken streamlines even when exceeding the limit of lines within a single grid cell.

```
import matplotlib.pyplot as plt
import numpy as np

w = 3
Y, X = np.mgrid[-w:w:100j, -w:w:100j]
U = -1 - X**2 + Y
V = 1 + X - Y**2
speed = np.sqrt(U**2 + V**2)

fig, axs = plt.subplots(3, 2, figsize=(7, 9), height_ratios=[1, 1, 2])
axs = axs.flat

#  Varying density along a streamline
axs[0].streamplot(X, Y, U, V, density=[0.5, 1])
axs[0].set_title('Varying Density')

# Varying color along a streamline
strm = axs[1].streamplot(X, Y, U, V, color=U, linewidth=2, cmap='autumn')
fig.colorbar(strm.lines)
axs[1].set_title('Varying Color')

#  Varying line width along a streamline
lw = 5*speed / speed.max()
axs[2].streamplot(X, Y, U, V, density=0.6, color='k', linewidth=lw)
axs[2].set_title('Varying Line Width')

# Controlling the starting points of the streamlines
seed_points = np.array([[-2, -1, 0, 1, 2, -1], [-2, -1,  0, 1, 2, 2]])

strm = axs[3].streamplot(X, Y, U, V, color=U, linewidth=2,
                         cmap='autumn', start_points=seed_points.T)
fig.colorbar(strm.lines)
axs[3].set_title('Controlling Starting Points')

# Displaying the starting points with blue symbols.
axs[3].plot(seed_points[0], seed_points[1], 'bo')
axs[3].set(xlim=(-w, w), ylim=(-w, w))
```

(continues on next page)

(continued from previous page)

```
# Create a mask
mask = np.zeros(U.shape, dtype=bool)
mask[40:60, 40:60] = True
U[:20, :20] = np.nan
U = np.ma.array(U, mask=mask)

axs[4].streamplot(X, Y, U, V, color='r')
axs[4].set_title('Streamplot with Masking')

axs[4].imshow(~mask, extent=(-w, w, -w, w), alpha=0.5, cmap='gray',
              aspect='auto')
axs[4].set_aspect('equal')

axs[5].streamplot(X, Y, U, V, broken_streamlines=False)
axs[5].set_title('Streamplot with unbroken streamlines')

plt.tight_layout()
plt.show()
```


Varying Density
Varying Color
Varying Line Width
Controlling Starting Points
Streamplot with Masking
Streamplot with unbroken streamlines

**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.streamplot* / *matplotlib.pyplot.streamplot*
- *matplotlib.gridspec.GridSpec*

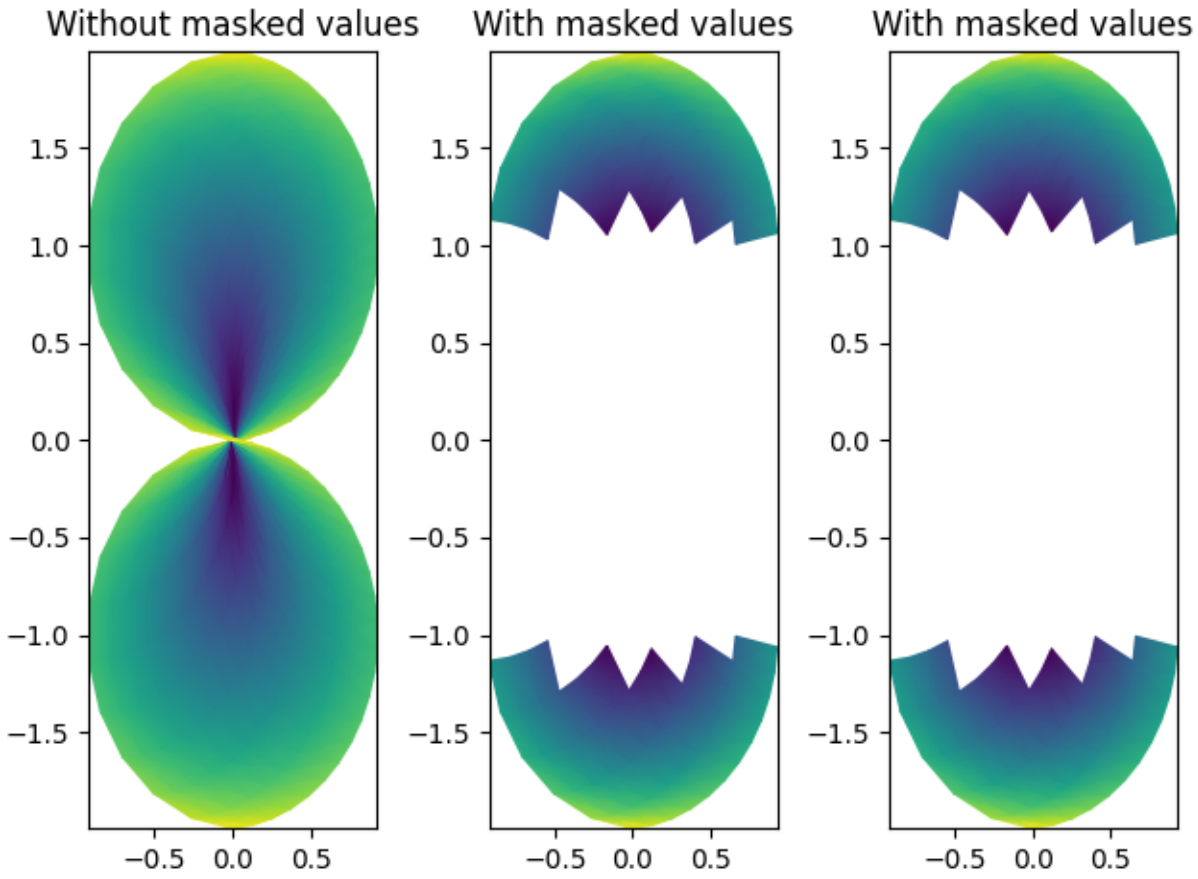
**Total running time of the script:** (0 minutes 4.064 seconds)

### QuadMesh Demo

*pcolormesh* uses a *QuadMesh*, a faster generalization of *pcolor*, but with some restrictions.

This demo illustrates a bug in quadmesh with masked data.

```
import numpy as np

from matplotlib import pyplot as plt

n = 12
x = np.linspace(-1.5, 1.5, n)
y = np.linspace(-1.5, 1.5, n * 2)
X, Y = np.meshgrid(x, y)
Qx = np.cos(Y) - np.cos(X)
Qz = np.sin(Y) + np.sin(X)
Z = np.sqrt(X**2 + Y**2) / 5
Z = (Z - Z.min()) / (Z.max() - Z.min())

# The color array can include masked values.
Zm = np.ma.masked_where(np.abs(Qz) < 0.5 * np.max(Qz), Z)

fig, axs = plt.subplots(nrows=1, ncols=3)
axs[0].pcolormesh(Qx, Qz, Z, shading='gouraud')
axs[0].set_title('Without masked values')

# You can control the color of the masked region.
cmap = plt.colormaps[plt.rcParams['image.cmap']].with_extremes(bad='y')
axs[1].pcolormesh(Qx, Qz, Zm, shading='gouraud', cmap=cmap)
axs[1].set_title('With masked values')

# Or use the default, which is transparent.
axs[2].pcolormesh(Qx, Qz, Zm, shading='gouraud')
axs[2].set_title('With masked values')

fig.tight_layout()
plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.pcolormesh* / *matplotlib.pyplot.pcolormesh*

**Total running time of the script:** (0 minutes 1.020 seconds)

### Advanced quiver and quiverkey functions

Demonstrates some more advanced options for *quiver*. For a simple example refer to *Quiver Simple Demo*.

Note: The plot autoscaling does not take into account the arrows, so those on the boundaries may reach out of the picture. This is not an easy problem to solve in a perfectly general way. The recommended workaround is to manually set the Axes limits in such a case.

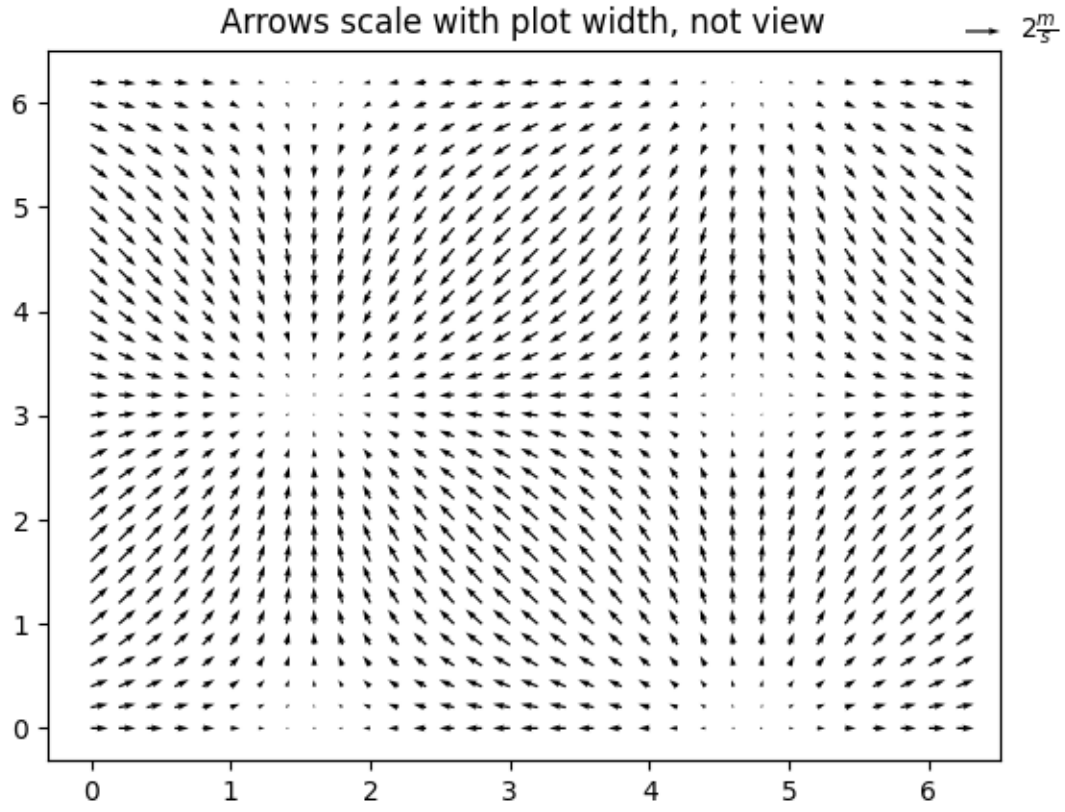
```
import matplotlib.pyplot as plt
import numpy as np

X, Y = np.meshgrid(np.arange(0, 2 * np.pi, .2), np.arange(0, 2 * np.pi, .2))
```

(continues on next page)

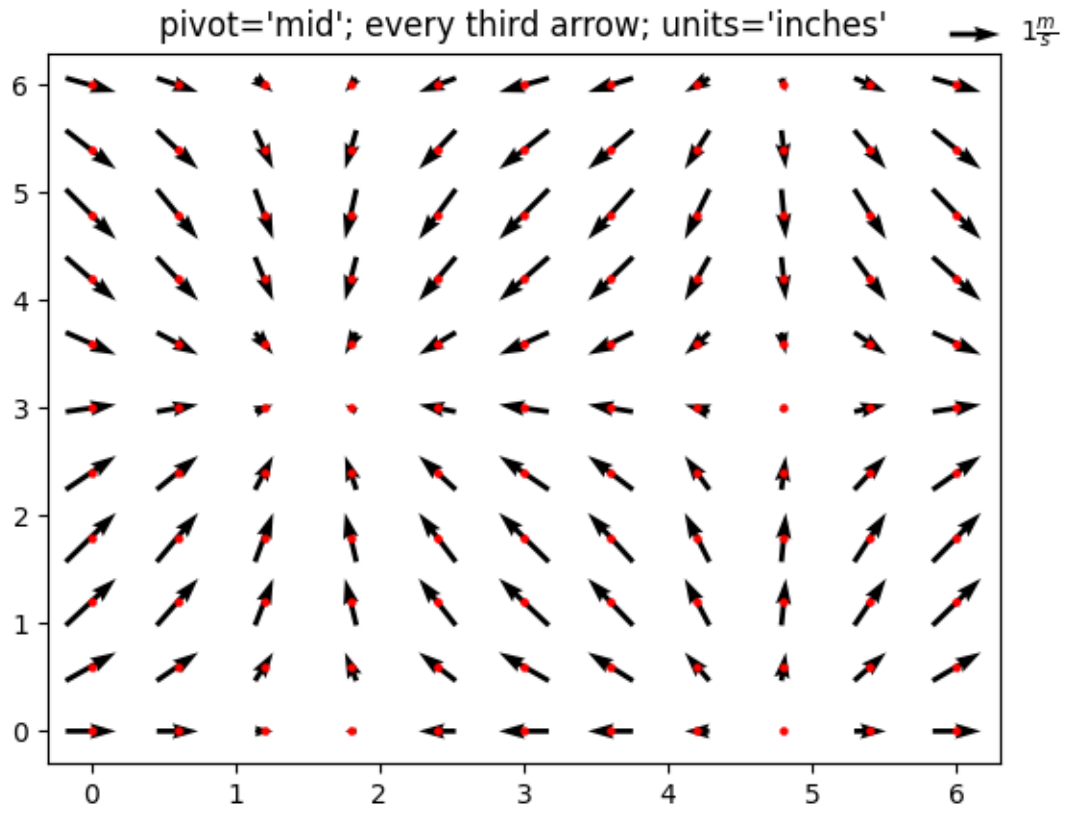
(continued from previous page)

```
U = np.cos(X)
V = np.sin(Y)
```

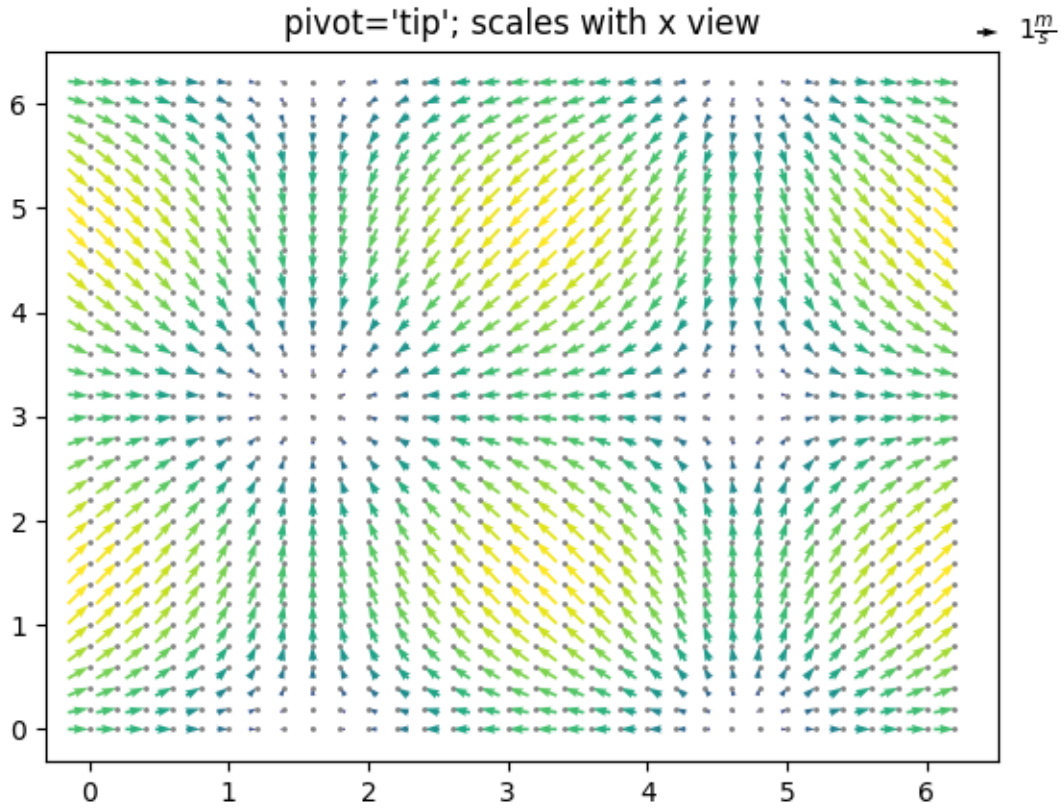
```
fig1, ax1 = plt.subplots()
ax1.set_title('Arrows scale with plot width, not view')
Q = ax1.quiver(X, Y, U, V, units='width')
qk = ax1.quiverkey(Q, 0.9, 0.9, 2, r'$2 \frac{m}{s}$', labelpos='E',
                   coordinates='figure')
```




```
fig2, ax2 = plt.subplots()
ax2.set_title("pivot='mid'; every third arrow; units='inches'")
Q = ax2.quiver(X[::3, ::3], Y[::3, ::3], U[::3, ::3], V[::3, ::3],
               pivot='mid', units='inches')
qk = ax2.quiverkey(Q, 0.9, 0.9, 1, r'$1 \frac{m}{s}$', labelpos='E',
                   coordinates='figure')
ax2.scatter(X[::3, ::3], Y[::3, ::3], color='r', s=5)
```




```
fig3, ax3 = plt.subplots()
ax3.set_title("pivot='tip'; scales with x view")
M = np.hypot(U, V)
Q = ax3.quiver(X, Y, U, V, M, units='x', pivot='tip', width=0.022,
               scale=1 / 0.15)
qk = ax3.quiverkey(Q, 0.9, 0.9, 1, r'$1 \frac{m}{s}$', labelpos='E',
                   coordinates='figure')
ax3.scatter(X, Y, color='0.5', s=1)

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.quiver* / *matplotlib.pyplot.quiver*
- *matplotlib.axes.Axes.quiverkey* / *matplotlib.pyplot.quiverkey*

**Total running time of the script:** (0 minutes 2.156 seconds)

## Quiver Simple Demo

A simple example of a *quiver* plot with a *quiverkey*.

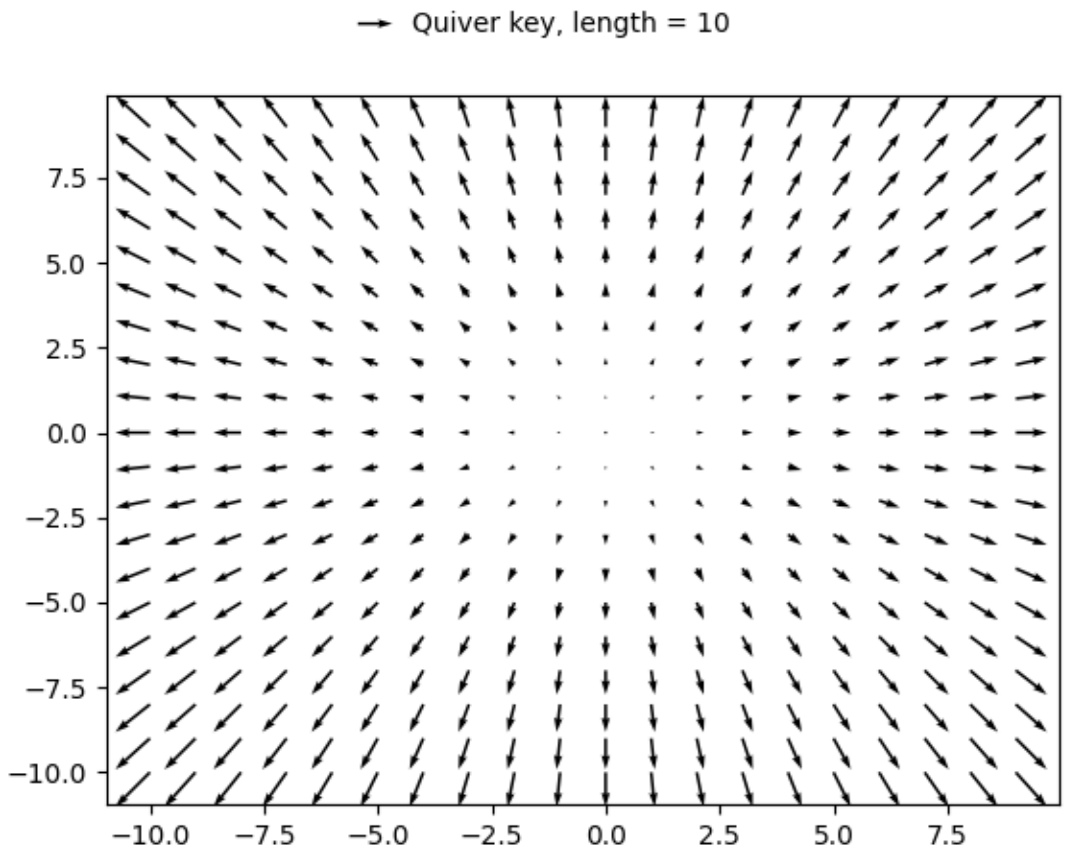
For more advanced options refer to *Advanced quiver and quiverkey functions*.

```
import matplotlib.pyplot as plt
import numpy as np

X = np.arange(-10, 10, 1)
Y = np.arange(-10, 10, 1)
```

(continues on next page)

(continued from previous page)

```
U, V = np.meshgrid(X, Y)

fig, ax = plt.subplots()
q = ax.quiver(X, Y, U, V)
ax.quiverkey(q, X=0.3, Y=1.1, U=10,
             label='Quiver key, length = 10', labelpos='E')

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.quiver* / *matplotlib.pyplot.quiver*
- *matplotlib.axes.Axes.quiverkey* / *matplotlib.pyplot.quiverkey*

### Shading example

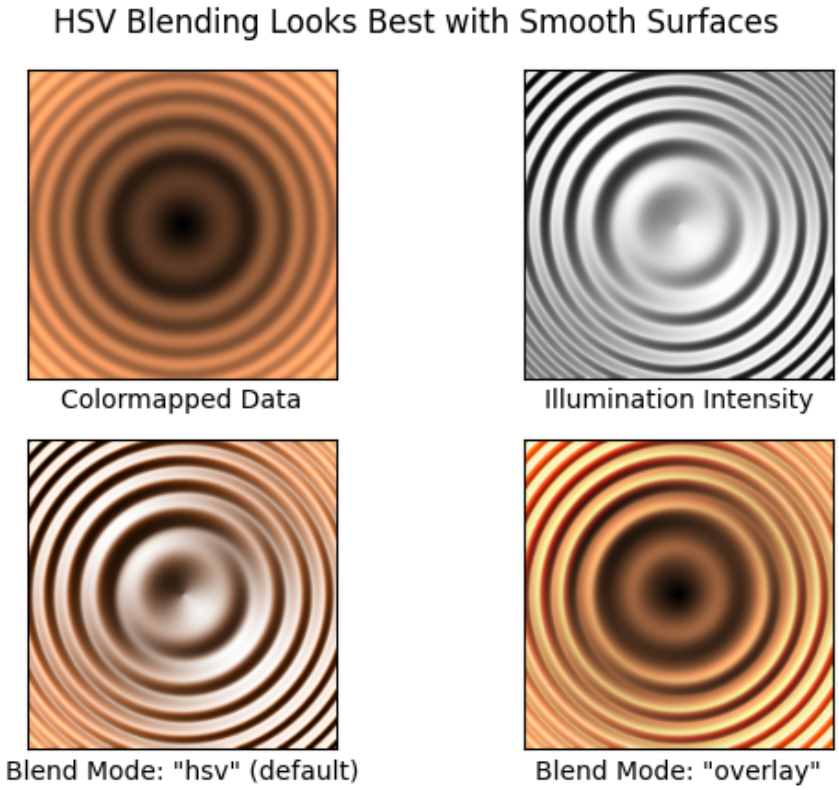
Example showing how to make shaded relief plots like Mathematica or Generic Mapping Tools.

```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib import cbook
from matplotlib.colors import LightSource


def main():
    # Test data
    x, y = np.mgrid[-5:5:0.05, -5:5:0.05]
    z = 5 * (np.sqrt(x**2 + y**2) + np.sin(x**2 + y**2))

    dem = cbook.get_sample_data('jacksboro_fault_dem.npz')
    elev = dem['elevation']

    fig = compare(z, plt.cm.copper)
    fig.suptitle('HSV Blending Looks Best with Smooth Surfaces', y=0.95)

    fig = compare(elev, plt.cm.gist_earth, ve=0.05)
    fig.suptitle('Overlay Blending Looks Best with Rough Surfaces', y=0.95)

    plt.show()


def compare(z, cmap, ve=1):
    # Create subplots and hide ticks
    fig, axs = plt.subplots(ncols=2, nrows=2)
    for ax in axs.flat:
        ax.set(xticks=[], yticks=[])

    # Illuminate the scene from the northwest
    ls = LightSource(azdeg=315, altdeg=45)

    axs[0, 0].imshow(z, cmap=cmap)
    axs[0, 0].set(xlabel='Colormapped Data')

    axs[0, 1].imshow(ls.hillshade(z, vert_exag=ve), cmap='gray')
    axs[0, 1].set(xlabel='Illumination Intensity')

    rgb = ls.shade(z, cmap=cmap, vert_exag=ve, blend_mode='hsv')
    axs[1, 0].imshow(rgb)
    axs[1, 0].set(xlabel='Blend Mode: "hsv" (default)')

    rgb = ls.shade(z, cmap=cmap, vert_exag=ve, blend_mode='overlay')
    axs[1, 1].imshow(rgb)
    axs[1, 1].set(xlabel='Blend Mode: "overlay"')

    return fig
```

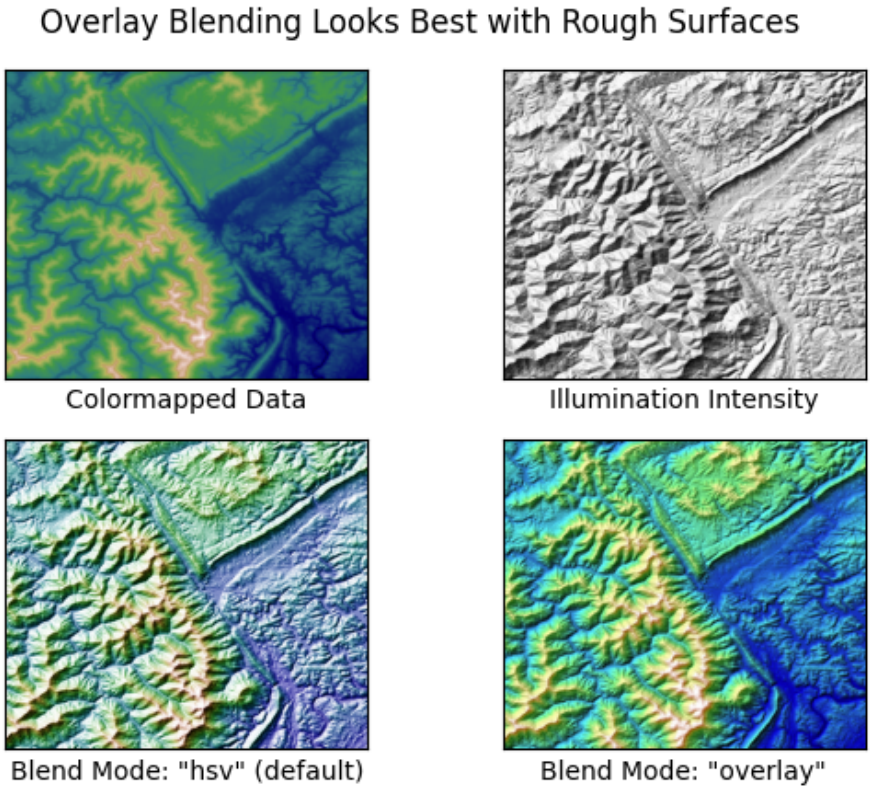
(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':
    main()
```




-




- 

**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.colors.LightSource*
- *matplotlib.axes.Axes.imshow* / *matplotlib.pyplot.imshow*

**Total running time of the script:** (0 minutes 1.095 seconds)

## Spectrogram

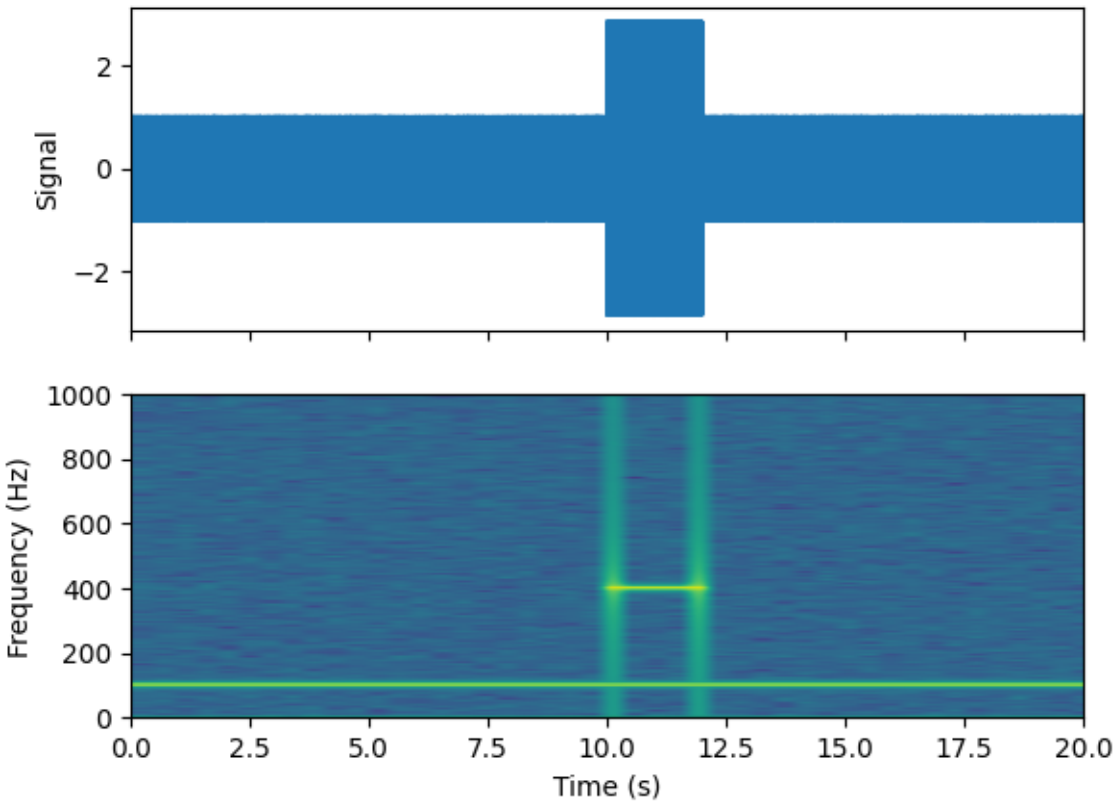
Plotting a spectrogram using *specgram*.

```
import matplotlib.pyplot as plt
import numpy as np

# Fixing random state for reproducibility
np.random.seed(19680801)

dt = 0.0005
t = np.arange(0.0, 20.5, dt)
```

(continues on next page)

(continued from previous page)

```
s1 = np.sin(2 * np.pi * 100 * t)
s2 = 2 * np.sin(2 * np.pi * 400 * t)

# create a transient "chirp"
s2[t <= 10] = s2[12 <= t] = 0

# add some noise into the mix
nse = 0.01 * np.random.random(size=len(t))

x = s1 + s2 + nse  # the signal
NFFT = 1024  # the length of the windowing segments
Fs = 1/dt  # the sampling frequency

fig, (ax1, ax2) = plt.subplots(nrows=2, sharex=True)
ax1.plot(t, x)
ax1.set_ylabel('Signal')

Pxx, freqs, bins, im = ax2.specgram(x, NFFT=NFFT, Fs=Fs)
# The `specgram` method returns 4 objects. They are:
# - Pxx: the periodogram
# - freqs: the frequency vector
# - bins: the centers of the time bins
# - im: the .image.AxesImage instance representing the data in the plot
ax2.set_xlabel('Time (s)')
ax2.set_ylabel('Frequency (Hz)')
ax2.set_xlim(0, 20)

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.specgram* / *matplotlib.pyplot.specgram*

## Spy Demos

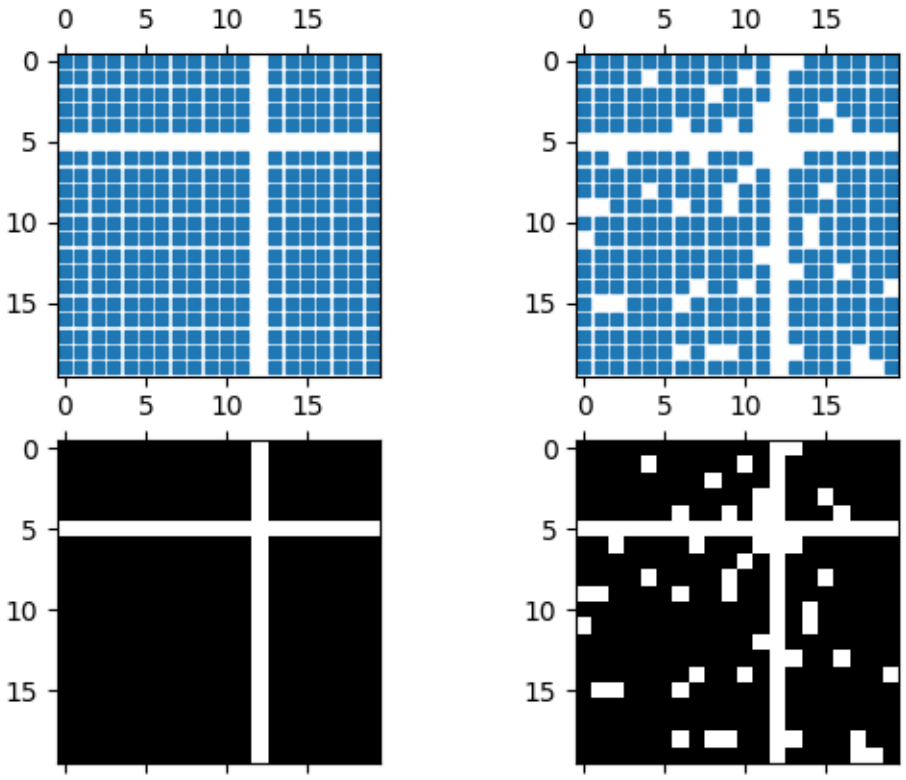
Plot the sparsity pattern of arrays.

```
import matplotlib.pyplot as plt
import numpy as np

# Fixing random state for reproducibility
np.random.seed(19680801)

fig, axs = plt.subplots(2, 2)
ax1 = axs[0, 0]
ax2 = axs[0, 1]
ax3 = axs[1, 0]
```

(continues on next page)

(continued from previous page)

```
ax4 = axs[1, 1]

x = np.random.randn(20, 20)
x[5, :] = 0.
x[:, 12] = 0.

ax1.spy(x, markersize=5)
ax2.spy(x, precision=0.1, markersize=5)

ax3.spy(x)
ax4.spy(x, precision=0.1)

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.spy* / *matplotlib.pyplot.spy*

### Tricontour Demo

Contour plots of unstructured triangular grids.

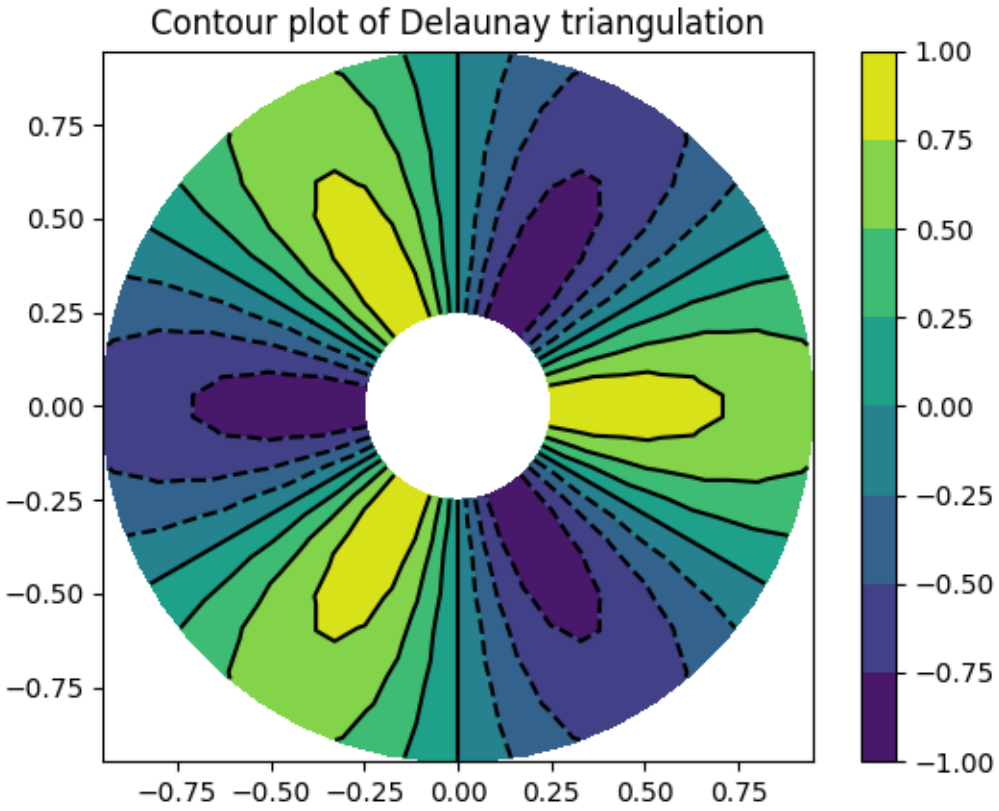
```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib.tri as tri
```

Creating a Triangulation without specifying the triangles results in the Delaunay triangulation of the points.

```
# First create the x and y coordinates of the points.
n_angles = 48
n_radii = 8
min_radius = 0.25
radii = np.linspace(min_radius, 0.95, n_radii)

angles = np.linspace(0, 2 * np.pi, n_angles, endpoint=False)
angles = np.repeat(angles[..., np.newaxis], n_radii, axis=1)
angles[:, 1::2] += np.pi / n_angles

x = (radii * np.cos(angles)).flatten()
y = (radii * np.sin(angles)).flatten()
z = (np.cos(radii) * np.cos(3 * angles)).flatten()

# Create the Triangulation; no triangles so Delaunay triangulation created.
triang = tri.Triangulation(x, y)

# Mask off unwanted triangles.
triang.set_mask(np.hypot(x[triang.triangles].mean(axis=1),
                         y[triang.triangles].mean(axis=1))
                < min_radius)
```

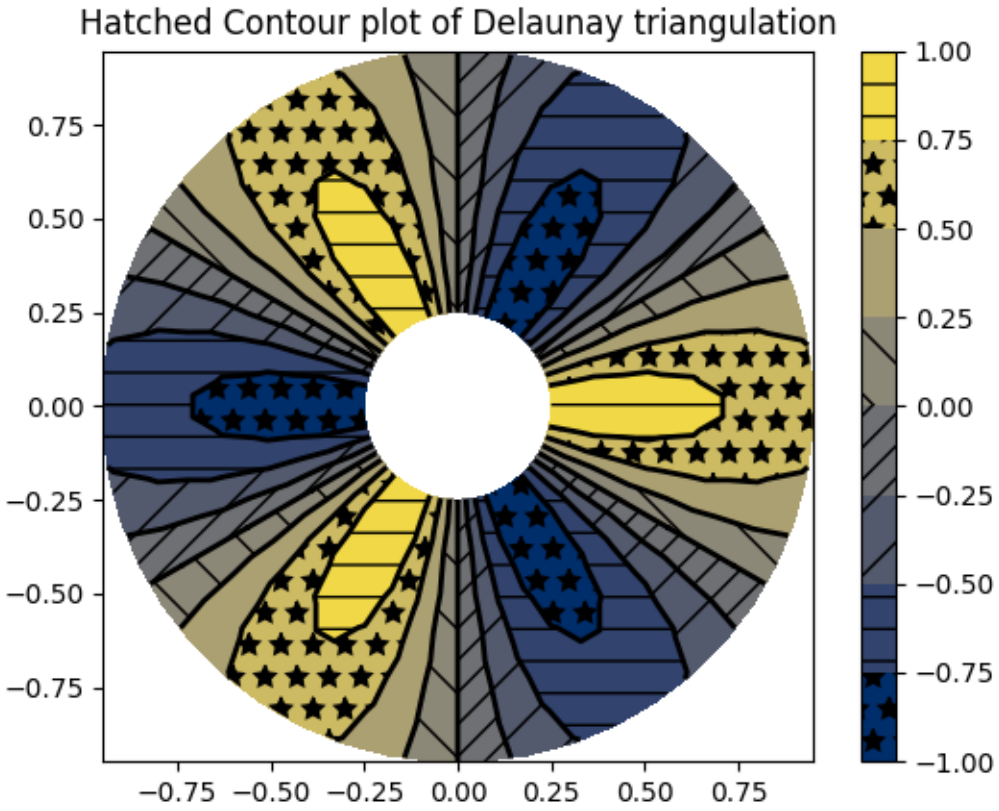
pcolor plot.

```
fig1, ax1 = plt.subplots()
ax1.set_aspect('equal')
tcf = ax1.tricontourf(triang, z)
fig1.colorbar(tcf)
ax1.tricontour(triang, z, colors='k')
ax1.set_title('Contour plot of Delaunay triangulation')
```



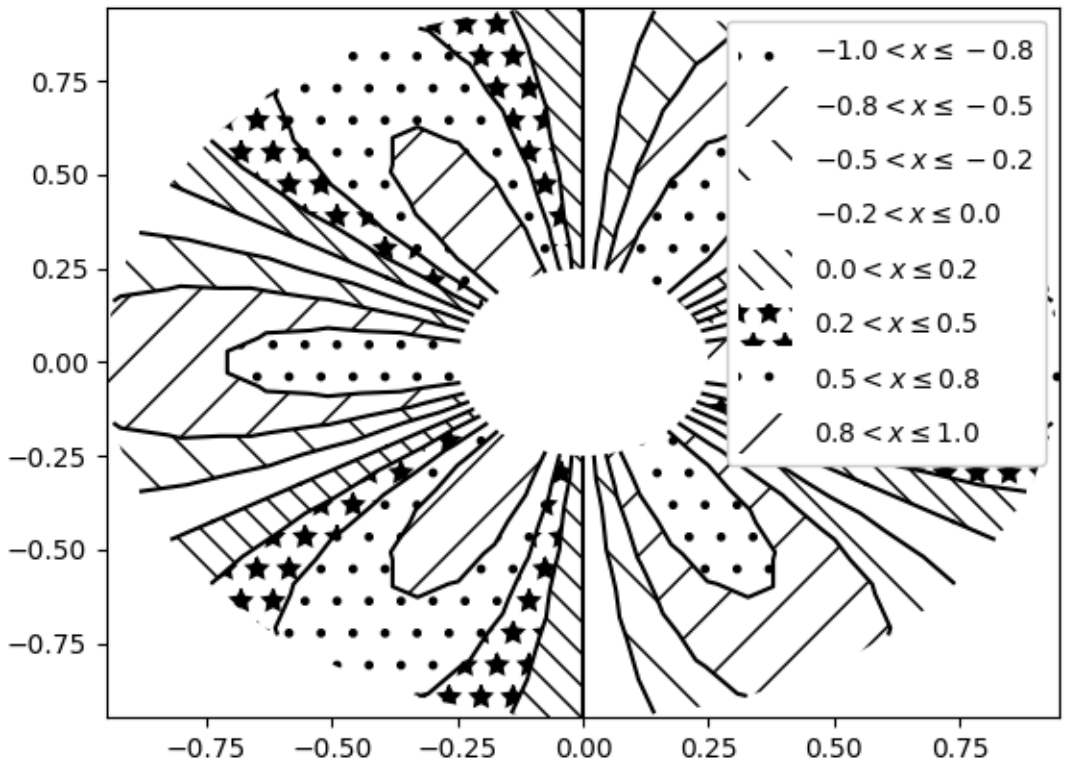

You could also specify hatching patterns along with different cmaps.

```
fig2, ax2 = plt.subplots()
ax2.set_aspect("equal")
tcf = ax2.tricontourf(
    triang,
    z,
    hatches=["*", "-", "/", "//", "\\", None],
    cmap="cividis"
)
fig2.colorbar(tcf)
ax2.tricontour(triang, z, linestyles="solid", colors="k", linewidths=2.0)
ax2.set_title("Hatched Contour plot of Delaunay triangulation")
```




You could also generate hatching patterns labeled with no color.

```
fig3, ax3 = plt.subplots()
n_levels = 7
tcf = ax3.tricontourf(
    triang,
    z,
    n_levels,
    colors="none",
    hatches=[".", "/", "\\", None, "\\\\", "*"],
)
ax3.tricontour(triang, z, n_levels, colors="black", linestyles="-")


# create a legend for the contour set
artists, labels = tcf.legend_elements(str_format="{:2.1f}".format)
ax3.legend(artists, labels, handleheight=2, framealpha=1)
```




You can specify your own triangulation rather than perform a Delaunay triangulation of the points, where each triangle is given by the indices of the three points that make up the triangle, ordered in either a clockwise or anticlockwise manner.

```
xy = np.asarray([
    [-0.101, 0.872], [-0.080, 0.883], [-0.069, 0.888], [-0.054, 0.890],
    [-0.045, 0.897], [-0.057, 0.895], [-0.073, 0.900], [-0.087, 0.898],
    [-0.090, 0.904], [-0.069, 0.907], [-0.069, 0.921], [-0.080, 0.919],
    [-0.073, 0.928], [-0.052, 0.930], [-0.048, 0.942], [-0.062, 0.949],
    [-0.054, 0.958], [-0.069, 0.954], [-0.087, 0.952], [-0.087, 0.959],
    [-0.080, 0.966], [-0.085, 0.973], [-0.087, 0.965], [-0.097, 0.965],
    [-0.097, 0.975], [-0.092, 0.984], [-0.101, 0.980], [-0.108, 0.980],
    [-0.104, 0.987], [-0.102, 0.993], [-0.115, 1.001], [-0.099, 0.996],
    [-0.101, 1.007], [-0.090, 1.010], [-0.087, 1.021], [-0.069, 1.021],
    [-0.052, 1.022], [-0.052, 1.017], [-0.069, 1.010], [-0.064, 1.005],
    [-0.048, 1.005], [-0.031, 1.005], [-0.031, 0.996], [-0.040, 0.987],
    [-0.045, 0.980], [-0.052, 0.975], [-0.040, 0.973], [-0.026, 0.968],
    [-0.020, 0.954], [-0.006, 0.947], [ 0.003, 0.935], [ 0.006, 0.926],
    [ 0.005, 0.921], [ 0.022, 0.923], [ 0.033, 0.912], [ 0.029, 0.905],
    [ 0.017, 0.900], [ 0.012, 0.895], [ 0.027, 0.893], [ 0.019, 0.886],
    [ 0.001, 0.883], [-0.012, 0.884], [-0.029, 0.883], [-0.038, 0.879],
    [-0.057, 0.881], [-0.062, 0.876], [-0.078, 0.876], [-0.087, 0.872],
    [-0.030, 0.907], [-0.007, 0.905], [-0.057, 0.916], [-0.025, 0.933],
```

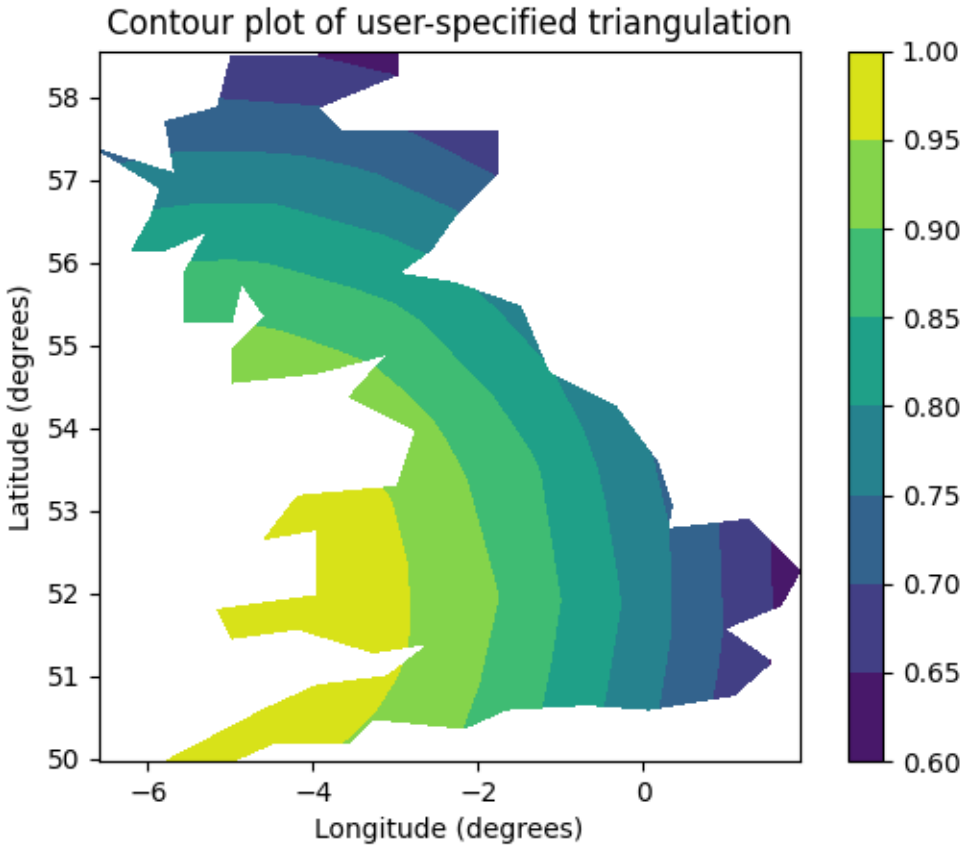
(continues on next page)

(continued from previous page)

```
    [-0.077, 0.990], [-0.059, 0.993]])
x = np.degrees(xy[:, 0])
y = np.degrees(xy[:, 1])
x0 = -5
y0 = 52
z = np.exp(-0.01 * ((x - x0) ** 2 + (y - y0) ** 2))

triangles = np.asarray([
    [67, 66,  1], [65,  2, 66], [ 1, 66,  2], [64,  2, 65], [63,  3, 64],
    [60, 59, 57], [ 2, 64,  3], [ 3, 63,  4], [ 0, 67,  1], [62,  4, 63],
    [57, 59, 56], [59, 58, 56], [61, 60, 69], [57, 69, 60], [ 4, 62, 68],
    [ 6,  5,  9], [61, 68, 62], [69, 68, 61], [ 9,  5, 70], [ 6,  8,  7],
    [ 4, 70,  5], [ 8,  6,  9], [56, 69, 57], [69, 56, 52], [70, 10,  9],
    [54, 53, 55], [56, 55, 53], [68, 70,  4], [52, 56, 53], [11, 10, 12],
    [69, 71, 68], [68, 13, 70], [10, 70, 13], [51, 50, 52], [13, 68, 71],
    [52, 71, 69], [12, 10, 13], [71, 52, 50], [71, 14, 13], [50, 49, 71],
    [49, 48, 71], [14, 16, 15], [14, 71, 48], [17, 19, 18], [17, 20, 19],
    [48, 16, 14], [48, 47, 16], [47, 46, 16], [16, 46, 45], [23, 22, 24],
    [21, 24, 22], [17, 16, 45], [20, 17, 45], [21, 25, 24], [27, 26, 28],
    [20, 72, 21], [25, 21, 72], [45, 72, 20], [25, 28, 26], [44, 73, 45],
    [72, 45, 73], [28, 25, 29], [29, 25, 31], [43, 73, 44], [73, 43, 40],
    [72, 73, 39], [72, 31, 25], [42, 40, 43], [31, 30, 29], [39, 73, 40],
    [42, 41, 40], [72, 33, 31], [32, 31, 33], [39, 38, 72], [33, 72, 38],
    [33, 38, 34], [37, 35, 38], [34, 38, 35], [35, 37, 36]])
```

Rather than create a Triangulation object, can simply pass x, y and triangles arrays to tripcolor directly. It would be better to use a Triangulation object if the same triangulation was to be used more than once to save duplicated calculations.

```
fig4, ax4 = plt.subplots()
ax4.set_aspect('equal')
tcf = ax4.tricontourf(x, y, triangles, z)
fig4.colorbar(tcf)
ax4.set_title('Contour plot of user-specified triangulation')
ax4.set_xlabel('Longitude (degrees)')
ax4.set_ylabel('Latitude (degrees)')

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.tricontourf* / *matplotlib.pyplot.tricontourf*
- *matplotlib.tri.Triangulation*
- *matplotlib.figure.Figure.colorbar* / *matplotlib.pyplot.colorbar*
- *matplotlib.axes.Axes.legend* / *matplotlib.pyplot.legend*
- *matplotlib.contour.ContourSet.legend_elements*

**Total running time of the script:** (0 minutes 1.742 seconds)

### Tricontour Smooth Delaunay

Demonstrates high-resolution tricontouring of a random set of points; a *matplotlib.tri.TriAnalyzer* is used to improve the plot quality.

The initial data points and triangular grid for this demo are:

- a set of random points is instantiated, inside [-1, 1] x [-1, 1] square
- A Delaunay triangulation of these points is then computed, of which a random subset of triangles is masked out by the user (based on *init_mask_frac* parameter). This simulates invalidated data.

The proposed generic procedure to obtain a high resolution contouring of such a data set is the following:

1. Compute an extended mask with a *matplotlib.tri.TriAnalyzer*, which will exclude badly shaped (flat) triangles from the border of the triangulation. Apply the mask to the triangulation (using set_mask).
2. Refine and interpolate the data using a *matplotlib.tri.UniformTriRefiner*.
3. Plot the refined data with *tricontour*.

```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.tri import TriAnalyzer, Triangulation, UniformTriRefiner


# -----------------------------------------------------------------------------
# Analytical test function
# -----------------------------------------------------------------------------
def experiment_res(x, y):
    """An analytic function representing experiment results."""
    x = 2 * x
    r1 = np.sqrt((0.5 - x)**2 + (0.5 - y)**2)
    theta1 = np.arctan2(0.5 - x, 0.5 - y)
    r2 = np.sqrt((-x - 0.2)**2 + (-y - 0.2)**2)
    theta2 = np.arctan2(-x - 0.2, -y - 0.2)
    z = (4 * (np.exp((r1/10)**2) - 1) * 30 * np.cos(3 * theta1) +
         (np.exp((r2/10)**2) - 1) * 30 * np.cos(5 * theta2) +
         2 * (x**2 + y**2))
    return (np.max(z) - z) / (np.max(z) - np.min(z))

# -----------------------------------------------------------------------------
# Generating the initial data test points and triangulation for the demo
# -----------------------------------------------------------------------------
# User parameters for data test points

# Number of test data points, tested from 3 to 5000 for subdiv=3
n_test = 200

# Number of recursive subdivisions of the initial mesh for smooth plots.
# Values >3 might result in a very high number of triangles for the refine
# mesh: new triangles numbering = (4**subdiv)*ntri
```

(continues on next page)

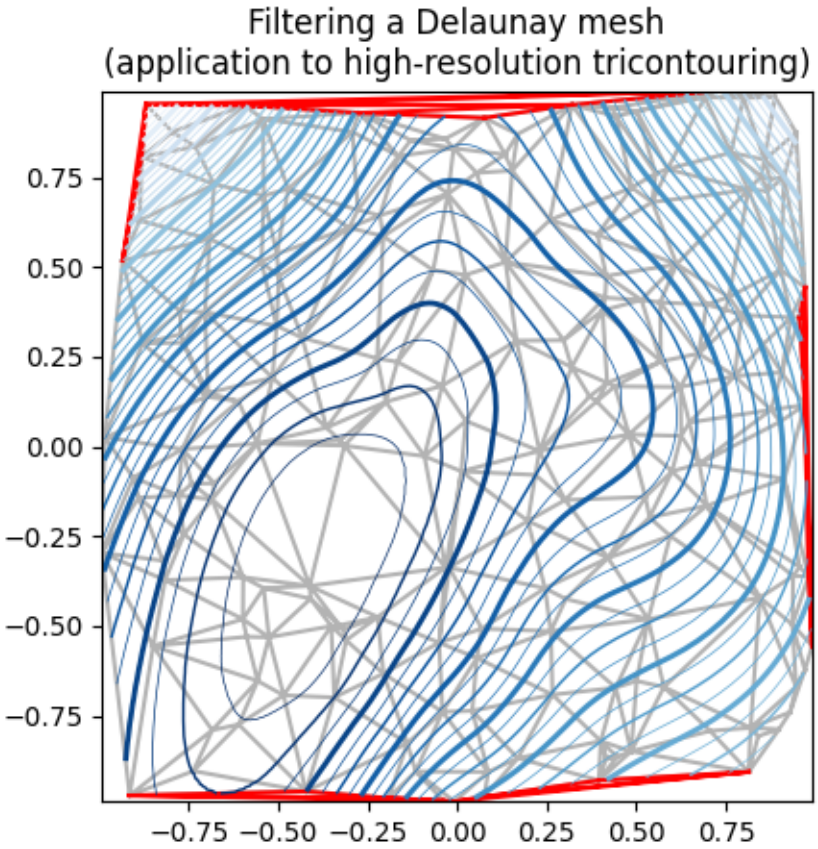
(continued from previous page)

```
subdiv = 3

# Float > 0. adjusting the proportion of (invalid) initial triangles which␣
 ↪will
# be masked out. Enter 0 for no mask.
init_mask_frac = 0.0

# Minimum circle ratio - border triangles with circle ratio below this will be
# masked if they touch a border. Suggested value 0.01; use -1 to keep all
# triangles.
min_circle_ratio = .01

# Random points
random_gen = np.random.RandomState(seed=19680801)
x_test = random_gen.uniform(-1., 1., size=n_test)
y_test = random_gen.uniform(-1., 1., size=n_test)
z_test = experiment_res(x_test, y_test)

# meshing with Delaunay triangulation
tri = Triangulation(x_test, y_test)
ntri = tri.triangles.shape[0]

# Some invalid data are masked out
mask_init = np.zeros(ntri, dtype=bool)
masked_tri = random_gen.randint(0, ntri, int(ntri * init_mask_frac))
mask_init[masked_tri] = True
tri.set_mask(mask_init)


# -----------------------------------------------------------------------------
# Improving the triangulation before high-res plots: removing flat triangles
# -----------------------------------------------------------------------------
# masking badly shaped triangles at the border of the triangular mesh.
mask = TriAnalyzer(tri).get_flat_tri_mask(min_circle_ratio)
tri.set_mask(mask)

# refining the data
refiner = UniformTriRefiner(tri)
tri_refi, z_test_refi = refiner.refine_field(z_test, subdiv=subdiv)

# analytical 'results' for comparison
z_expected = experiment_res(tri_refi.x, tri_refi.y)

# for the demo: loading the 'flat' triangles for plot
flat_tri = Triangulation(x_test, y_test)
flat_tri.set_mask(~mask)


# -----------------------------------------------------------------------------
# Now the plots
# -----------------------------------------------------------------------------
# User options for plots
```

(continues on next page)

(continued from previous page)

```
plot_tri = True          # plot of base triangulation
plot_masked_tri = True   # plot of excessively flat excluded triangles
plot_refi_tri = False    # plot of refined triangulation
plot_expected = False    # plot of analytical function values for comparison


# Graphical options for tricontouring
levels = np.arange(0., 1., 0.025)

fig, ax = plt.subplots()
ax.set_aspect('equal')
ax.set_title("Filtering a Delaunay mesh\n"
             "(application to high-resolution tricontouring)")

# 1) plot of the refined (computed) data contours:
ax.tricontour(tri_refi, z_test_refi, levels=levels, cmap='Blues',
              linewidths=[2.0, 0.5, 1.0, 0.5])
# 2) plot of the expected (analytical) data contours (dashed):
if plot_expected:
    ax.tricontour(tri_refi, z_expected, levels=levels, cmap='Blues',
                  linestyles='--')
# 3) plot of the fine mesh on which interpolation was done:
if plot_refi_tri:
    ax.triplot(tri_refi, color='0.97')
# 4) plot of the initial 'coarse' mesh:
if plot_tri:
    ax.triplot(tri, color='0.7')
# 4) plot of the unvalidated triangles from naive Delaunay Triangulation:
if plot_masked_tri:
    ax.triplot(flat_tri, color='red')

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.tricontour* / *matplotlib.pyplot.tricontour*
- *matplotlib.axes.Axes.tricontourf* / *matplotlib.pyplot.tricontourf*
- *matplotlib.axes.Axes.triplot* / *matplotlib.pyplot.triplot*
- *matplotlib.tri*
- *matplotlib.tri.Triangulation*
- *matplotlib.tri.TriAnalyzer*
- *matplotlib.tri.UniformTriRefiner*

### Tricontour Smooth User

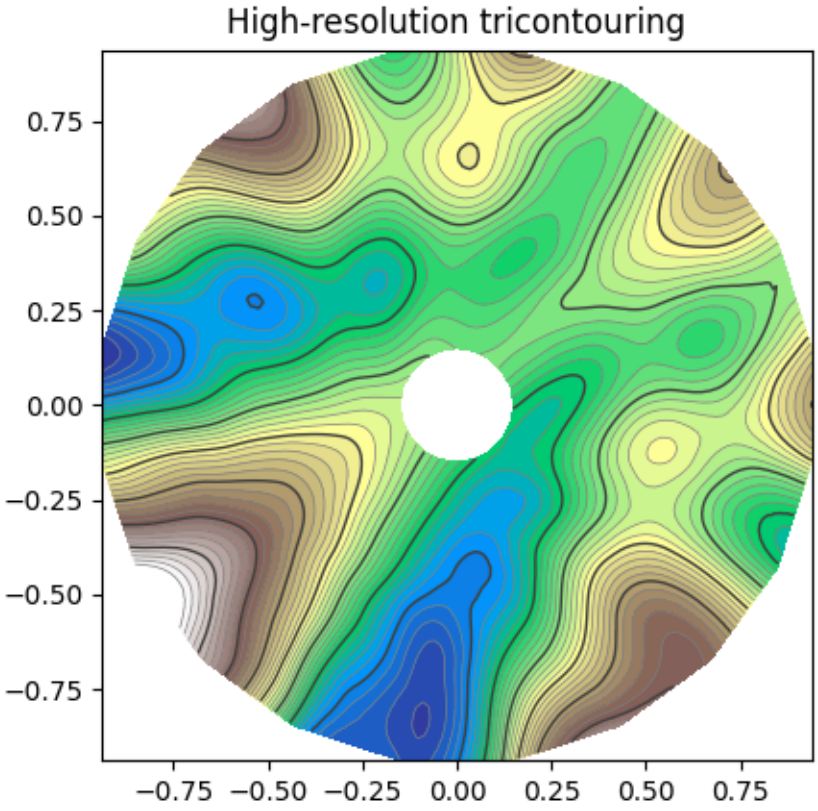
Demonstrates high-resolution tricontouring on user-defined triangular grids with *matplotlib.tri.UniformTriRefiner*.

```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib.tri as tri


# -----------------------------------------------------------------------------
# Analytical test function
# -----------------------------------------------------------------------------
def function_z(x, y):
    r1 = np.sqrt((0.5 - x)**2 + (0.5 - y)**2)
    theta1 = np.arctan2(0.5 - x, 0.5 - y)
    r2 = np.sqrt((-x - 0.2)**2 + (-y - 0.2)**2)
    theta2 = np.arctan2(-x - 0.2, -y - 0.2)
    z = -(2 * (np.exp((r1 / 10)**2) - 1) * 30. * np.cos(7. * theta1) +
          (np.exp((r2 / 10)**2) - 1) * 30. * np.cos(11. * theta2) +
          0.7 * (x**2 + y**2))
    return (np.max(z) - z) / (np.max(z) - np.min(z))

# -----------------------------------------------------------------------------
# Creating a Triangulation
# -----------------------------------------------------------------------------
# First create the x and y coordinates of the points.
n_angles = 20
n_radii = 10
min_radius = 0.15
radii = np.linspace(min_radius, 0.95, n_radii)

angles = np.linspace(0, 2 * np.pi, n_angles, endpoint=False)
angles = np.repeat(angles[..., np.newaxis], n_radii, axis=1)
angles[:, 1::2] += np.pi / n_angles

x = (radii * np.cos(angles)).flatten()
y = (radii * np.sin(angles)).flatten()
z = function_z(x, y)

# Now create the Triangulation.
# (Creating a Triangulation without specifying the triangles results in the
# Delaunay triangulation of the points.)
triang = tri.Triangulation(x, y)

# Mask off unwanted triangles.
triang.set_mask(np.hypot(x[triang.triangles].mean(axis=1),
                         y[triang.triangles].mean(axis=1))
                < min_radius)

# -----------------------------------------------------------------------------
# Refine data
```

(continues on next page)

(continued from previous page)

```
# -----------------------------------------------------------------------------
refiner = tri.UniformTriRefiner(triang)
tri_refi, z_test_refi = refiner.refine_field(z, subdiv=3)

# -----------------------------------------------------------------------------
# Plot the triangulation and the high-res iso-contours
# -----------------------------------------------------------------------------
fig, ax = plt.subplots()
ax.set_aspect('equal')
ax.triplot(triang, lw=0.5, color='white')

levels = np.arange(0., 1., 0.025)
ax.tricontourf(tri_refi, z_test_refi, levels=levels, cmap='terrain')
ax.tricontour(tri_refi, z_test_refi, levels=levels,
              colors=['0.25', '0.5', '0.5', '0.5', '0.5'],
              linewidths=[1.0, 0.5, 0.5, 0.5, 0.5])

ax.set_title("High-resolution tricontouring")

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.tricontour* / *matplotlib.pyplot.tricontour*
- *matplotlib.axes.Axes.tricontourf* / *matplotlib.pyplot.tricontourf*
- *matplotlib.tri*
- *matplotlib.tri.Triangulation*
- *matplotlib.tri.UniformTriRefiner*

### Trigradient Demo

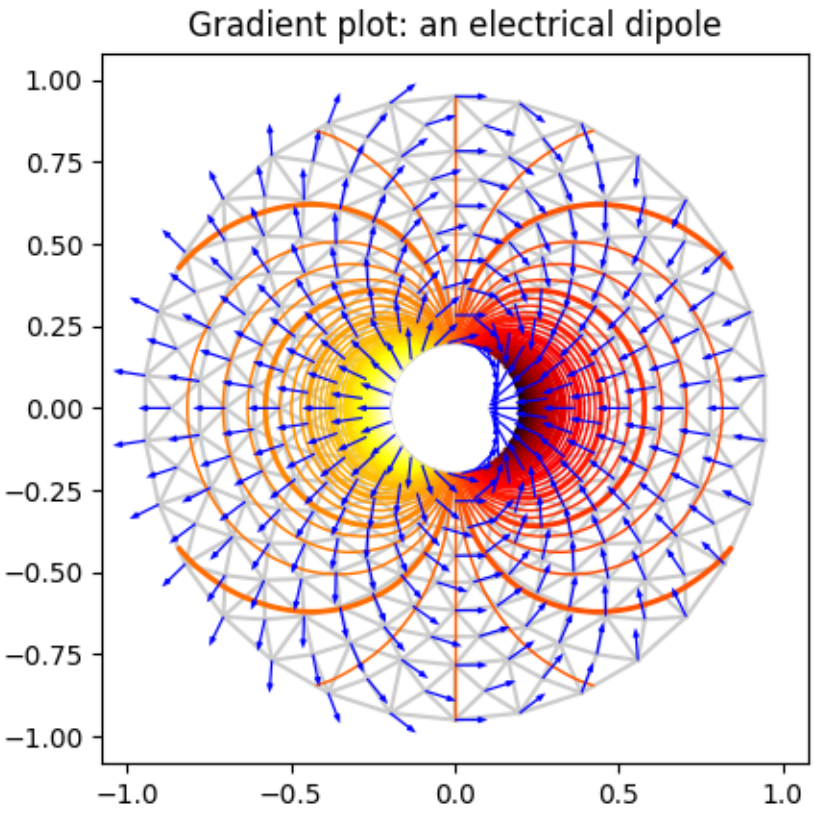
Demonstrates computation of gradient with *matplotlib.tri.CubicTriInterpolator*.

```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.tri import (CubicTriInterpolator, Triangulation,
                            UniformTriRefiner)


# -----------------------------------------------------------------------------
# Electrical potential of a dipole
# -----------------------------------------------------------------------------
def dipole_potential(x, y):
    """The electric dipole potential V, at position *x*, *y*."""
    r_sq = x**2 + y**2
    theta = np.arctan2(y, x)
    z = np.cos(theta)/r_sq
    return (np.max(z) - z) / (np.max(z) - np.min(z))


# -----------------------------------------------------------------------------
# Creating a Triangulation
# -----------------------------------------------------------------------------
# First create the x and y coordinates of the points.
n_angles = 30
n_radii = 10
min_radius = 0.2
radii = np.linspace(min_radius, 0.95, n_radii)

angles = np.linspace(0, 2 * np.pi, n_angles, endpoint=False)
angles = np.repeat(angles[..., np.newaxis], n_radii, axis=1)
angles[:, 1::2] += np.pi / n_angles

x = (radii*np.cos(angles)).flatten()
y = (radii*np.sin(angles)).flatten()
V = dipole_potential(x, y)

# Create the Triangulation; no triangles specified so Delaunay triangulation
```

(continues on next page)

(continued from previous page)

```
# created.
triang = Triangulation(x, y)

# Mask off unwanted triangles.
triang.set_mask(np.hypot(x[triang.triangles].mean(axis=1),
                         y[triang.triangles].mean(axis=1))
                < min_radius)

# -----------------------------------------------------------------------------
# Refine data - interpolates the electrical potential V
# -----------------------------------------------------------------------------
refiner = UniformTriRefiner(triang)
tri_refi, z_test_refi = refiner.refine_field(V, subdiv=3)

# -----------------------------------------------------------------------------
# Computes the electrical field (Ex, Ey) as gradient of electrical potential
# -----------------------------------------------------------------------------
tci = CubicTriInterpolator(triang, -V)
# Gradient requested here at the mesh nodes but could be anywhere else:
(Ex, Ey) = tci.gradient(triang.x, triang.y)
E_norm = np.sqrt(Ex**2 + Ey**2)

# -----------------------------------------------------------------------------
# Plot the triangulation, the potential iso-contours and the vector field
# -----------------------------------------------------------------------------
fig, ax = plt.subplots()
ax.set_aspect('equal')
# Enforce the margins, and enlarge them to give room for the vectors.
ax.use_sticky_edges = False
ax.margins(0.07)

ax.triplot(triang, color='0.8')

levels = np.arange(0., 1., 0.01)
ax.tricontour(tri_refi, z_test_refi, levels=levels, cmap='hot',
              linewidths=[2.0, 1.0, 1.0, 1.0])
# Plots direction of the electrical vector field
ax.quiver(triang.x, triang.y, Ex/E_norm, Ey/E_norm,
          units='xy', scale=10., zorder=3, color='blue',
          width=0.007, headwidth=3., headlength=4.)

ax.set_title('Gradient plot: an electrical dipole')
plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.tricontour* / *matplotlib.pyplot.tricontour*
- *matplotlib.axes.Axes.triplot* / *matplotlib.pyplot.triplot*
- *matplotlib.tri*
- *matplotlib.tri.Triangulation*
- *matplotlib.tri.CubicTriInterpolator*
- *matplotlib.tri.CubicTriInterpolator.gradient*
- *matplotlib.tri.UniformTriRefiner*
- *matplotlib.axes.Axes.quiver* / *matplotlib.pyplot.quiver*

### Triinterp Demo

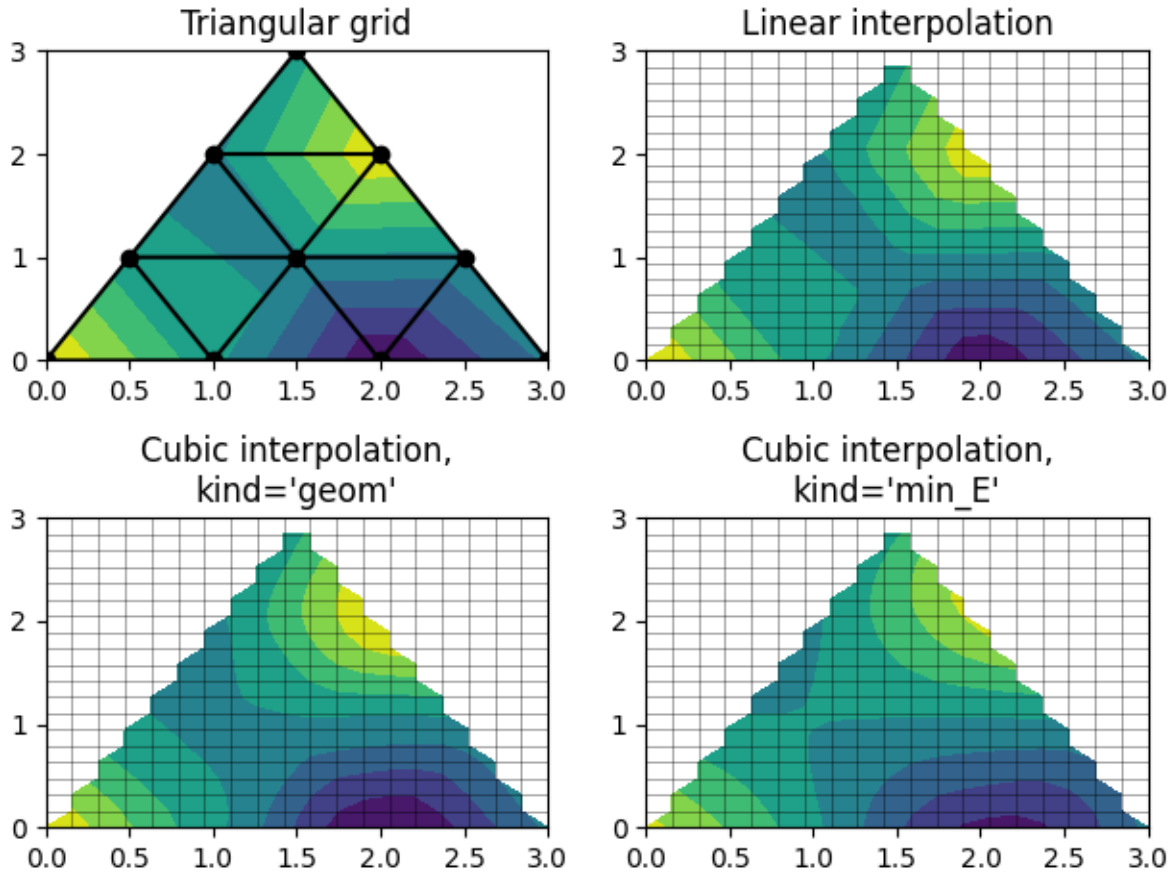
Interpolation from triangular grid to quad grid.

```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib.tri as mtri

# Create triangulation.
x = np.asarray([0, 1, 2, 3, 0.5, 1.5, 2.5, 1, 2, 1.5])
y = np.asarray([0, 0, 0, 0, 1.0, 1.0, 1.0, 2, 2, 3.0])
triangles = [[0, 1, 4], [1, 2, 5], [2, 3, 6], [1, 5, 4], [2, 6, 5], [4, 5, 7],
             [5, 6, 8], [5, 8, 7], [7, 8, 9]]
triang = mtri.Triangulation(x, y, triangles)

# Interpolate to regularly-spaced quad grid.
z = np.cos(1.5 * x) * np.cos(1.5 * y)
xi, yi = np.meshgrid(np.linspace(0, 3, 20), np.linspace(0, 3, 20))

interp_lin = mtri.LinearTriInterpolator(triang, z)
zi_lin = interp_lin(xi, yi)

interp_cubic_geom = mtri.CubicTriInterpolator(triang, z, kind='geom')
zi_cubic_geom = interp_cubic_geom(xi, yi)

interp_cubic_min_E = mtri.CubicTriInterpolator(triang, z, kind='min_E')
zi_cubic_min_E = interp_cubic_min_E(xi, yi)

# Set up the figure
fig, axs = plt.subplots(nrows=2, ncols=2)
axs = axs.flatten()

# Plot the triangulation.
axs[0].tricontourf(triang, z)
axs[0].triplot(triang, 'ko-')
axs[0].set_title('Triangular grid')

# Plot linear interpolation to quad grid.
axs[1].contourf(xi, yi, zi_lin)
axs[1].plot(xi, yi, 'k-', lw=0.5, alpha=0.5)
axs[1].plot(xi.T, yi.T, 'k-', lw=0.5, alpha=0.5)
axs[1].set_title("Linear interpolation")

# Plot cubic interpolation to quad grid, kind=geom
axs[2].contourf(xi, yi, zi_cubic_geom)
axs[2].plot(xi, yi, 'k-', lw=0.5, alpha=0.5)
axs[2].plot(xi.T, yi.T, 'k-', lw=0.5, alpha=0.5)
axs[2].set_title("Cubic interpolation,\nkind='geom'")

# Plot cubic interpolation to quad grid, kind=min_E
axs[3].contourf(xi, yi, zi_cubic_min_E)
axs[3].plot(xi, yi, 'k-', lw=0.5, alpha=0.5)
```

(continues on next page)

(continued from previous page)

```
axs[3].plot(xi.T, yi.T, 'k-', lw=0.5, alpha=0.5)
axs[3].set_title("Cubic interpolation,\nkind='min_E'")

fig.tight_layout()
plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.tricontourf* / *matplotlib.pyplot.tricontourf*
- *matplotlib.axes.Axes.triplot* / *matplotlib.pyplot.triplot*
- *matplotlib.axes.Axes.contourf* / *matplotlib.pyplot.contourf*
- *matplotlib.axes.Axes.plot* / *matplotlib.pyplot.plot*
- *matplotlib.tri*
- *matplotlib.tri.LinearTriInterpolator*
- *matplotlib.tri.CubicTriInterpolator*

- *matplotlib.tri.Triangulation*

### Tripcolor Demo

Pseudocolor plots of unstructured triangular grids.

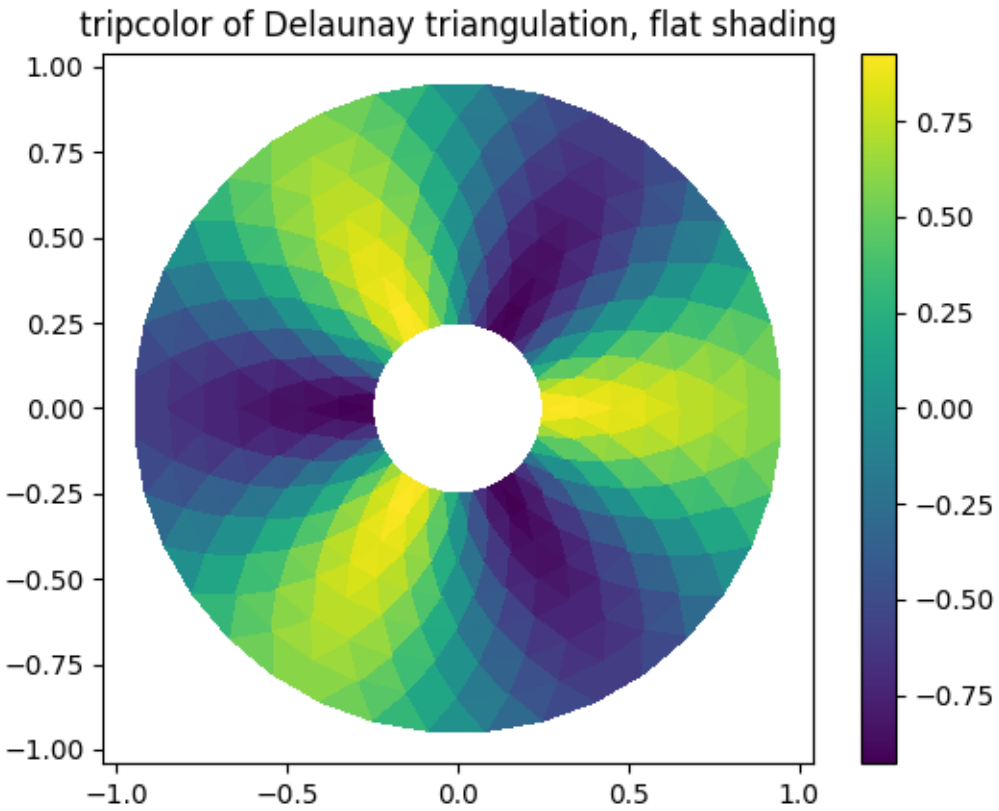
```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib.tri as tri
```

Creating a Triangulation without specifying the triangles results in the Delaunay triangulation of the points.

```
# First create the x and y coordinates of the points.
n_angles = 36
n_radii = 8
min_radius = 0.25
radii = np.linspace(min_radius, 0.95, n_radii)

angles = np.linspace(0, 2 * np.pi, n_angles, endpoint=False)
angles = np.repeat(angles[..., np.newaxis], n_radii, axis=1)
angles[:, 1::2] += np.pi / n_angles

x = (radii * np.cos(angles)).flatten()
y = (radii * np.sin(angles)).flatten()
z = (np.cos(radii) * np.cos(3 * angles)).flatten()

# Create the Triangulation; no triangles so Delaunay triangulation created.
triang = tri.Triangulation(x, y)

# Mask off unwanted triangles.
triang.set_mask(np.hypot(x[triang.triangles].mean(axis=1),
                         y[triang.triangles].mean(axis=1))
                < min_radius)
```

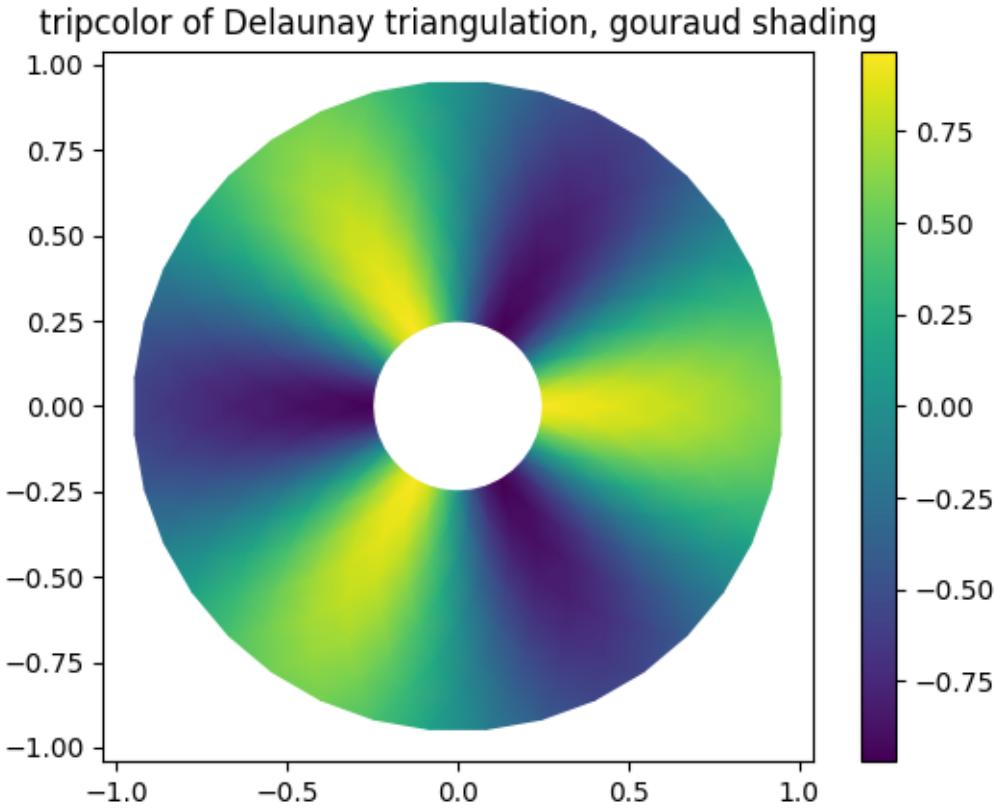
tripcolor plot.

```
fig1, ax1 = plt.subplots()
ax1.set_aspect('equal')
tpc = ax1.tripcolor(triang, z, shading='flat')
fig1.colorbar(tpc)
ax1.set_title('tripcolor of Delaunay triangulation, flat shading')
```




Illustrate Gouraud shading.

```
fig2, ax2 = plt.subplots()
ax2.set_aspect('equal')
tpc = ax2.tripcolor(triang, z, shading='gouraud')
fig2.colorbar(tpc)
ax2.set_title('tripcolor of Delaunay triangulation, gouraud shading')
```




You can specify your own triangulation rather than perform a Delaunay triangulation of the points, where each triangle is given by the indices of the three points that make up the triangle, ordered in either a clockwise or anticlockwise manner.

```
xy = np.asarray([
    [-0.101, 0.872], [-0.080, 0.883], [-0.069, 0.888], [-0.054, 0.890],
    [-0.045, 0.897], [-0.057, 0.895], [-0.073, 0.900], [-0.087, 0.898],
    [-0.090, 0.904], [-0.069, 0.907], [-0.069, 0.921], [-0.080, 0.919],
    [-0.073, 0.928], [-0.052, 0.930], [-0.048, 0.942], [-0.062, 0.949],
    [-0.054, 0.958], [-0.069, 0.954], [-0.087, 0.952], [-0.087, 0.959],
    [-0.080, 0.966], [-0.085, 0.973], [-0.087, 0.965], [-0.097, 0.965],
    [-0.097, 0.975], [-0.092, 0.984], [-0.101, 0.980], [-0.108, 0.980],
    [-0.104, 0.987], [-0.102, 0.993], [-0.115, 1.001], [-0.099, 0.996],
    [-0.101, 1.007], [-0.090, 1.010], [-0.087, 1.021], [-0.069, 1.021],
    [-0.052, 1.022], [-0.052, 1.017], [-0.069, 1.010], [-0.064, 1.005],
    [-0.048, 1.005], [-0.031, 1.005], [-0.031, 0.996], [-0.040, 0.987],
    [-0.045, 0.980], [-0.052, 0.975], [-0.040, 0.973], [-0.026, 0.968],
    [-0.020, 0.954], [-0.006, 0.947], [ 0.003, 0.935], [ 0.006, 0.926],
    [ 0.005, 0.921], [ 0.022, 0.923], [ 0.033, 0.912], [ 0.029, 0.905],
    [ 0.017, 0.900], [ 0.012, 0.895], [ 0.027, 0.893], [ 0.019, 0.886],
    [ 0.001, 0.883], [-0.012, 0.884], [-0.029, 0.883], [-0.038, 0.879],
    [-0.057, 0.881], [-0.062, 0.876], [-0.078, 0.876], [-0.087, 0.872],
    [-0.030, 0.907], [-0.007, 0.905], [-0.057, 0.916], [-0.025, 0.933],
```

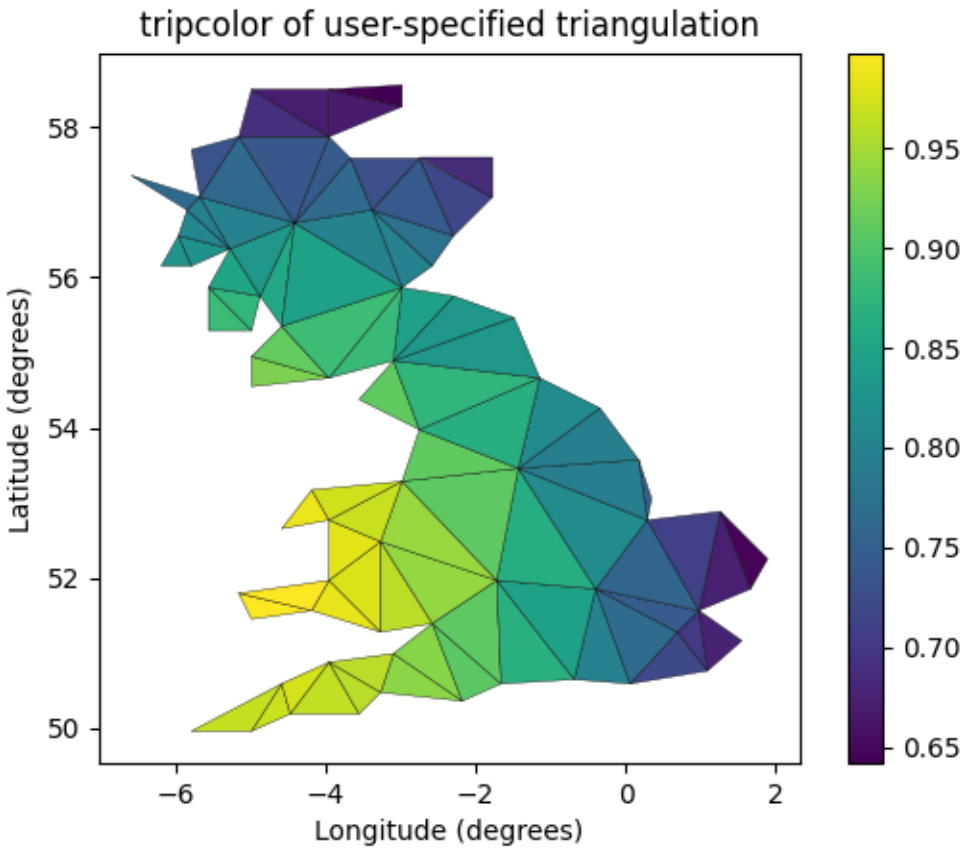
(continues on next page)

(continued from previous page)

```
    [-0.077, 0.990], [-0.059, 0.993]])
x, y = np.rad2deg(xy).T

triangles = np.asarray([
    [67, 66,  1], [65,  2, 66], [ 1, 66,  2], [64,  2, 65], [63,  3, 64],
    [60, 59, 57], [ 2, 64,  3], [ 3, 63,  4], [ 0, 67,  1], [62,  4, 63],
    [57, 59, 56], [59, 58, 56], [61, 60, 69], [57, 69, 60], [ 4, 62, 68],
    [ 6,  5,  9], [61, 68, 62], [69, 68, 61], [ 9,  5, 70], [ 6,  8,  7],
    [ 4, 70,  5], [ 8,  6,  9], [56, 69, 57], [69, 56, 52], [70, 10,  9],
    [54, 53, 55], [56, 55, 53], [68, 70,  4], [52, 56, 53], [11, 10, 12],
    [69, 71, 68], [68, 13, 70], [10, 70, 13], [51, 50, 52], [13, 68, 71],
    [52, 71, 69], [12, 10, 13], [71, 52, 50], [71, 14, 13], [50, 49, 71],
    [49, 48, 71], [14, 16, 15], [14, 71, 48], [17, 19, 18], [17, 20, 19],
    [48, 16, 14], [48, 47, 16], [47, 46, 16], [16, 46, 45], [23, 22, 24],
    [21, 24, 22], [17, 16, 45], [20, 17, 45], [21, 25, 24], [27, 26, 28],
    [20, 72, 21], [25, 21, 72], [45, 72, 20], [25, 28, 26], [44, 73, 45],
    [72, 45, 73], [28, 25, 29], [29, 25, 31], [43, 73, 44], [73, 43, 40],
    [72, 73, 39], [72, 31, 25], [42, 40, 43], [31, 30, 29], [39, 73, 40],
    [42, 41, 40], [72, 33, 31], [32, 31, 33], [39, 38, 72], [33, 72, 38],
    [33, 38, 34], [37, 35, 38], [34, 38, 35], [35, 37, 36]])

xmid = x[triangles].mean(axis=1)
ymid = y[triangles].mean(axis=1)
x0 = -5
y0 = 52
zfaces = np.exp(-0.01 * ((xmid - x0) * (xmid - x0) +
                         (ymid - y0) * (ymid - y0)))
```

Rather than create a Triangulation object, can simply pass x, y and triangles arrays to tripcolor directly. It would be better to use a Triangulation object if the same triangulation was to be used more than once to save duplicated calculations. Can specify one color value per face rather than one per point by using the *facecolors* keyword argument.

```
fig3, ax3 = plt.subplots()
ax3.set_aspect('equal')
tpc = ax3.tripcolor(x, y, triangles, facecolors=zfaces, edgecolors='k')
fig3.colorbar(tpc)
ax3.set_title('tripcolor of user-specified triangulation')
ax3.set_xlabel('Longitude (degrees)')
ax3.set_ylabel('Latitude (degrees)')

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.tripcolor* / *matplotlib.pyplot.tripcolor*
- *matplotlib.tri*
- *matplotlib.tri.Triangulation*

**Total running time of the script:** (0 minutes 1.559 seconds)

### Triplot Demo

Creating and plotting unstructured triangular grids.

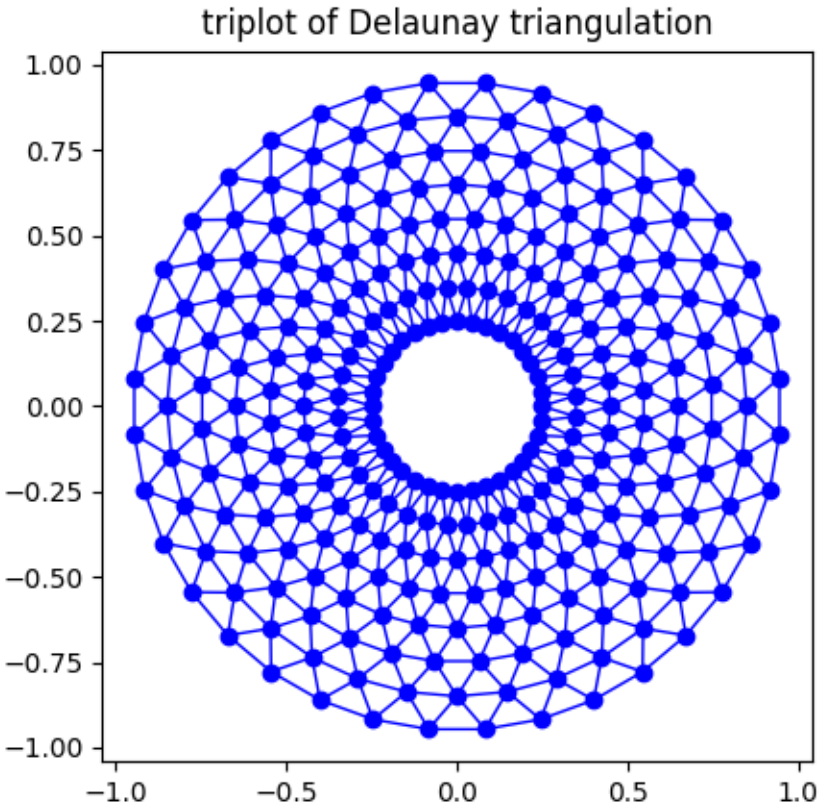
```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib.tri as tri
```

Creating a Triangulation without specifying the triangles results in the Delaunay triangulation of the points.

```
# First create the x and y coordinates of the points.
n_angles = 36
n_radii = 8
min_radius = 0.25
radii = np.linspace(min_radius, 0.95, n_radii)

angles = np.linspace(0, 2 * np.pi, n_angles, endpoint=False)
angles = np.repeat(angles[..., np.newaxis], n_radii, axis=1)
angles[:, 1::2] += np.pi / n_angles

x = (radii * np.cos(angles)).flatten()
y = (radii * np.sin(angles)).flatten()

# Create the Triangulation; no triangles so Delaunay triangulation created.
triang = tri.Triangulation(x, y)

# Mask off unwanted triangles.
triang.set_mask(np.hypot(x[triang.triangles].mean(axis=1),
                         y[triang.triangles].mean(axis=1))
                < min_radius)
```

Plot the triangulation.

```
fig1, ax1 = plt.subplots()
ax1.set_aspect('equal')
ax1.triplot(triang, 'bo-', lw=1)
ax1.set_title('triplot of Delaunay triangulation')
```




You can specify your own triangulation rather than perform a Delaunay triangulation of the points, where each triangle is given by the indices of the three points that make up the triangle, ordered in either a clockwise or anticlockwise manner.

```
xy = np.asarray([
    [-0.101, 0.872], [-0.080, 0.883], [-0.069, 0.888], [-0.054, 0.890],
    [-0.045, 0.897], [-0.057, 0.895], [-0.073, 0.900], [-0.087, 0.898],
    [-0.090, 0.904], [-0.069, 0.907], [-0.069, 0.921], [-0.080, 0.919],
    [-0.073, 0.928], [-0.052, 0.930], [-0.048, 0.942], [-0.062, 0.949],
    [-0.054, 0.958], [-0.069, 0.954], [-0.087, 0.952], [-0.087, 0.959],
    [-0.080, 0.966], [-0.085, 0.973], [-0.087, 0.965], [-0.097, 0.965],
    [-0.097, 0.975], [-0.092, 0.984], [-0.101, 0.980], [-0.108, 0.980],
    [-0.104, 0.987], [-0.102, 0.993], [-0.115, 1.001], [-0.099, 0.996],
    [-0.101, 1.007], [-0.090, 1.010], [-0.087, 1.021], [-0.069, 1.021],
    [-0.052, 1.022], [-0.052, 1.017], [-0.069, 1.010], [-0.064, 1.005],
    [-0.048, 1.005], [-0.031, 1.005], [-0.031, 0.996], [-0.040, 0.987],
    [-0.045, 0.980], [-0.052, 0.975], [-0.040, 0.973], [-0.026, 0.968],
    [-0.020, 0.954], [-0.006, 0.947], [ 0.003, 0.935], [ 0.006, 0.926],
    [ 0.005, 0.921], [ 0.022, 0.923], [ 0.033, 0.912], [ 0.029, 0.905],
    [ 0.017, 0.900], [ 0.012, 0.895], [ 0.027, 0.893], [ 0.019, 0.886],
    [ 0.001, 0.883], [-0.012, 0.884], [-0.029, 0.883], [-0.038, 0.879],
    [-0.057, 0.881], [-0.062, 0.876], [-0.078, 0.876], [-0.087, 0.872],
    [-0.030, 0.907], [-0.007, 0.905], [-0.057, 0.916], [-0.025, 0.933],
```

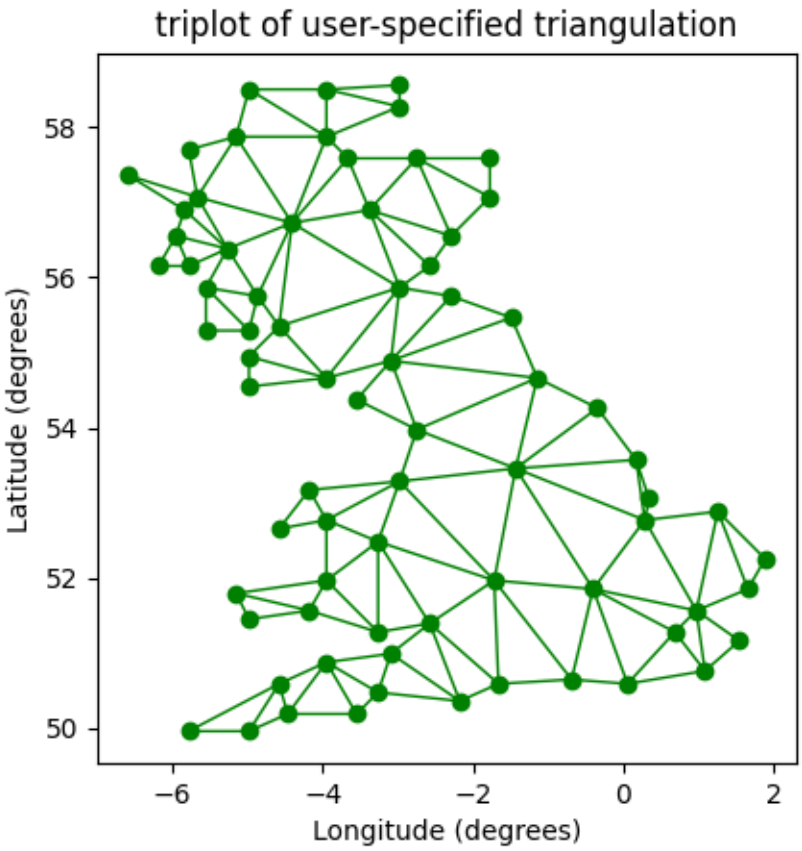
(continues on next page)

(continued from previous page)

```
    [-0.077, 0.990], [-0.059, 0.993]])
x = np.degrees(xy[:, 0])
y = np.degrees(xy[:, 1])

triangles = np.asarray([
    [67, 66,  1], [65,  2, 66], [ 1, 66,  2], [64,  2, 65], [63,  3, 64],
    [60, 59, 57], [ 2, 64,  3], [ 3, 63,  4], [ 0, 67,  1], [62,  4, 63],
    [57, 59, 56], [59, 58, 56], [61, 60, 69], [57, 69, 60], [ 4, 62, 68],
    [ 6,  5,  9], [61, 68, 62], [69, 68, 61], [ 9,  5, 70], [ 6,  8,  7],
    [ 4, 70,  5], [ 8,  6,  9], [56, 69, 57], [69, 56, 52], [70, 10,  9],
    [54, 53, 55], [56, 55, 53], [68, 70,  4], [52, 56, 53], [11, 10, 12],
    [69, 71, 68], [68, 13, 70], [10, 70, 13], [51, 50, 52], [13, 68, 71],
    [52, 71, 69], [12, 10, 13], [71, 52, 50], [71, 14, 13], [50, 49, 71],
    [49, 48, 71], [14, 16, 15], [14, 71, 48], [17, 19, 18], [17, 20, 19],
    [48, 16, 14], [48, 47, 16], [47, 46, 16], [16, 46, 45], [23, 22, 24],
    [21, 24, 22], [17, 16, 45], [20, 17, 45], [21, 25, 24], [27, 26, 28],
    [20, 72, 21], [25, 21, 72], [45, 72, 20], [25, 28, 26], [44, 73, 45],
    [72, 45, 73], [28, 25, 29], [29, 25, 31], [43, 73, 44], [73, 43, 40],
    [72, 73, 39], [72, 31, 25], [42, 40, 43], [31, 30, 29], [39, 73, 40],
    [42, 41, 40], [72, 33, 31], [32, 31, 33], [39, 38, 72], [33, 72, 38],
    [33, 38, 34], [37, 35, 38], [34, 38, 35], [35, 37, 36]])
```

Rather than create a Triangulation object, can simply pass x, y and triangles arrays to triplot directly. It would be better to use a Triangulation object if the same triangulation was to be used more than once to save duplicated calculations.

```
fig2, ax2 = plt.subplots()
ax2.set_aspect('equal')
ax2.triplot(x, y, triangles, 'go-', lw=1.0)
ax2.set_title('triplot of user-specified triangulation')
ax2.set_xlabel('Longitude (degrees)')
ax2.set_ylabel('Latitude (degrees)')

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.triplot* / *matplotlib.pyplot.triplot*
- *matplotlib.tri*
- *matplotlib.tri.Triangulation*

### Watermark image

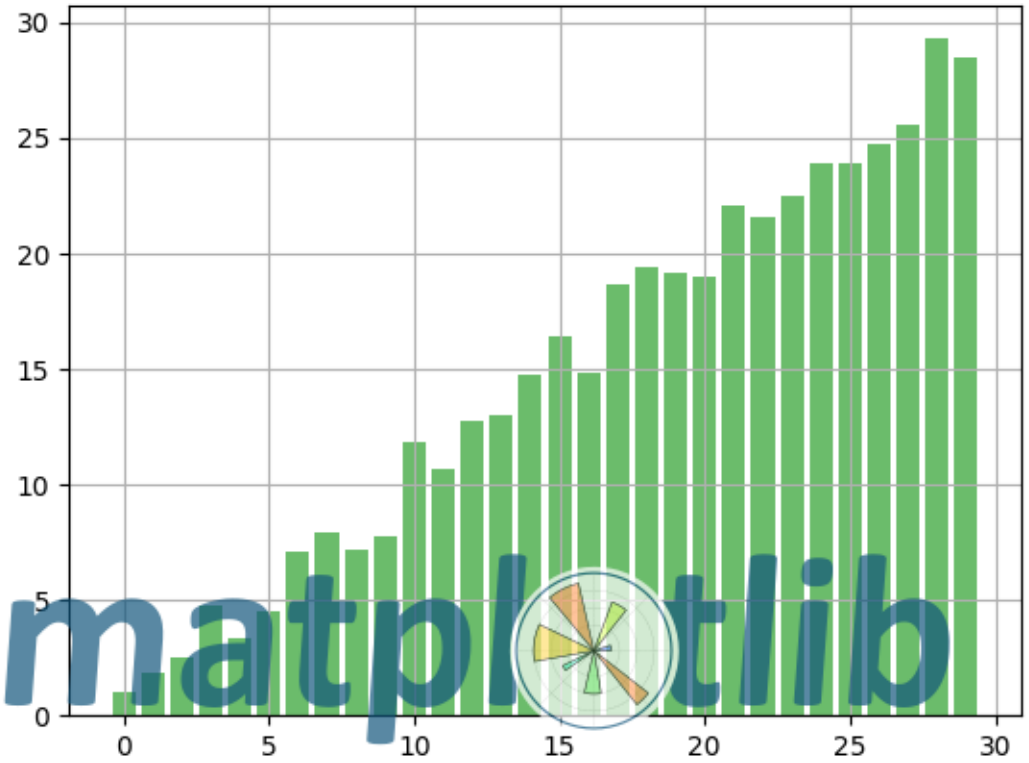
Overlay an image on a plot by moving it to the front (`zorder=3`) and making it semi-transparent (`alpha=0.7`).

```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib.cbook as cbook
import matplotlib.image as image

```

(continues on next page)

(continued from previous page)

```
with cbook.get_sample_data('logo2.png') as file:
    im = image.imread(file)

fig, ax = plt.subplots()

np.random.seed(19680801)
x = np.arange(30)
y = x + np.random.randn(30)
ax.bar(x, y, color='#6bbc6b')
ax.grid()

fig.figimage(im, 25, 25, zorder=3, alpha=.7)

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.image*
- *matplotlib.image.imread* / *matplotlib.pyplot.imread*

- *matplotlib.figure.Figure.figimage*

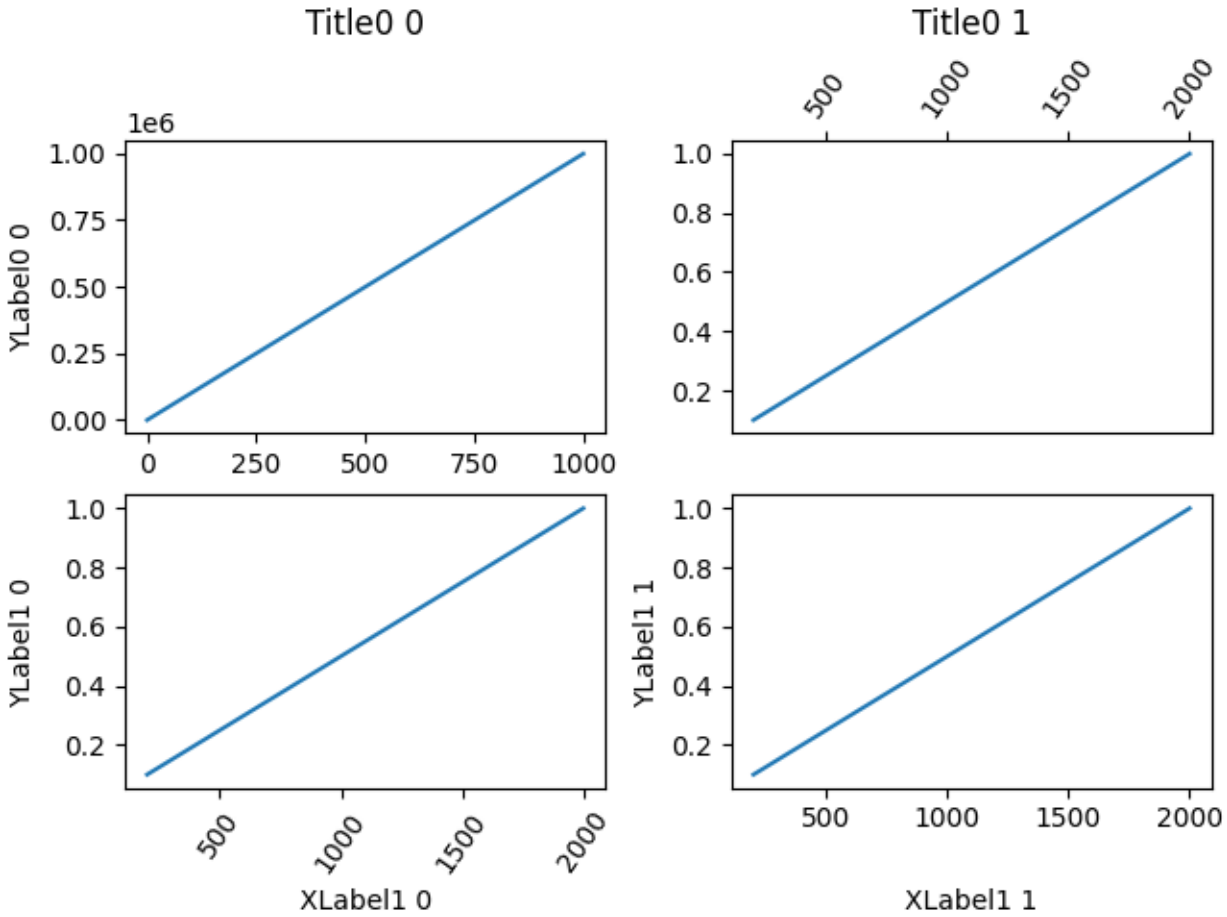
### 6.25.3 Subplots, axes and figures

#### Aligning Labels and Titles

Aligning xlabel, ylabel, and title using *Figure.align_xlabels*, *Figure.align_ylabels*, and *Figure.align_titles*.

*Figure.align_labels* wraps the x and y label functions.

Note that the xlabel "XLabel1 1" would normally be much closer to the x-axis, "YLabel0 0" would be much closer to the y-axis, and title "Title0 0" would be much closer to the top of their respective axes.




```
import matplotlib.pyplot as plt
import numpy as np

fig, axs = plt.subplots(2, 2, layout='constrained')

ax = axs[0][0]
ax.plot(np.arange(0, 1e6, 1000))
```

(continues on next page)

(continued from previous page)

```
ax.set_title('Title0 0')
ax.set_ylabel('YLabel0 0')

ax = axs[0][1]
ax.plot(np.arange(1., 0., -0.1) * 2000., np.arange(1., 0., -0.1))
ax.set_title('Title0 1')
ax.xaxis.tick_top()
ax.tick_params(axis='x', rotation=55)


for i in range(2):
    ax = axs[1][i]
    ax.plot(np.arange(1., 0., -0.1) * 2000., np.arange(1., 0., -0.1))
    ax.set_ylabel('YLabel1 %d' % i)
    ax.set_xlabel('XLabel1 %d' % i)
    if i == 0:
        ax.tick_params(axis='x', rotation=55)

fig.align_labels()  # same as fig.align_xlabels(); fig.align_ylabels()
fig.align_titles()

plt.show()
```

### Programmatically controlling subplot adjustment

---

**Note:** This example is primarily intended to show some advanced concepts in Matplotlib.

If you are only looking for having enough space for your labels, it is almost always simpler and good enough to either set the subplot parameters manually using *Figure.subplots_adjust*, or use one of the automatic layout mechanisms (*Constrained layout guide* or *Tight layout guide*).

---

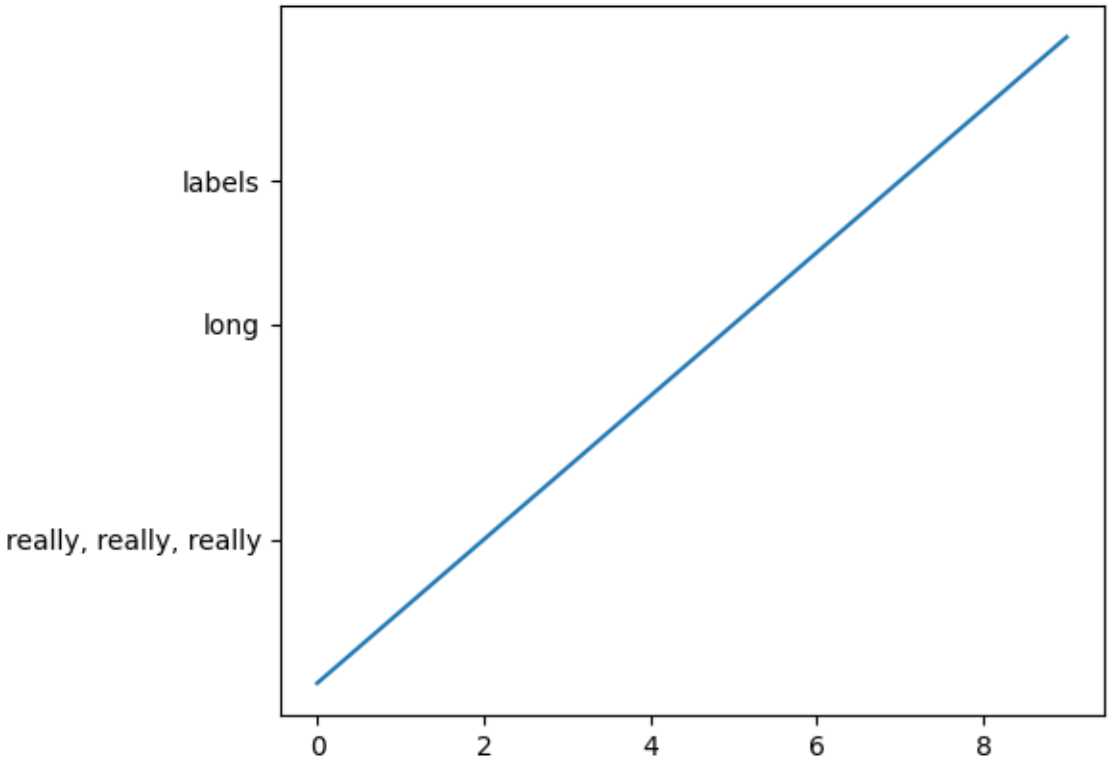
This example describes a user-defined way to read out Artist sizes and set the subplot parameters accordingly. Its main purpose is to illustrate some advanced concepts like reading out text positions, working with bounding boxes and transforms and using *events*. But it can also serve as a starting point if you want to automate the layouting and need more flexibility than tight layout and constrained layout.

Below, we collect the bounding boxes of all y-labels and move the left border of the subplot to the right so that it leaves enough room for the union of all the bounding boxes.

There's one catch with calculating text bounding boxes: Querying the text bounding boxes (*Text.get_window_extent*) needs a renderer (*RendererBase* instance), to calculate the text size. This renderer is only available after the figure has been drawn (*Figure.draw*).

A solution to this is putting the adjustment logic in a draw callback. This function is executed after the figure has been drawn. It can now check if the subplot leaves enough room for the text. If not, the subplot parameters are updated and second draw is triggered.

```
import matplotlib.pyplot as plt

import matplotlib.transforms as mtransforms

fig, ax = plt.subplots()
ax.plot(range(10))
ax.set_yticks([2, 5, 7], labels=['really, really, really', 'long', 'labels'])


def on_draw(event):
    bboxes = []
    for label in ax.get_yticklabels():
        # Bounding box in pixels
        bbox_px = label.get_window_extent()
        # Transform to relative figure coordinates. This is the inverse of
        # transFigure.
        bbox_fig = bbox_px.transformed(fig.transFigure.inverted())
        bboxes.append(bbox_fig)
    # the bbox that bounds all the bboxes, again in relative figure coords
    bbox = mtransforms.Bbox.union(bboxes)
    if fig.subplotpars.left < bbox.width:
        # Move the subplot left edge more to the right
        fig.subplots_adjust(left=1.1*bbox.width)  # pad a little
        fig.canvas.draw()


fig.canvas.mpl_connect('draw_event', on_draw)

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.artist.Artist.get_window_extent*
- *matplotlib.transforms.Bbox*
- *matplotlib.transforms.BboxBase.transformed*
- *matplotlib.transforms.BboxBase.union*
- *matplotlib.transforms.Transform.inverted*
- *matplotlib.figure.Figure.subplots_adjust*
- *matplotlib.gridspec.SubplotParams*
- *matplotlib.backend_bases.FigureCanvasBase.mpl_connect*

### Axes box aspect

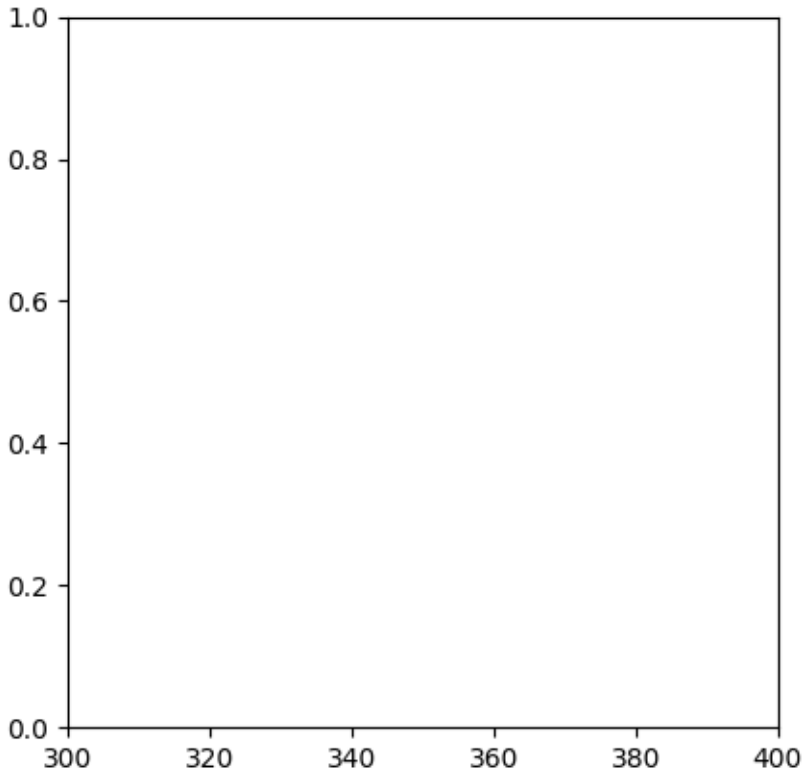
This demo shows how to set the aspect of an Axes box directly via *set_box_aspect*. The box aspect is the ratio between Axes height and Axes width in physical units, independent of the data limits. This is useful to e.g. produce a square plot, independent of the data it contains, or to have a usual plot with the same axes dimensions next to an image plot with fixed (data-)aspect.

The following lists a few use cases for *set_box_aspect*.

### A square Axes, independent of data

Produce a square Axes, no matter what the data limits are.

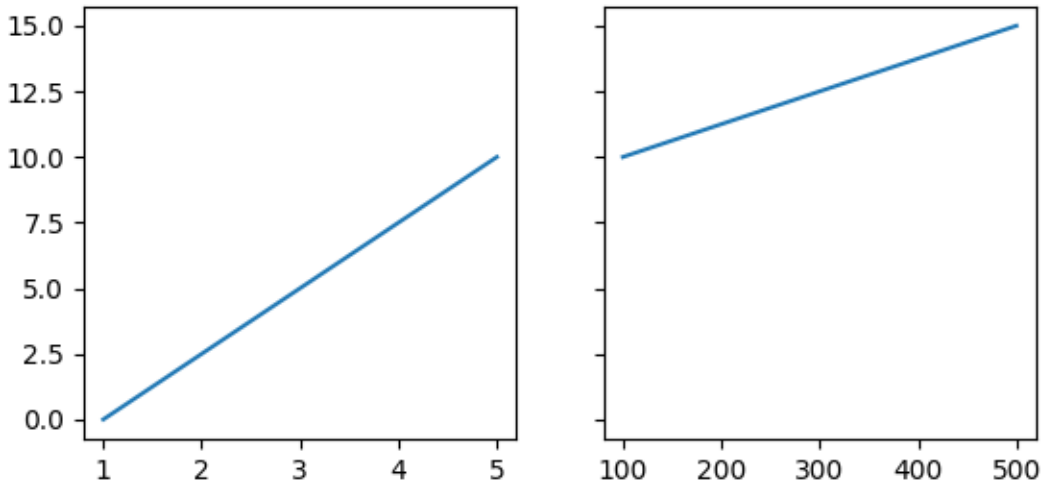
```
import matplotlib.pyplot as plt
import numpy as np

fig1, ax = plt.subplots()

ax.set_xlim(300, 400)
ax.set_box_aspect(1)

plt.show()
```



### Shared square Axes

Produce shared subplots that are squared in size.

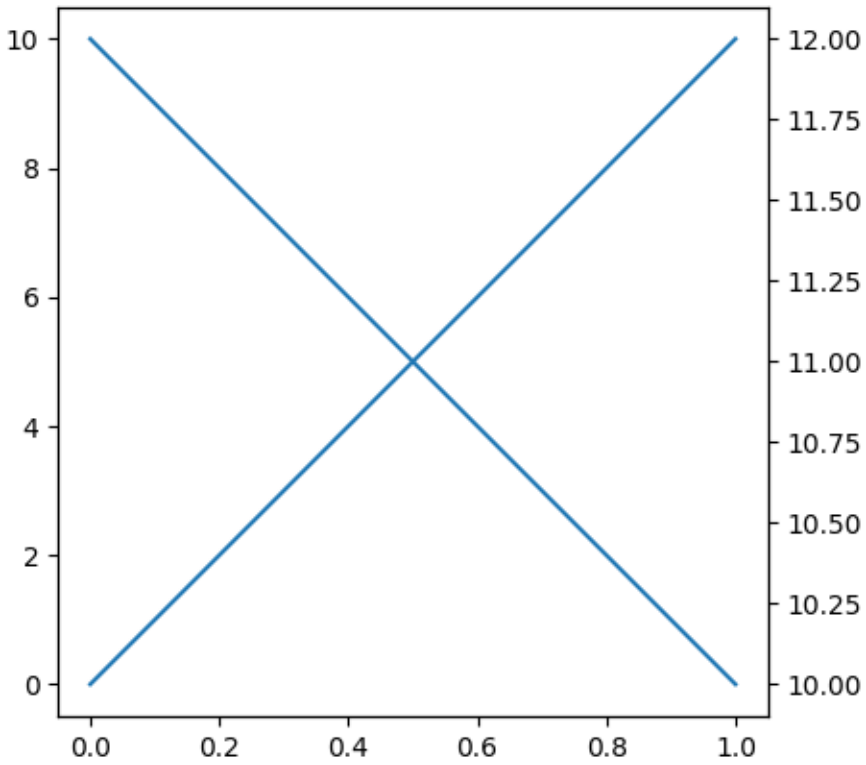
```
fig2, (ax, ax2) = plt.subplots(ncols=2, sharey=True)

ax.plot([1, 5], [0, 10])
ax2.plot([100, 500], [10, 15])

ax.set_box_aspect(1)
ax2.set_box_aspect(1)

plt.show()
```



### Square twin Axes

Produce a square Axes, with a twin Axes. The twinned Axes takes over the box aspect of the parent.

```
fig3, ax = plt.subplots()

ax2 = ax.twinx()

ax.plot([0, 10])
ax2.plot([12, 10])

ax.set_box_aspect(1)

plt.show()
```

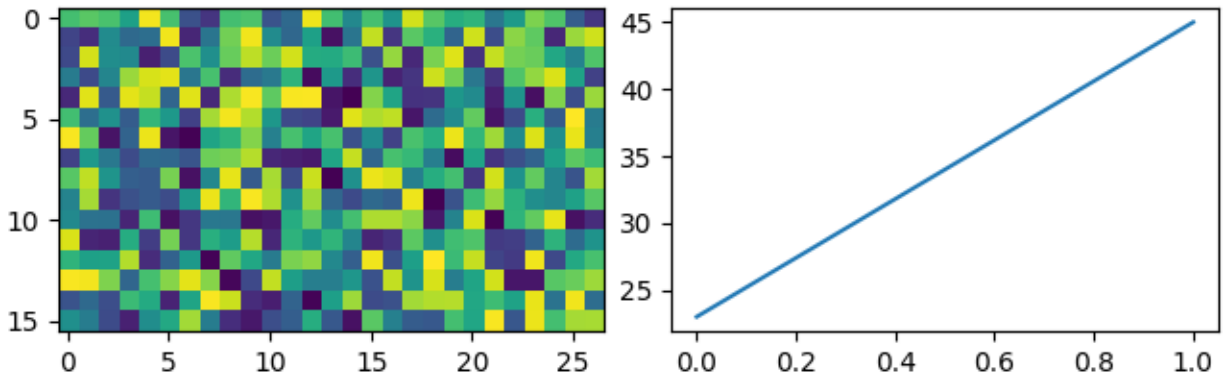


### Normal plot next to image

When creating an image plot with fixed data aspect and the default `adjustable="box"` next to a normal plot, the Axes would be unequal in height. *`set_box_aspect`* provides an easy solution to that by allowing to have the normal plot's Axes use the images dimensions as box aspect.

This example also shows that *constrained layout* interplays nicely with a fixed box aspect.

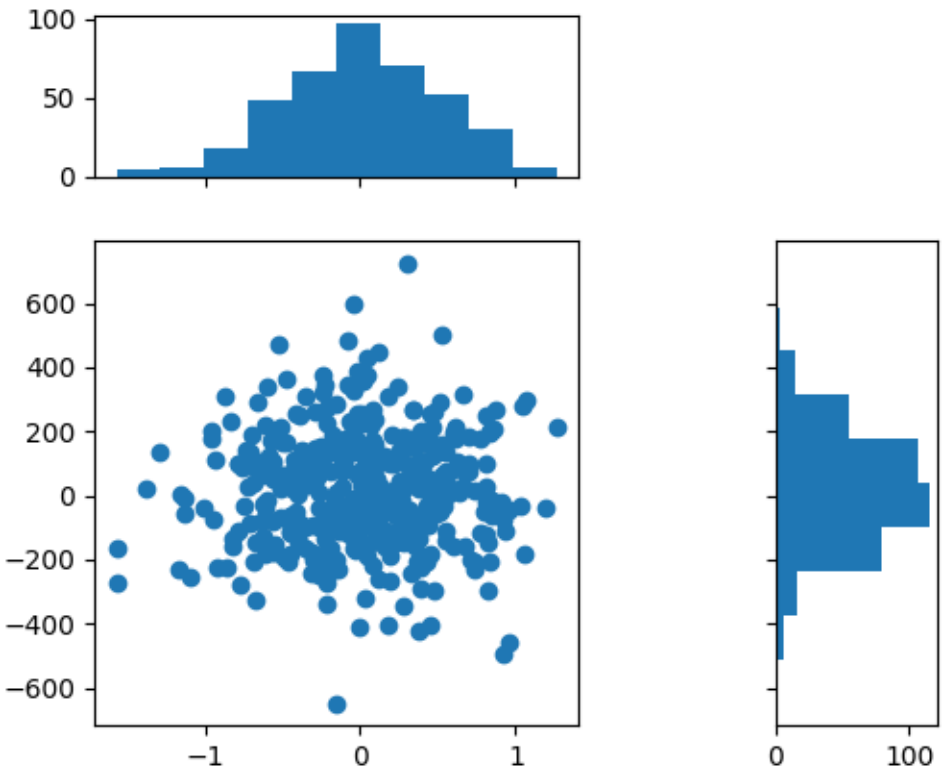
```
fig4, (ax, ax2) = plt.subplots(ncols=2, layout="constrained")

np.random.seed(19680801)  # Fixing random state for reproducibility
im = np.random.rand(16, 27)
ax.imshow(im)

ax2.plot([23, 45])
ax2.set_box_aspect(im.shape[0]/im.shape[1])

plt.show()
```



### Square joint/marginal plot

It may be desirable to show marginal distributions next to a plot of joint data. The following creates a square plot with the box aspect of the marginal Axes being equal to the width- and height-ratios of the gridspec. This ensures that all Axes align perfectly, independent on the size of the figure.

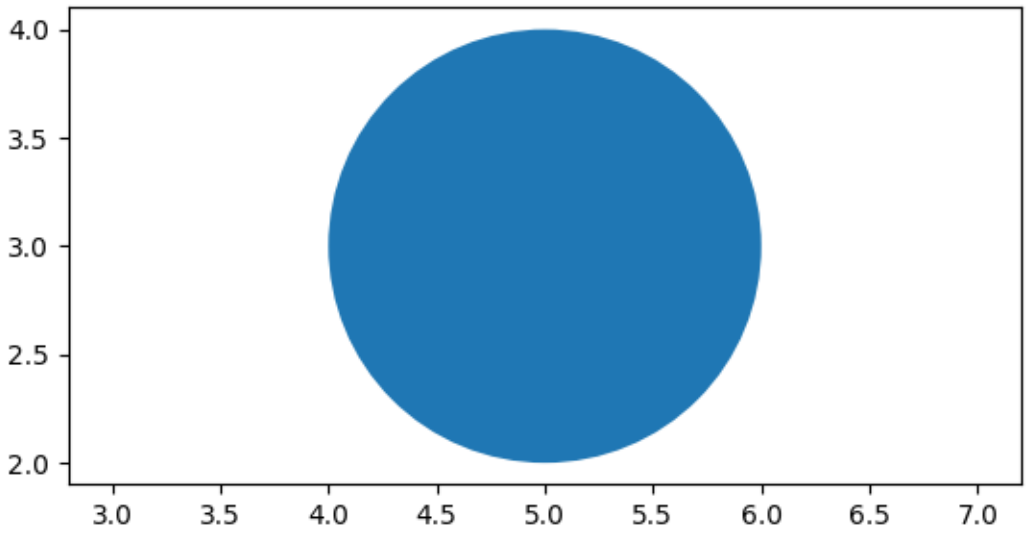
```
fig5, axs = plt.subplots(2, 2, sharex="col", sharey="row",
                         gridspec_kw=dict(height_ratios=[1, 3],
                                          width_ratios=[3, 1]))
axs[0, 1].set_visible(False)
axs[0, 0].set_box_aspect(1/3)
axs[1, 0].set_box_aspect(1)
axs[1, 1].set_box_aspect(3/1)

np.random.seed(19680801)  # Fixing random state for reproducibility
x, y = np.random.randn(2, 400) * [[.5], [180]]
axs[1, 0].scatter(x, y)
axs[0, 0].hist(x)
axs[1, 1].hist(y, orientation="horizontal")

plt.show()
```



### Set data aspect with box aspect

When setting the box aspect, one may still set the data aspect as well. Here we create an Axes with a box twice as long as it is tall and use an "equal" data aspect for its contents, i.e. the circle actually stays circular.

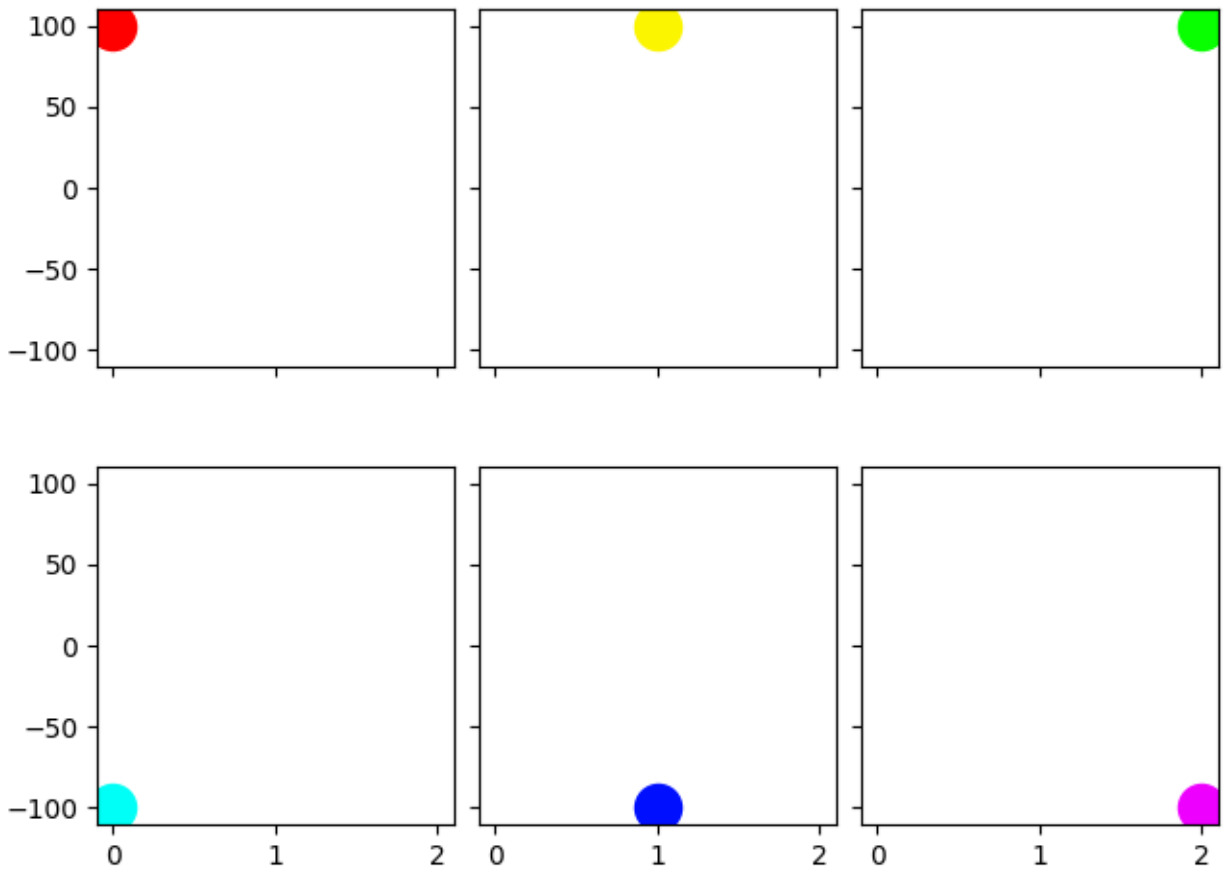
```
fig6, ax = plt.subplots()

ax.add_patch(plt.Circle((5, 3), 1))
ax.set_aspect("equal", adjustable="datalim")
ax.set_box_aspect(0.5)
ax.autoscale()

plt.show()
```



### Box aspect for many subplots

It is possible to pass the box aspect to an Axes at initialization. The following creates a 2 by 3 subplot grid with all square Axes.

```
fig7, axs = plt.subplots(2, 3, subplot_kw=dict(box_aspect=1),
                         sharex=True, sharey=True, layout="constrained")

for i, ax in enumerate(axs.flat):
    ax.scatter(i % 3, -((i // 3) - 0.5)*200, c=[plt.cm.hsv(i / 6)], s=300)
plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.set_box_aspect*

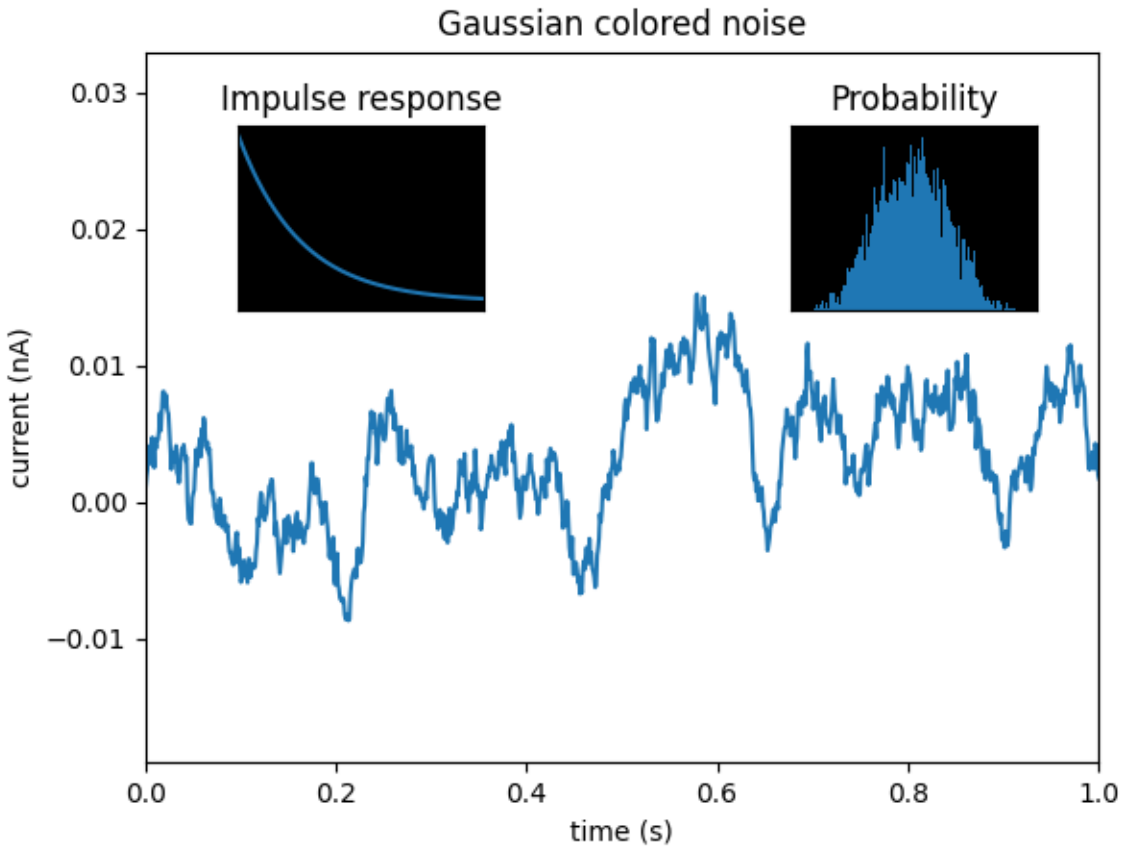
**Total running time of the script:** (0 minutes 2.087 seconds)

### Axes Demo

Example use of `fig.add_axes` to create inset Axes within the main plot Axes.

Please see also the *Module - axes_grid1* section, and the following three examples:

- *Zoom region inset Axes*
- *Inset locator demo*
- *Inset locator demo 2*




```
import matplotlib.pyplot as plt
import numpy as np

np.random.seed(19680801)  # Fixing random state for reproducibility.

# create some data to use for the plot
dt = 0.001
t = np.arange(0.0, 10.0, dt)
r = np.exp(-t[:1000] / 0.05)  # impulse response
x = np.random.randn(len(t))
s = np.convolve(x, r)[:len(x)] * dt  # colored noise
```

(continues on next page)

(continued from previous page)

```
fig, main_ax = plt.subplots()
main_ax.plot(t, s)
main_ax.set_xlim(0, 1)
main_ax.set_ylim(1.1 * np.min(s), 2 * np.max(s))
main_ax.set_xlabel('time (s)')
main_ax.set_ylabel('current (nA)')
main_ax.set_title('Gaussian colored noise')

# this is an inset Axes over the main Axes
right_inset_ax = fig.add_axes([.65, .6, .2, .2], facecolor='k')
right_inset_ax.hist(s, 400, density=True)
right_inset_ax.set(title='Probability', xticks=[], yticks=[])

# this is another inset Axes over the main Axes
left_inset_ax = fig.add_axes([.2, .6, .2, .2], facecolor='k')
left_inset_ax.plot(t[:len(r)], r)
left_inset_ax.set(title='Impulse response', xlim=(0, .2), xticks=[],␣
 ↪yticks=[])

plt.show()
```

### Controlling view limits using margins and sticky_edges

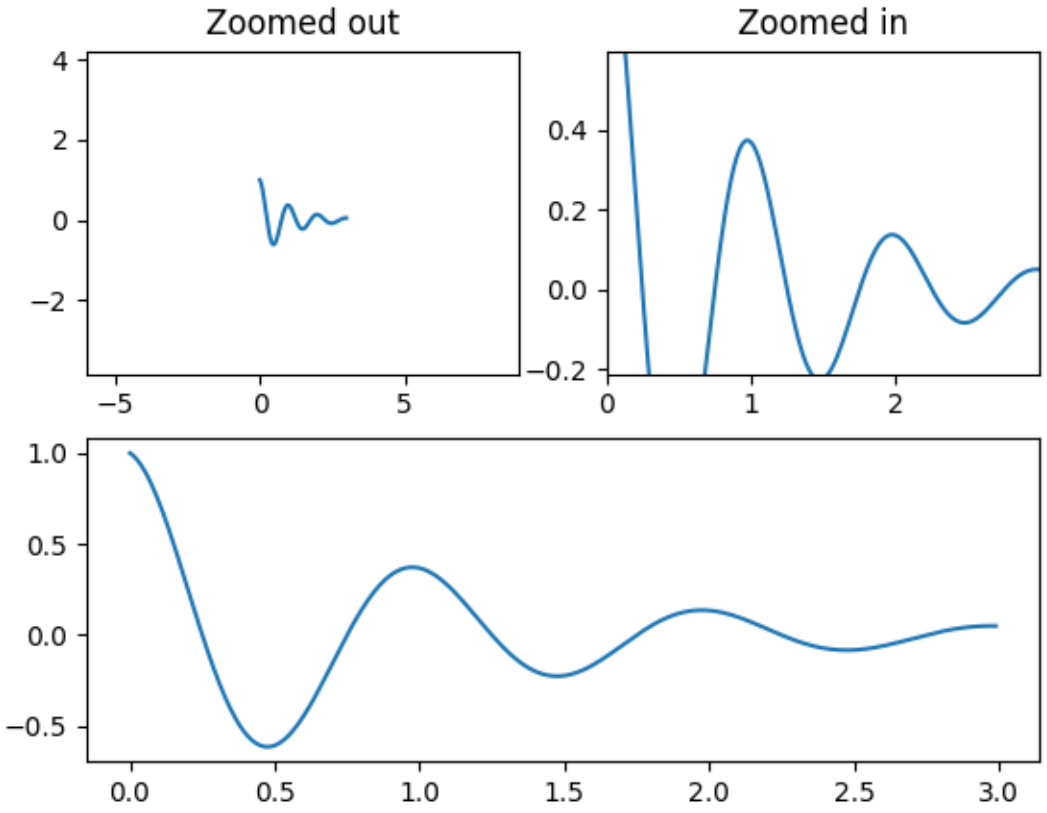
The first figure in this example shows how to zoom in and out of a plot using *margins* instead of *set_xlim* and *set_ylim*. The second figure demonstrates the concept of edge "stickiness" introduced by certain methods and artists and how to effectively work around that.

```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.patches import Polygon


def f(t):
    return np.exp(-t) * np.cos(2*np.pi*t)


t1 = np.arange(0.0, 3.0, 0.01)

ax1 = plt.subplot(212)
ax1.margins(0.05)           # Default margin is 0.05, value 0 means fit
ax1.plot(t1, f(t1))

ax2 = plt.subplot(221)
ax2.margins(2, 2)           # Values >0.0 zoom out
ax2.plot(t1, f(t1))
ax2.set_title('Zoomed out')

ax3 = plt.subplot(222)
ax3.margins(x=0, y=-0.25)   # Values in (-0.5, 0.0) zooms in to center
```

(continues on next page)

(continued from previous page)

```
ax3.plot(t1, f(t1))
ax3.set_title('Zoomed in')

plt.show()
```




## On the "stickiness" of certain plotting methods

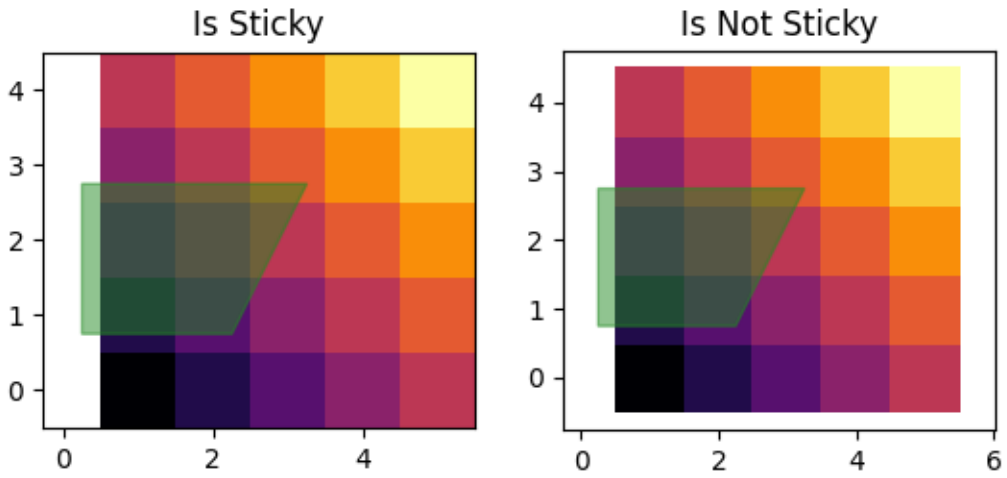
Some plotting functions make the axis limits "sticky" or immune to the will of the *margins* methods. For instance, *imshow* and *pcolor* expect the user to want the limits to be tight around the pixels shown in the plot. If this behavior is not desired, you need to set *use_sticky_edges* to `False`. Consider the following example:

```
y, x = np.mgrid[:5, 1:6]
poly_coords = [
    (0.25, 2.75), (3.25, 2.75),
    (2.25, 0.75), (0.25, 0.75)
]
fig, (ax1, ax2) = plt.subplots(ncols=2)

# Here we set the stickiness of the Axes object...
```

(continues on next page)

(continued from previous page)

```
# ax1 we'll leave as the default, which uses sticky edges
# and we'll turn off stickiness for ax2
ax2.use_sticky_edges = False

for ax, status in zip((ax1, ax2), ('Is', 'Is Not')):
    cells = ax.pcolor(x, y, x+y, cmap='inferno', shading='auto')  # sticky
    ax.add_patch(
        Polygon(poly_coords, color='forestgreen', alpha=0.5)
    )  # not sticky
    ax.margins(x=0.1, y=0.05)
    ax.set_aspect('equal')
    ax.set_title(f'{status} Sticky')

plt.show()
```




**References**

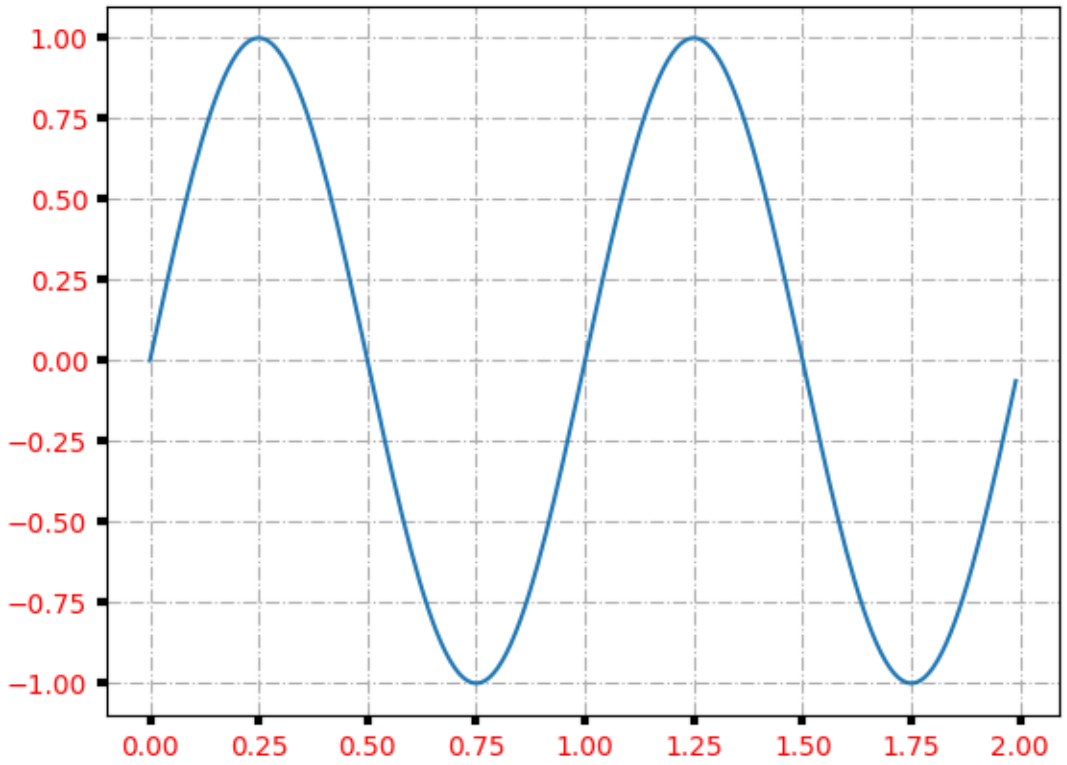
The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.margins* / *matplotlib.pyplot.margins*
- *matplotlib.axes.Axes.use_sticky_edges*

- *matplotlib.axes.Axes.pcolor* / *matplotlib.pyplot.pcolor*
- *matplotlib.patches.Polygon*

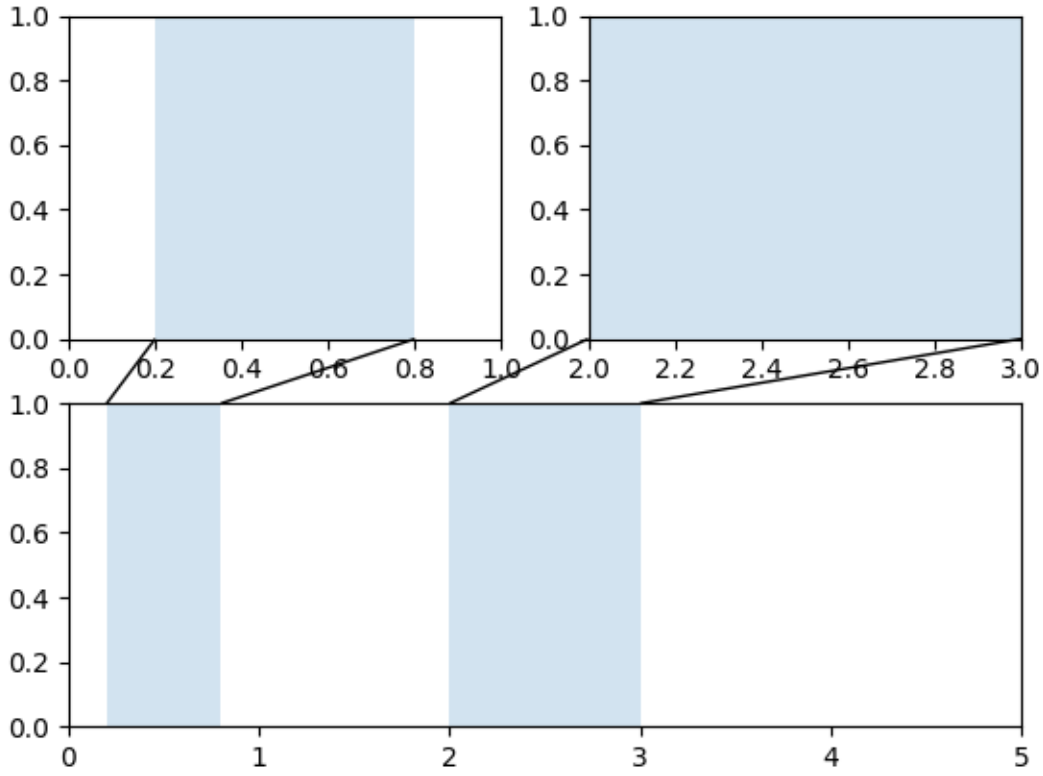
### Axes Props

You can control the axis tick and grid properties




```
import matplotlib.pyplot as plt
import numpy as np

t = np.arange(0.0, 2.0, 0.01)
s = np.sin(2 * np.pi * t)

fig, ax = plt.subplots()
ax.plot(t, s)

ax.grid(True, linestyle='-.')
ax.tick_params(labelcolor='r', labelsize='medium', width=3)

plt.show()
```

### Axes Zoom Effect




```
import matplotlib.pyplot as plt

from matplotlib.transforms import (Bbox, TransformedBbox,
                                   blended_transform_factory)
from mpl_toolkits.axes_grid1.inset_locator import (BboxConnector,
                                                   BboxConnectorPatch,
                                                   BboxPatch)


def connect_bbox(bbox1, bbox2,
                 loc1a, loc2a, loc1b, loc2b,
                 prop_lines, prop_patches=None):
    if prop_patches is None:
        prop_patches = {
            **prop_lines,
            "alpha": prop_lines.get("alpha", 1) * 0.2,
            "clip_on": False,
        }

    c1 = BboxConnector(
        bbox1, bbox2, loc1=loc1a, loc2=loc2a, clip_on=False, **prop_lines)
```

(continues on next page)

(continued from previous page)

```
    c2 = BboxConnector(
        bbox1, bbox2, loc1=loc1b, loc2=loc2b, clip_on=False, **prop_lines)

    bbox_patch1 = BboxPatch(bbox1, **prop_patches)
    bbox_patch2 = BboxPatch(bbox2, **prop_patches)

    p = BboxConnectorPatch(bbox1, bbox2,
                           loc1a=loc1a, loc2a=loc2a, loc1b=loc1b, loc2b=loc2b,
                           clip_on=False,
                           **prop_patches)

    return c1, c2, bbox_patch1, bbox_patch2, p


def zoom_effect01(ax1, ax2, xmin, xmax, **kwargs):
    """
    Connect *ax1* and *ax2*. The *xmin*-to-*xmax* range in both Axes will
    be marked.

    Parameters
    ----------
    ax1
        The main Axes.
    ax2
        The zoomed Axes.
    xmin, xmax
        The limits of the colored area in both plot Axes.
    **kwargs
        Arguments passed to the patch constructor.
    """

    bbox = Bbox.from_extents(xmin, 0, xmax, 1)

    mybbox1 = TransformedBbox(bbox, ax1.get_xaxis_transform())
    mybbox2 = TransformedBbox(bbox, ax2.get_xaxis_transform())

    prop_patches = {**kwargs, "ec": "none", "alpha": 0.2}

    c1, c2, bbox_patch1, bbox_patch2, p = connect_bbox(
        mybbox1, mybbox2,
        loc1a=3, loc2a=2, loc1b=4, loc2b=1,
        prop_lines=kwargs, prop_patches=prop_patches)

    ax1.add_patch(bbox_patch1)
    ax2.add_patch(bbox_patch2)
    ax2.add_patch(c1)
    ax2.add_patch(c2)
    ax2.add_patch(p)

    return c1, c2, bbox_patch1, bbox_patch2, p
```

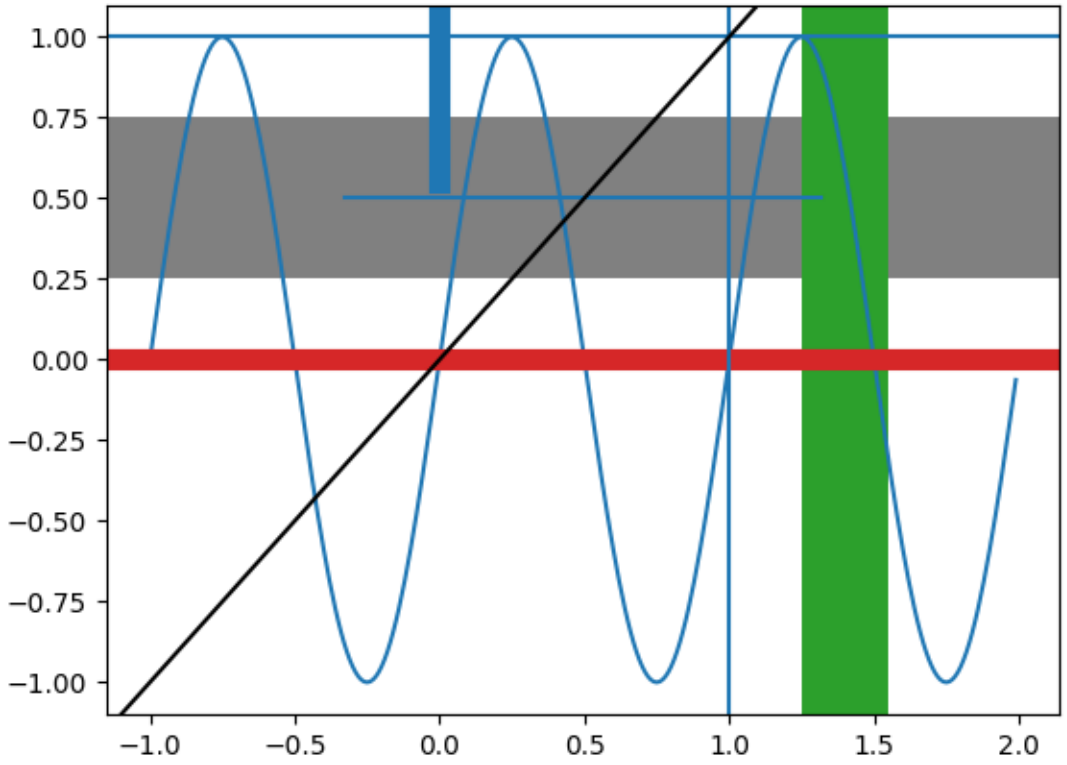
(continues on next page)

(continued from previous page)

```
def zoom_effect02(ax1, ax2, **kwargs):
    """
    ax1 : the main Axes
    ax1 : the zoomed Axes

    Similar to zoom_effect01.  The xmin & xmax will be taken from the
    ax1.viewLim.
    """

    tt = ax1.transScale + (ax1.transLimits + ax2.transAxes)
    trans = blended_transform_factory(ax2.transData, tt)

    mybbox1 = ax1.bbox
    mybbox2 = TransformedBbox(ax1.viewLim, trans)

    prop_patches = {**kwargs, "ec": "none", "alpha": 0.2}

    c1, c2, bbox_patch1, bbox_patch2, p = connect_bbox(
        mybbox1, mybbox2,
        loc1a=3, loc2a=2, loc1b=4, loc2b=1,
        prop_lines=kwargs, prop_patches=prop_patches)

    ax1.add_patch(bbox_patch1)
    ax2.add_patch(bbox_patch2)
    ax2.add_patch(c1)
    ax2.add_patch(c2)
    ax2.add_patch(p)

    return c1, c2, bbox_patch1, bbox_patch2, p


axs = plt.figure().subplot_mosaic([
    ["zoom1", "zoom2"],
    ["main", "main"],
])

axs["main"].set(xlim=(0, 5))
zoom_effect01(axs["zoom1"], axs["main"], 0.2, 0.8)
axs["zoom2"].set(xlim=(2, 3))
zoom_effect02(axs["zoom2"], axs["main"])

plt.show()
```

### axhspan Demo

Create lines or rectangles that span the Axes in either the horizontal or vertical direction, and lines than span the Axes with an arbitrary orientation.




```
import matplotlib.pyplot as plt
import numpy as np

t = np.arange(-1, 2, .01)
s = np.sin(2 * np.pi * t)

fig, ax = plt.subplots()

ax.plot(t, s)
# Thick red horizontal line at y=0 that spans the xrange.
ax.axhline(linewidth=8, color='#d62728')
# Horizontal line at y=1 that spans the xrange.
ax.axhline(y=1)
# Vertical line at x=1 that spans the yrange.
ax.axvline(x=1)
# Thick blue vertical line at x=0 that spans the upper quadrant of the yrange.
ax.axvline(x=0, ymin=0.75, linewidth=8, color='#1f77b4')
# Default hline at y=.5 that spans the middle half of the Axes.
```

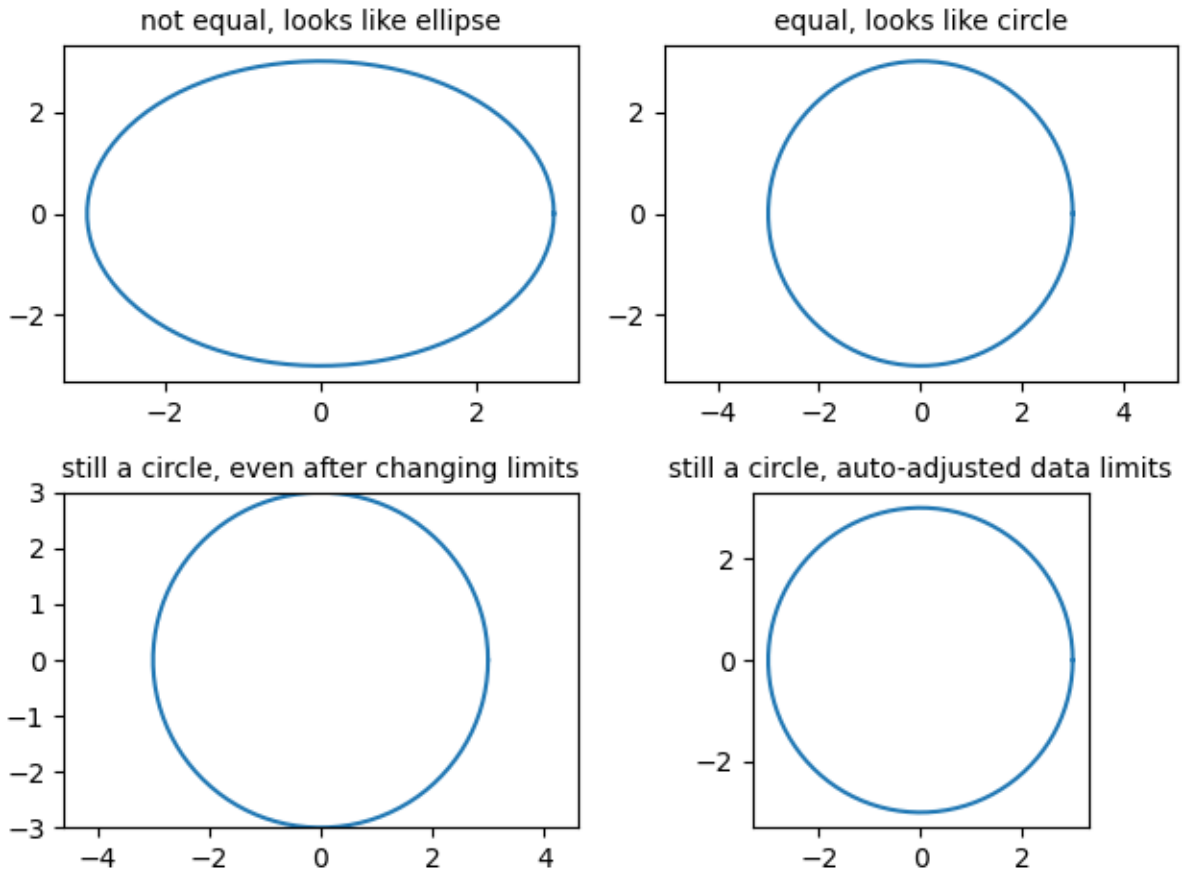
(continues on next page)

(continued from previous page)

```
ax.axhline(y=.5, xmin=0.25, xmax=0.75)
# Infinite black line going through (0, 0) to (1, 1).
ax.axline((0, 0), (1, 1), color='k')
# 50%-gray rectangle spanning the Axes' width from y=0.25 to y=0.75.
ax.axhspan(0.25, 0.75, facecolor='0.5')
# Green rectangle spanning the Axes' height from x=1.25 to x=1.55.
ax.axvspan(1.25, 1.55, facecolor='#2ca02c')

plt.show()
```

### Equal axis aspect ratio

How to set and adjust plots with equal axis aspect ratios.




```
import matplotlib.pyplot as plt
import numpy as np

# Plot circle of radius 3.

an = np.linspace(0, 2 * np.pi, 100)
```

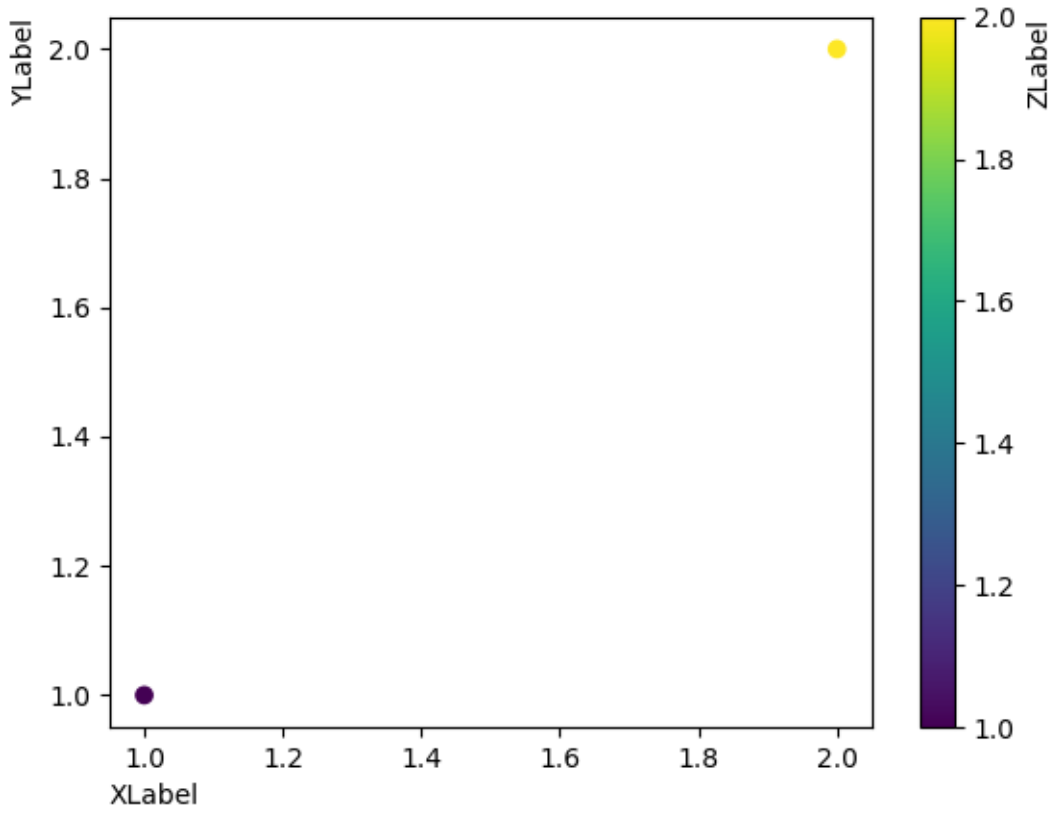
(continues on next page)

(continued from previous page)

```
fig, axs = plt.subplots(2, 2)

axs[0, 0].plot(3 * np.cos(an), 3 * np.sin(an))
axs[0, 0].set_title('not equal, looks like ellipse', fontsize=10)

axs[0, 1].plot(3 * np.cos(an), 3 * np.sin(an))
axs[0, 1].axis('equal')
axs[0, 1].set_title('equal, looks like circle', fontsize=10)

axs[1, 0].plot(3 * np.cos(an), 3 * np.sin(an))
axs[1, 0].axis('equal')
axs[1, 0].set(xlim=(-3, 3), ylim=(-3, 3))
axs[1, 0].set_title('still a circle, even after changing limits', fontsize=10)

axs[1, 1].plot(3 * np.cos(an), 3 * np.sin(an))
axs[1, 1].set_aspect('equal', 'box')
axs[1, 1].set_title('still a circle, auto-adjusted data limits', fontsize=10)

fig.tight_layout()

plt.show()
```

### Axis Label Position

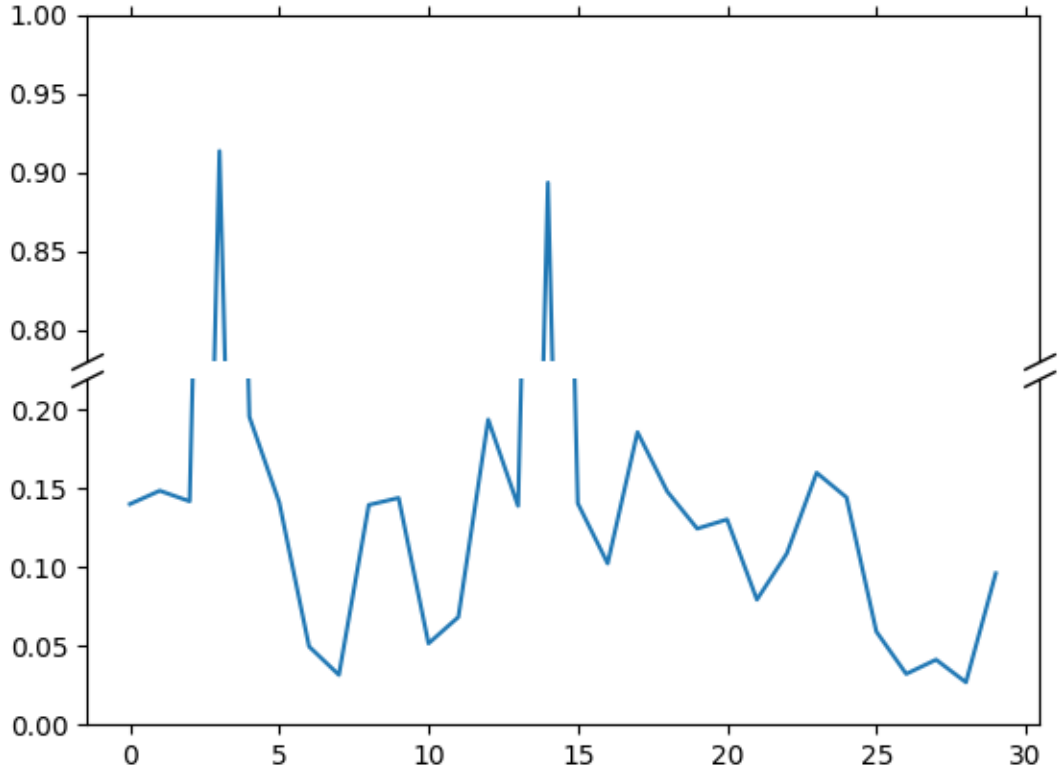
Choose axis label position when calling *set_xlabel* and *set_ylabel* as well as for colorbar.




```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()

sc = ax.scatter([1, 2], [1, 2], c=[1, 2])
ax.set_ylabel('YLabel', loc='top')
ax.set_xlabel('XLabel', loc='left')
cbar = fig.colorbar(sc)
cbar.set_label("ZLabel", loc='top')

plt.show()
```

### Broken Axis

Broken axis example, where the y-axis will have a portion cut out.




```
import matplotlib.pyplot as plt
import numpy as np

np.random.seed(19680801)

pts = np.random.rand(30)*.2
# Now let's make two outlier points which are far away from everything.
pts[[3, 14]] += .8

# If we were to simply plot pts, we'd lose most of the interesting
# details due to the outliers. So let's 'break' or 'cut-out' the y-axis
# into two portions - use the top (ax1) for the outliers, and the bottom
# (ax2) for the details of the majority of our data
fig, (ax1, ax2) = plt.subplots(2, 1, sharex=True)
fig.subplots_adjust(hspace=0.05)  # adjust space between Axes

# plot the same data on both Axes
ax1.plot(pts)
ax2.plot(pts)
```

(continues on next page)

(continued from previous page)

```
# zoom-in / limit the view to different portions of the data
ax1.set_ylim(.78, 1.)  # outliers only
ax2.set_ylim(0, .22)  # most of the data

# hide the spines between ax and ax2
ax1.spines.bottom.set_visible(False)
ax2.spines.top.set_visible(False)
ax1.xaxis.tick_top()
ax1.tick_params(labeltop=False)  # don't put tick labels at the top
ax2.xaxis.tick_bottom()

# Now, let's turn towards the cut-out slanted lines.
# We create line objects in axes coordinates, in which (0,0), (0,1),
# (1,0), and (1,1) are the four corners of the Axes.
# The slanted lines themselves are markers at those locations, such that the
# lines keep their angle and position, independent of the Axes size or scale
# Finally, we need to disable clipping.

d = .5  # proportion of vertical to horizontal extent of the slanted line
kwargs = dict(marker=[(-1, -d), (1, d)], markersize=12,
              linestyle="none", color='k', mec='k', mew=1, clip_on=False)
ax1.plot([0, 1], [0, 0], transform=ax1.transAxes, **kwargs)
ax2.plot([0, 1], [1, 1], transform=ax2.transAxes, **kwargs)


plt.show()
```

### Custom Figure subclasses

You can pass a *Figure* subclass to *pyplot.figure* if you want to change the default behavior of the figure.

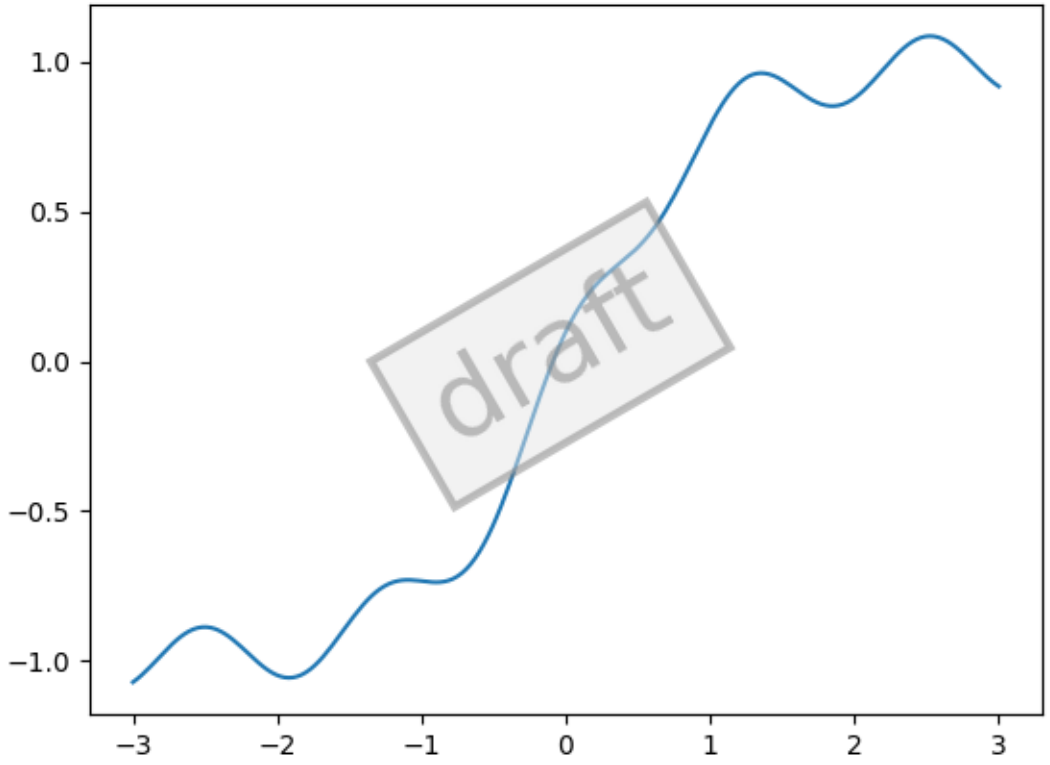
This example defines a *Figure* subclass `WatermarkFigure` that accepts an additional parameter `watermark` to display a custom watermark text. The figure is created using the `FigureClass` parameter of *pyplot.figure*. The additional `watermark` parameter is passed on to the subclass constructor.

```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.figure import Figure


class WatermarkFigure(Figure):
    """A figure with a text watermark."""

    def __init__(self, *args, watermark=None, **kwargs):
        super().__init__(*args, **kwargs)

        if watermark is not None:
```

(continues on next page)

(continued from previous page)

```
            bbox = dict(boxstyle='square', lw=3, ec='gray',
                        fc=(0.9, 0.9, .9, .5), alpha=0.5)
            self.text(0.5, 0.5, watermark,
                      ha='center', va='center', rotation=30,
                      fontsize=40, color='gray', alpha=0.5, bbox=bbox)


x = np.linspace(-3, 3, 201)
y = np.tanh(x) + 0.1 * np.cos(5 * x)

plt.figure(FigureClass=WatermarkFigure, watermark='draft')
plt.plot(x, y)
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.pyplot.figure*
- *matplotlib.figure.Figure*
- *matplotlib.figure.Figure.text*

### Resizing Axes with constrained layout

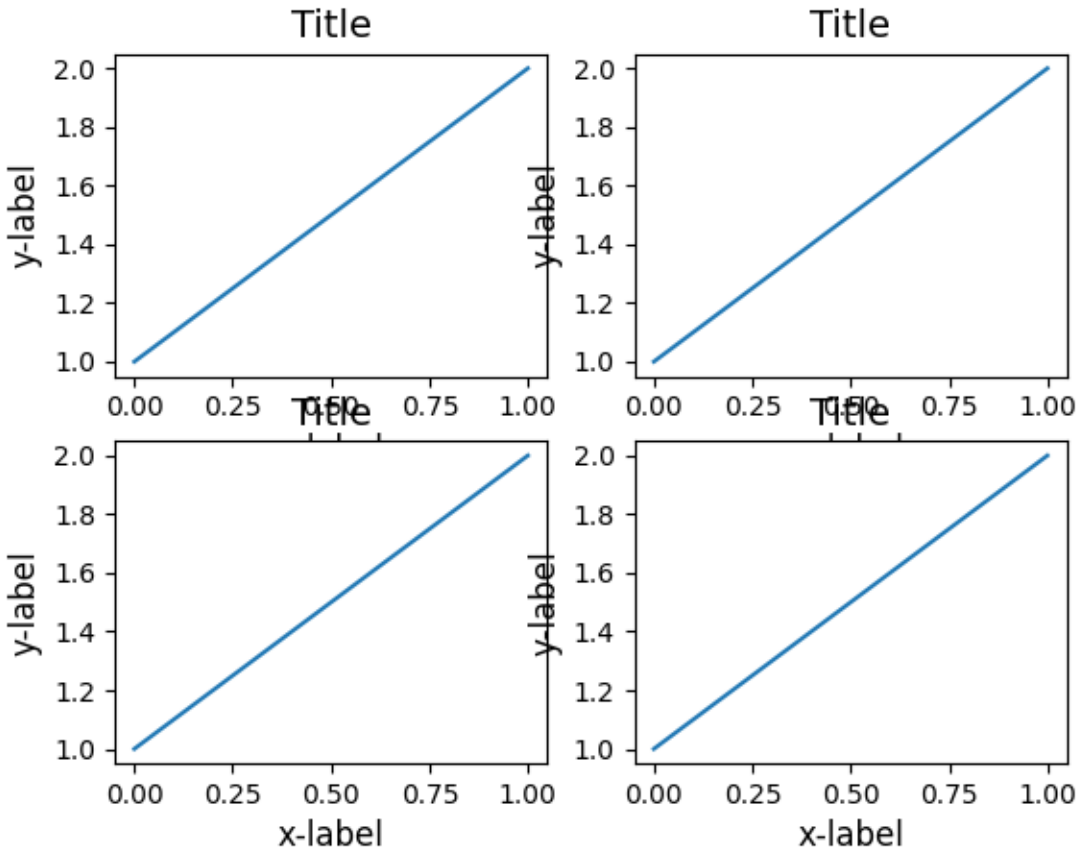
*Constrained layout* attempts to resize subplots in a figure so that there are no overlaps between Axes objects and labels on the Axes.

See *Constrained layout guide* for more details and *Tight layout guide* for an alternative.

```
import matplotlib.pyplot as plt


def example_plot(ax):
    ax.plot([1, 2])
    ax.set_xlabel('x-label', fontsize=12)
    ax.set_ylabel('y-label', fontsize=12)
    ax.set_title('Title', fontsize=14)
```

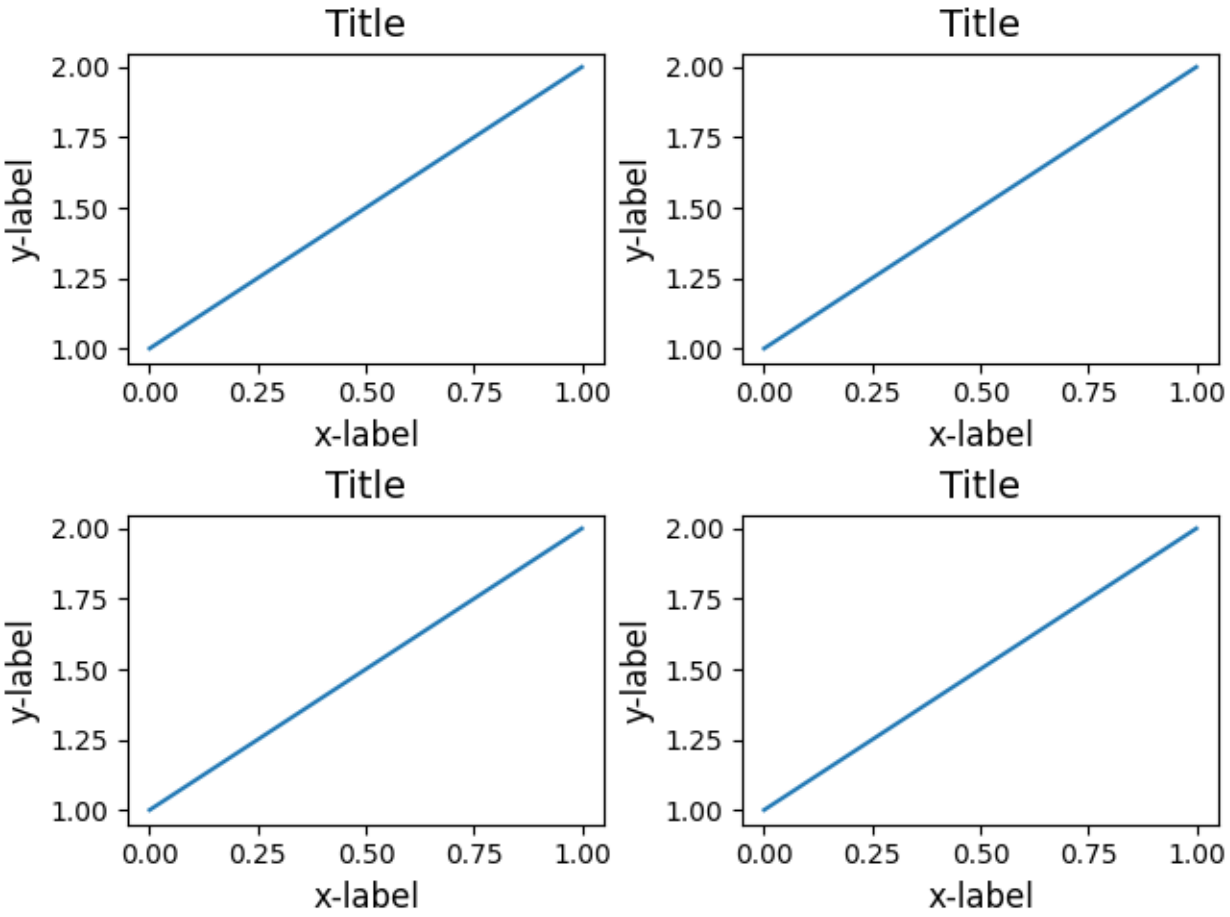
If we don't use *constrained layout*, then labels overlap the Axes

```
fig, axs = plt.subplots(nrows=2, ncols=2, layout=None)

for ax in axs.flat:
    example_plot(ax)
```




adding `layout='constrained'` automatically adjusts.

```
fig, axs = plt.subplots(nrows=2, ncols=2, layout='constrained')

for ax in axs.flat:
    example_plot(ax)
```



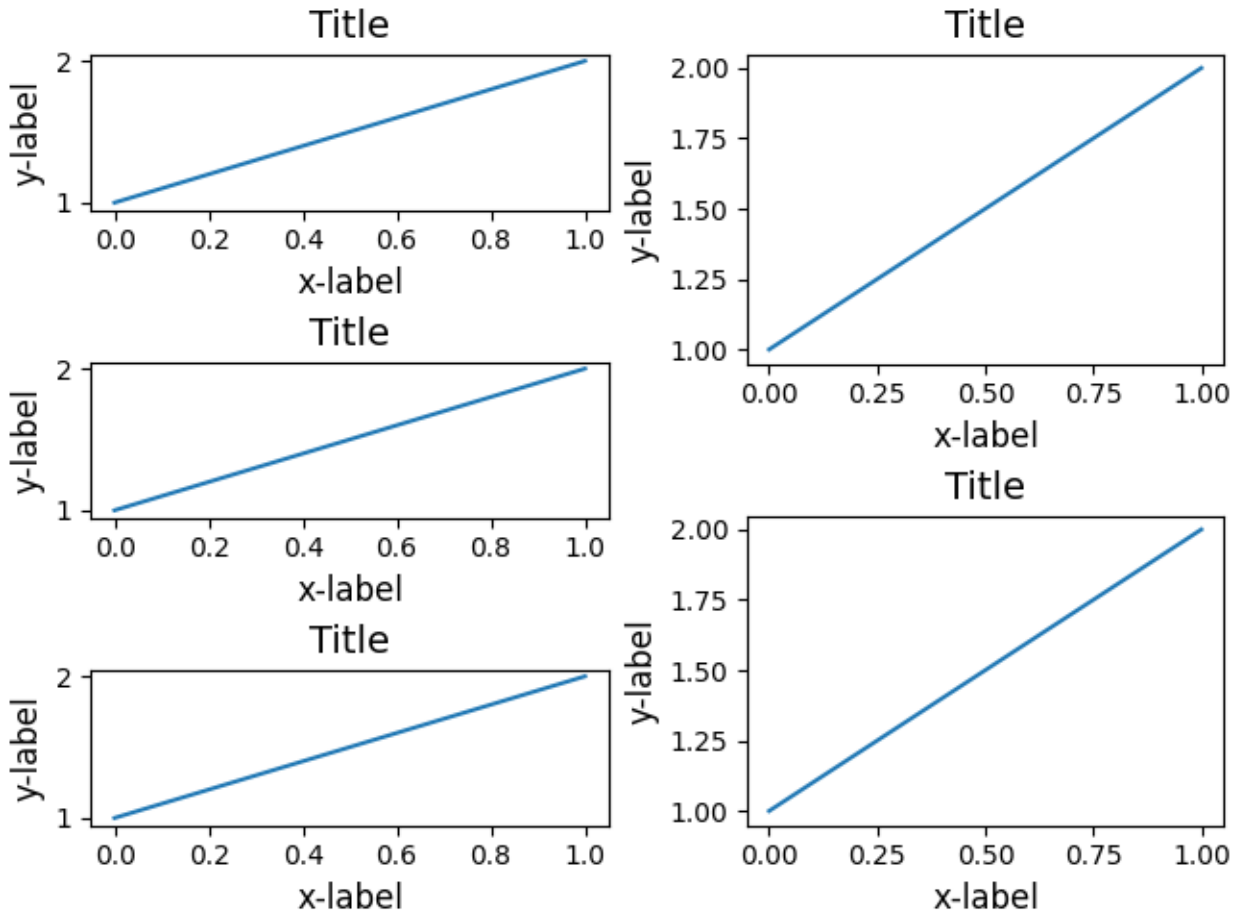

Below is a more complicated example using nested gridspecs.

```
fig = plt.figure(layout='constrained')

import matplotlib.gridspec as gridspec

gs0 = gridspec.GridSpec(1, 2, figure=fig)

gs1 = gridspec.GridSpecFromSubplotSpec(3, 1, subplot_spec=gs0[0])
for n in range(3):
    ax = fig.add_subplot(gs1[n])
    example_plot(ax)


gs2 = gridspec.GridSpecFromSubplotSpec(2, 1, subplot_spec=gs0[1])
for n in range(2):
    ax = fig.add_subplot(gs2[n])
    example_plot(ax)
```

(continues on next page)

(continued from previous page)

```
plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.gridspec.GridSpec*
- *matplotlib.gridspec.GridSpecFromSubplotSpec*

**Total running time of the script:** (0 minutes 1.871 seconds)

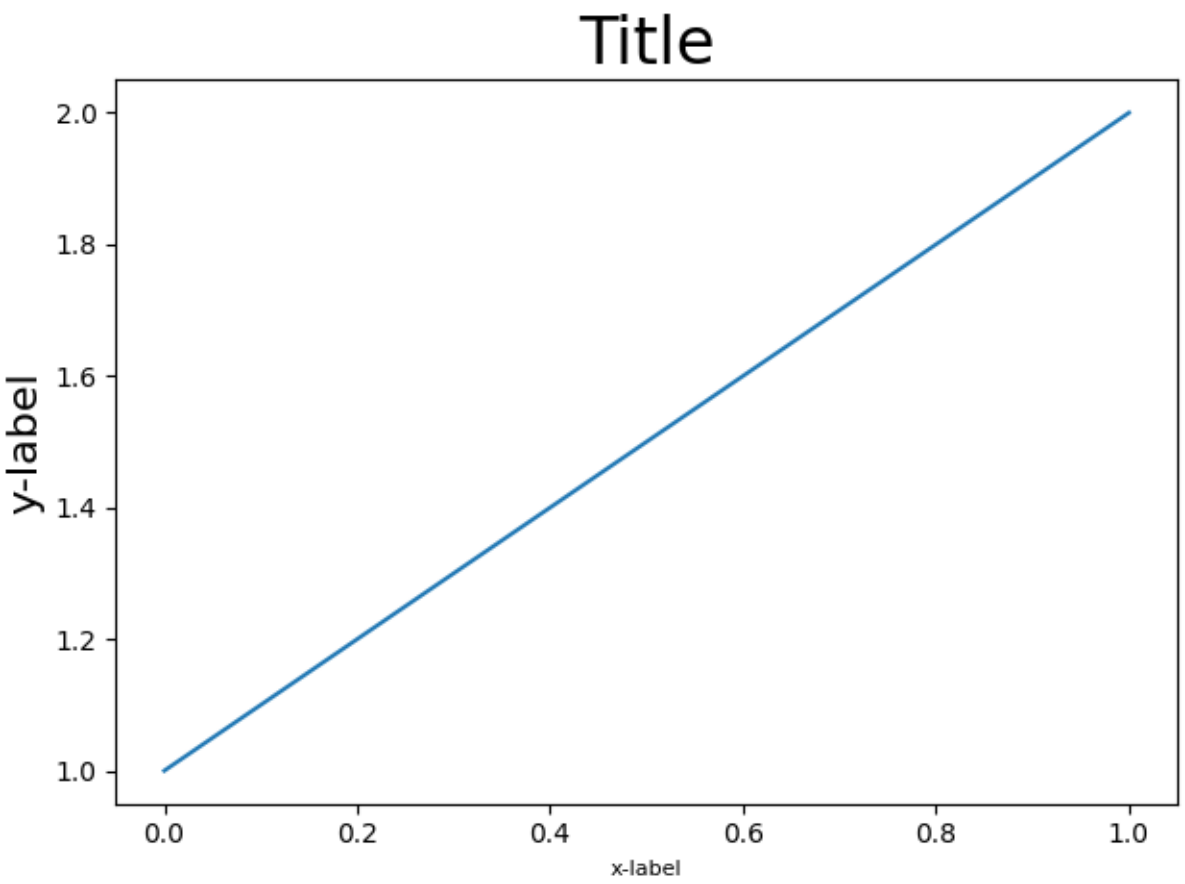
### Resizing Axes with tight layout

*tight_layout* attempts to resize subplots in a figure so that there are no overlaps between Axes objects and labels on the Axes.

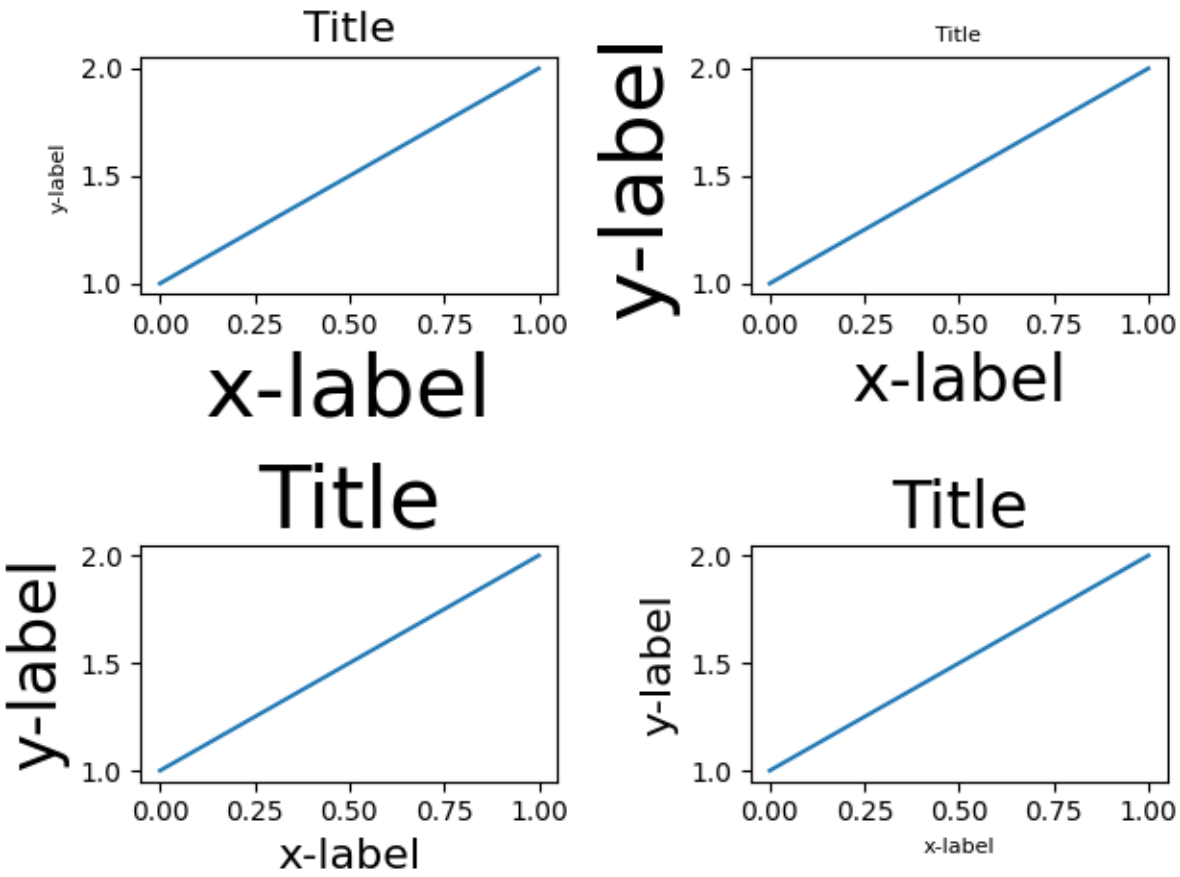
See *Tight layout guide* for more details and *Constrained layout guide* for an alternative.

```
import itertools
import warnings

import matplotlib.pyplot as plt

fontsizes = itertools.cycle([8, 16, 24, 32])


def example_plot(ax):
    ax.plot([1, 2])
    ax.set_xlabel('x-label', fontsize=next(fontsizes))
    ax.set_ylabel('y-label', fontsize=next(fontsizes))
    ax.set_title('Title', fontsize=next(fontsizes))
```

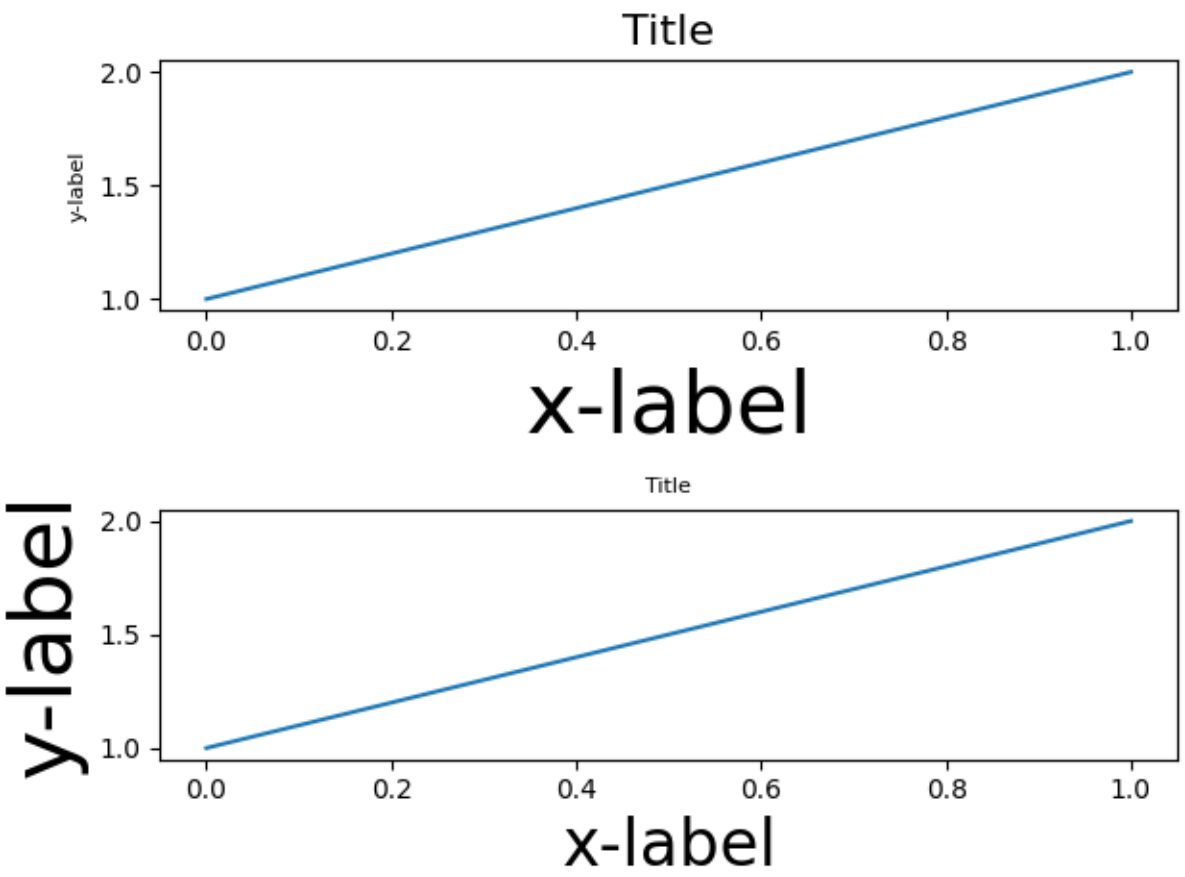
```
fig, ax = plt.subplots()
example_plot(ax)
fig.tight_layout()
```



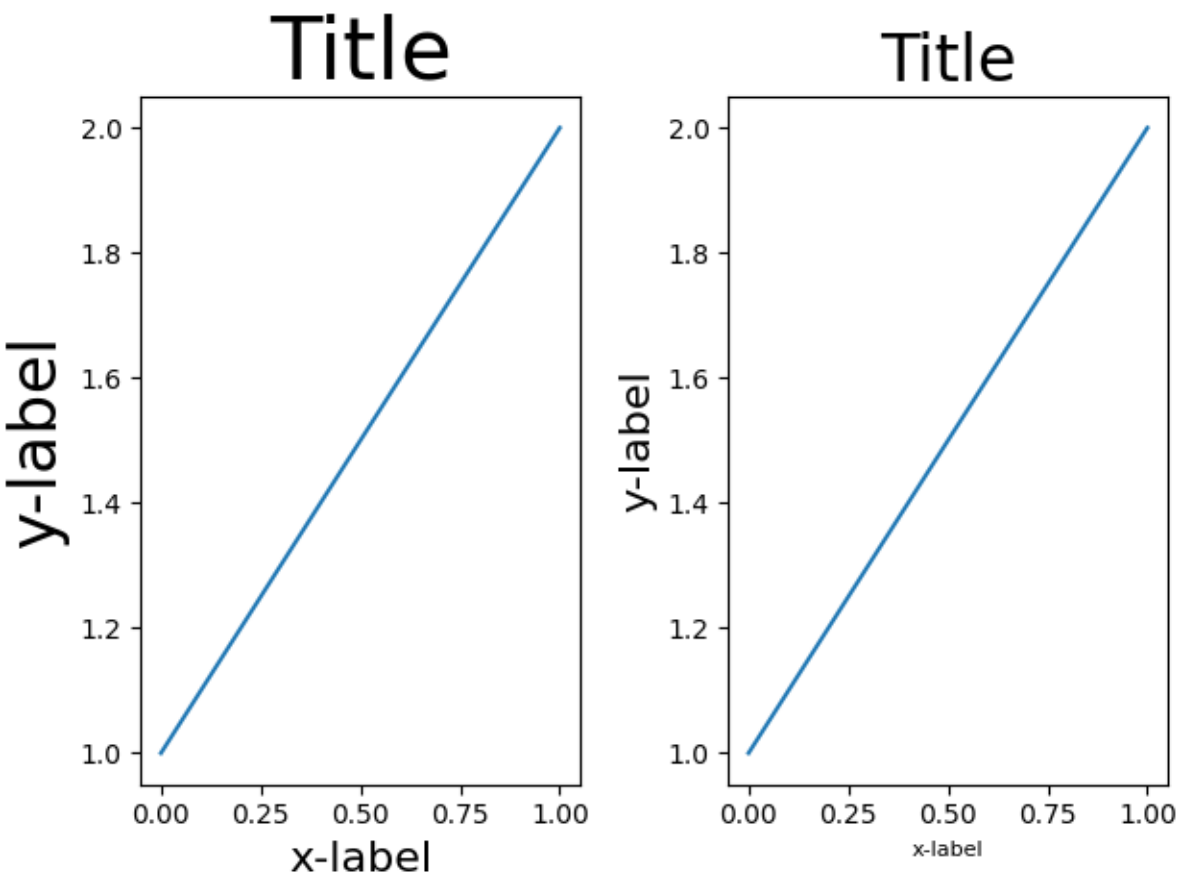

```
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(nrows=2, ncols=2)
example_plot(ax1)
example_plot(ax2)
example_plot(ax3)
example_plot(ax4)
fig.tight_layout()
```



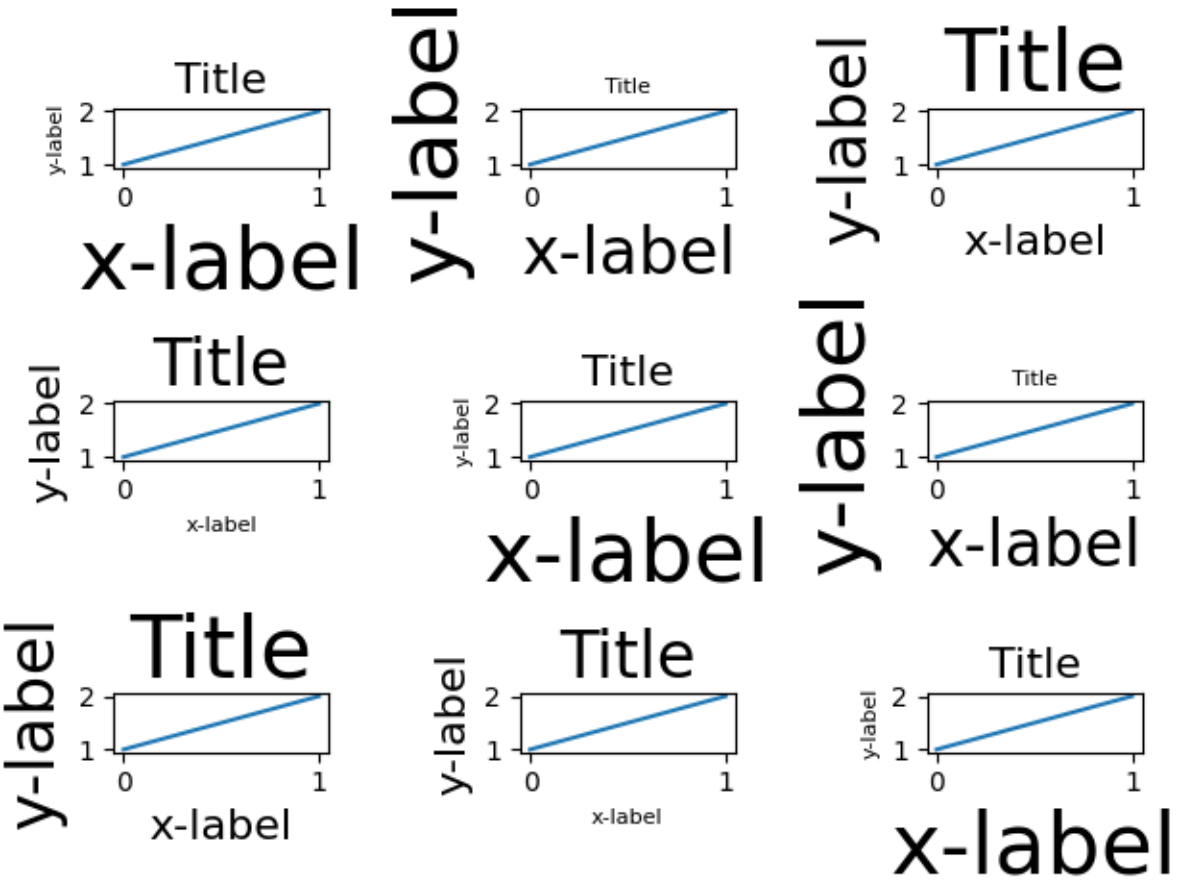

```
fig, (ax1, ax2) = plt.subplots(nrows=2, ncols=1)
example_plot(ax1)
example_plot(ax2)
fig.tight_layout()
```



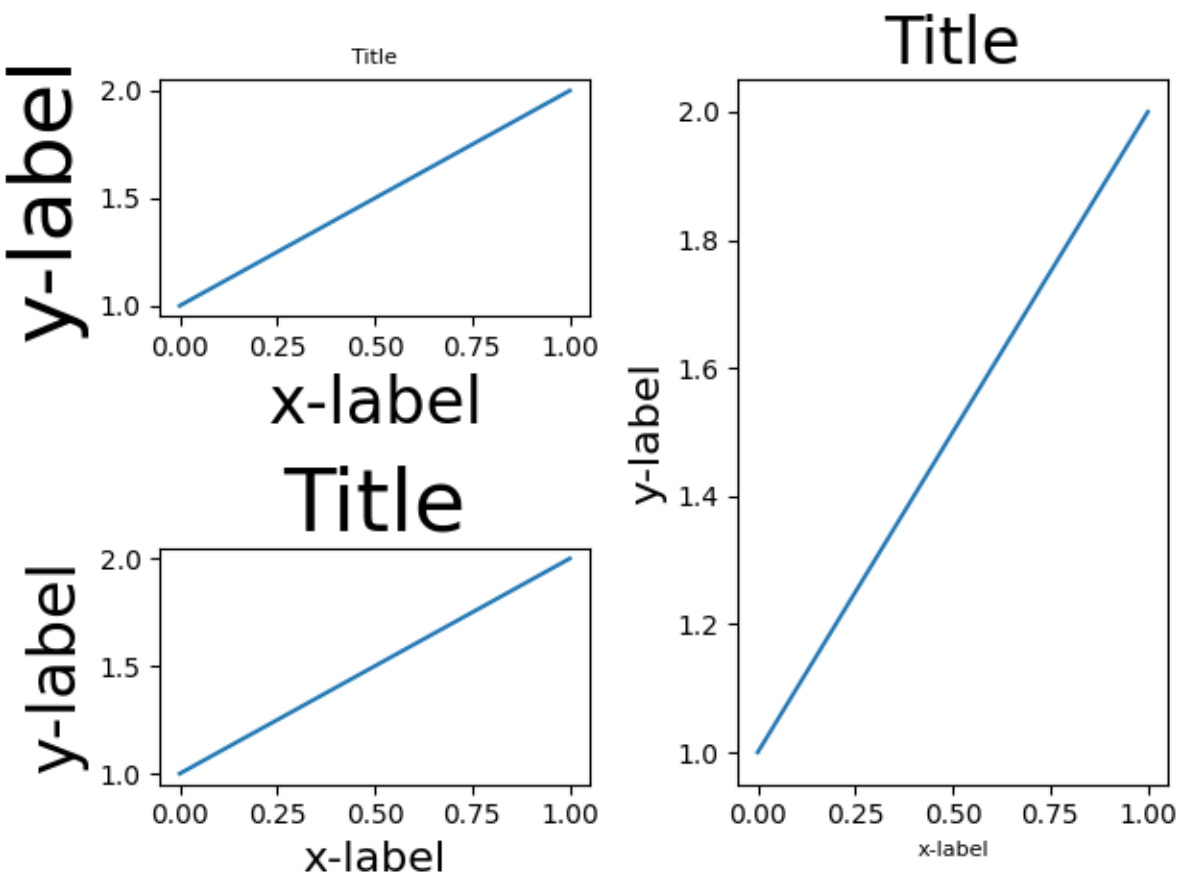

```
fig, (ax1, ax2) = plt.subplots(nrows=1, ncols=2)
example_plot(ax1)
example_plot(ax2)
fig.tight_layout()
```



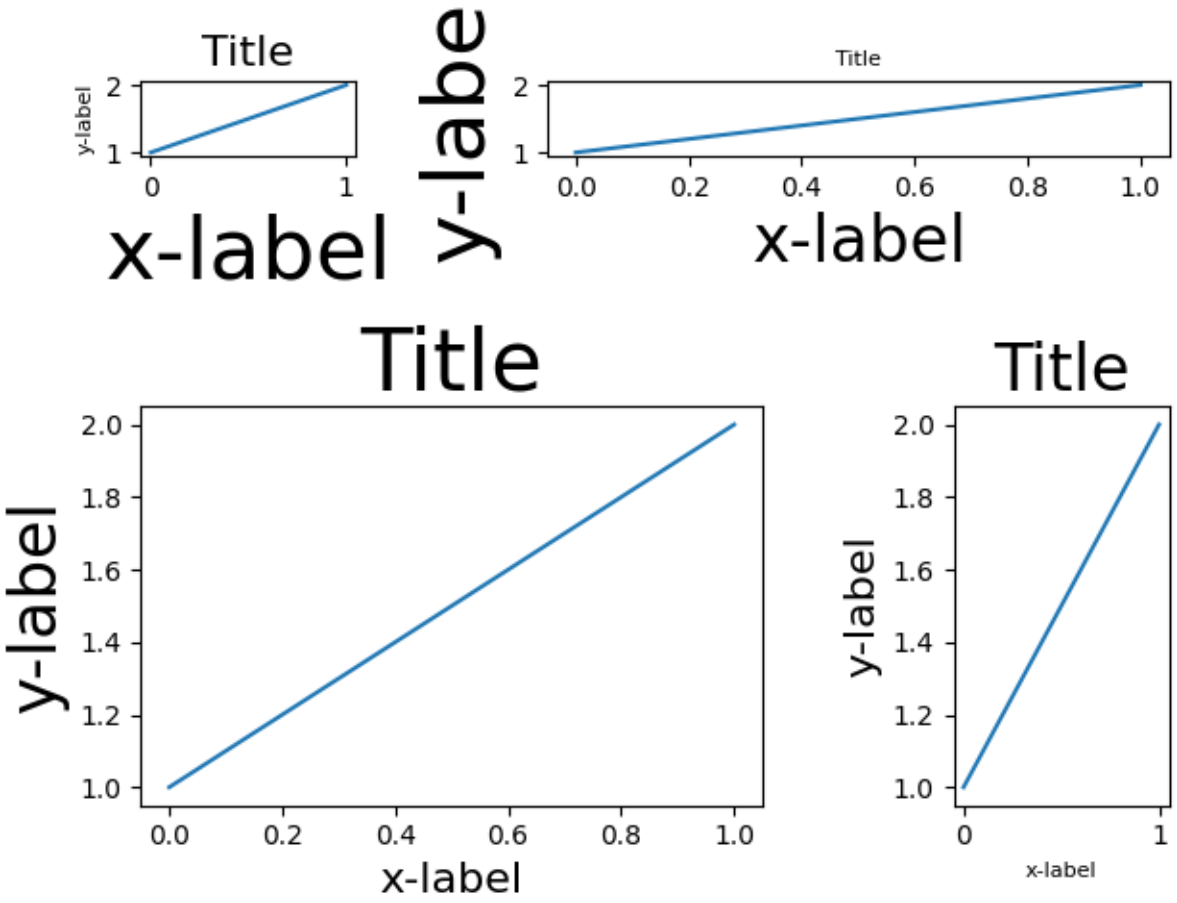

```
fig, axs = plt.subplots(nrows=3, ncols=3)
for ax in axs.flat:
    example_plot(ax)
fig.tight_layout()
```




```
plt.figure()
ax1 = plt.subplot(221)
ax2 = plt.subplot(223)
ax3 = plt.subplot(122)
example_plot(ax1)
example_plot(ax2)
example_plot(ax3)
plt.tight_layout()
```




```
plt.figure()
ax1 = plt.subplot2grid((3, 3), (0, 0))
ax2 = plt.subplot2grid((3, 3), (0, 1), colspan=2)
ax3 = plt.subplot2grid((3, 3), (1, 0), colspan=2, rowspan=2)
ax4 = plt.subplot2grid((3, 3), (1, 2), rowspan=2)
example_plot(ax1)
example_plot(ax2)
example_plot(ax3)
example_plot(ax4)
plt.tight_layout()
```

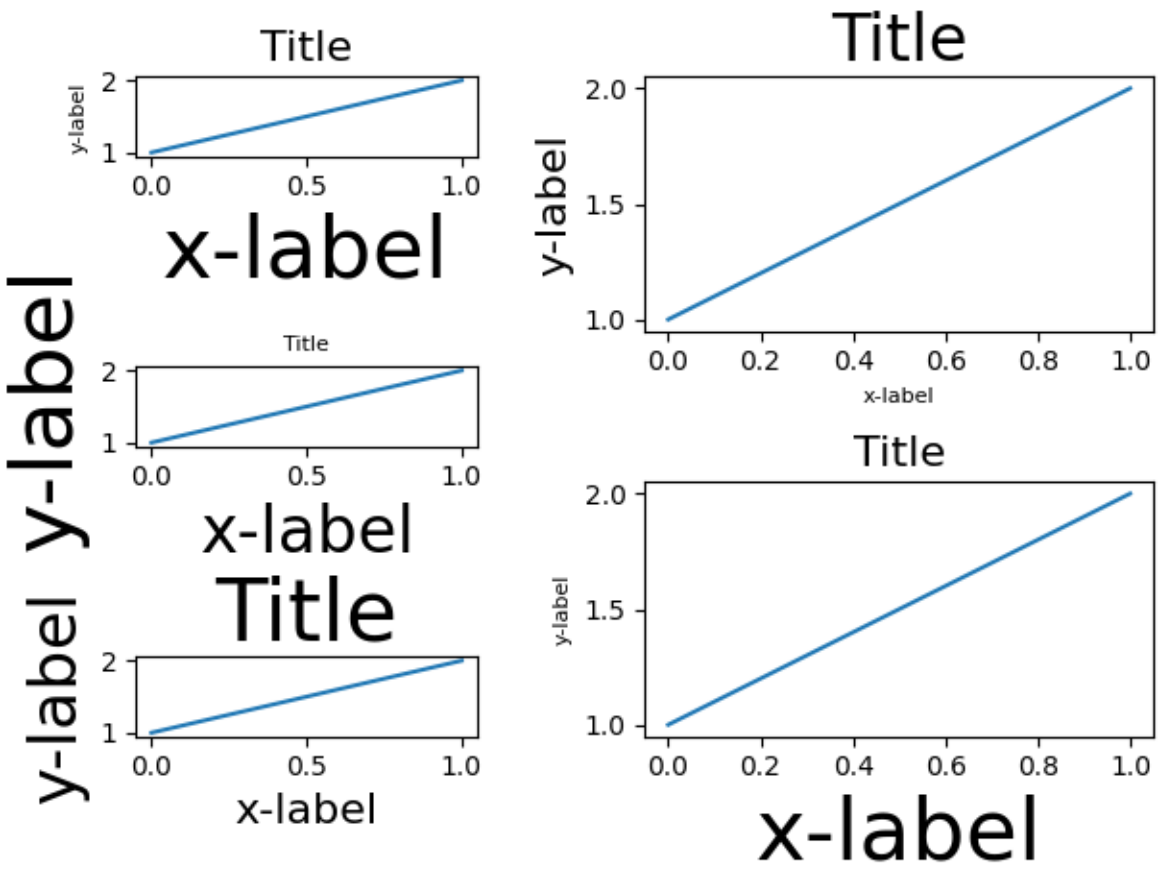



```
fig = plt.figure()

gs1 = fig.add_gridspec(3, 1)
ax1 = fig.add_subplot(gs1[0])
ax2 = fig.add_subplot(gs1[1])
ax3 = fig.add_subplot(gs1[2])
example_plot(ax1)
example_plot(ax2)
example_plot(ax3)
gs1.tight_layout(fig, rect=[None, None, 0.45, None])

gs2 = fig.add_gridspec(2, 1)
ax4 = fig.add_subplot(gs2[0])
ax5 = fig.add_subplot(gs2[1])
example_plot(ax4)
example_plot(ax5)
with warnings.catch_warnings():
    # gs2.tight_layout cannot handle the subplots from the first gridspec
    # (gs1), so it will raise a warning. We are going to match the gridspecs
    # manually so we can filter the warning away.
    warnings.simplefilter("ignore", UserWarning)
    gs2.tight_layout(fig, rect=[0.45, None, None, None])
```

(continues on next page)

(continued from previous page)

```
# now match the top and bottom of two gridspecs.
top = min(gs1.top, gs2.top)
bottom = max(gs1.bottom, gs2.bottom)

gs1.update(top=top, bottom=bottom)
gs2.update(top=top, bottom=bottom)

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

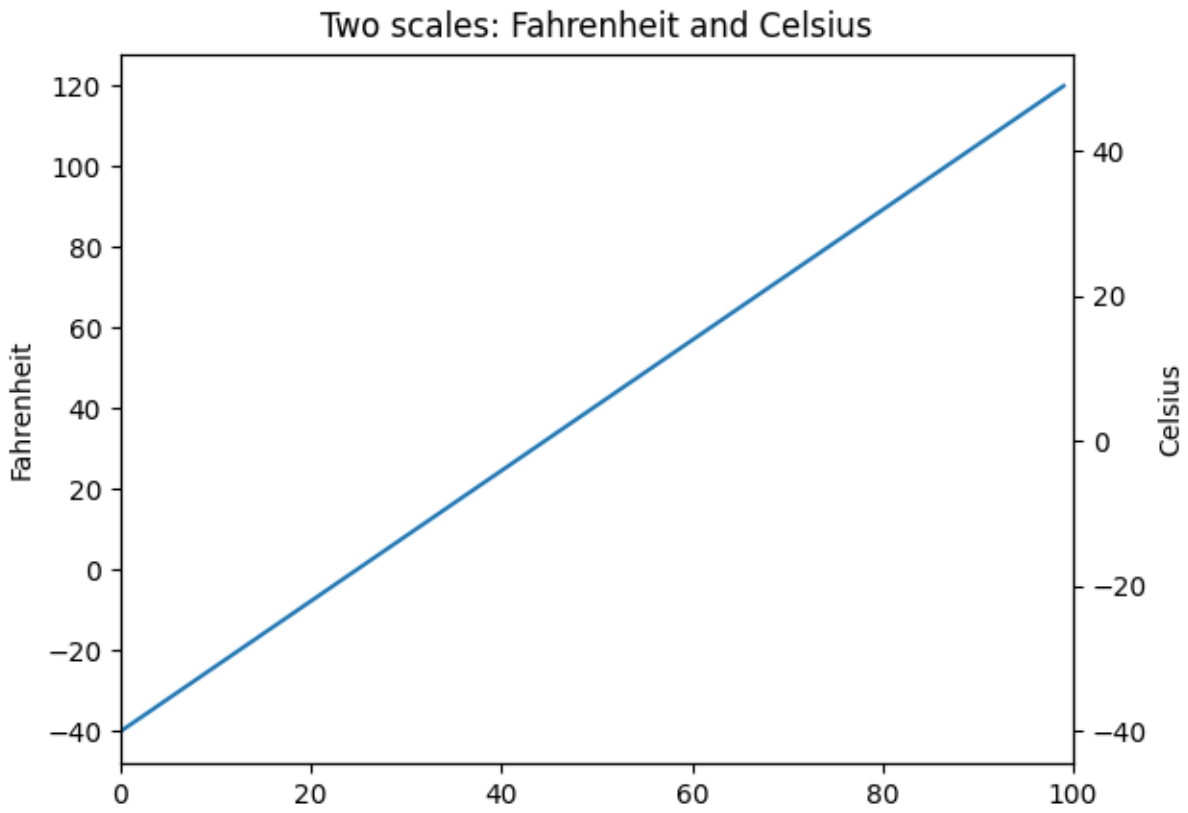
- *matplotlib.figure.Figure.tight_layout* / *matplotlib.pyplot.tight_layout*
- *matplotlib.figure.Figure.add_gridspec*
- *matplotlib.figure.Figure.add_subplot*
- *matplotlib.pyplot.subplot2grid*

**Total running time of the script:** (0 minutes 4.600 seconds)

### Different scales on the same Axes

Demo of how to display two scales on the left and right y-axis.

This example uses the Fahrenheit and Celsius scales.




```
import matplotlib.pyplot as plt
import numpy as np


def fahrenheit2celsius(temp):
    """
    Returns temperature in Celsius given Fahrenheit temperature.
    """
    return (5. / 9.) * (temp - 32)


def make_plot():

    # Define a closure function to register as a callback
```

(continues on next page)

(continued from previous page)

```
    def convert_ax_c_to_celsius(ax_f):
        """
        Update second axis according to first axis.
        """
        y1, y2 = ax_f.get_ylim()
        ax_c.set_ylim(fahrenheit2celsius(y1), fahrenheit2celsius(y2))
        ax_c.figure.canvas.draw()

    fig, ax_f = plt.subplots()
    ax_c = ax_f.twinx()

    # automatically update ylim of ax2 when ylim of ax1 changes.
    ax_f.callbacks.connect("ylim_changed", convert_ax_c_to_celsius)
    ax_f.plot(np.linspace(-40, 120, 100))
    ax_f.set_xlim(0, 100)

    ax_f.set_title('Two scales: Fahrenheit and Celsius')
    ax_f.set_ylabel('Fahrenheit')
    ax_c.set_ylabel('Celsius')

    plt.show()

make_plot()
```

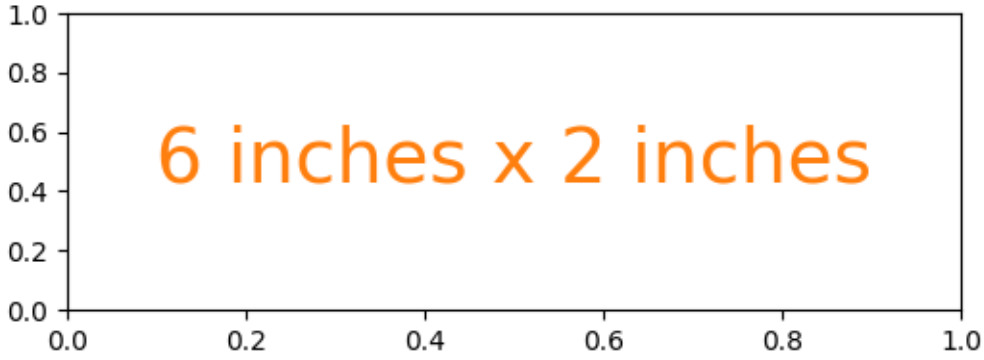
### Figure size in different units

The native figure size unit in Matplotlib is inches, deriving from print industry standards. However, users may need to specify their figures in other units like centimeters or pixels. This example illustrates how to do this efficiently.

```
import matplotlib.pyplot as plt

text_kwargs = dict(ha='center', va='center', fontsize=28, color='C1')
```

### Figure size in inches (default)

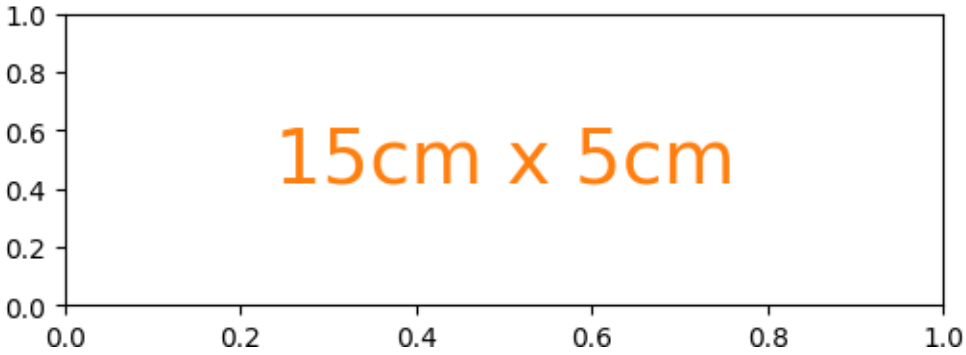
```
plt.subplots(figsize=(6, 2))
plt.text(0.5, 0.5, '6 inches x 2 inches', **text_kwargs)
plt.show()
```




### Figure size in centimeter

Multiplying centimeter-based numbers with a conversion factor from cm to inches, gives the right numbers. Naming the conversion factor `cm` makes the conversion almost look like appending a unit to the number, which is nicely readable.

```
cm = 1/2.54  # centimeters in inches
plt.subplots(figsize=(15*cm, 5*cm))
plt.text(0.5, 0.5, '15cm x 5cm', **text_kwargs)
plt.show()
```

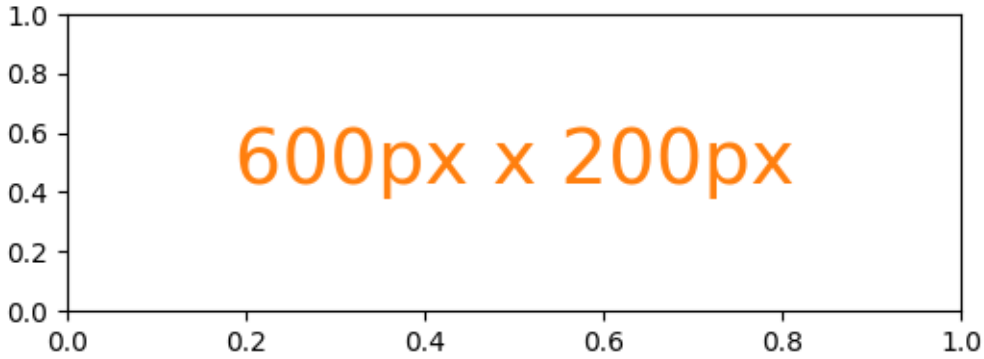


### Figure size in pixel

Similarly, one can use a conversion from pixels.

Note that you could break this if you use *savefig* with a different explicit dpi value.

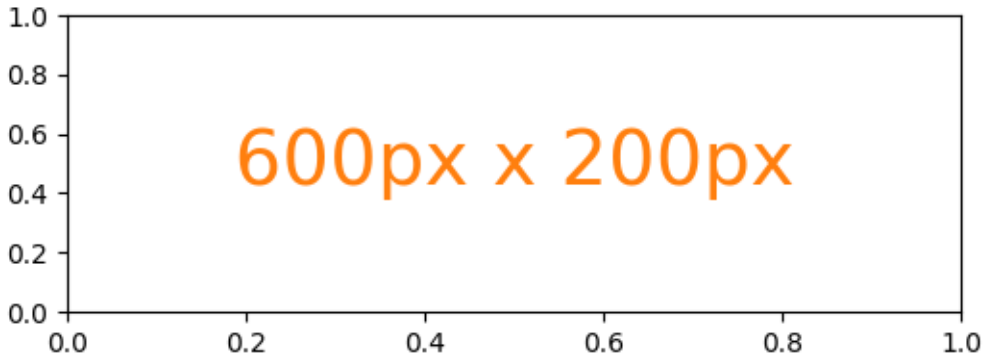
```
px = 1/plt.rcParams['figure.dpi']  # pixel in inches
plt.subplots(figsize=(600*px, 200*px))
plt.text(0.5, 0.5, '600px x 200px', **text_kwargs)
plt.show()
```




Quick interactive work is usually rendered to the screen, making pixels a good size of unit. But defining the conversion factor may feel a little tedious for quick iterations.

Because of the default `rcParams['figure.dpi'] = 100`, one can mentally divide the needed pixel value by 100[1]:

```
plt.subplots(figsize=(6, 2))
plt.text(0.5, 0.5, '600px x 200px', **text_kwargs)
plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

[1] Unfortunately, this does not work well for the `matplotlib inline` backend in Jupyter because that backend saves the figure with `bbox_inches='tight'`, which crops the figure and makes the actual size unpredictable.

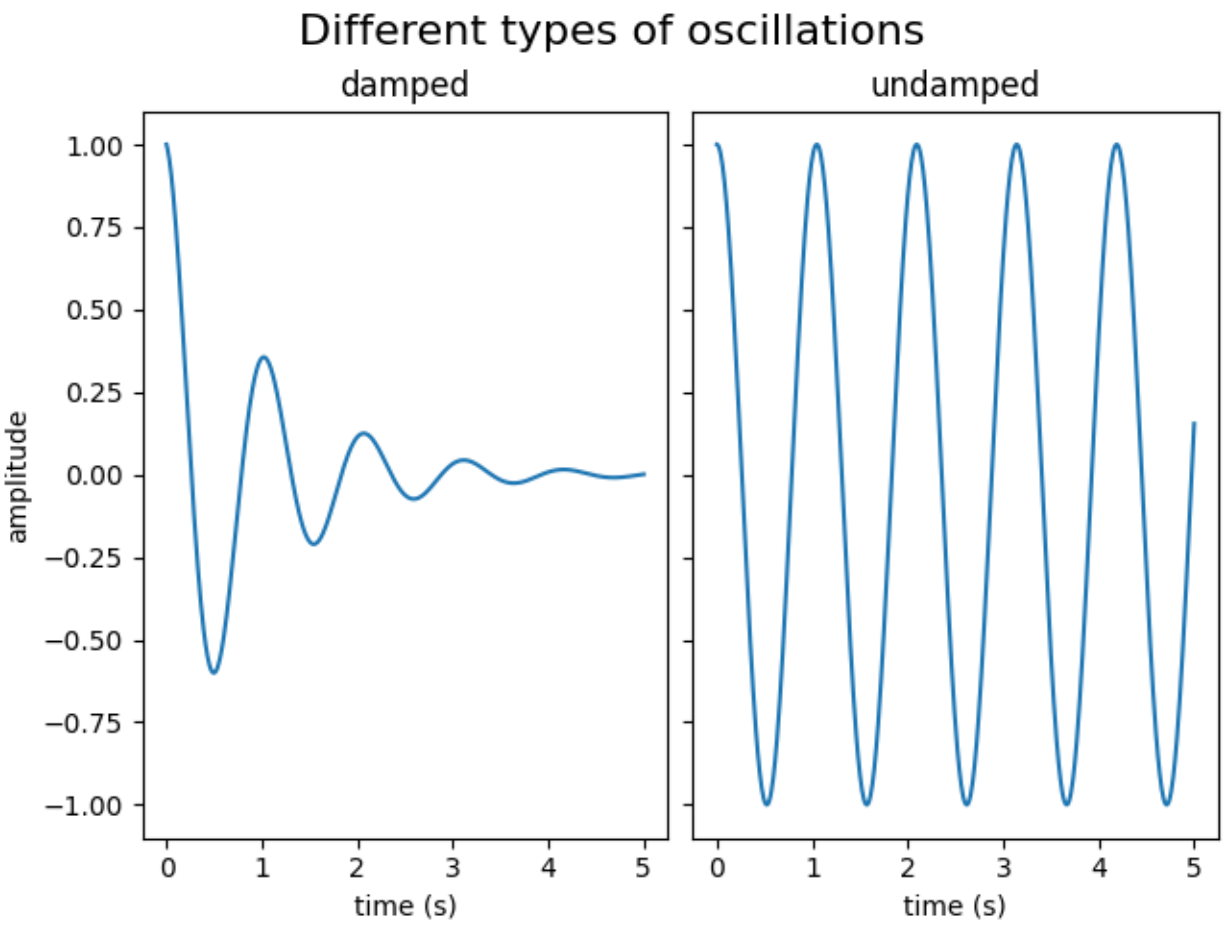
- *matplotlib.pyplot.figure*
- *matplotlib.pyplot.subplots*
- *matplotlib.pyplot.subplot_mosaic*

### Figure labels: suptitle, supxlabel, supylabel

Each Axes can have a title (or actually three - one each with *loc* "left", "center", and "right"), but is sometimes desirable to give a whole figure (or *SubFigure*) an overall title, using *Figure.suptitle*.

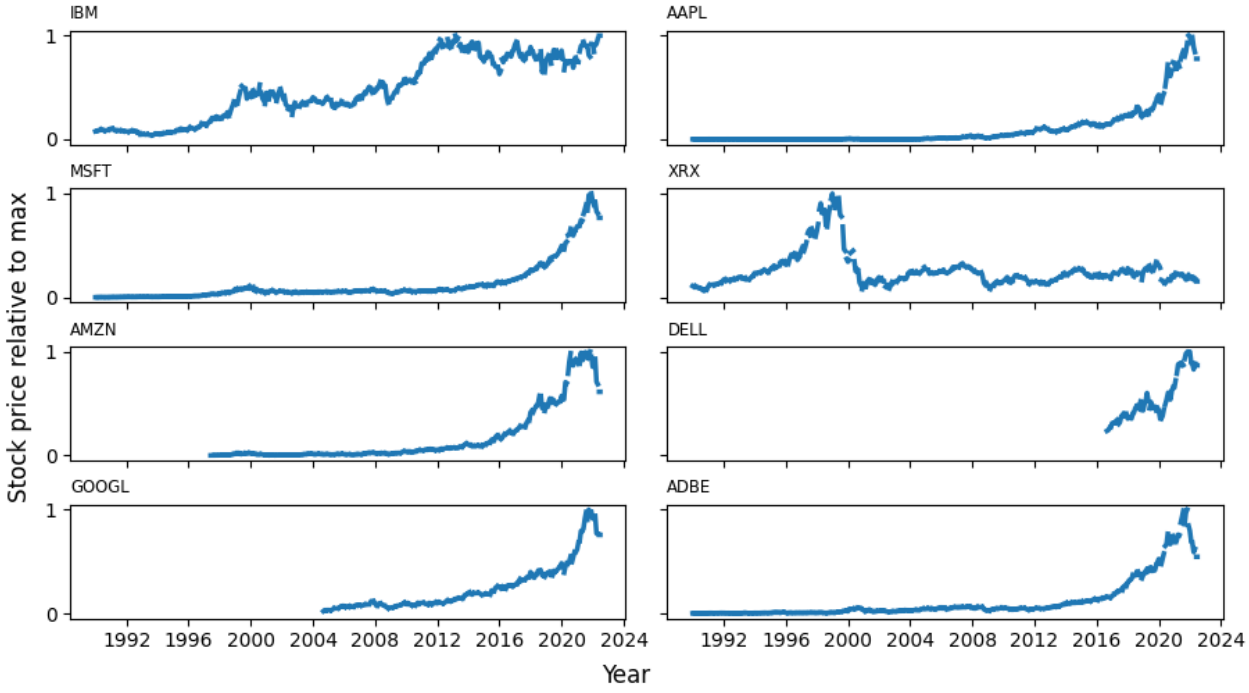
We can also add figure-level x- and y-labels using *Figure.supxlabel* and *Figure.supylabel*.

```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.cbook import get_sample_data

x = np.linspace(0.0, 5.0, 501)

fig, (ax1, ax2) = plt.subplots(1, 2, layout='constrained', sharey=True)
ax1.plot(x, np.cos(6*x) * np.exp(-x))
ax1.set_title('damped')
ax1.set_xlabel('time (s)')
ax1.set_ylabel('amplitude')

ax2.plot(x, np.cos(6*x))
ax2.set_xlabel('time (s)')
ax2.set_title('undamped')

fig.suptitle('Different types of oscillations', fontsize=16)
```




A global x- or y-label can be set using the *Figure.supxlabel* and *Figure.supylabel* methods.

```
with get_sample_data('Stocks.csv') as file:
    stocks = np.genfromtxt(
        file, delimiter=',', names=True, dtype=None,
        converters={0: lambda x: np.datetime64(x, 'D')}, skip_header=1)

fig, axs = plt.subplots(4, 2, figsize=(9, 5), layout='constrained',
                        sharex=True, sharey=True)
for nn, ax in enumerate(axs.flat):
    column_name = stocks.dtype.names[1+nn]
    y = stocks[column_name]
    line, = ax.plot(stocks['Date'], y / np.nanmax(y), lw=2.5)
    ax.set_title(column_name, fontsize='small', loc='left')
fig.supxlabel('Year')
fig.supylabel('Stock price relative to max')

plt.show()
```

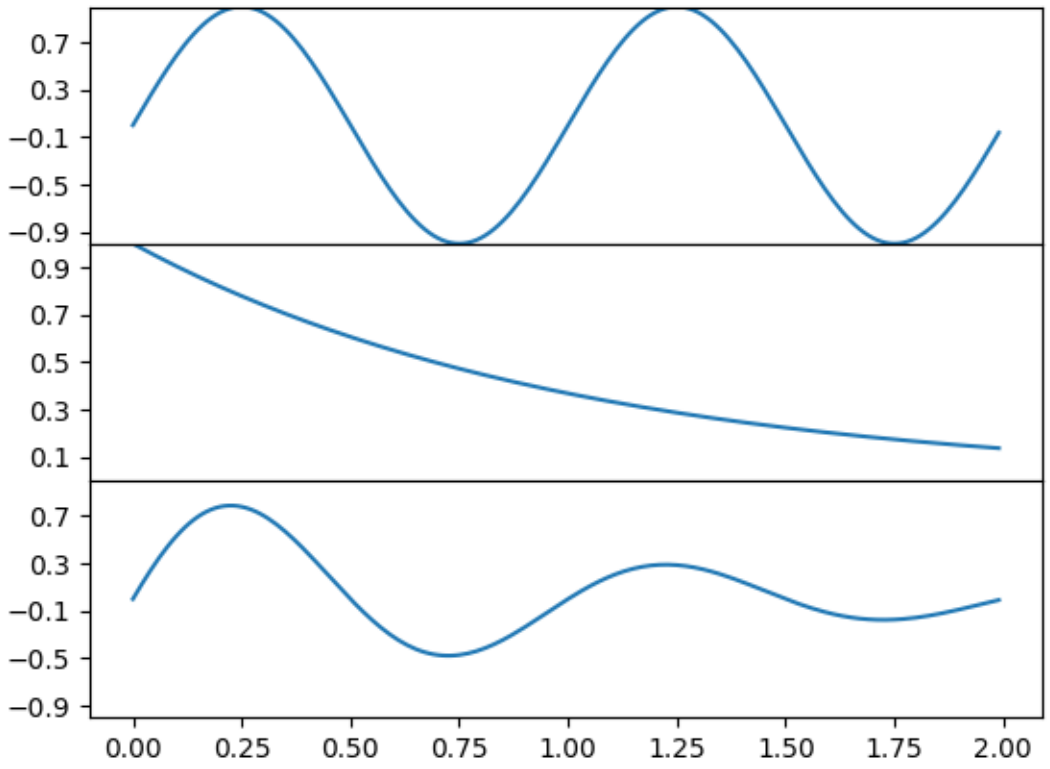



**Total running time of the script:** (0 minutes 3.007 seconds)

## Creating adjacent subplots

To create plots that share a common axis (visually) you can set the hspace between the subplots to zero. Passing sharex=True when creating the subplots will automatically turn off all x ticks and labels except those on the bottom axis.

In this example the plots share a common x-axis, but you can follow the same logic to supply a common y-axis.




```
import matplotlib.pyplot as plt
import numpy as np

t = np.arange(0.0, 2.0, 0.01)

s1 = np.sin(2 * np.pi * t)
s2 = np.exp(-t)
s3 = s1 * s2

fig, axs = plt.subplots(3, 1, sharex=True)
# Remove vertical space between Axes
fig.subplots_adjust(hspace=0)

# Plot each graph, and manually set the y tick values
axs[0].plot(t, s1)
axs[0].set_yticks(np.arange(-0.9, 1.0, 0.4))
axs[0].set_ylim(-1, 1)

axs[1].plot(t, s2)
axs[1].set_yticks(np.arange(0.1, 1.0, 0.2))
axs[1].set_ylim(0, 1)

axs[2].plot(t, s3)
```

(continues on next page)

(continued from previous page)

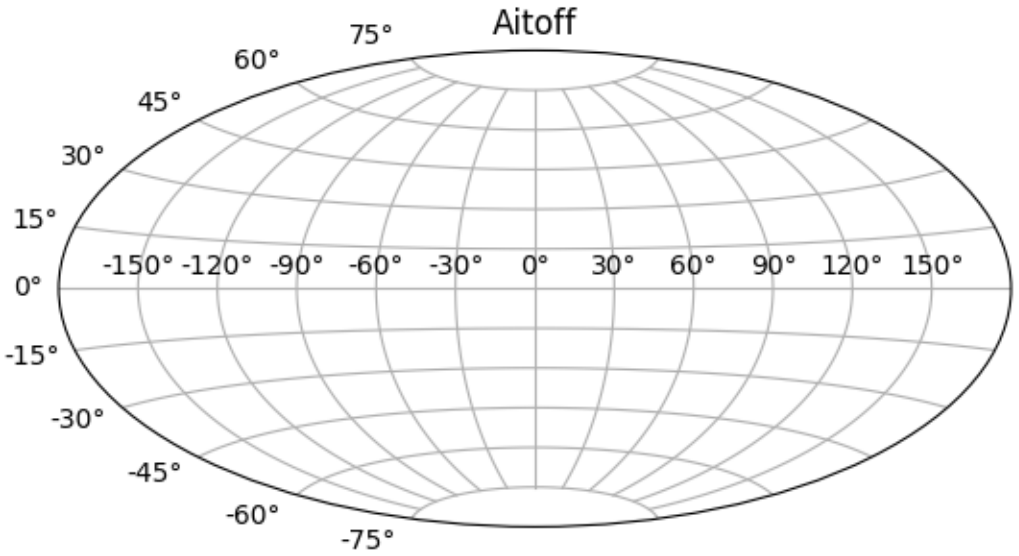
```
axs[2].set_yticks(np.arange(-0.9, 1.0, 0.4))
axs[2].set_ylim(-1, 1)

plt.show()
```

### Geographic Projections

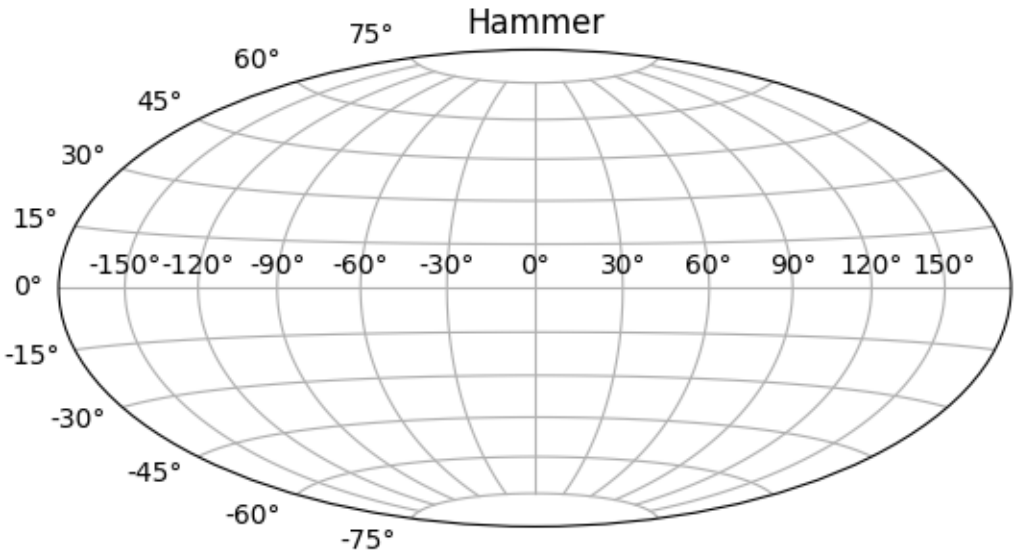
This shows 4 possible geographic projections. Cartopy supports more projections.

```
import matplotlib.pyplot as plt
```

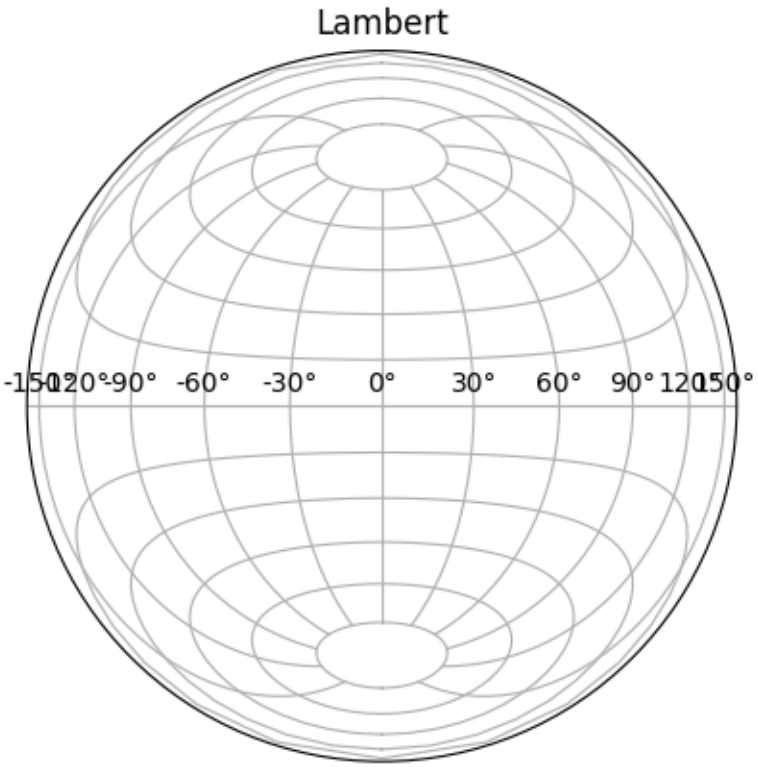
```
plt.figure()
plt.subplot(projection="aitoff")
plt.title("Aitoff")
plt.grid(True)
```



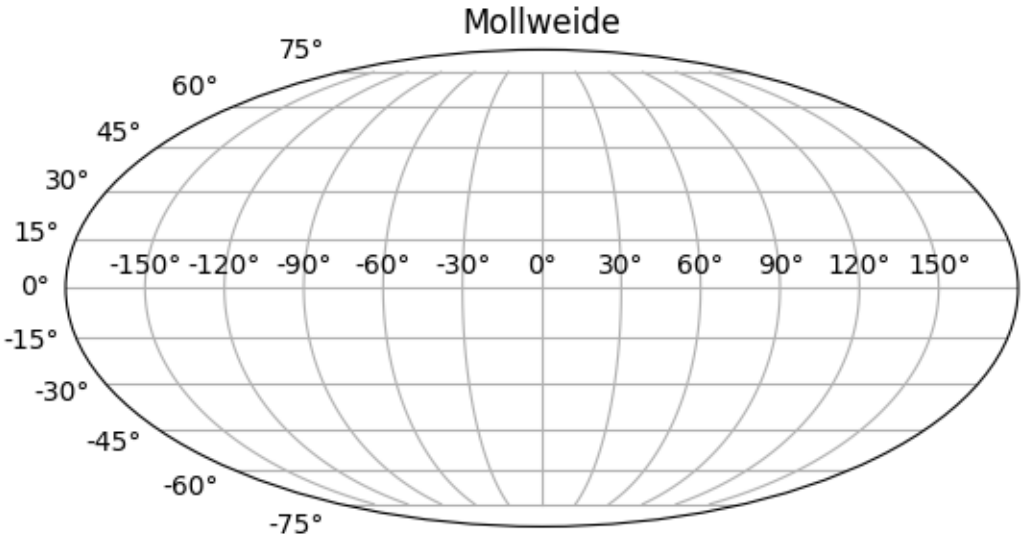

```
plt.figure()
plt.subplot(projection="hammer")
plt.title("Hammer")
plt.grid(True)
```




```
plt.figure()
plt.subplot(projection="lambert")
plt.title("Lambert")
plt.grid(True)
```




```
plt.figure()
plt.subplot(projection="mollweide")
plt.title("Mollweide")
plt.grid(True)

plt.show()
```



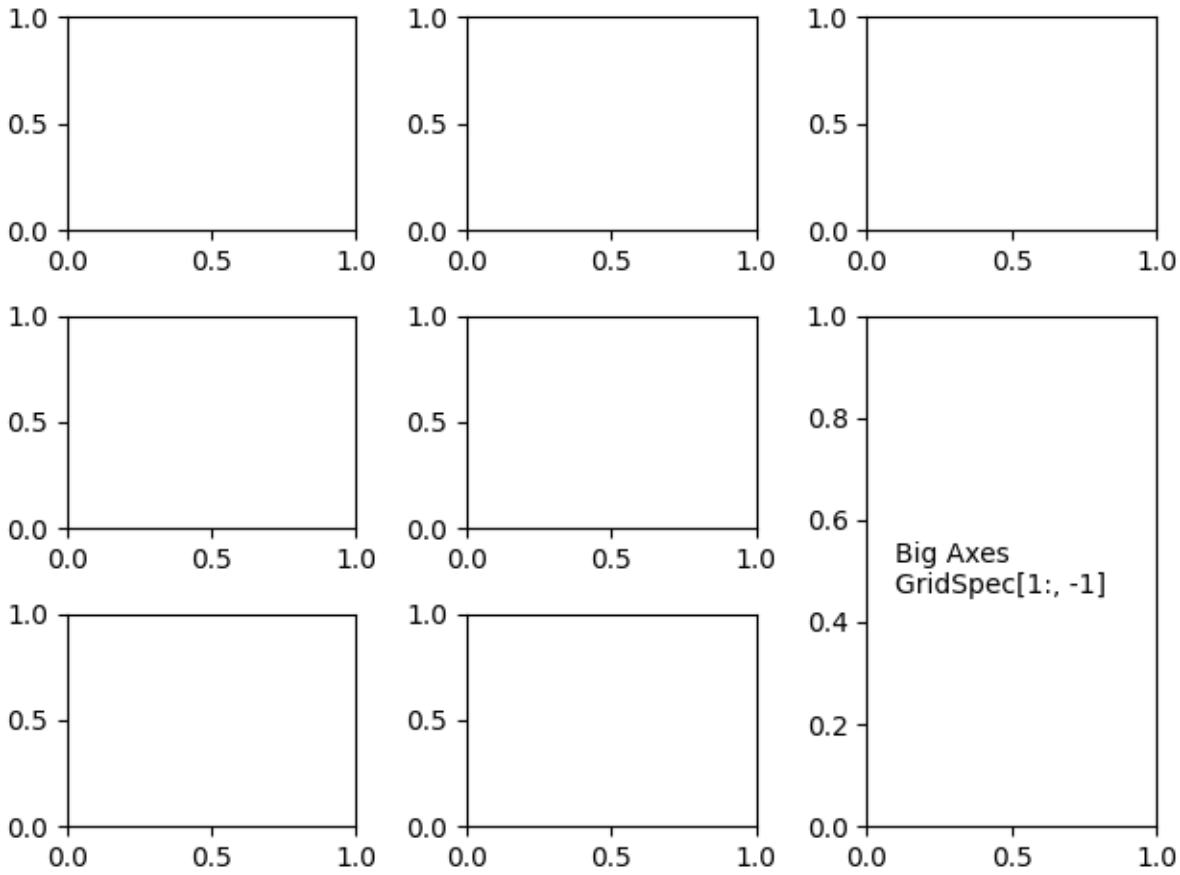

**Total running time of the script:** (0 minutes 1.108 seconds)

### Combining two subplots using subplots and GridSpec

Sometimes we want to combine two subplots in an Axes layout created with `subplots`. We can get the `GridSpec` from the Axes and then remove the covered Axes and fill the gap with a new bigger Axes. Here we create a layout with the bottom two Axes in the last column combined.

To start with this layout (rather than removing the overlapping Axes) use `subplot_mosaic`.

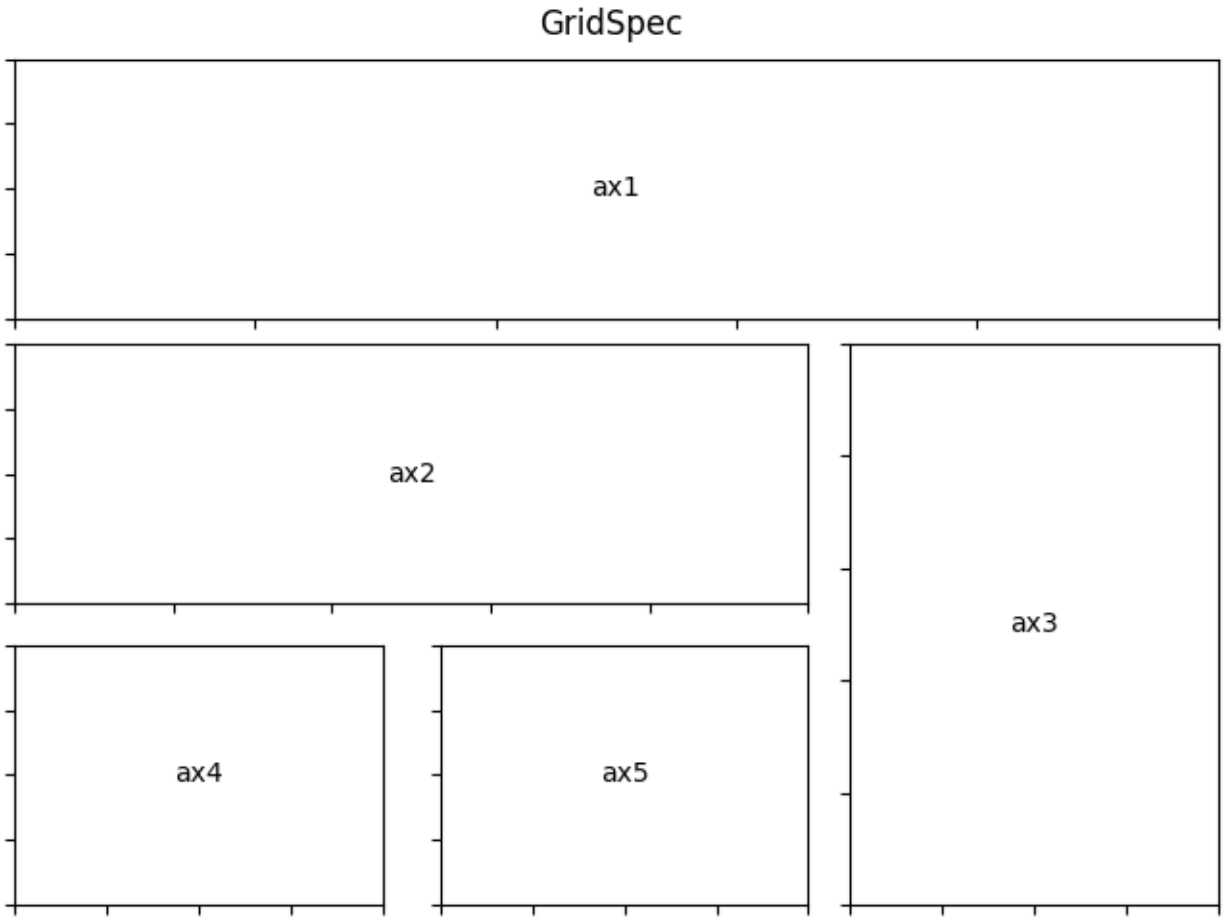
See also *Arranging multiple Axes in a Figure*.




```
import matplotlib.pyplot as plt

fig, axs = plt.subplots(ncols=3, nrows=3)
gs = axs[1, 2].get_gridspec()
# remove the underlying Axes
for ax in axs[1:, -1]:
    ax.remove()
axbig = fig.add_subplot(gs[1:, -1])
axbig.annotate('Big Axes \nGridSpec[1:, -1]', (0.1, 0.5),
               xycoords='axes fraction', va='center')

fig.tight_layout()

plt.show()
```

### Using Gridspec to make multi-column/row subplot layouts

*GridSpec* is a flexible way to layout subplot grids. Here is an example with a 3x3 grid, and axes spanning all three columns, two columns, and two rows.




```
import matplotlib.pyplot as plt

from matplotlib.gridspec import GridSpec


def format_axes(fig):
    for i, ax in enumerate(fig.axes):
        ax.text(0.5, 0.5, "ax%d" % (i+1), va="center", ha="center")
        ax.tick_params(labelbottom=False, labelleft=False)

fig = plt.figure(layout="constrained")

gs = GridSpec(3, 3, figure=fig)
ax1 = fig.add_subplot(gs[0, :])
# identical to ax1 = plt.subplot(gs.new_subplotspec((0, 0), colspan=3))
ax2 = fig.add_subplot(gs[1, :-1])
ax3 = fig.add_subplot(gs[1:, -1])
ax4 = fig.add_subplot(gs[-1, 0])
```

(continues on next page)

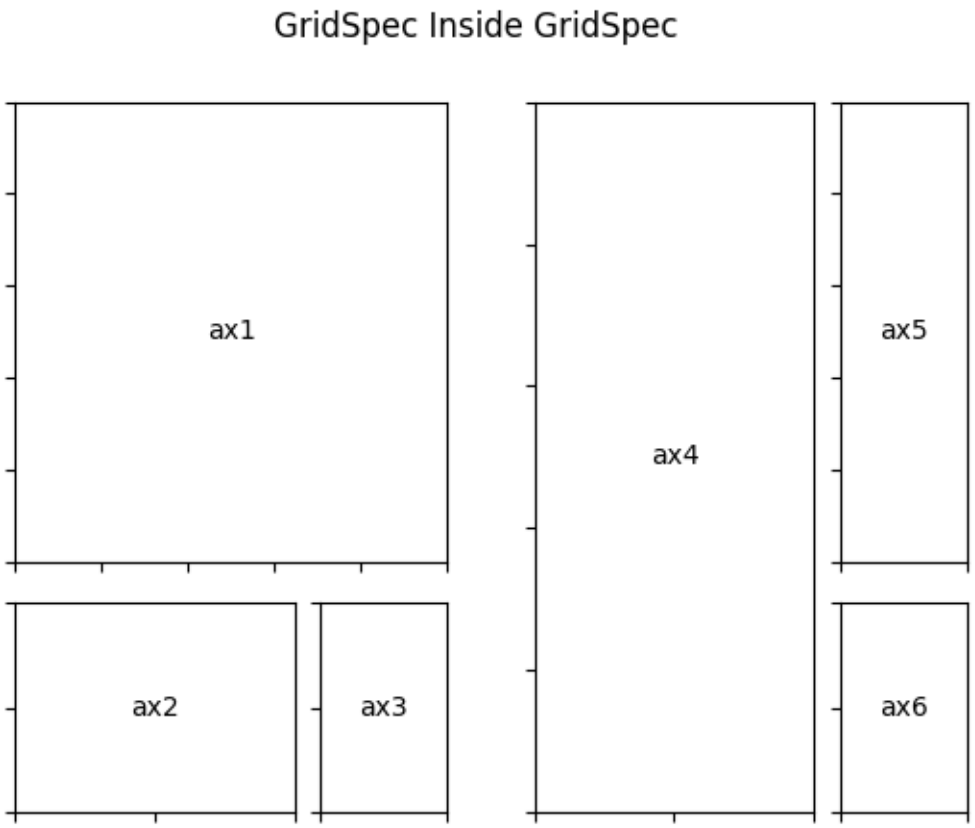
(continued from previous page)

```
ax5 = fig.add_subplot(gs[-1, -2])

fig.suptitle("GridSpec")
format_axes(fig)

plt.show()
```

### Nested Gridspecs

GridSpecs can be nested, so that a subplot from a parent GridSpec can set the position for a nested grid of subplots.

Note that the same functionality can be achieved more directly with `subfigures`; see *Figure subfigures*.




```
import matplotlib.pyplot as plt

import matplotlib.gridspec as gridspec


def format_axes(fig):
```

(continues on next page)

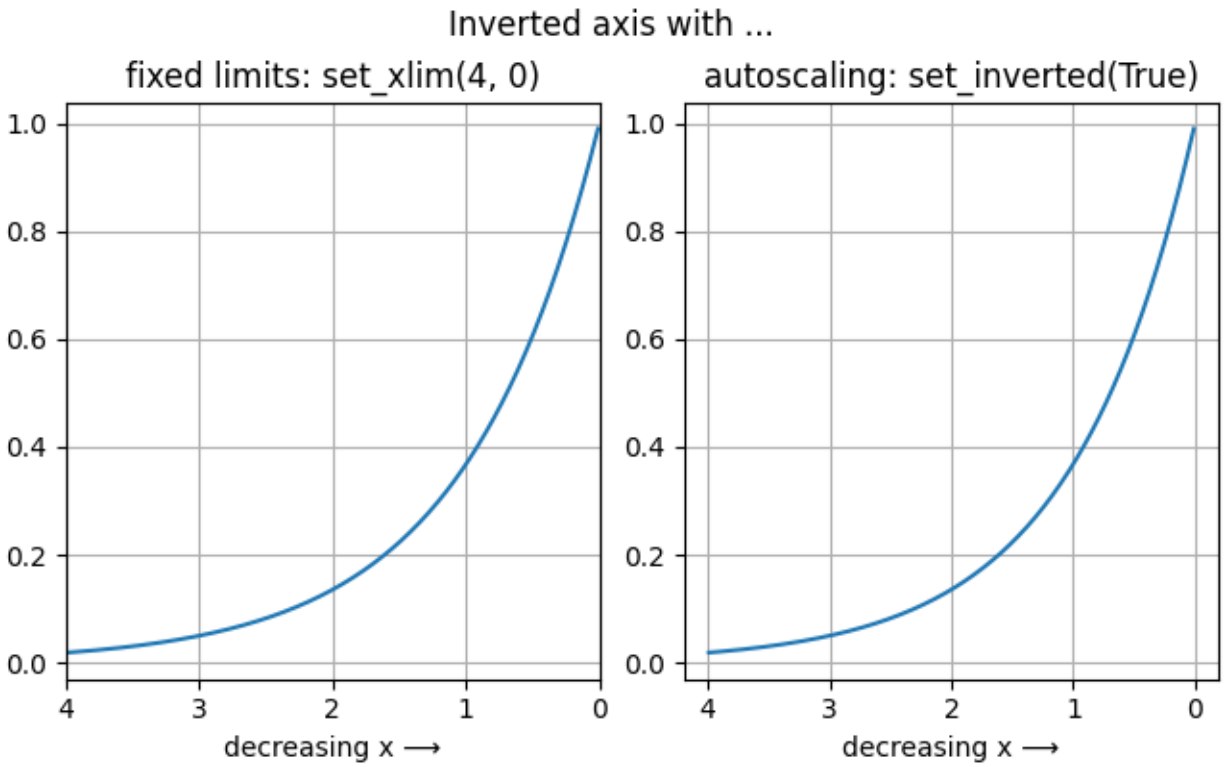
(continued from previous page)

```
    for i, ax in enumerate(fig.axes):
        ax.text(0.5, 0.5, "ax%d" % (i+1), va="center", ha="center")
        ax.tick_params(labelbottom=False, labelleft=False)


# gridspec inside gridspec
fig = plt.figure()

gs0 = gridspec.GridSpec(1, 2, figure=fig)

gs00 = gridspec.GridSpecFromSubplotSpec(3, 3, subplot_spec=gs0[0])

ax1 = fig.add_subplot(gs00[:-1, :])
ax2 = fig.add_subplot(gs00[-1, :-1])
ax3 = fig.add_subplot(gs00[-1, -1])

# the following syntax does the same as the GridSpecFromSubplotSpec call␣
 ↪above:
gs01 = gs0[1].subgridspec(3, 3)

ax4 = fig.add_subplot(gs01[:, :-1])
ax5 = fig.add_subplot(gs01[:-1, -1])
ax6 = fig.add_subplot(gs01[-1, -1])

plt.suptitle("GridSpec Inside GridSpec")
format_axes(fig)

plt.show()
```

### Inverted axis

This example demonstrates two ways to invert the direction of an axis:

- If you want to set *explicit axis limits* anyway, e.g. via *set_xlim*, you can swap the limit values: `set_xlim(4, 0)` instead of `set_xlim(0, 4)`.
- Use *Axis.set_inverted* if you only want to invert the axis *without modifying the limits*, i.e. keep existing limits or existing autoscaling behavior.




```
import matplotlib.pyplot as plt
import numpy as np

x = np.arange(0.01, 4.0, 0.01)
y = np.exp(-x)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(6.4,  4), layout="constrained")
fig.suptitle('Inverted axis with ...')

ax1.plot(x, y)
ax1.set_xlim(4, 0)    # inverted fixed limits
ax1.set_title('fixed limits: set_xlim(4, 0)')
ax1.set_xlabel('decreasing x →')
ax1.grid(True)

ax2.plot(x, y)
ax2.xaxis.set_inverted(True)   # inverted axis with autoscaling
ax2.set_title('autoscaling: set_inverted(True)')
ax2.set_xlabel('decreasing x →')
ax2.grid(True)

plt.show()
```

### Managing multiple figures in pyplot

*matplotlib.pyplot* uses the concept of a *current figure* and *current Axes*. Figures are identified via a figure number that is passed to *figure*. The figure with the given number is set as *current figure*. Additionally, if no figure with the number exists, a new one is created.

---

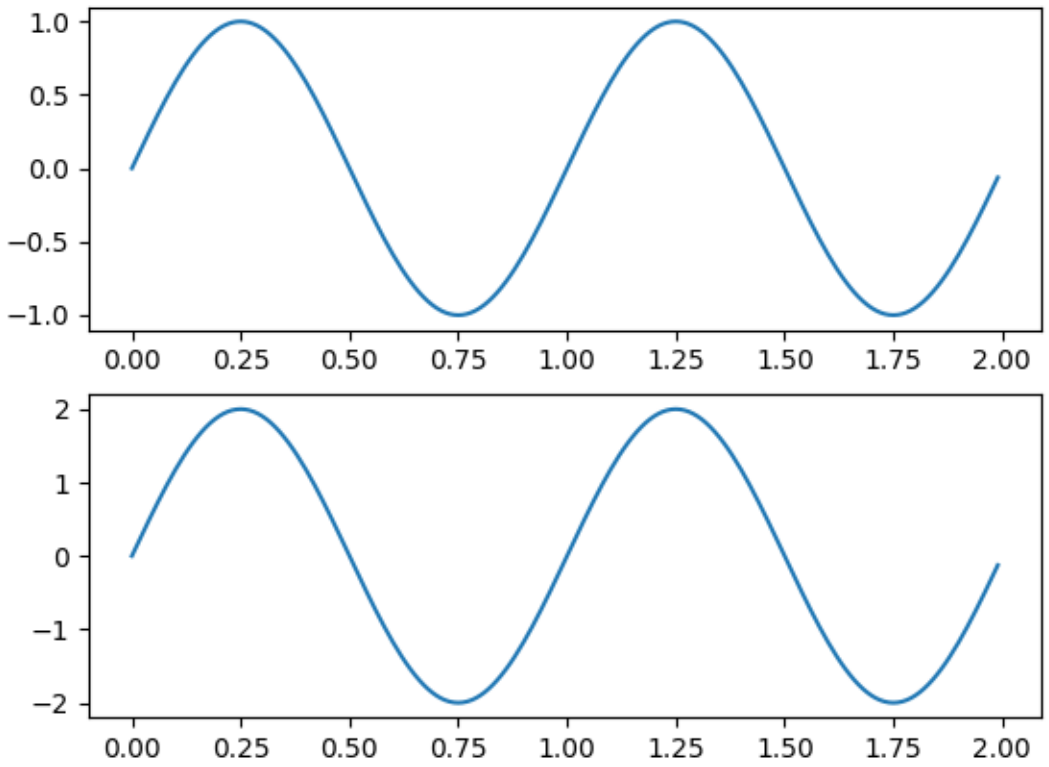
**Note:** We discourage working with multiple figures through the implicit pyplot interface because managing the *current figure* is cumbersome and error-prone. Instead, we recommend using the explicit approach and call methods on Figure and Axes instances. See *Matplotlib Application Interfaces (APIs)* for an explanation of the trade-offs between the implicit and explicit interfaces.

---

```
import matplotlib.pyplot as plt
import numpy as np

t = np.arange(0.0, 2.0, 0.01)
s1 = np.sin(2*np.pi*t)
s2 = np.sin(4*np.pi*t)
```

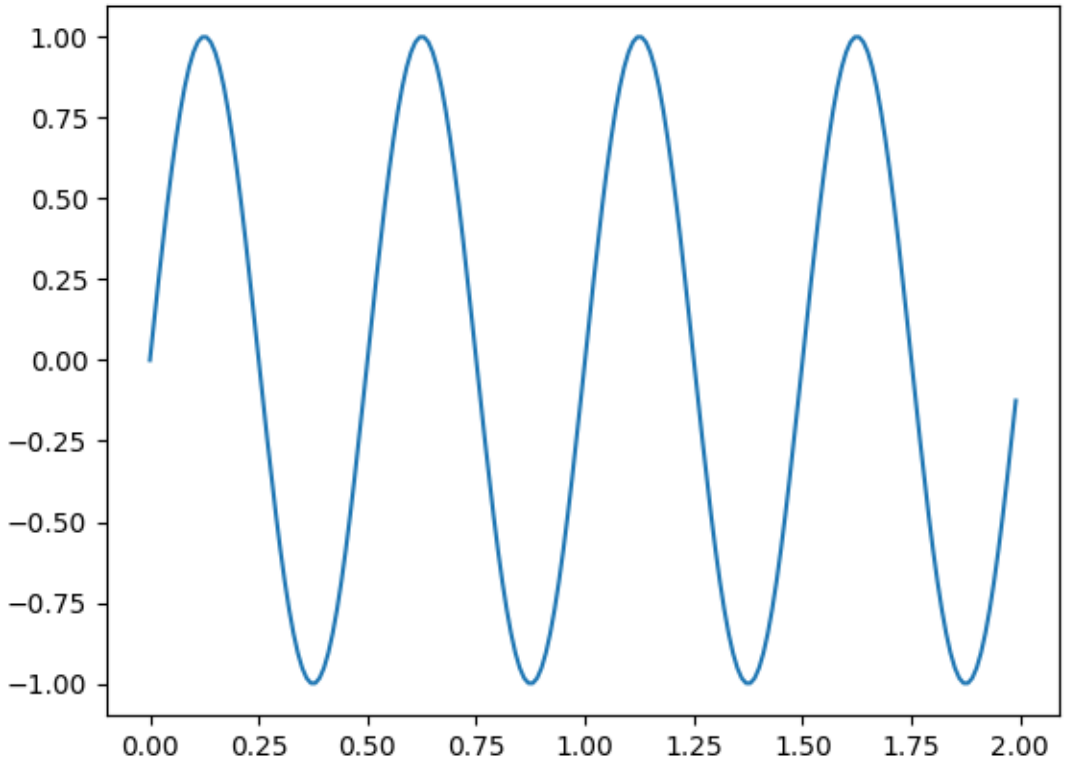
Create figure 1

```
plt.figure(1)
plt.subplot(211)
plt.plot(t, s1)
plt.subplot(212)
plt.plot(t, 2*s1)
```



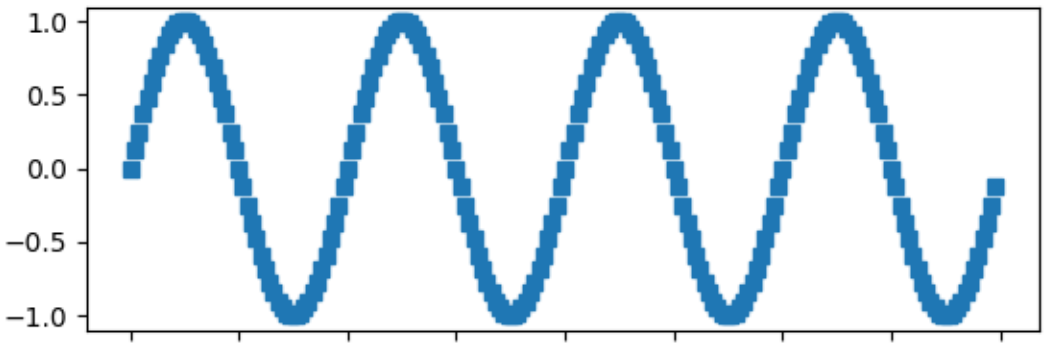

Create figure 2

```
plt.figure(2)
plt.plot(t, s2)
```




Now switch back to figure 1 and make some changes

```
plt.figure(1)
plt.subplot(211)
plt.plot(t, s2, 's')
ax = plt.gca()
ax.set_xticklabels([])

plt.show()
```




Tags: component: figure plot-type: line

### Secondary Axis

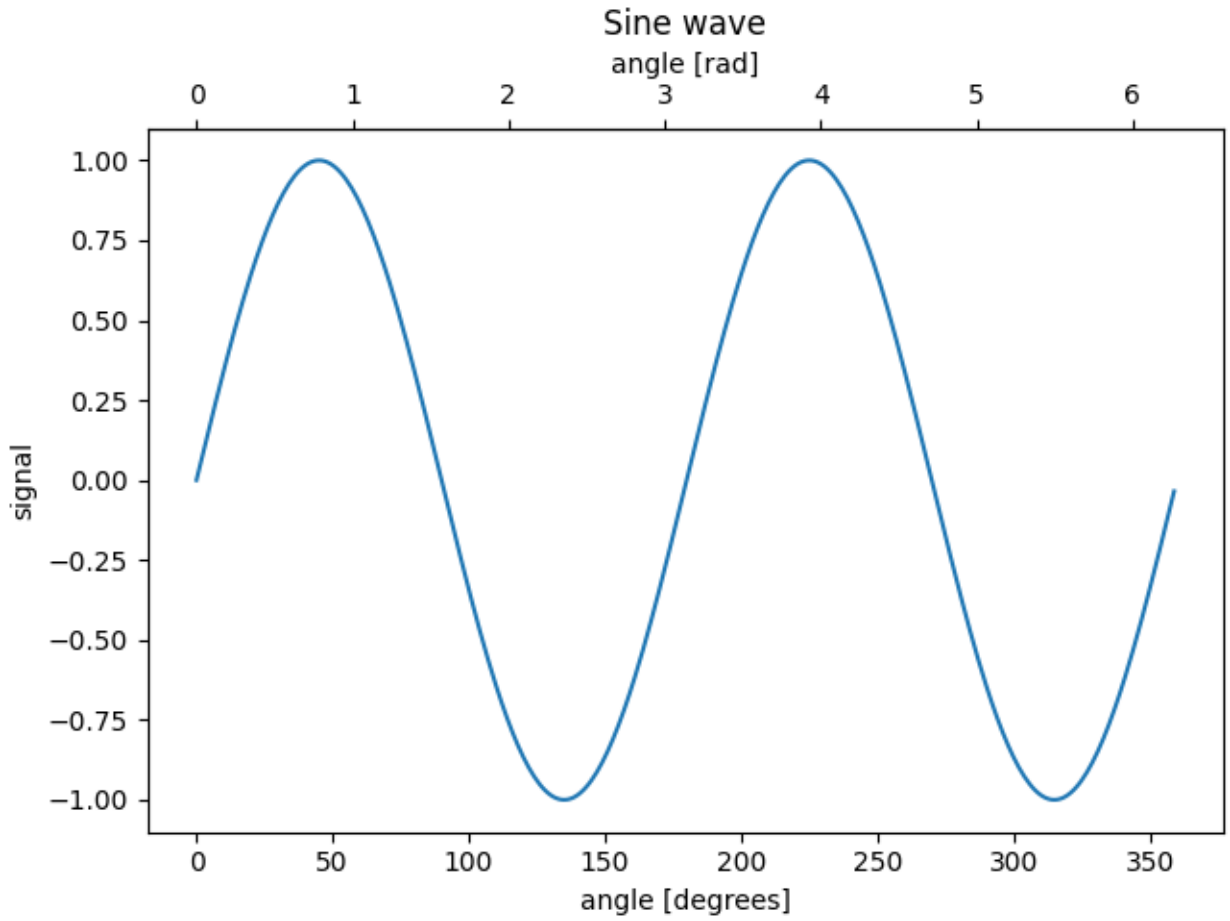
Sometimes we want a secondary axis on a plot, for instance to convert radians to degrees on the same plot. We can do this by making a child axes with only one axis visible via *axes.Axes.secondary_xaxis* and *axes.Axes.secondary_yaxis*. This secondary axis can have a different scale than the main axis by providing both a forward and an inverse conversion function in a tuple to the *functions* keyword argument:

```
import datetime

import matplotlib.pyplot as plt
import numpy as np

import matplotlib.dates as mdates
from matplotlib.ticker import AutoMinorLocator

fig, ax = plt.subplots(layout='constrained')
x = np.arange(0, 360, 1)
y = np.sin(2 * x * np.pi / 180)
ax.plot(x, y)
ax.set_xlabel('angle [degrees]')
```

(continues on next page)

(continued from previous page)

```
ax.set_ylabel('signal')
ax.set_title('Sine wave')


def deg2rad(x):
    return x * np.pi / 180


def rad2deg(x):
    return x * 180 / np.pi


secax = ax.secondary_xaxis('top', functions=(deg2rad, rad2deg))
secax.set_xlabel('angle [rad]')
plt.show()
```



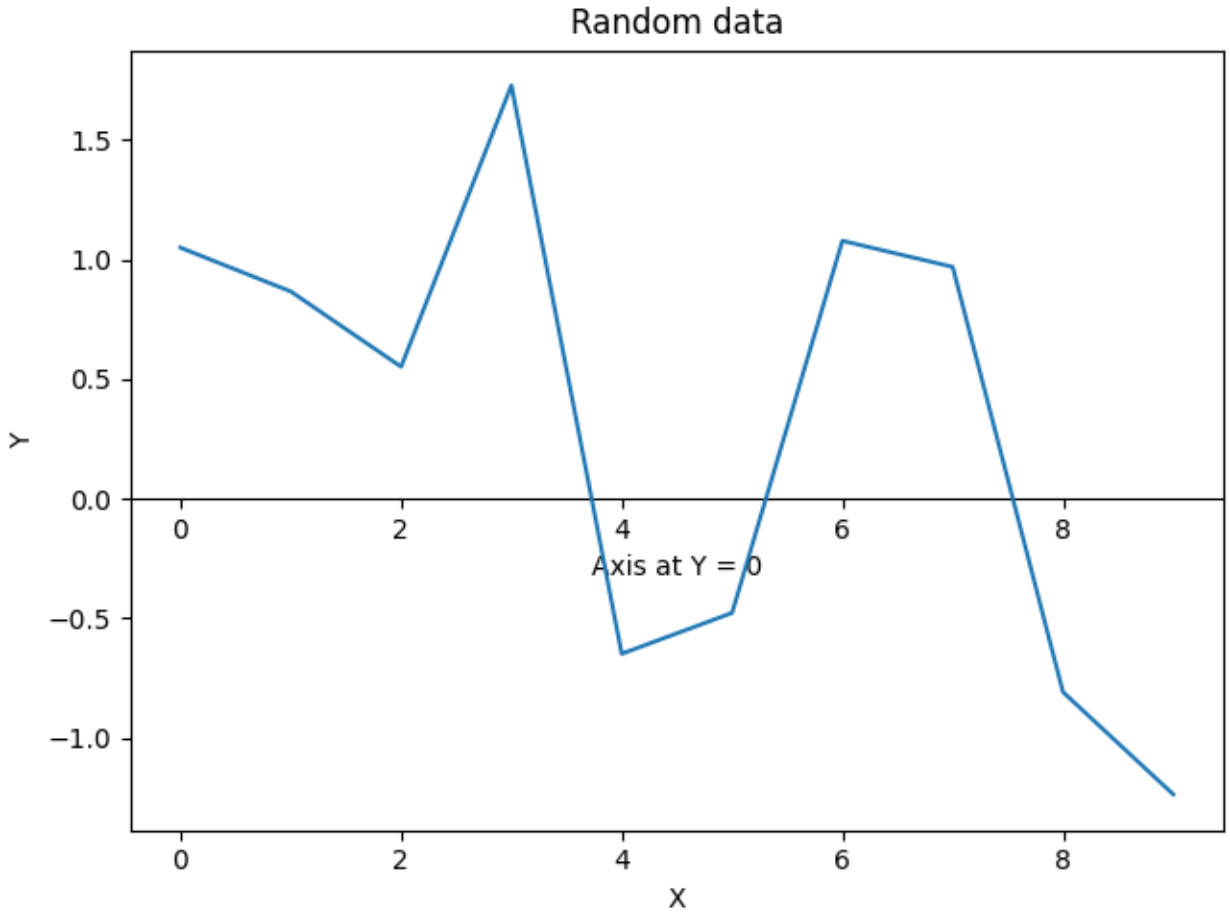

By default, the secondary axis is drawn in the Axes coordinate space. We can also provide a custom transform to place it in a different coordinate space. Here we put the axis at Y = 0 in data coordinates.

```
fig, ax = plt.subplots(layout='constrained')
x = np.arange(0, 10)
np.random.seed(19680801)
```

(continues on next page)

(continued from previous page)

```
y = np.random.randn(len(x))
ax.plot(x, y)
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_title('Random data')

# Pass ax.transData as a transform to place the axis relative to our data
secax = ax.secondary_xaxis(0, transform=ax.transData)
secax.set_xlabel('Axis at Y = 0')
plt.show()
```




Here is the case of converting from wavenumber to wavelength in a log-log scale.

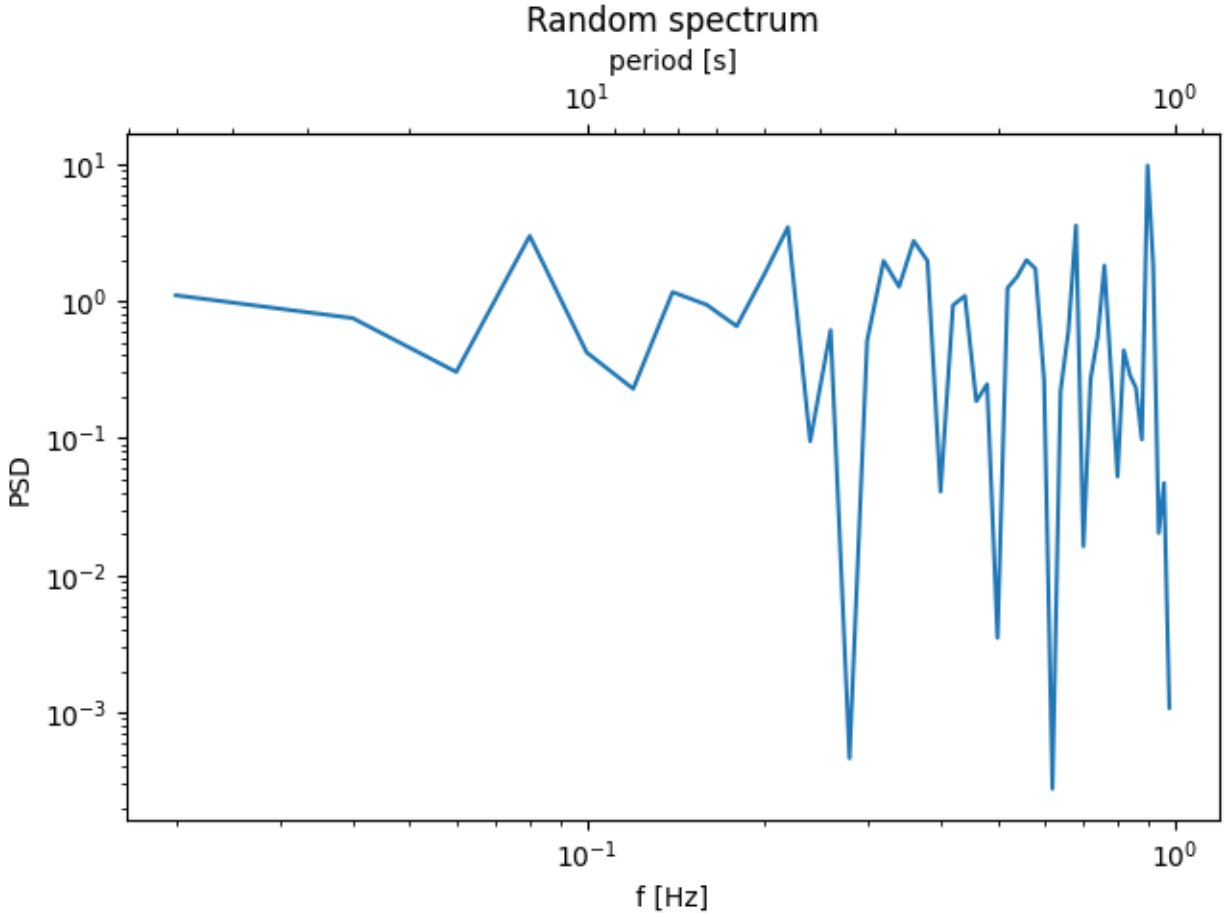
**Note:** In this case, the xscale of the parent is logarithmic, so the child is made logarithmic as well.

```
fig, ax = plt.subplots(layout='constrained')
x = np.arange(0.02, 1, 0.02)
np.random.seed(19680801)
y = np.random.randn(len(x)) ** 2
ax.loglog(x, y)
```

(continues on next page)

(continued from previous page)

```
ax.set_xlabel('f [Hz]')
ax.set_ylabel('PSD')
ax.set_title('Random spectrum')


def one_over(x):
    """Vectorized 1/x, treating x==0 manually"""
    x = np.array(x, float)
    near_zero = np.isclose(x, 0)
    x[near_zero] = np.inf
    x[~near_zero] = 1 / x[~near_zero]
    return x


# the function "1/x" is its own inverse
inverse = one_over


secax = ax.secondary_xaxis('top', functions=(one_over, inverse))
secax.set_xlabel('period [s]')
plt.show()
```




Sometime we want to relate the axes in a transform that is ad-hoc from the data, and is derived empirically.

In that case we can set the forward and inverse transforms functions to be linear interpolations from the one data set to the other.

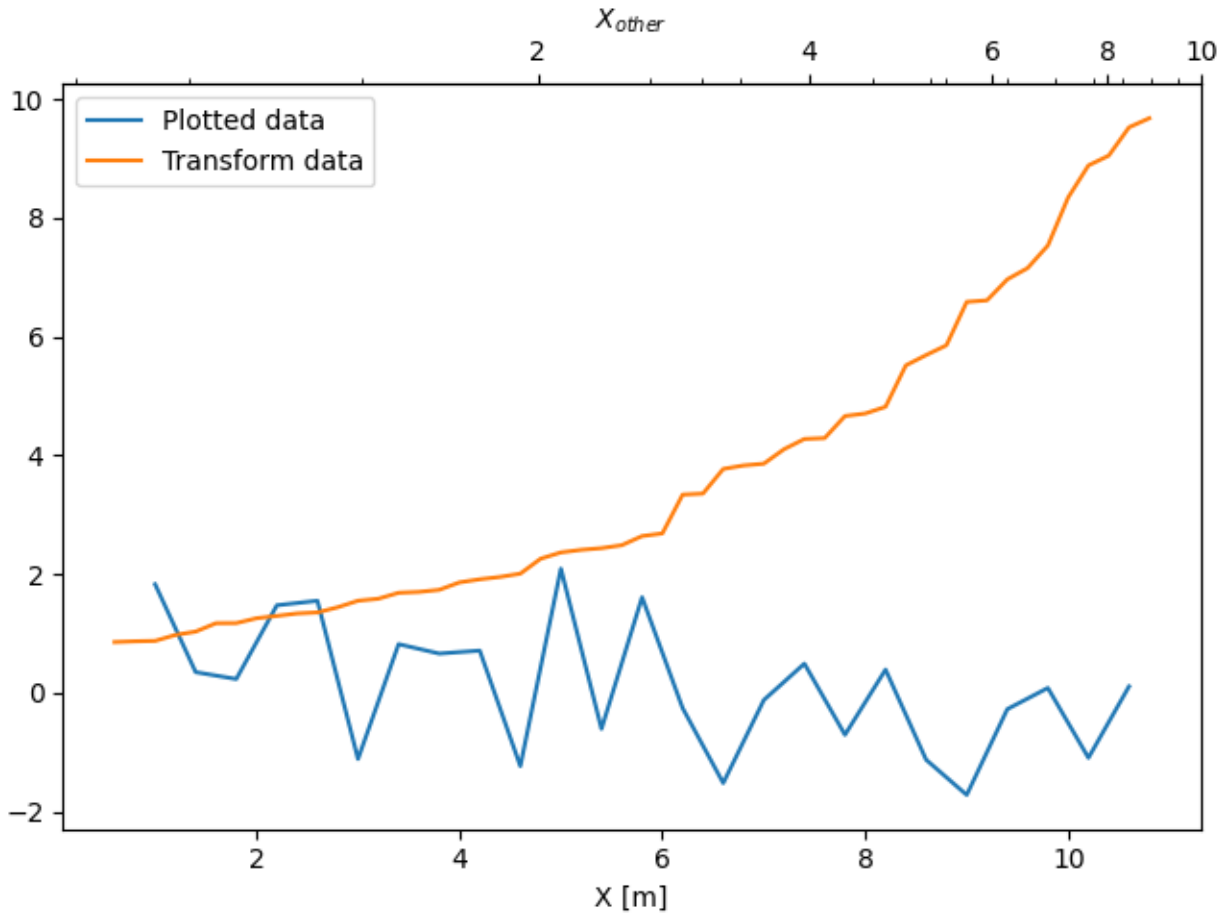
---

**Note:** In order to properly handle the data margins, the mapping functions (`forward` and `inverse` in this example) need to be defined beyond the nominal plot limits.

In the specific case of the numpy linear interpolation, `numpy.interp`, this condition can be arbitrarily enforced by providing optional keyword arguments *left*, *right* such that values outside the data range are mapped well outside the plot limits.

---

```
fig, ax = plt.subplots(layout='constrained')
xdata = np.arange(1, 11, 0.4)
ydata = np.random.randn(len(xdata))
ax.plot(xdata, ydata, label='Plotted data')

xold = np.arange(0, 11, 0.2)
# fake data set relating x coordinate to another data-derived coordinate.
# xnew must be monotonic, so we sort...
xnew = np.sort(10 * np.exp(-xold / 4) + np.random.randn(len(xold)) / 3)

ax.plot(xold[3:], xnew[3:], label='Transform data')
ax.set_xlabel('X [m]')
ax.legend()


def forward(x):
    return np.interp(x, xold, xnew)


def inverse(x):
    return np.interp(x, xnew, xold)


secax = ax.secondary_xaxis('top', functions=(forward, inverse))
secax.xaxis.set_minor_locator(AutoMinorLocator())
secax.set_xlabel('$X_{other}$')

plt.show()
```



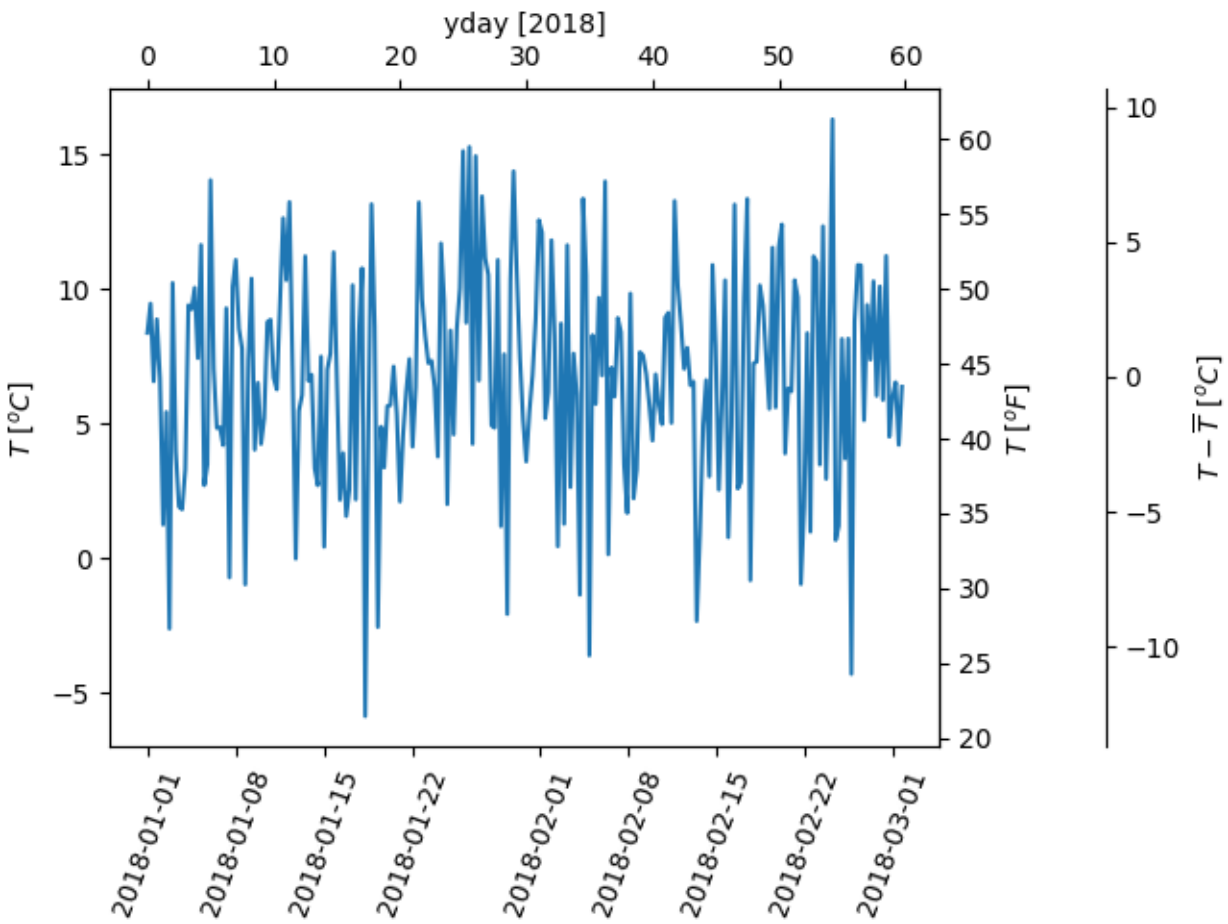

A final example translates np.datetime64 to yearday on the x axis and from Celsius to Fahrenheit on the y axis. Note the addition of a third y axis, and that it can be placed using a float for the location argument

```
dates = [datetime.datetime(2018, 1, 1) + datetime.timedelta(hours=k * 6)
         for k in range(240)]
temperature = np.random.randn(len(dates)) * 4 + 6.7
fig, ax = plt.subplots(layout='constrained')

ax.plot(dates, temperature)
ax.set_ylabel(r'$T\ [^oC]$')
ax.xaxis.set_tick_params(rotation=70)


def date2yday(x):
    """Convert matplotlib datenum to days since 2018-01-01."""
    y = x - mdates.date2num(datetime.datetime(2018, 1, 1))
    return y


def yday2date(x):
    """Return a matplotlib datenum for *x* days after 2018-01-01."""
    y = x + mdates.date2num(datetime.datetime(2018, 1, 1))
    return y
```

(continues on next page)

(continued from previous page)

```
secax_x = ax.secondary_xaxis('top', functions=(date2yday, yday2date))
secax_x.set_xlabel('yday [2018]')


def celsius_to_fahrenheit(x):
    return x * 1.8 + 32


def fahrenheit_to_celsius(x):
    return (x - 32) / 1.8


secax_y = ax.secondary_yaxis(
    'right', functions=(celsius_to_fahrenheit, fahrenheit_to_celsius))
secax_y.set_ylabel(r'$T\ [^oF]$')


def celsius_to_anomaly(x):
    return (x - np.mean(temperature))


def anomaly_to_celsius(x):
    return (x + np.mean(temperature))


# use of a float for the position:
secax_y2 = ax.secondary_yaxis(
    1.2, functions=(celsius_to_anomaly, anomaly_to_celsius))
secax_y2.set_ylabel(r'$T - \overline{T}\ [^oC]$')


plt.show()
```




**References**

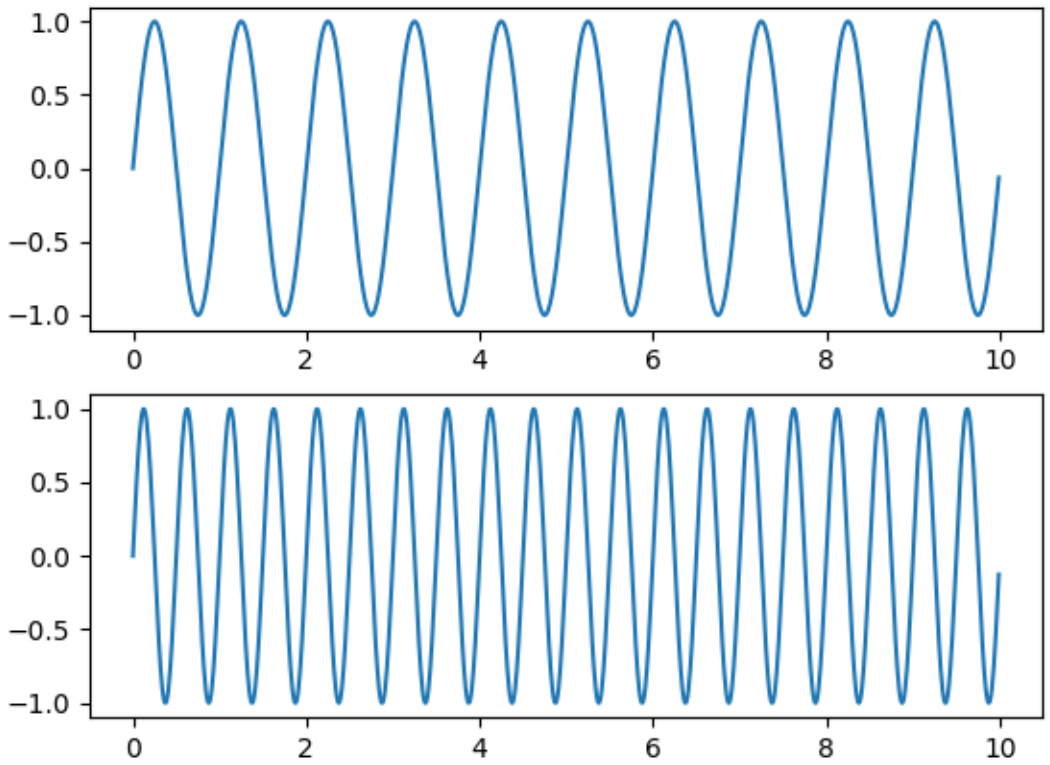
The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.secondary_xaxis*
- *matplotlib.axes.Axes.secondary_yaxis*

**Total running time of the script:** (0 minutes 4.129 seconds)

## Sharing axis limits and views

It's common to make two or more plots which share an axis, e.g., two subplots with time as a common axis. When you pan and zoom around on one, you want the other to move around with you. To facilitate this, matplotlib Axes support a `sharex` and `sharey` attribute. When you create a *subplot* or *axes*, you can pass in a keyword indicating what Axes you want to share with.




```
import matplotlib.pyplot as plt
import numpy as np

t = np.arange(0, 10, 0.01)

ax1 = plt.subplot(211)
ax1.plot(t, np.sin(2*np.pi*t))

ax2 = plt.subplot(212, sharex=ax1)
ax2.plot(t, np.sin(4*np.pi*t))

plt.show()
```

### Shared axis

You can share the x- or y-axis limits for one axis with another by passing an `Axes` instance as a *sharex* or *sharey* keyword argument.

Changing the axis limits on one Axes will be reflected automatically in the other, and vice-versa, so when you navigate with the toolbar the Axes will follow each other on their shared axis. Ditto for changes in the axis scaling (e.g., log vs. linear). However, it is possible to have differences in tick labeling, e.g., you can selectively turn off the tick labels on one Axes.

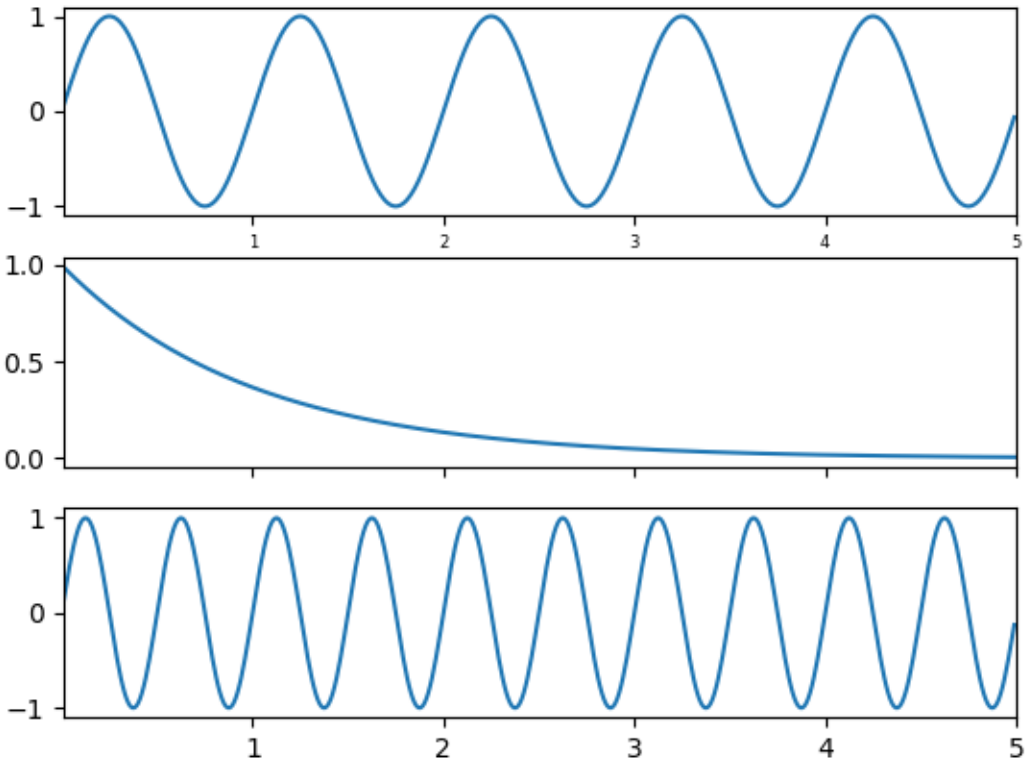
The example below shows how to customize the tick labels on the various axes. Shared axes share the tick locator, tick formatter, view limits, and transformation (e.g., log, linear). But the ticklabels themselves do not share properties. This is a feature and not a bug, because you may want to make the tick labels smaller on the upper axes, e.g., in the example below.

If you want to turn off the ticklabels for a given Axes (e.g., on subplot(211) or subplot(212)), you cannot do the standard trick:

```
setp(ax2, xticklabels=[])
```

because this changes the tick Formatter, which is shared among all Axes. But you can alter the visibility of the labels, which is a property:

```
setp(ax2.get_xticklabels(), visible=False)
```



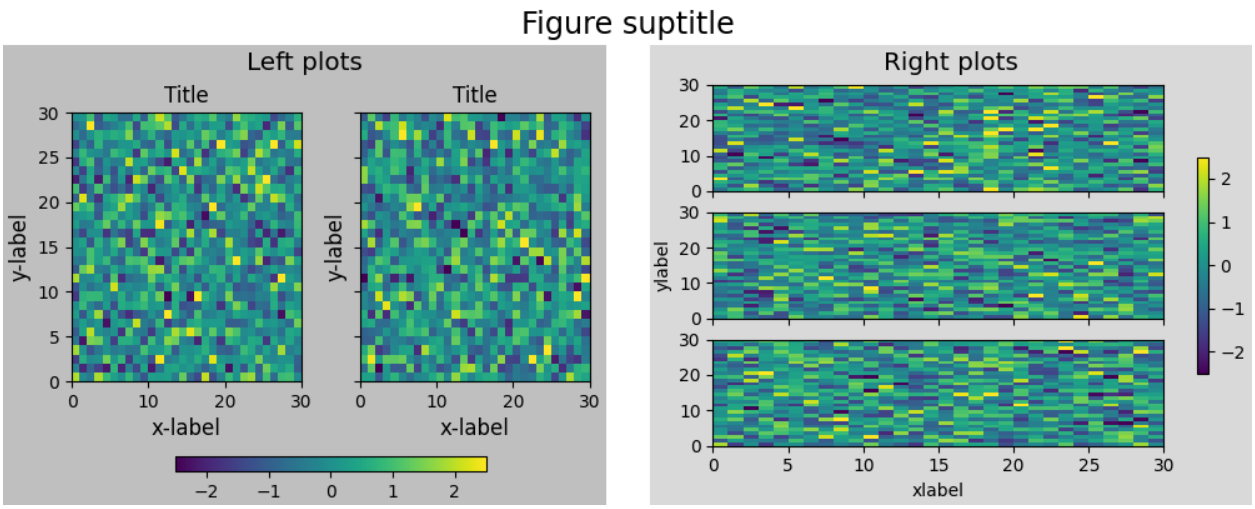

```
import matplotlib.pyplot as plt
import numpy as np

t = np.arange(0.01, 5.0, 0.01)
s1 = np.sin(2 * np.pi * t)
s2 = np.exp(-t)
s3 = np.sin(4 * np.pi * t)

ax1 = plt.subplot(311)
plt.plot(t, s1)
plt.tick_params('x', labelsize=6)

# share x only
ax2 = plt.subplot(312, sharex=ax1)
plt.plot(t, s2)
# make these tick labels invisible
plt.tick_params('x', labelbottom=False)

# share x and y
ax3 = plt.subplot(313, sharex=ax1, sharey=ax1)
plt.plot(t, s3)
plt.xlim(0.01, 5.0)
plt.show()
```

### Figure subfigures

Sometimes it is desirable to have a figure with two different layouts in it. This can be achieved with *nested gridspecs*, but having a virtual figure with its own artists is helpful, so Matplotlib also has "subfigures", accessed by calling *matplotlib.figure.Figure.add_subfigure* in a way that is analogous to *matplotlib.figure.Figure.add_subplot*, or *matplotlib.figure.Figure.subfigures* to make an array of subfigures. Note that subfigures can also have their own child subfigures.

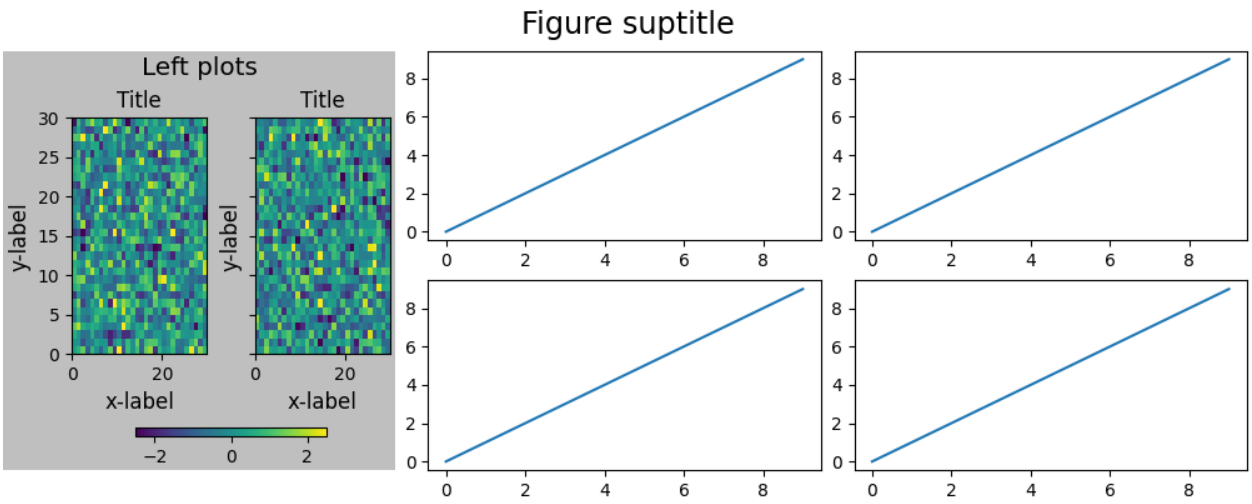
**Note:** The *subfigure* concept is new in v3.4, and the API is still provisional.

```
import matplotlib.pyplot as plt
import numpy as np


def example_plot(ax, fontsize=12, hide_labels=False):
    pc = ax.pcolormesh(np.random.randn(30, 30), vmin=-2.5, vmax=2.5)
    if not hide_labels:
        ax.set_xlabel('x-label', fontsize=fontsize)
        ax.set_ylabel('y-label', fontsize=fontsize)
        ax.set_title('Title', fontsize=fontsize)
    return pc

np.random.seed(19680808)
# gridspec inside gridspec
fig = plt.figure(layout='constrained', figsize=(10, 4))
subfigs = fig.subfigures(1, 2, wspace=0.07)

axsLeft = subfigs[0].subplots(1, 2, sharey=True)
subfigs[0].set_facecolor('0.75')
for ax in axsLeft:
    pc = example_plot(ax)
subfigs[0].suptitle('Left plots', fontsize='x-large')
subfigs[0].colorbar(pc, shrink=0.6, ax=axsLeft, location='bottom')

axsRight = subfigs[1].subplots(3, 1, sharex=True)
for nn, ax in enumerate(axsRight):
    pc = example_plot(ax, hide_labels=True)
    if nn == 2:
        ax.set_xlabel('xlabel')
    if nn == 1:
        ax.set_ylabel('ylabel')

subfigs[1].set_facecolor('0.85')
subfigs[1].colorbar(pc, shrink=0.6, ax=axsRight)
subfigs[1].suptitle('Right plots', fontsize='x-large')

fig.suptitle('Figure suptitle', fontsize='xx-large')

plt.show()
```



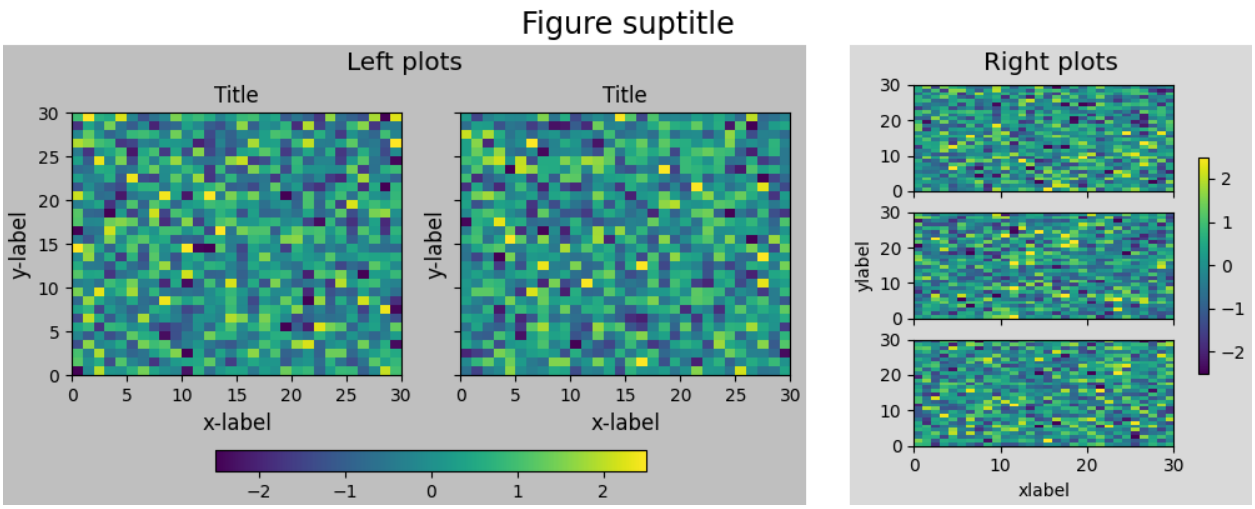

It is possible to mix subplots and subfigures using *matplotlib.figure.Figure.add_subfigure*. This requires getting the gridspec that the subplots are laid out on.

```
fig, axs = plt.subplots(2, 3, layout='constrained', figsize=(10, 4))
gridspec = axs[0, 0].get_subplotspec().get_gridspec()

# clear the left column for the subfigure:
for a in axs[:, 0]:
    a.remove()

# plot data in remaining Axes:
for a in axs[:, 1:].flat:
    a.plot(np.arange(10))

# make the subfigure in the empty gridspec slots:
subfig = fig.add_subfigure(gridspec[:, 0])

axsLeft = subfig.subplots(1, 2, sharey=True)
subfig.set_facecolor('0.75')
for ax in axsLeft:
    pc = example_plot(ax)
subfig.suptitle('Left plots', fontsize='x-large')
subfig.colorbar(pc, shrink=0.6, ax=axsLeft, location='bottom')

fig.suptitle('Figure suptitle', fontsize='xx-large')
plt.show()
```



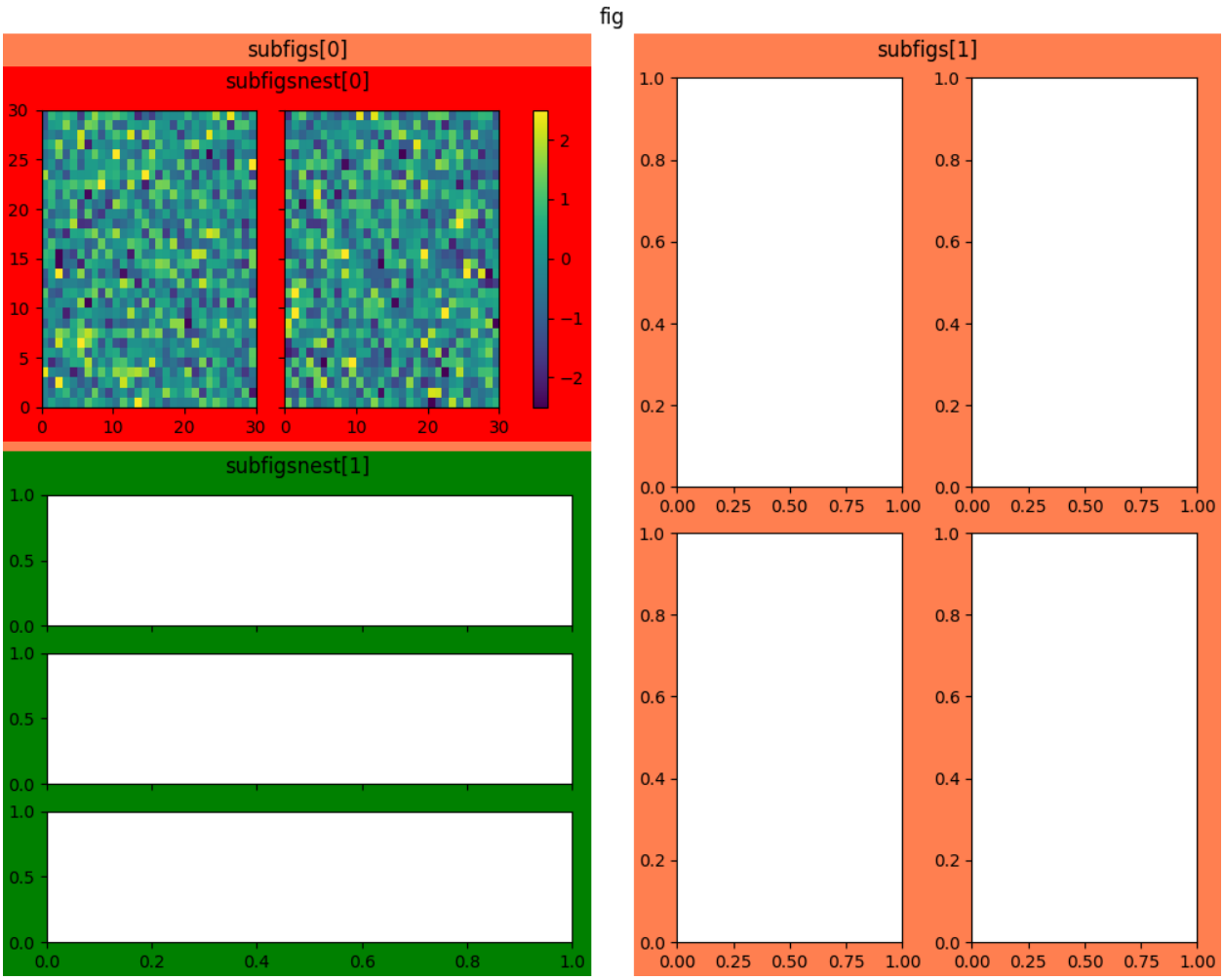

Subfigures can have different widths and heights. This is exactly the same example as the first example, but *width_ratios* has been changed:

```
fig = plt.figure(layout='constrained', figsize=(10, 4))
subfigs = fig.subfigures(1, 2, wspace=0.07, width_ratios=[2, 1])

axsLeft = subfigs[0].subplots(1, 2, sharey=True)
subfigs[0].set_facecolor('0.75')
for ax in axsLeft:
    pc = example_plot(ax)
subfigs[0].suptitle('Left plots', fontsize='x-large')
subfigs[0].colorbar(pc, shrink=0.6, ax=axsLeft, location='bottom')

axsRight = subfigs[1].subplots(3, 1, sharex=True)
for nn, ax in enumerate(axsRight):
    pc = example_plot(ax, hide_labels=True)
    if nn == 2:
        ax.set_xlabel('xlabel')
    if nn == 1:
        ax.set_ylabel('ylabel')

subfigs[1].set_facecolor('0.85')
subfigs[1].colorbar(pc, shrink=0.6, ax=axsRight)
subfigs[1].suptitle('Right plots', fontsize='x-large')

fig.suptitle('Figure suptitle', fontsize='xx-large')

plt.show()
```




Subfigures can be also be nested:

```
fig = plt.figure(layout='constrained', figsize=(10, 8))

fig.suptitle('fig')

subfigs = fig.subfigures(1, 2, wspace=0.07)

subfigs[0].set_facecolor('coral')
subfigs[0].suptitle('subfigs[0]')

subfigs[1].set_facecolor('coral')
subfigs[1].suptitle('subfigs[1]')

subfigsnest = subfigs[0].subfigures(2, 1, height_ratios=[1, 1.4])
subfigsnest[0].suptitle('subfigsnest[0]')
subfigsnest[0].set_facecolor('r')
axsnest0 = subfigsnest[0].subplots(1, 2, sharey=True)
for nn, ax in enumerate(axsnest0):
    pc = example_plot(ax, hide_labels=True)
subfigsnest[0].colorbar(pc, ax=axsnest0)

subfigsnest[1].suptitle('subfigsnest[1]')
subfigsnest[1].set_facecolor('g')
axsnest1 = subfigsnest[1].subplots(3, 1, sharex=True)

axsRight = subfigs[1].subplots(2, 2)

plt.show()
```




**Total running time of the script:** (0 minutes 4.052 seconds)

## Multiple subplots

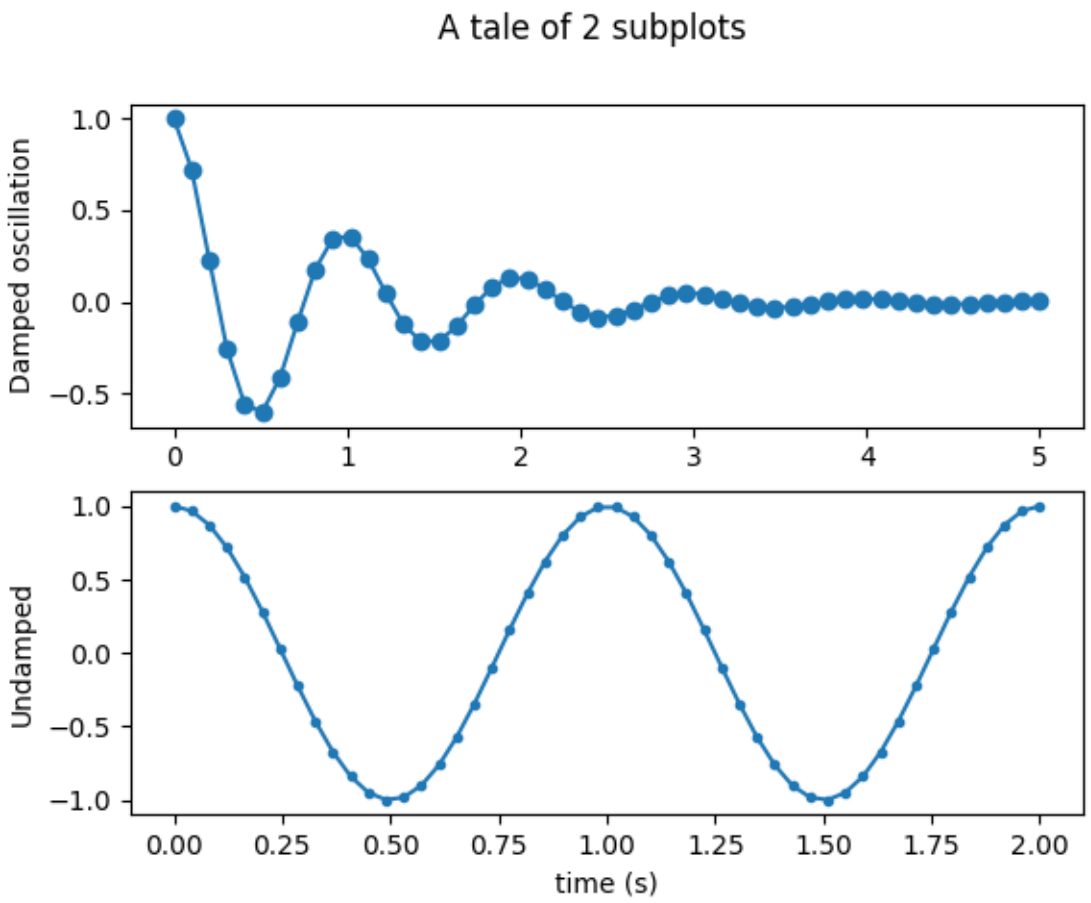
Simple demo with multiple subplots.

For more options, see *Creating multiple subplots using plt.subplots*.

```
import matplotlib.pyplot as plt
import numpy as np

# Create some fake data.
x1 = np.linspace(0.0, 5.0)
y1 = np.cos(2 * np.pi * x1) * np.exp(-x1)
x2 = np.linspace(0.0, 2.0)
y2 = np.cos(2 * np.pi * x2)
```

`subplots()` is the recommended method to generate simple subplot arrangements:

```
fig, (ax1, ax2) = plt.subplots(2, 1)
fig.suptitle('A tale of 2 subplots')

ax1.plot(x1, y1, 'o-')
ax1.set_ylabel('Damped oscillation')

ax2.plot(x2, y2, '.-')
ax2.set_xlabel('time (s)')
ax2.set_ylabel('Undamped')

plt.show()
```



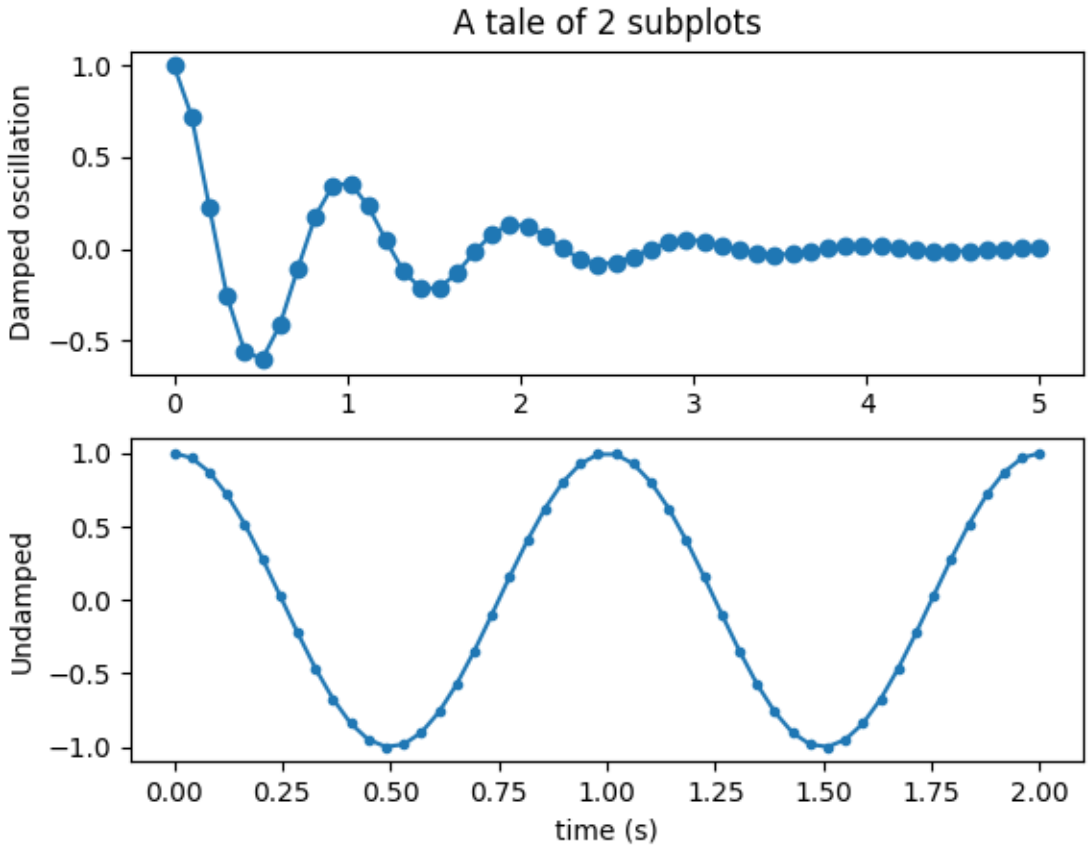

Subplots can also be generated one at a time using *subplot()*:

```
plt.subplot(2, 1, 1)
plt.plot(x1, y1, 'o-')
plt.title('A tale of 2 subplots')
plt.ylabel('Damped oscillation')

plt.subplot(2, 1, 2)
plt.plot(x2, y2, '.-')
plt.xlabel('time (s)')
plt.ylabel('Undamped')
```

(continues on next page)

(continued from previous page)

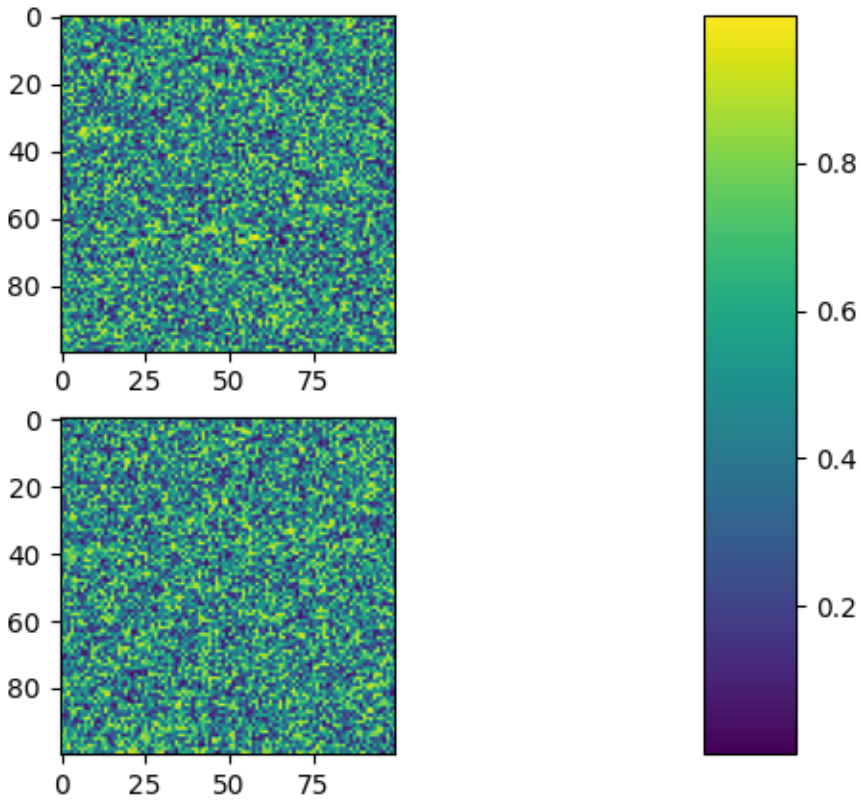
```
plt.show()
```




## Subplots spacings and margins

Adjusting the spacing of margins and subplots using *pyplot.subplots_adjust*.

**Note:** There is also a tool window to adjust the margins and spacings of displayed figures interactively. It can be opened via the toolbar or by calling *pyplot.subplot_tool*.




```
import matplotlib.pyplot as plt
import numpy as np

# Fixing random state for reproducibility
np.random.seed(19680801)

plt.subplot(211)
plt.imshow(np.random.random((100, 100)))
plt.subplot(212)
plt.imshow(np.random.random((100, 100)))

plt.subplots_adjust(bottom=0.1, right=0.8, top=0.9)
cax = plt.axes((0.85, 0.1, 0.075, 0.8))
plt.colorbar(cax=cax)

plt.show()
```

### Creating multiple subplots using `plt.subplots`

*pyplot.subplots* creates a figure and a grid of subplots with a single call, while providing reasonable control over how the individual plots are created. For more advanced use cases you can use *GridSpec* for a more general subplot layout or *Figure.add_subplot* for adding subplots at arbitrary locations within the figure.

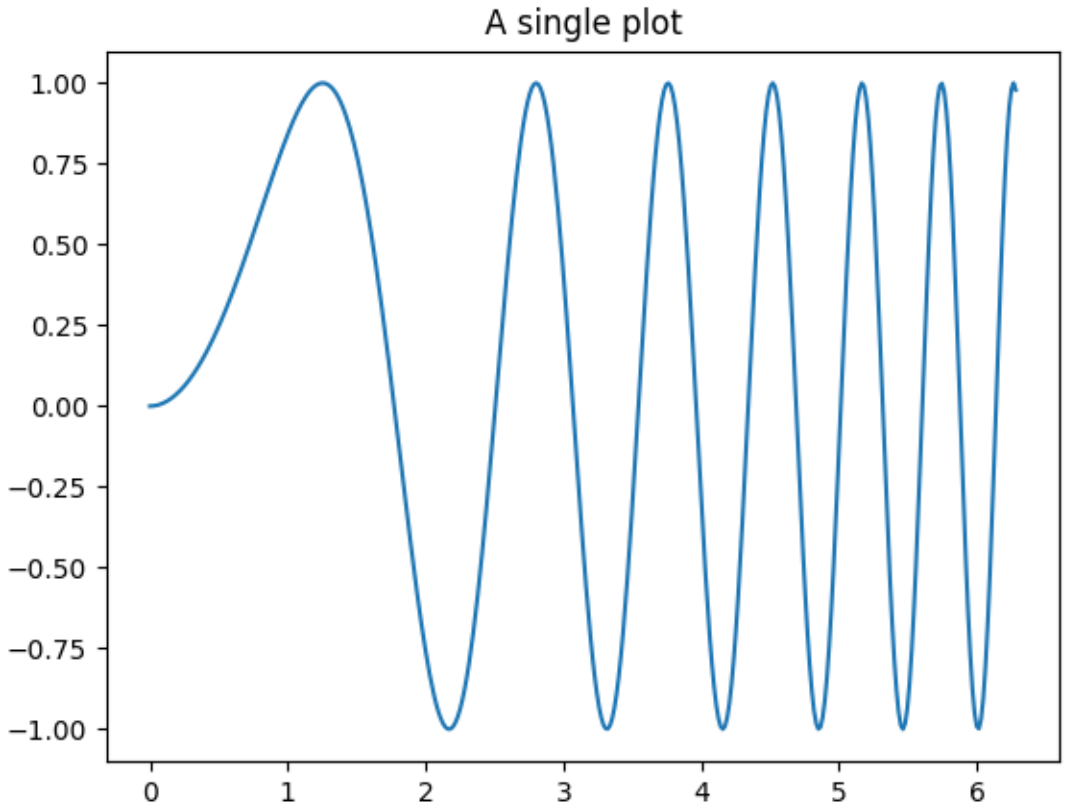
```
import matplotlib.pyplot as plt
import numpy as np

# Some example data to display
x = np.linspace(0, 2 * np.pi, 400)
y = np.sin(x ** 2)
```

### A figure with just one subplot

`subplots()` without arguments returns a *Figure* and a single *Axes*.

This is actually the simplest and recommended way of creating a single Figure and Axes.

```
fig, ax = plt.subplots()
ax.plot(x, y)
ax.set_title('A single plot')
```

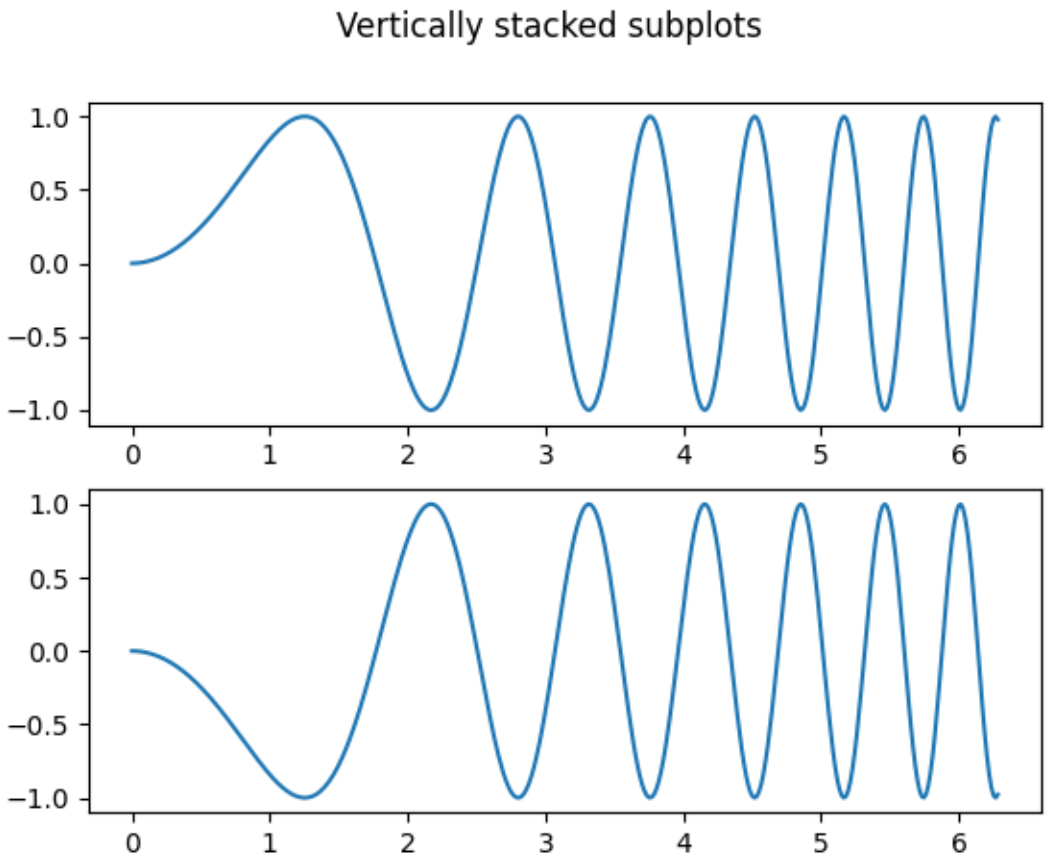



### Stacking subplots in one direction

The first two optional arguments of *pyplot.subplots* define the number of rows and columns of the subplot grid.

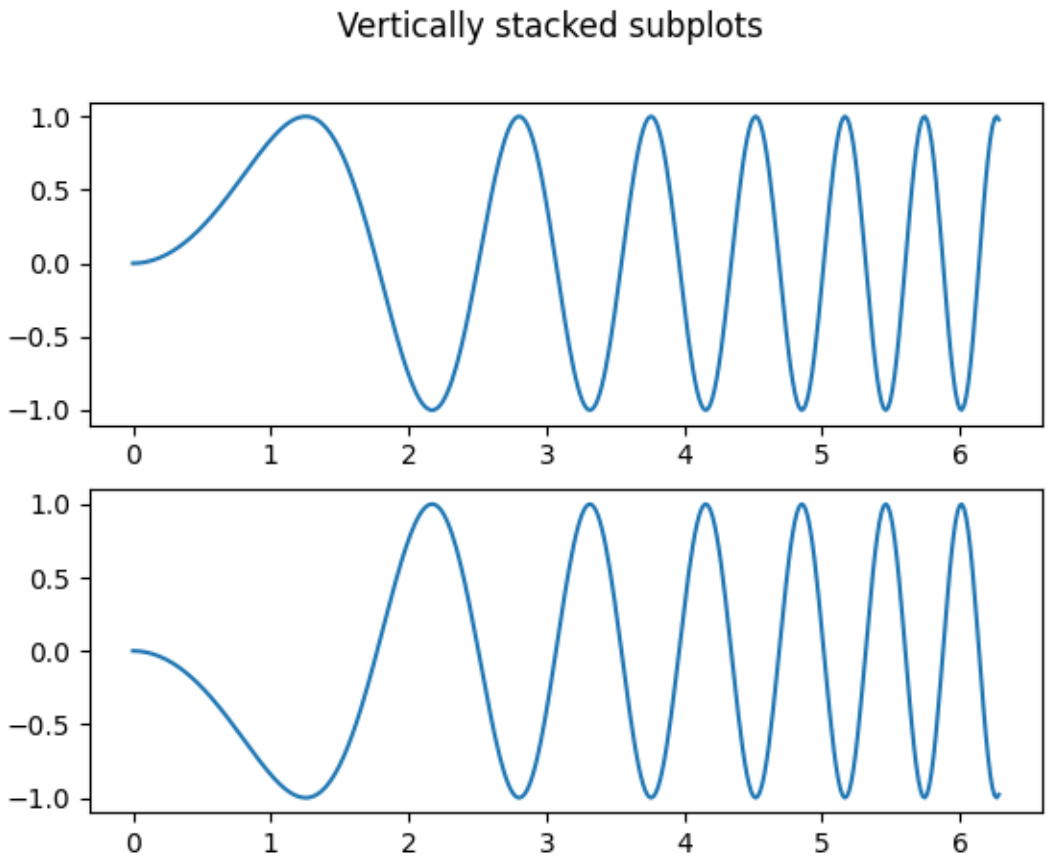
When stacking in one direction only, the returned `axs` is a 1D numpy array containing the list of created Axes.

```
fig, axs = plt.subplots(2)
fig.suptitle('Vertically stacked subplots')
axs[0].plot(x, y)
axs[1].plot(x, -y)
```



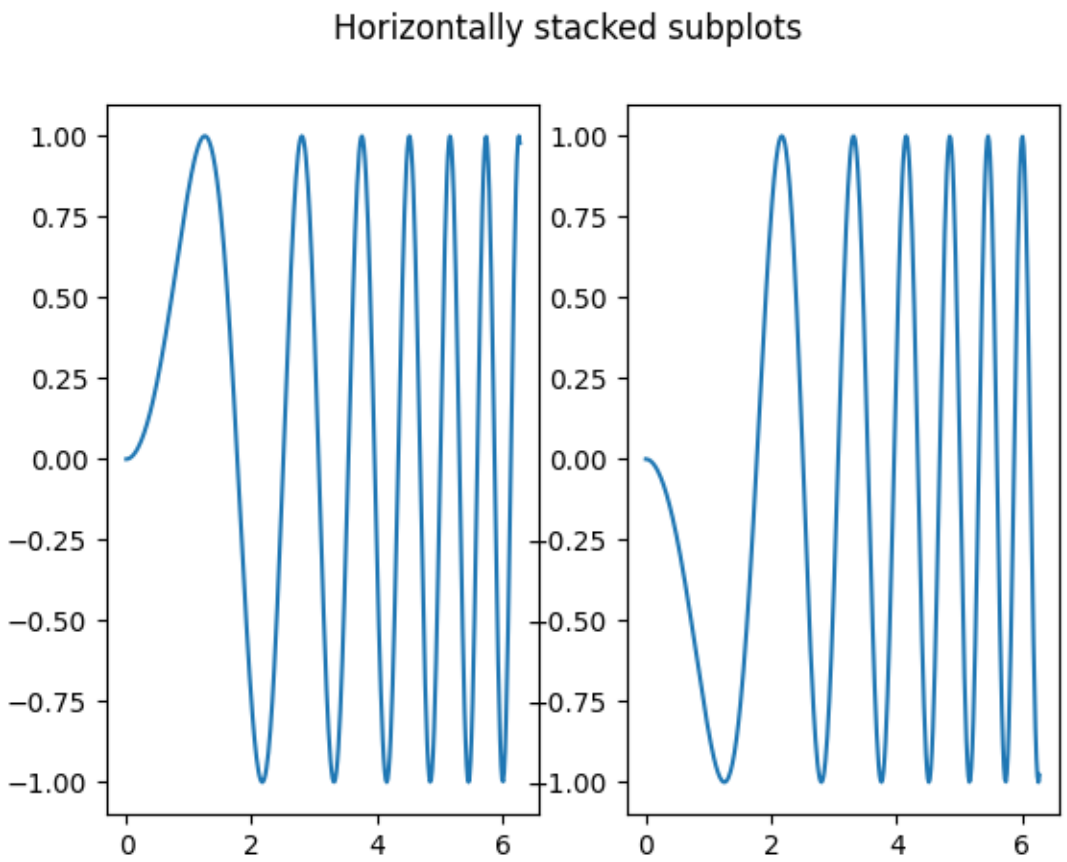

If you are creating just a few Axes, it's handy to unpack them immediately to dedicated variables for each Axes. That way, we can use `ax1` instead of the more verbose `axs[0]`.

```
fig, (ax1, ax2) = plt.subplots(2)
fig.suptitle('Vertically stacked subplots')
ax1.plot(x, y)
ax2.plot(x, -y)
```




To obtain side-by-side subplots, pass parameters `1, 2` for one row and two columns.

```
fig, (ax1, ax2) = plt.subplots(1, 2)
fig.suptitle('Horizontally stacked subplots')
ax1.plot(x, y)
ax2.plot(x, -y)
```




### Stacking subplots in two directions

When stacking in two directions, the returned `axs` is a 2D NumPy array.

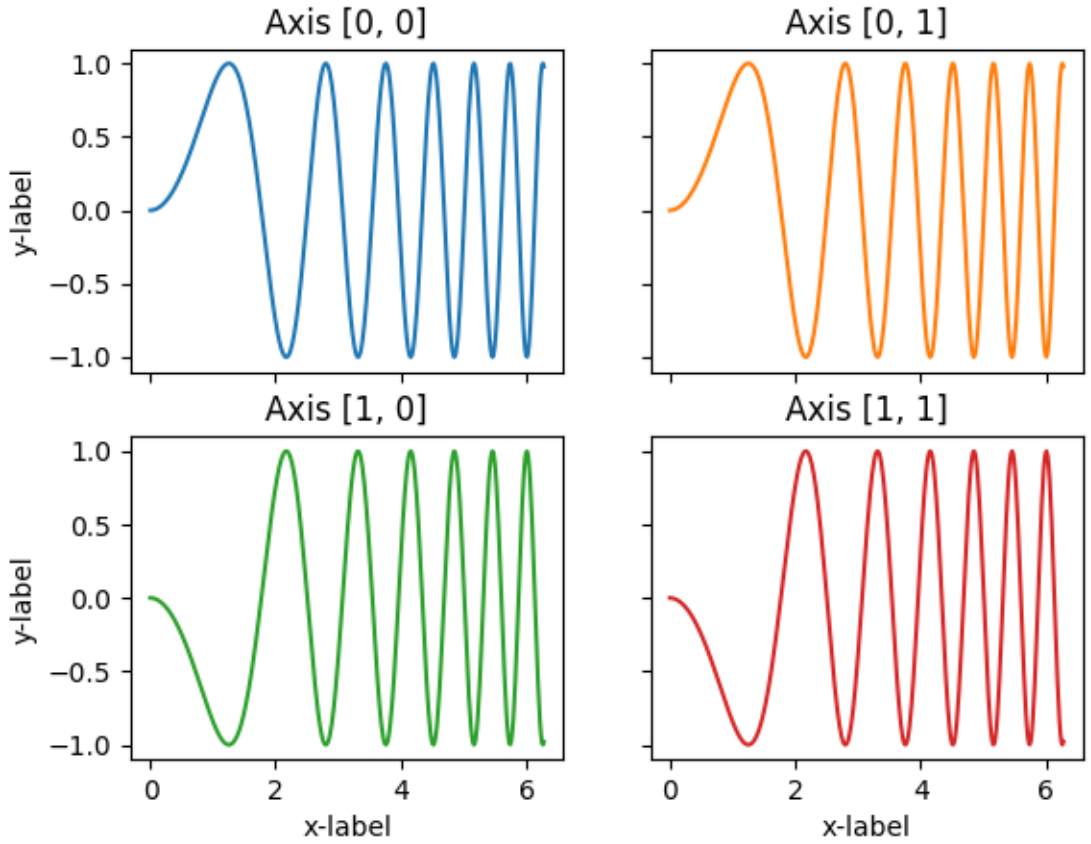
If you have to set parameters for each subplot it's handy to iterate over all subplots in a 2D grid using `for ax in axs.flat:`.

```
fig, axs = plt.subplots(2, 2)
axs[0, 0].plot(x, y)
axs[0, 0].set_title('Axis [0, 0]')
axs[0, 1].plot(x, y, 'tab:orange')
axs[0, 1].set_title('Axis [0, 1]')
axs[1, 0].plot(x, -y, 'tab:green')
axs[1, 0].set_title('Axis [1, 0]')
axs[1, 1].plot(x, -y, 'tab:red')
axs[1, 1].set_title('Axis [1, 1]')

for ax in axs.flat:
    ax.set(xlabel='x-label', ylabel='y-label')

# Hide x labels and tick labels for top plots and y ticks for right plots.
```

(continues on next page)

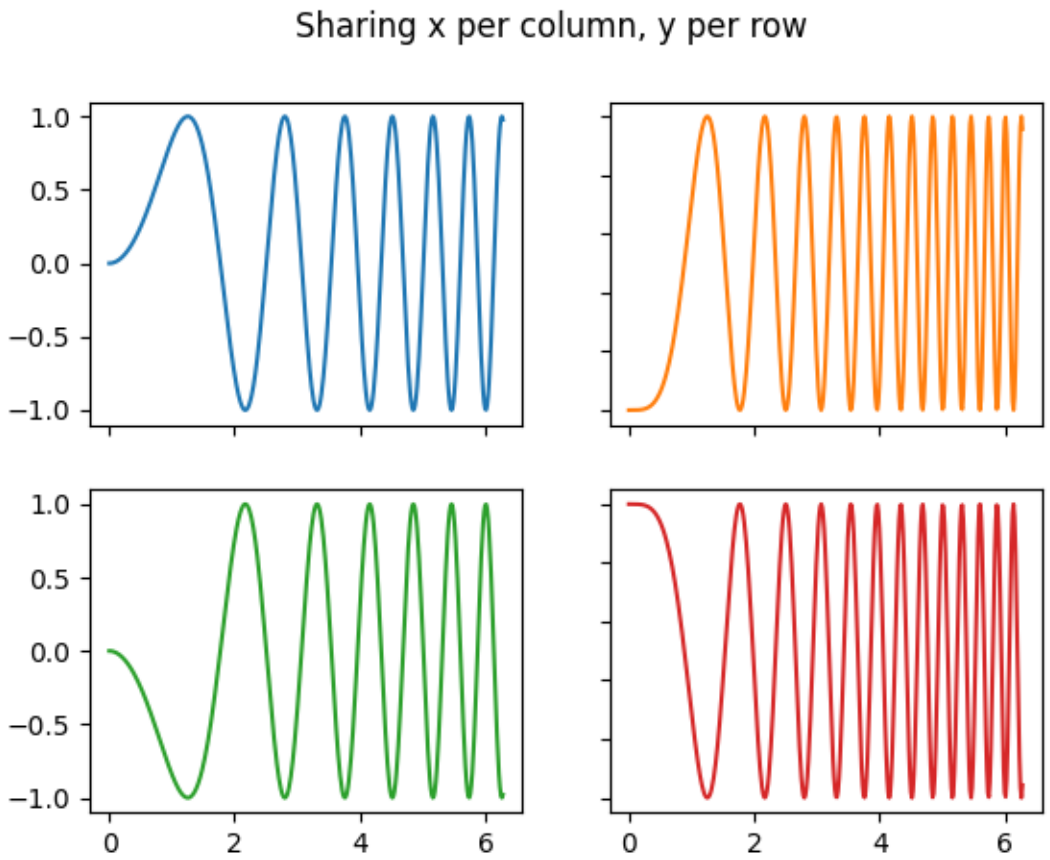
(continued from previous page)

```
for ax in axs.flat:
    ax.label_outer()
```




You can use tuple-unpacking also in 2D to assign all subplots to dedicated variables:

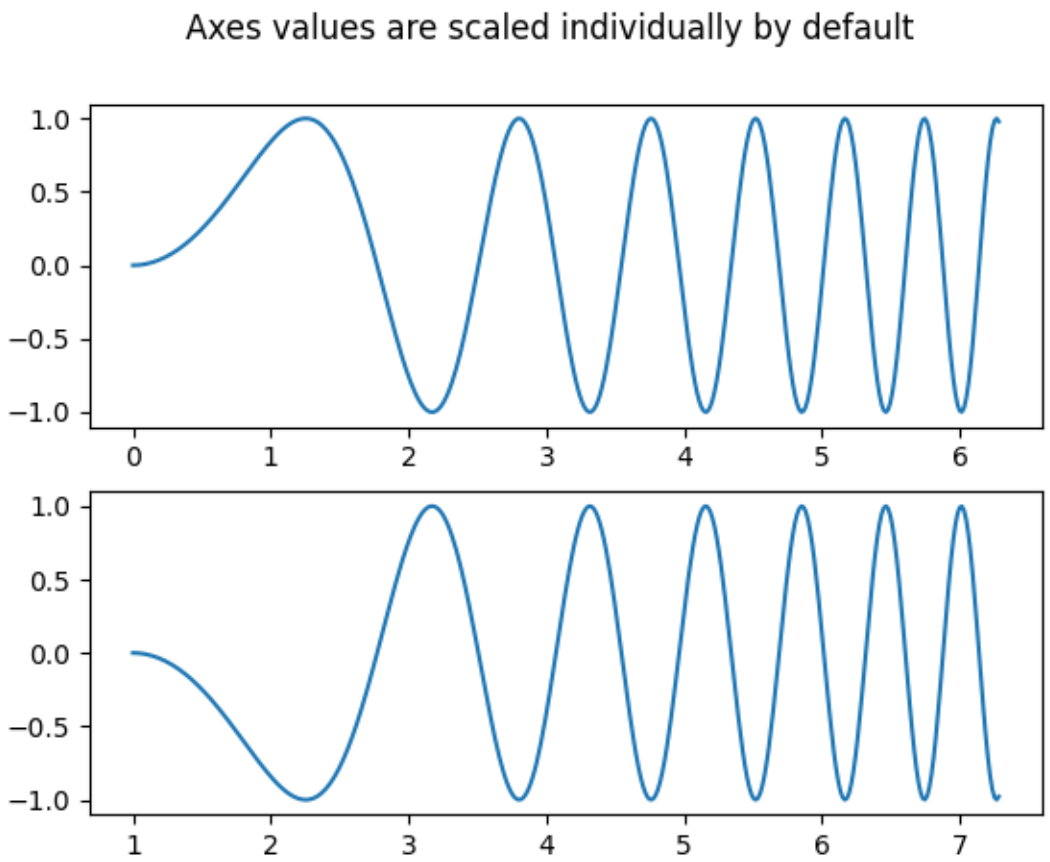
```
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2)
fig.suptitle('Sharing x per column, y per row')
ax1.plot(x, y)
ax2.plot(x, y**2, 'tab:orange')
ax3.plot(x, -y, 'tab:green')
ax4.plot(x, -y**2, 'tab:red')

for ax in fig.get_axes():
    ax.label_outer()
```




### Sharing axes

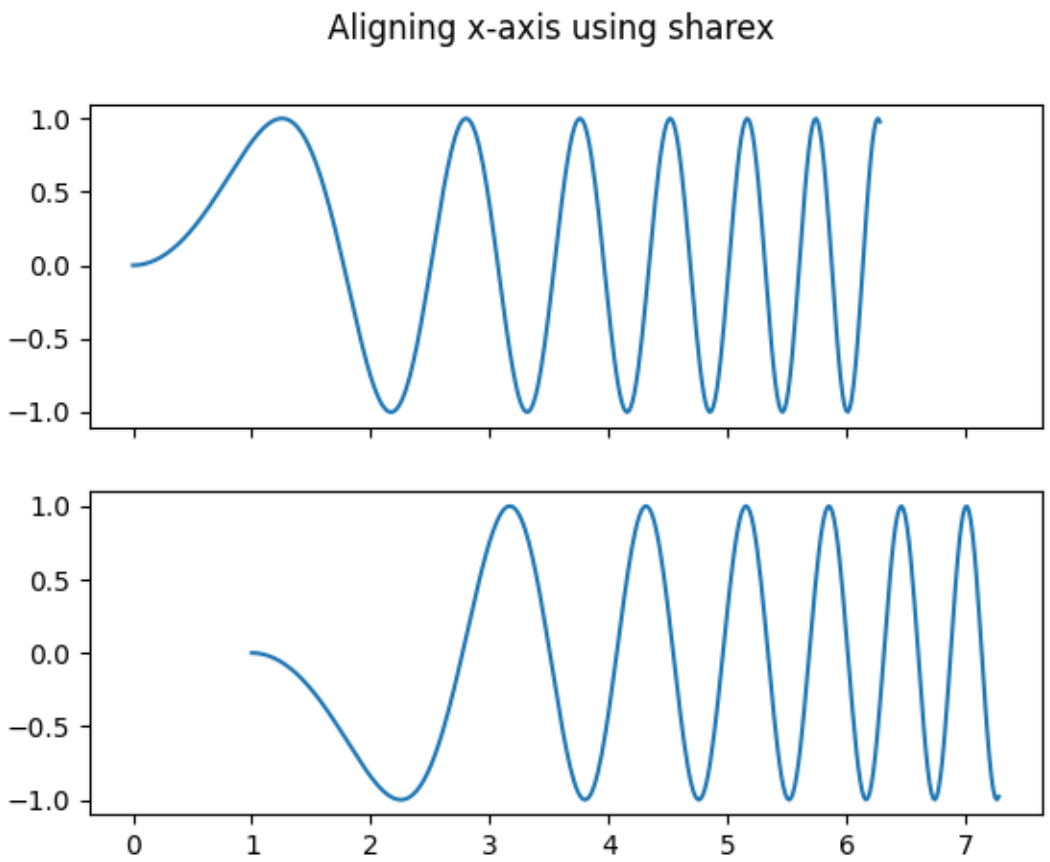
By default, each Axes is scaled individually. Thus, if the ranges are different the tick values of the subplots do not align.

```
fig, (ax1, ax2) = plt.subplots(2)
fig.suptitle('Axes values are scaled individually by default')
ax1.plot(x, y)
ax2.plot(x + 1, -y)
```



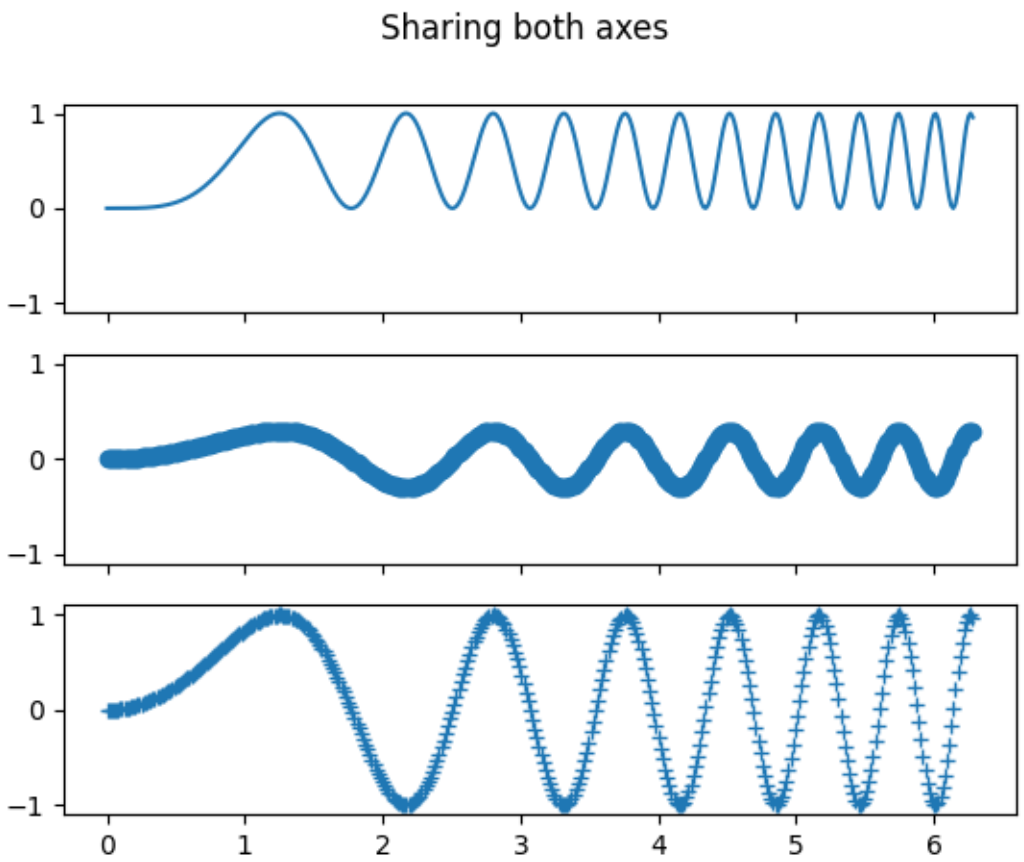

You can use *sharex* or *sharey* to align the horizontal or vertical axis.

```
fig, (ax1, ax2) = plt.subplots(2, sharex=True)
fig.suptitle('Aligning x-axis using sharex')
ax1.plot(x, y)
ax2.plot(x + 1, -y)
```




Setting *sharex* or *sharey* to `True` enables global sharing across the whole grid, i.e. also the y-axes of vertically stacked subplots have the same scale when using `sharey=True`.

```
fig, axs = plt.subplots(3, sharex=True, sharey=True)
fig.suptitle('Sharing both axes')
axs[0].plot(x, y ** 2)
axs[1].plot(x, 0.3 * y, 'o')
axs[2].plot(x, y, '+')
```

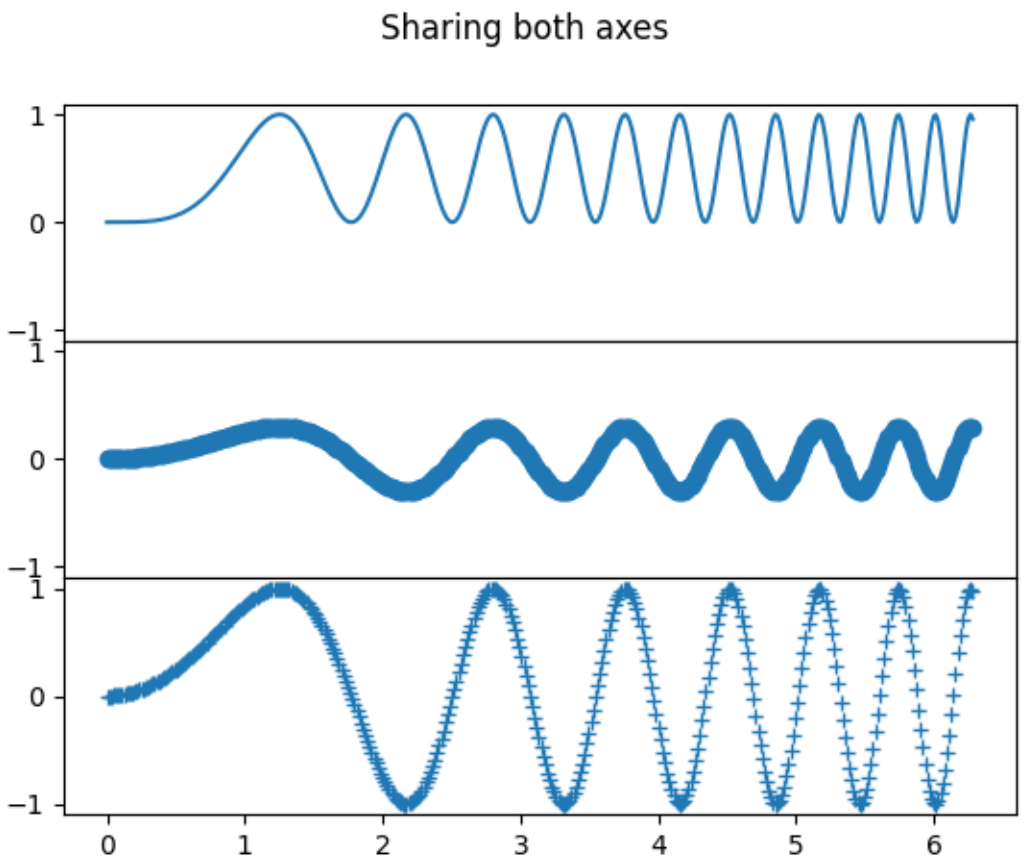



For subplots that are sharing axes one set of tick labels is enough. Tick labels of inner Axes are automatically removed by *sharex* and *sharey*. Still there remains an unused empty space between the subplots.

To precisely control the positioning of the subplots, one can explicitly create a *GridSpec* with *Figure.add_gridspec*, and then call its *subplots* method. For example, we can reduce the height between vertical subplots using `add_gridspec(hspace=0)`.

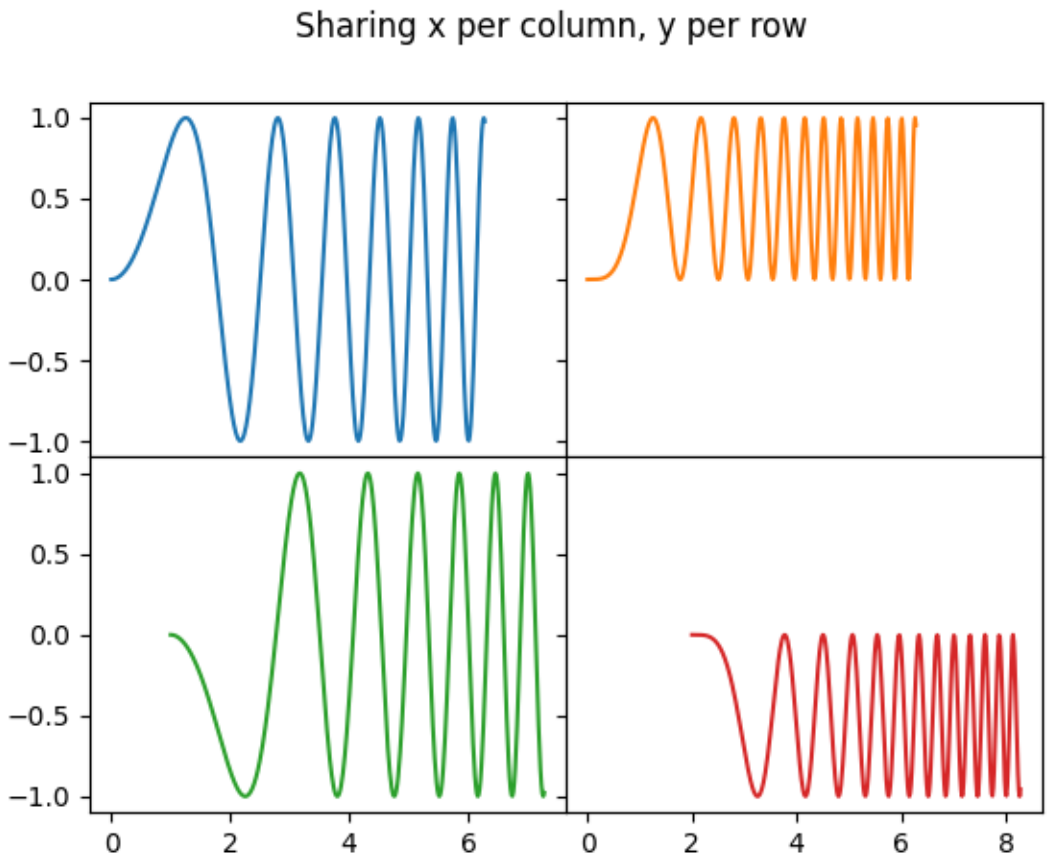
*label_outer* is a handy method to remove labels and ticks from subplots that are not at the edge of the grid.

```
fig = plt.figure()
gs = fig.add_gridspec(3, hspace=0)
axs = gs.subplots(sharex=True, sharey=True)
fig.suptitle('Sharing both axes')
axs[0].plot(x, y ** 2)
axs[1].plot(x, 0.3 * y, 'o')
axs[2].plot(x, y, '+')

# Hide x labels and tick labels for all but bottom plot.
for ax in axs:
    ax.label_outer()
```



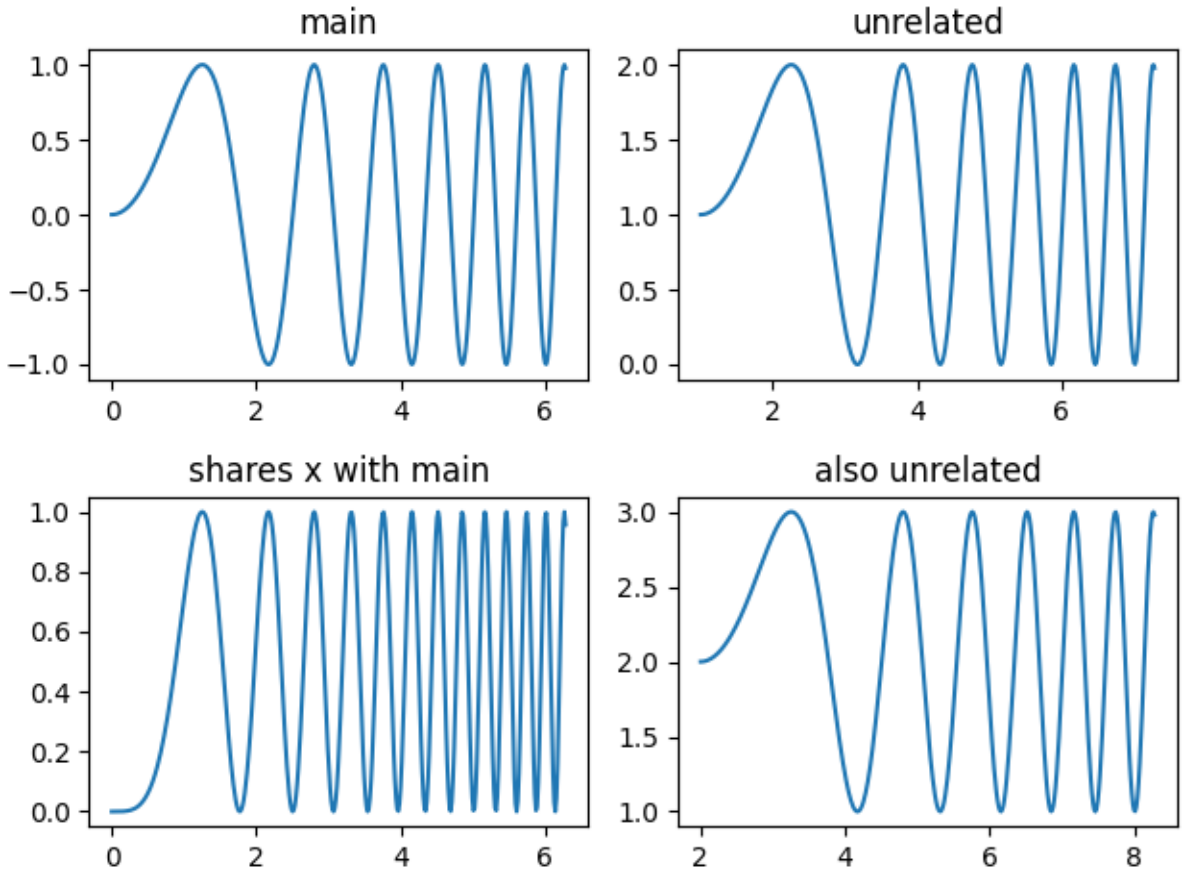

Apart from `True` and `False`, both *sharex* and *sharey* accept the values 'row' and 'col' to share the values only per row or column.

```
fig = plt.figure()
gs = fig.add_gridspec(2, 2, hspace=0, wspace=0)
(ax1, ax2), (ax3, ax4) = gs.subplots(sharex='col', sharey='row')
fig.suptitle('Sharing x per column, y per row')
ax1.plot(x, y)
ax2.plot(x, y**2, 'tab:orange')
ax3.plot(x + 1, -y, 'tab:green')
ax4.plot(x + 2, -y**2, 'tab:red')

for ax in fig.get_axes():
    ax.label_outer()
```




If you want a more complex sharing structure, you can first create the grid of Axes with no sharing, and then call *axes.Axes.sharex* or *axes.Axes.sharey* to add sharing info a posteriori.

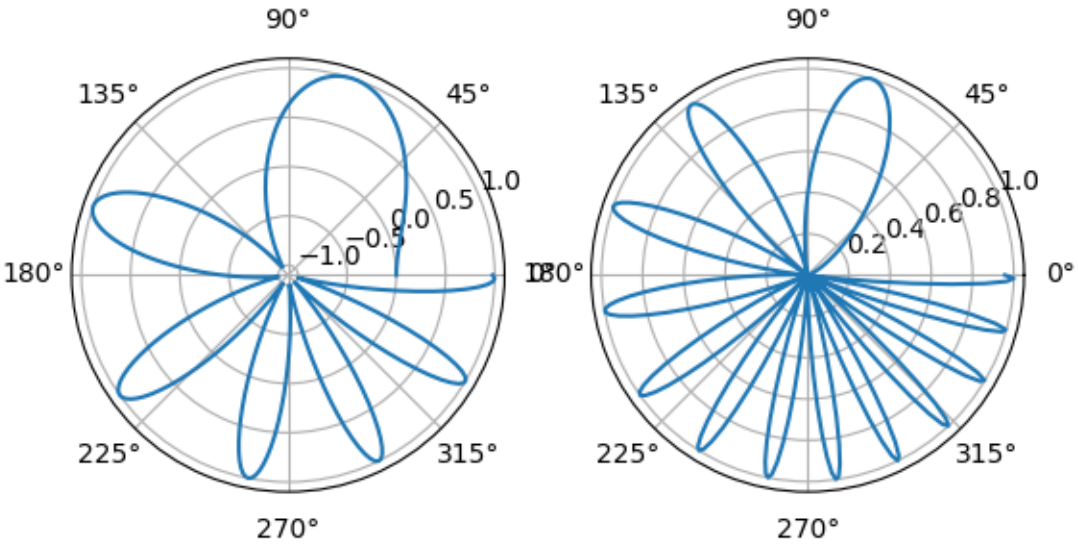
```
fig, axs = plt.subplots(2, 2)
axs[0, 0].plot(x, y)
axs[0, 0].set_title("main")
axs[1, 0].plot(x, y**2)
axs[1, 0].set_title("shares x with main")
axs[1, 0].sharex(axs[0, 0])
axs[0, 1].plot(x + 1, y + 1)
axs[0, 1].set_title("unrelated")
axs[1, 1].plot(x + 2, y + 2)
axs[1, 1].set_title("also unrelated")
fig.tight_layout()
```




### Polar Axes

The parameter *subplot_kw* of *pyplot.subplots* controls the subplot properties (see also *Figure.add_subplot*). In particular, this can be used to create a grid of polar Axes.

```
fig, (ax1, ax2) = plt.subplots(1, 2, subplot_kw=dict(projection='polar'))
ax1.plot(x, y)
ax2.plot(x, y ** 2)

plt.show()
```




**Total running time of the script:** (0 minutes 6.572 seconds)

### Plots with different scales

Two plots on the same Axes with different left and right scales.

The trick is to use *two different Axes* that share the same *x* axis. You can use separate `matplotlib.ticker` formatters and locators as desired since the two Axes are independent.

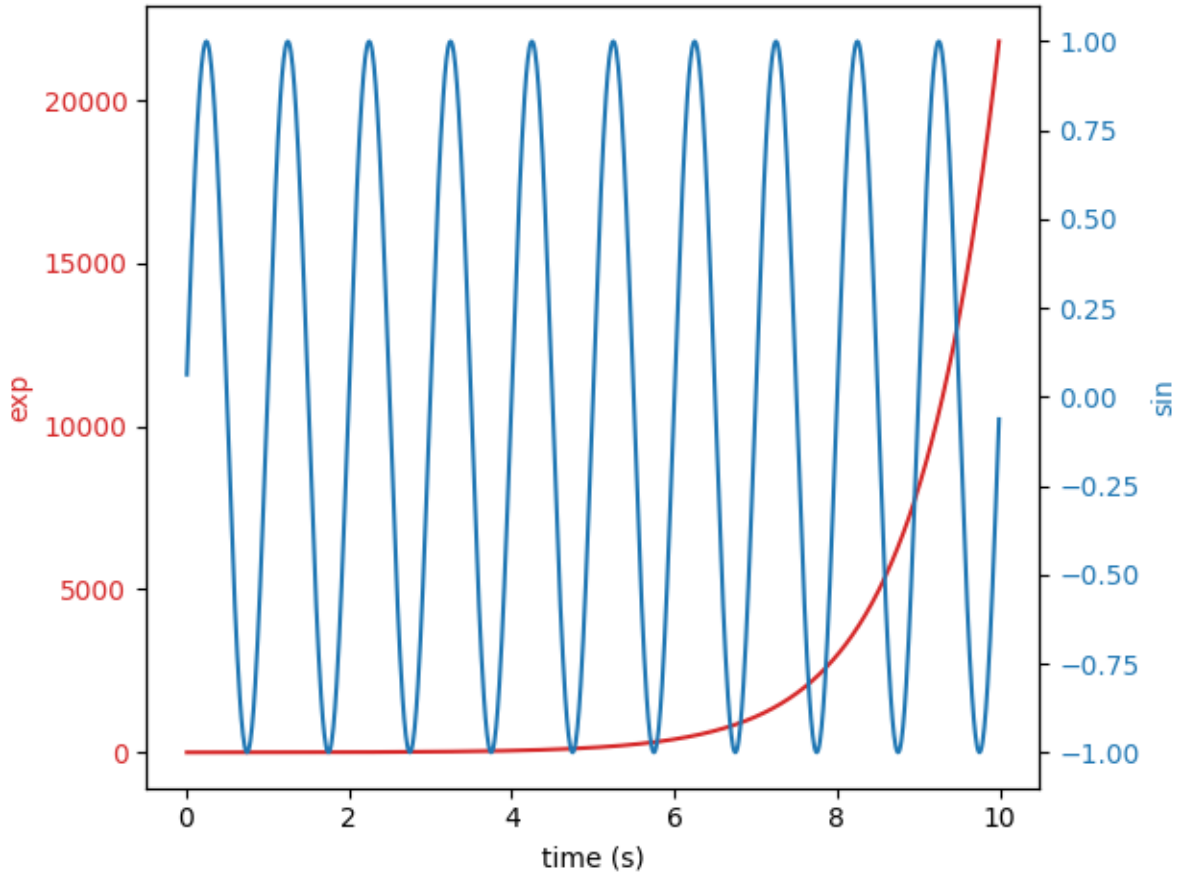
Such Axes are generated by calling the `Axes.twinx` method. Likewise, `Axes.twiny` is available to generate Axes that share a *y* axis but have different top and bottom scales.

```
import matplotlib.pyplot as plt
import numpy as np

# Create some mock data
t = np.arange(0.01, 10.0, 0.01)
data1 = np.exp(t)
data2 = np.sin(2 * np.pi * t)

fig, ax1 = plt.subplots()
```

(continues on next page)

(continued from previous page)

```
color = 'tab:red'
ax1.set_xlabel('time (s)')
ax1.set_ylabel('exp', color=color)
ax1.plot(t, data1, color=color)
ax1.tick_params(axis='y', labelcolor=color)

ax2 = ax1.twinx()  # instantiate a second Axes that shares the same x-axis

color = 'tab:blue'
ax2.set_ylabel('sin', color=color)  # we already handled the x-label with ax1
ax2.plot(t, data2, color=color)
ax2.tick_params(axis='y', labelcolor=color)

fig.tight_layout()  # otherwise the right y-label is slightly clipped
plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

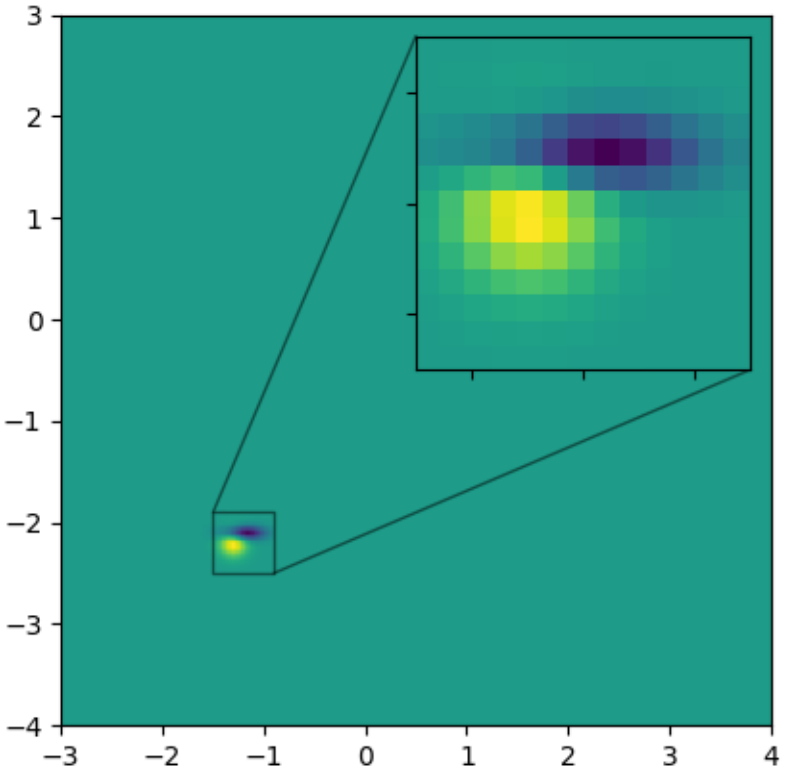
- *matplotlib.axes.Axes.twinx* / *matplotlib.pyplot.twinx*
- *matplotlib.axes.Axes.twiny* / *matplotlib.pyplot.twiny*

- *matplotlib.axes.Axes.tick_params* / *matplotlib.pyplot.tick_params*

### Zoom region inset Axes

Example of an inset Axes and a rectangle showing where the zoom is located.

```
import numpy as np

from matplotlib import cbook
from matplotlib import pyplot as plt

fig, ax = plt.subplots()

# make data
Z = cbook.get_sample_data("axes_grid/bivariate_normal.npy")  # 15x15 array
Z2 = np.zeros((150, 150))
ny, nx = Z.shape
Z2[30:30+ny, 30:30+nx] = Z
extent = (-3, 4, -4, 3)

ax.imshow(Z2, extent=extent, origin="lower")

# inset Axes....
x1, x2, y1, y2 = -1.5, -0.9, -2.5, -1.9  # subregion of the original image
axins = ax.inset_axes(
    [0.5, 0.5, 0.47, 0.47],
    xlim=(x1, x2), ylim=(y1, y2), xticklabels=[], yticklabels=[])
axins.imshow(Z2, extent=extent, origin="lower")

ax.indicate_inset_zoom(axins, edgecolor="black")

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.inset_axes*
- *matplotlib.axes.Axes.indicate_inset_zoom*
- *matplotlib.axes.Axes.imshow*

### 6.25.4 Statistics

#### Percentiles as horizontal bar chart

Bar charts are useful for visualizing counts, or summary statistics with error bars. Also see the *Grouped bar chart with labels* or the *Horizontal bar chart* example for simpler versions of those features.

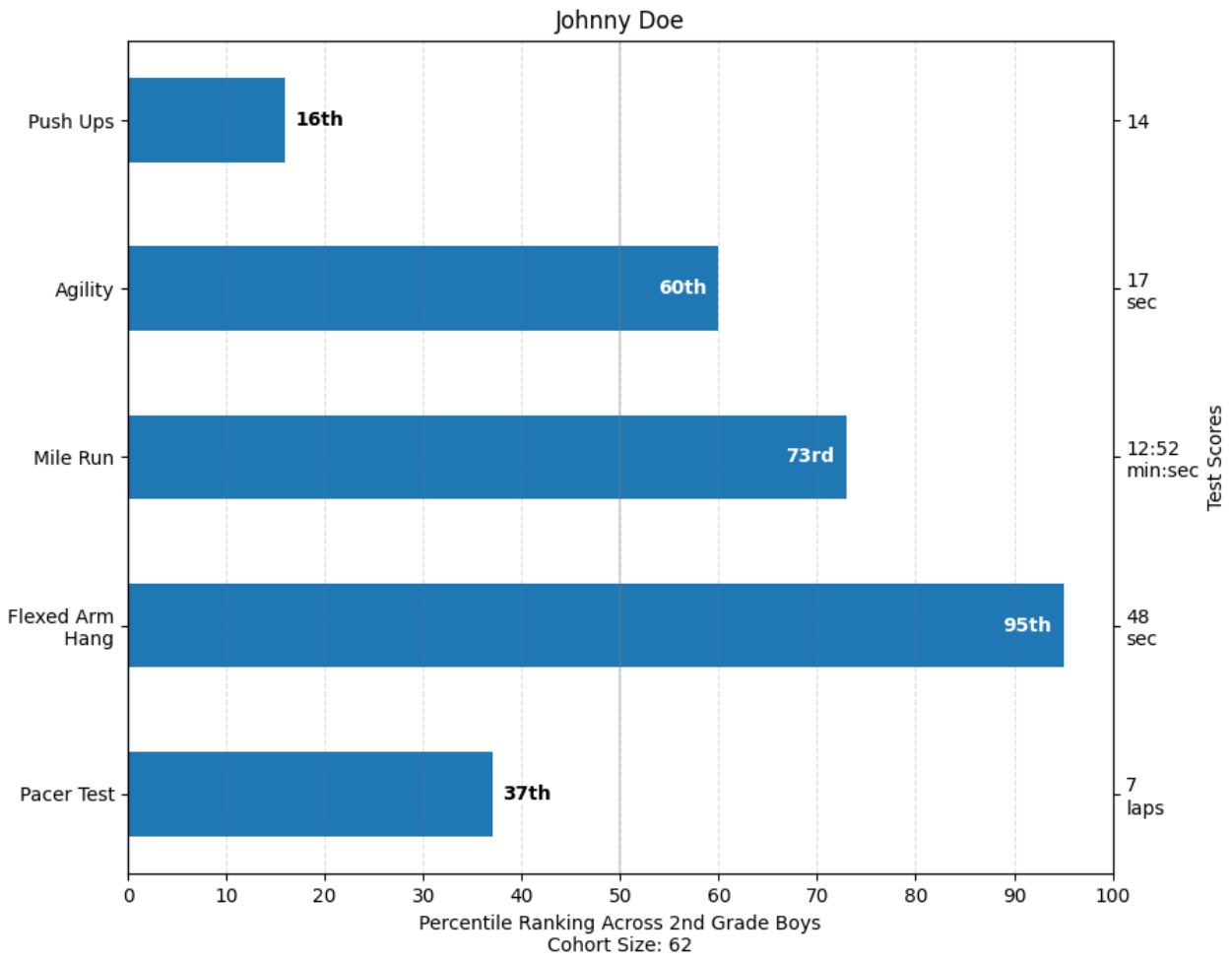
This example comes from an application in which grade school gym teachers wanted to be able to show parents how their child did across a handful of fitness tests, and importantly, relative to how other children did. To extract the plotting code for demo purposes, we'll just make up some data for little Johnny Doe.

```
from collections import namedtuple

import matplotlib.pyplot as plt
import numpy as np

Student = namedtuple('Student', ['name', 'grade', 'gender'])
Score = namedtuple('Score', ['value', 'unit', 'percentile'])


def to_ordinal(num):
    """Convert an integer to an ordinal string, e.g. 2 -> '2nd'."""
    suffixes = {str(i): v
                for i, v in enumerate(['th', 'st', 'nd', 'rd', 'th',
                                       'th', 'th', 'th', 'th', 'th'])}
    v = str(num)
    # special case early teens
    if v in {'11', '12', '13'}:
        return v + 'th'
    return v + suffixes[v[-1]]


def format_score(score):
    """
    Create score labels for the right y-axis as the test name followed by the
    measurement unit (if any), split over two lines.
    """
    return f'{score.value}\n{score.unit}' if score.unit else str(score.value)


def plot_student_results(student, scores_by_test, cohort_size):
    fig, ax1 = plt.subplots(figsize=(9, 7), layout='constrained')
    fig.canvas.manager.set_window_title('Eldorado K-8 Fitness Chart')

    ax1.set_title(student.name)
    ax1.set_xlabel(
        'Percentile Ranking Across {grade} Grade {gender}s\n'
        'Cohort Size: {cohort_size}'.format(
            grade=to_ordinal(student.grade),
            gender=student.gender.title(),
            cohort_size=cohort_size))

    test_names = list(scores_by_test.keys())
    percentiles = [score.percentile for score in scores_by_test.values()]

    rects = ax1.barh(test_names, percentiles, align='center', height=0.5)
    # Partition the percentile values to be able to draw large numbers in
    # white within the bar, and small numbers in black outside the bar.
    large_percentiles = [to_ordinal(p) if p > 40 else '' for p in percentiles]
    small_percentiles = [to_ordinal(p) if p <= 40 else '' for p in␣
↪percentiles]
    ax1.bar_label(rects, small_percentiles,
                  padding=5, color='black', fontweight='bold')
```

(continues on next page)

(continued from previous page)

```
    ax1.bar_label(rects, large_percentiles,
                  padding=-32, color='white', fontweight='bold')

    ax1.set_xlim([0, 100])
    ax1.set_xticks([0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100])
    ax1.xaxis.grid(True, linestyle='--', which='major',
                   color='grey', alpha=.25)
    ax1.axvline(50, color='grey', alpha=0.25)  # median position

    # Set the right-hand Y-axis ticks and labels
    ax2 = ax1.twinx()
    # Set equal limits on both yaxis so that the ticks line up
    ax2.set_ylim(ax1.get_ylim())
    # Set the tick locations and labels
    ax2.set_yticks(
        np.arange(len(scores_by_test)),
        labels=[format_score(score) for score in scores_by_test.values()])

    ax2.set_ylabel('Test Scores')


student = Student(name='Johnny Doe', grade=2, gender='Boy')
scores_by_test = {
    'Pacer Test': Score(7, 'laps', percentile=37),
    'Flexed Arm\n Hang': Score(48, 'sec', percentile=95),
    'Mile Run': Score('12:52', 'min:sec', percentile=73),
    'Agility': Score(17, 'sec', percentile=60),
    'Push Ups': Score(14, '', percentile=16),
}

plot_student_results(student, scores_by_test, cohort_size=62)
plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.bar* / *matplotlib.pyplot.bar*
- *matplotlib.axes.Axes.bar_label* / *matplotlib.pyplot.bar_label*
- *matplotlib.axes.Axes.twinx* / *matplotlib.pyplot.twinx*

## Artist customization in box plots

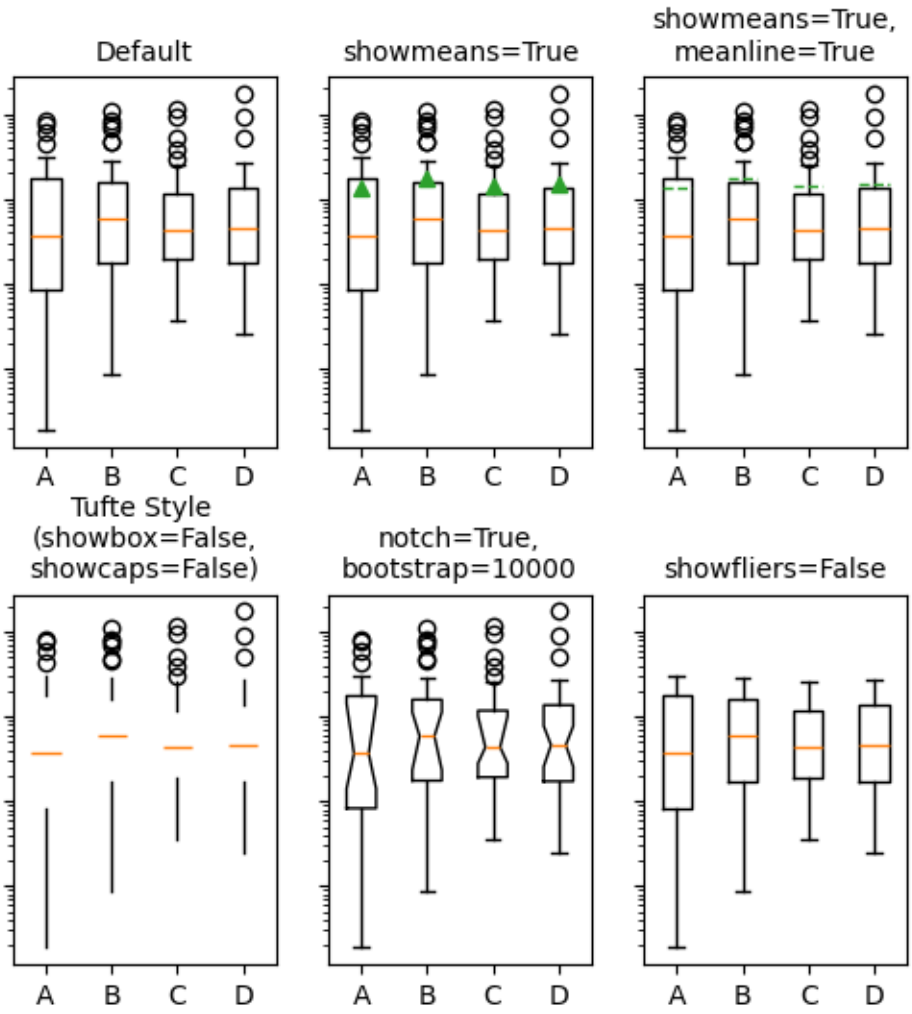
This example demonstrates how to use the various keyword arguments to fully customize box plots. The first figure demonstrates how to remove and add individual components (note that the mean is the only value not shown by default). The second figure demonstrates how the styles of the artists can be customized. It also demonstrates how to set the limit of the whiskers to specific percentiles (lower right Axes)

A good general reference on boxplots and their history can be found here: https://vita.had.co.nz/papers/boxplots.pdf

```
import matplotlib.pyplot as plt
import numpy as np

# fake data
np.random.seed(19680801)
data = np.random.lognormal(size=(37, 4), mean=1.5, sigma=1.75)
labels = list('ABCD')
fs = 10  # fontsize
```

Demonstrate how to toggle the display of different elements:

```
fig, axs = plt.subplots(nrows=2, ncols=3, figsize=(6, 6), sharey=True)
axs[0, 0].boxplot(data, tick_labels=labels)
axs[0, 0].set_title('Default', fontsize=fs)

axs[0, 1].boxplot(data, tick_labels=labels, showmeans=True)
axs[0, 1].set_title('showmeans=True', fontsize=fs)

axs[0, 2].boxplot(data, tick_labels=labels, showmeans=True, meanline=True)
axs[0, 2].set_title('showmeans=True,\nmeanline=True', fontsize=fs)

axs[1, 0].boxplot(data, tick_labels=labels, showbox=False, showcaps=False)
tufte_title = 'Tufte Style \n(showbox=False,\nshowcaps=False)'
axs[1, 0].set_title(tufte_title, fontsize=fs)

axs[1, 1].boxplot(data, tick_labels=labels, notch=True, bootstrap=10000)
axs[1, 1].set_title('notch=True,\nbootstrap=10000', fontsize=fs)

axs[1, 2].boxplot(data, tick_labels=labels, showfliers=False)
axs[1, 2].set_title('showfliers=False', fontsize=fs)

for ax in axs.flat:
    ax.set_yscale('log')
    ax.set_yticklabels([])

fig.subplots_adjust(hspace=0.4)
plt.show()
```



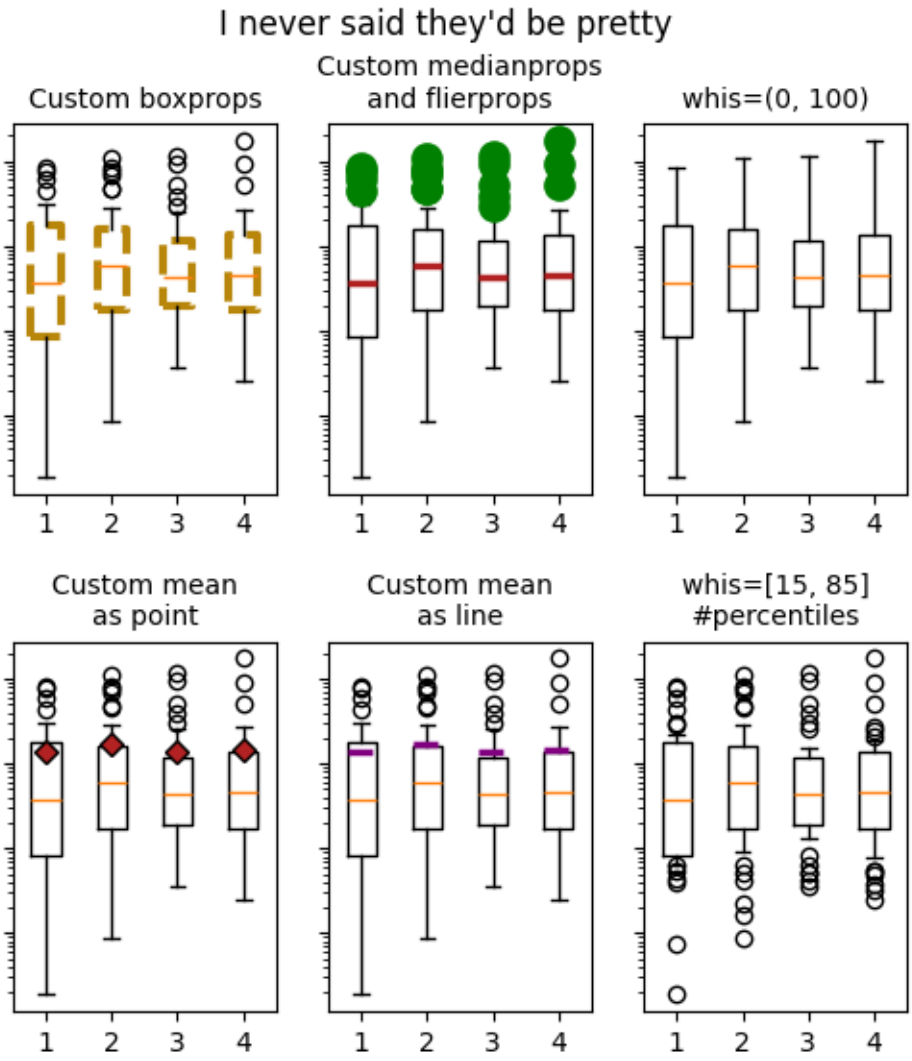

Demonstrate how to customize the display different elements:

```
boxprops = dict(linestyle='--', linewidth=3, color='darkgoldenrod')
flierprops = dict(marker='o', markerfacecolor='green', markersize=12,
                  markeredgecolor='none')
medianprops = dict(linestyle='-.', linewidth=2.5, color='firebrick')
meanpointprops = dict(marker='D', markeredgecolor='black',
                      markerfacecolor='firebrick')
meanlineprops = dict(linestyle='--', linewidth=2.5, color='purple')

fig, axs = plt.subplots(nrows=2, ncols=3, figsize=(6, 6), sharey=True)
axs[0, 0].boxplot(data, boxprops=boxprops)
axs[0, 0].set_title('Custom boxprops', fontsize=fs)

axs[0, 1].boxplot(data, flierprops=flierprops, medianprops=medianprops)
axs[0, 1].set_title('Custom medianprops\nand flierprops', fontsize=fs)
```

(continues on next page)

(continued from previous page)

```
axs[0, 2].boxplot(data, whis=(0, 100))
axs[0, 2].set_title('whis=(0, 100)', fontsize=fs)

axs[1, 0].boxplot(data, meanprops=meanpointprops, meanline=False,
                  showmeans=True)
axs[1, 0].set_title('Custom mean\nas point', fontsize=fs)

axs[1, 1].boxplot(data, meanprops=meanlineprops, meanline=True,
                  showmeans=True)
axs[1, 1].set_title('Custom mean\nas line', fontsize=fs)

axs[1, 2].boxplot(data, whis=[15, 85])
axs[1, 2].set_title('whis=[15, 85]\n#percentiles', fontsize=fs)

for ax in axs.flat:
    ax.set_yscale('log')
    ax.set_yticklabels([])

fig.suptitle("I never said they'd be pretty")
fig.subplots_adjust(hspace=0.4)
plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.boxplot* / *matplotlib.pyplot.boxplot*

**Total running time of the script:** (0 minutes 2.155 seconds)

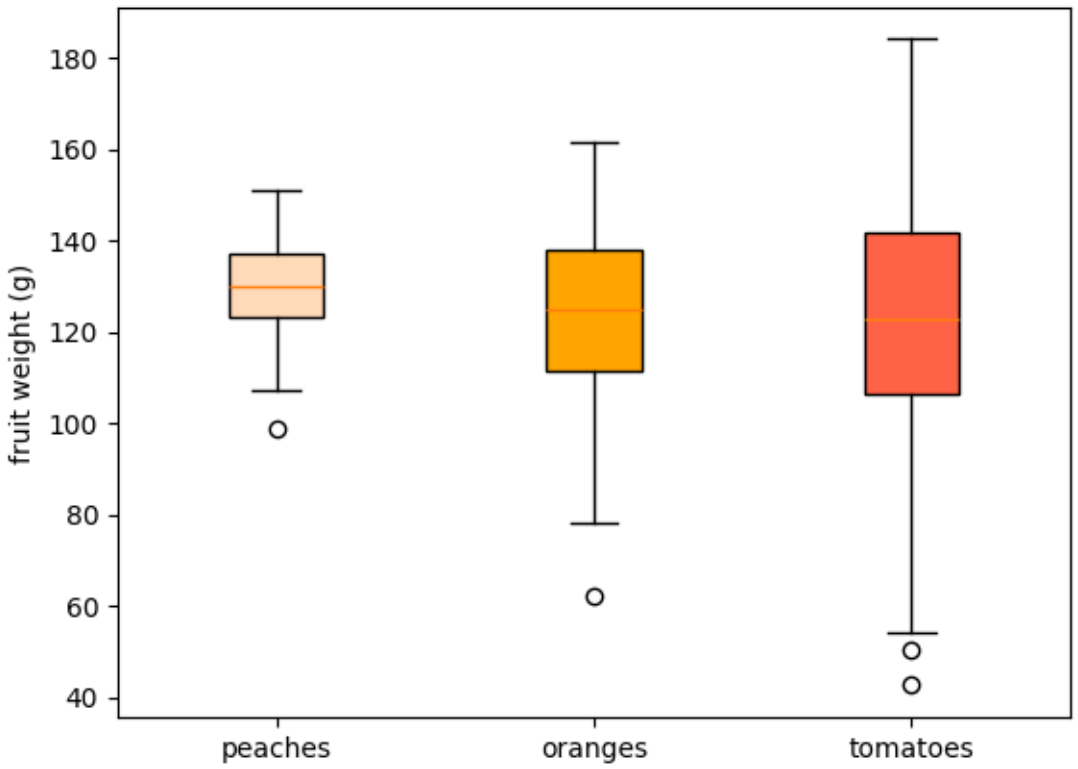
### Box plots with custom fill colors

To color each box of a box plot individually:

1) use the keyword argument `patch_artist=True` to create filled boxes.

2) loop through the created boxes and adapt their color.

```
import matplotlib.pyplot as plt
import numpy as np

np.random.seed(19680801)
fruit_weights = [
    np.random.normal(130, 10, size=100),
    np.random.normal(125, 20, size=100),
    np.random.normal(120, 30, size=100),
]
labels = ['peaches', 'oranges', 'tomatoes']
colors = ['peachpuff', 'orange', 'tomato']

fig, ax = plt.subplots()
ax.set_ylabel('fruit weight (g)')

bplot = ax.boxplot(fruit_weights,
                   patch_artist=True,  # fill with color
                   tick_labels=labels)  # will be used to label x-ticks

# fill with colors
for patch, color in zip(bplot['boxes'], colors):
    patch.set_facecolor(color)

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.boxplot* / *matplotlib.pyplot.boxplot*

## Boxplots

Visualizing boxplots with matplotlib.

The following examples show off how to visualize boxplots with Matplotlib. There are many options to control their appearance and the statistics that they use to summarize the data.

```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.patches import Polygon

# Fixing random state for reproducibility
np.random.seed(19680801)
```

(continues on next page)

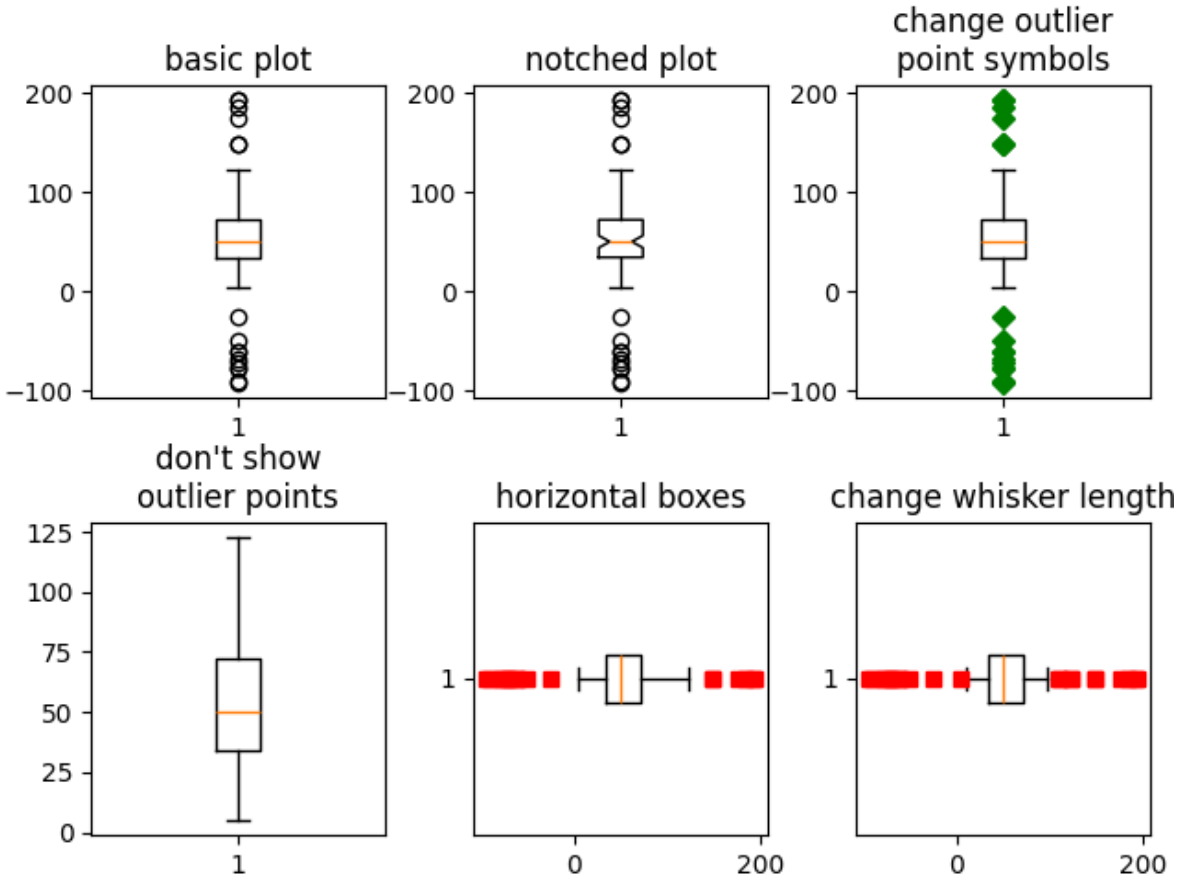
(continued from previous page)

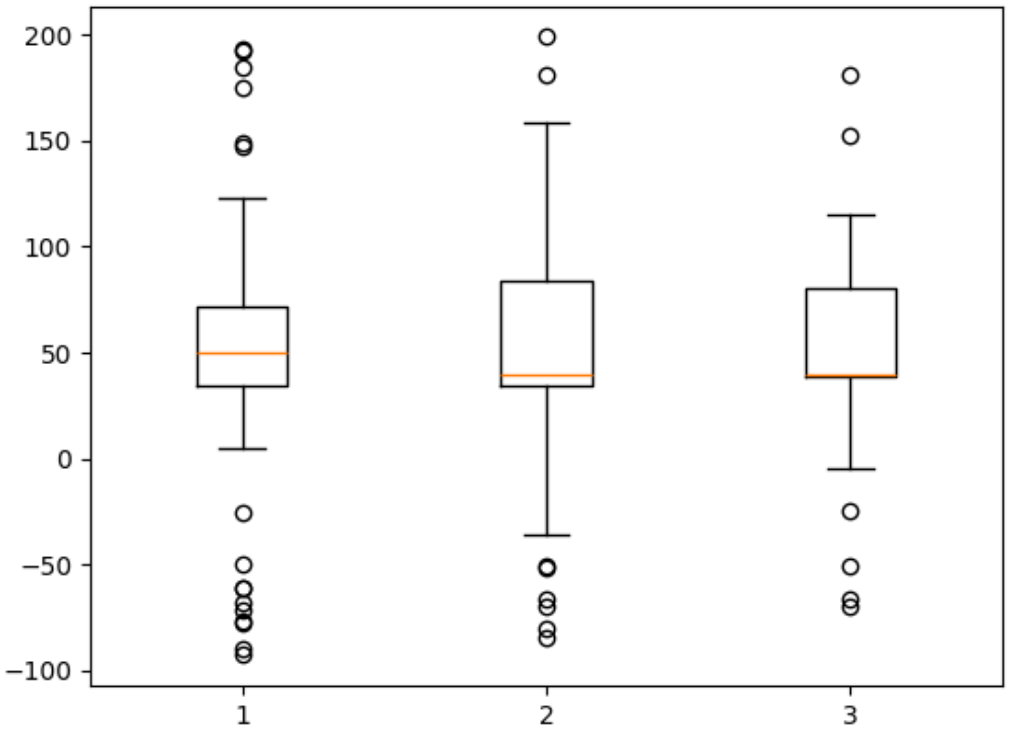
```
# fake up some data
spread = np.random.rand(50) * 100
center = np.ones(25) * 50
flier_high = np.random.rand(10) * 100 + 100
flier_low = np.random.rand(10) * -100
data = np.concatenate((spread, center, flier_high, flier_low))

fig, axs = plt.subplots(2, 3)

# basic plot
axs[0, 0].boxplot(data)
axs[0, 0].set_title('basic plot')

# notched plot
axs[0, 1].boxplot(data, 1)
axs[0, 1].set_title('notched plot')

# change outlier point symbols
axs[0, 2].boxplot(data, 0, 'gD')
axs[0, 2].set_title('change outlier\npoint symbols')

# don't show outlier points
axs[1, 0].boxplot(data, 0, '')
axs[1, 0].set_title("don't show\noutlier points")

# horizontal boxes
axs[1, 1].boxplot(data, 0, 'rs', 0)
axs[1, 1].set_title('horizontal boxes')

# change whisker length
axs[1, 2].boxplot(data, 0, 'rs', 0, 0.75)
axs[1, 2].set_title('change whisker length')

fig.subplots_adjust(left=0.08, right=0.98, bottom=0.05, top=0.9,
                    hspace=0.4, wspace=0.3)

# fake up some more data
spread = np.random.rand(50) * 100
center = np.ones(25) * 40
flier_high = np.random.rand(10) * 100 + 100
flier_low = np.random.rand(10) * -100
d2 = np.concatenate((spread, center, flier_high, flier_low))
# Making a 2-D array only works if all the columns are the
# same length.  If they are not, then use a list instead.
# This is actually more efficient because boxplot converts
# a 2-D array into a list of vectors internally anyway.
data = [data, d2, d2[::2]]

# Multiple box plots on one Axes
fig, ax = plt.subplots()
ax.boxplot(data)
```

(continues on next page)

(continued from previous page)

```
plt.show()
```






- 

Below we'll generate data from five different probability distributions, each with different characteristics. We want to play with how an IID bootstrap resample of the data preserves the distributional properties of the original sample, and a boxplot is one visual tool to make this assessment

```
random_dists = ['Normal(1, 1)', 'Lognormal(1, 1)', 'Exp(1)', 'Gumbel(6, 4)',
                'Triangular(2, 9, 11)']
N = 500

norm = np.random.normal(1, 1, N)
logn = np.random.lognormal(1, 1, N)
expo = np.random.exponential(1, N)
gumb = np.random.gumbel(6, 4, N)
tria = np.random.triangular(2, 9, 11, N)

# Generate some random indices that we'll use to resample the original data
# arrays. For code brevity, just use the same random indices for each array
bootstrap_indices = np.random.randint(0, N, N)
data = [
    norm, norm[bootstrap_indices],
    logn, logn[bootstrap_indices],
    expo, expo[bootstrap_indices],
    gumb, gumb[bootstrap_indices],
    tria, tria[bootstrap_indices],
]
```

(continues on next page)

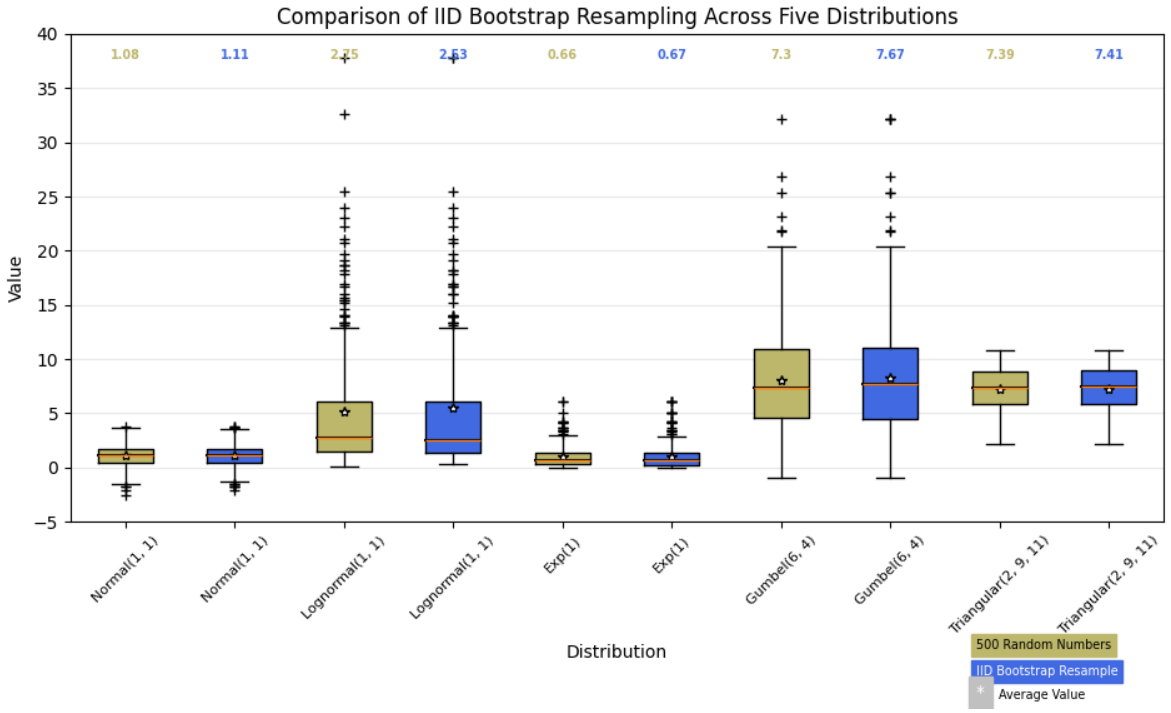
(continued from previous page)

```
fig, ax1 = plt.subplots(figsize=(10, 6))
fig.canvas.manager.set_window_title('A Boxplot Example')
fig.subplots_adjust(left=0.075, right=0.95, top=0.9, bottom=0.25)

bp = ax1.boxplot(data, notch=False, sym='+', vert=True, whis=1.5)
plt.setp(bp['boxes'], color='black')
plt.setp(bp['whiskers'], color='black')
plt.setp(bp['fliers'], color='red', marker='+')

# Add a horizontal grid to the plot, but make it very light in color
# so we can use it for reading data values but not be distracting
ax1.yaxis.grid(True, linestyle='-', which='major', color='lightgrey',
               alpha=0.5)

ax1.set(
    axisbelow=True,  # Hide the grid behind plot objects
    title='Comparison of IID Bootstrap Resampling Across Five Distributions',
    xlabel='Distribution',
    ylabel='Value',
)

# Now fill the boxes with desired colors
box_colors = ['darkkhaki', 'royalblue']
num_boxes = len(data)
medians = np.empty(num_boxes)
for i in range(num_boxes):
    box = bp['boxes'][i]
    box_x = []
    box_y = []
    for j in range(5):
        box_x.append(box.get_xdata()[j])
        box_y.append(box.get_ydata()[j])
    box_coords = np.column_stack([box_x, box_y])
    # Alternate between Dark Khaki and Royal Blue
    ax1.add_patch(Polygon(box_coords, facecolor=box_colors[i % 2]))
    # Now draw the median lines back over what we just filled in
    med = bp['medians'][i]
    median_x = []
    median_y = []
    for j in range(2):
        median_x.append(med.get_xdata()[j])
        median_y.append(med.get_ydata()[j])
        ax1.plot(median_x, median_y, 'k')
    medians[i] = median_y[0]
    # Finally, overplot the sample averages, with horizontal alignment
    # in the center of each box
    ax1.plot(np.average(med.get_xdata()), np.average(data[i]),
             color='w', marker='*', markeredgecolor='k')

# Set the axes ranges and axes labels
ax1.set_xlim(0.5, num_boxes + 0.5)
```

(continues on next page)

(continued from previous page)

```
top = 40
bottom = -5
ax1.set_ylim(bottom, top)
ax1.set_xticklabels(np.repeat(random_dists, 2),
                    rotation=45, fontsize=8)

# Due to the Y-axis scale being different across samples, it can be
# hard to compare differences in medians across the samples. Add upper
# X-axis tick labels with the sample medians to aid in comparison
# (just use two decimal places of precision)
pos = np.arange(num_boxes) + 1
upper_labels = [str(round(s, 2)) for s in medians]
weights = ['bold', 'semibold']
for tick, label in zip(range(num_boxes), ax1.get_xticklabels()):
    k = tick % 2
    ax1.text(pos[tick], .95, upper_labels[tick],
             transform=ax1.get_xaxis_transform(),
             horizontalalignment='center', size='x-small',
             weight=weights[k], color=box_colors[k])

# Finally, add a basic legend
fig.text(0.80, 0.08, f'{N} Random Numbers',
         backgroundcolor=box_colors[0], color='black', weight='roman',
         size='x-small')
fig.text(0.80, 0.045, 'IID Bootstrap Resample',
         backgroundcolor=box_colors[1],
         color='white', weight='roman', size='x-small')
fig.text(0.80, 0.015, '*', color='white', backgroundcolor='silver',
         weight='roman', size='medium')
fig.text(0.815, 0.013, ' Average Value', color='black', weight='roman',
         size='x-small')

plt.show()
```



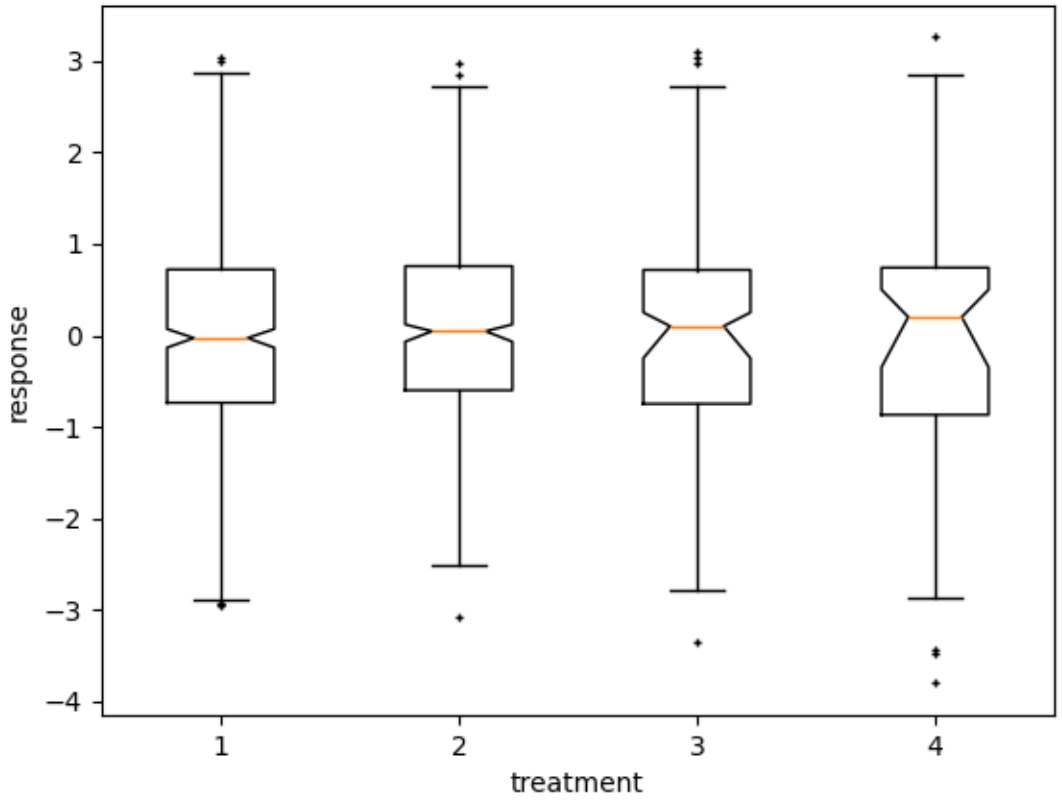

Here we write a custom function to bootstrap confidence intervals. We can then use the boxplot along with this function to show these intervals.

```
def fake_bootstrapper(n):
    """
    This is just a placeholder for the user's method of
    bootstrapping the median and its confidence intervals.

    Returns an arbitrary median and confidence interval packed into a tuple.
    """
    if n == 1:
        med = 0.1
        ci = (-0.25, 0.25)
    else:
        med = 0.2
        ci = (-0.35, 0.50)
    return med, ci

inc = 0.1
e1 = np.random.normal(0, 1, size=500)
e2 = np.random.normal(0, 1, size=500)
e3 = np.random.normal(0, 1 + inc, size=500)
e4 = np.random.normal(0, 1 + 2*inc, size=500)

treatments = [e1, e2, e3, e4]
med1, ci1 = fake_bootstrapper(1)
med2, ci2 = fake_bootstrapper(2)
medians = [None, None, med1, med2]
conf_intervals = [None, None, ci1, ci2]
```

(continues on next page)

(continued from previous page)

```
fig, ax = plt.subplots()
pos = np.arange(len(treatments)) + 1
bp = ax.boxplot(treatments, sym='k+', positions=pos,
                notch=True, bootstrap=5000,
                usermedians=medians,
                conf_intervals=conf_intervals)

ax.set_xlabel('treatment')
ax.set_ylabel('response')
plt.setp(bp['whiskers'], color='k', linestyle='-')
plt.setp(bp['fliers'], markersize=3.0)
plt.show()
```



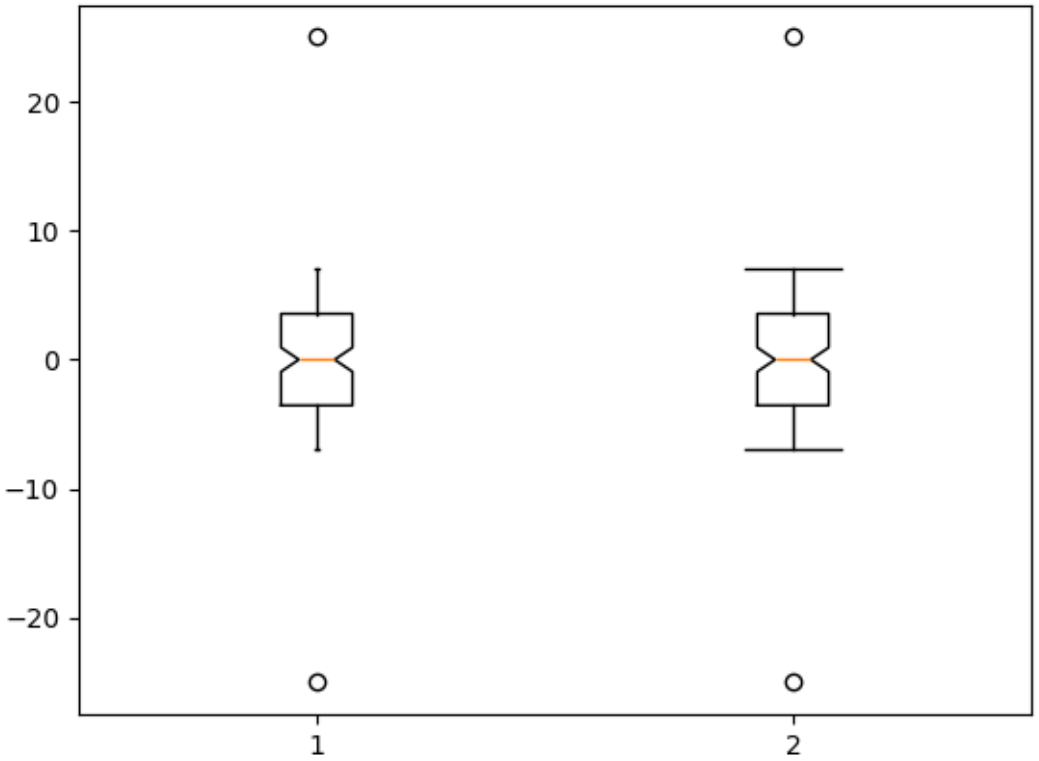

Here we customize the widths of the caps .

```
x = np.linspace(-7, 7, 140)
x = np.hstack([-25, x, 25])
fig, ax = plt.subplots()

ax.boxplot([x, x], notch=True, capwidths=[0.01, 0.2])
```

(continues on next page)

(continued from previous page)

```
plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.boxplot* / *matplotlib.pyplot.boxplot*
- *matplotlib.artist.Artist.set* / *matplotlib.pyplot.setp*

**Total running time of the script:** (0 minutes 2.207 seconds)

### Box plot vs. violin plot comparison

Note that although violin plots are closely related to Tukey's (1977) box plots, they add useful information such as the distribution of the sample data (density trace).

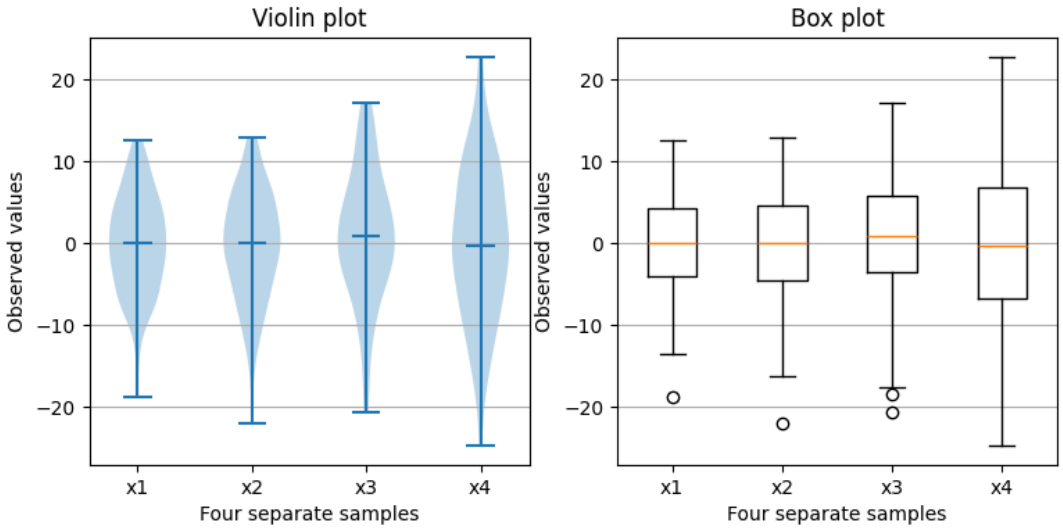
By default, box plots show data points outside 1.5 * the inter-quartile range as outliers above or below the whiskers whereas violin plots show the whole range of the data.

A good general reference on boxplots and their history can be found here: http://vita.had.co.nz/papers/boxplots.pdf

Violin plots require matplotlib >= 1.4.

For more information on violin plots, the scikit-learn docs have a great section: https://scikit-learn.org/stable/modules/density.html

```
import matplotlib.pyplot as plt
import numpy as np

fig, axs = plt.subplots(nrows=1, ncols=2, figsize=(9, 4))

# Fixing random state for reproducibility
np.random.seed(19680801)


# generate some random test data
all_data = [np.random.normal(0, std, 100) for std in range(6, 10)]

# plot violin plot
axs[0].violinplot(all_data,
                  showmeans=False,
                  showmedians=True)
axs[0].set_title('Violin plot')

# plot box plot
axs[1].boxplot(all_data)
axs[1].set_title('Box plot')

# adding horizontal grid lines
for ax in axs:
    ax.yaxis.grid(True)
    ax.set_xticks([y + 1 for y in range(len(all_data))],
                  labels=['x1', 'x2', 'x3', 'x4'])
    ax.set_xlabel('Four separate samples')
    ax.set_ylabel('Observed values')

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.boxplot* / *matplotlib.pyplot.boxplot*
- *matplotlib.axes.Axes.violinplot* / *matplotlib.pyplot.violinplot*

## Separate calculation and plotting of boxplots

Drawing a *boxplot* for a given data set, consists of two main operations, that can also be used separately:

1. Calculating the boxplot statistics: *matplotlib.cbook.boxplot_stats*
2. Drawing the boxplot: *matplotlib.axes.Axes.bxp*

Thus, `ax.boxplot(data)` is equivalent to

```
stats = cbook.boxplot_stats(data)
ax.bxp(stats)
```

All styling keyword arguments are identical between *boxplot* and *bxp*, and they are passed through from *boxplot* to *bxp*. However, the *tick_labels* parameter of *boxplot* translates to a generic *labels* parameter in *boxplot_stats*, because the labels are data-related and attached to the returned per-dataset dictionaries.

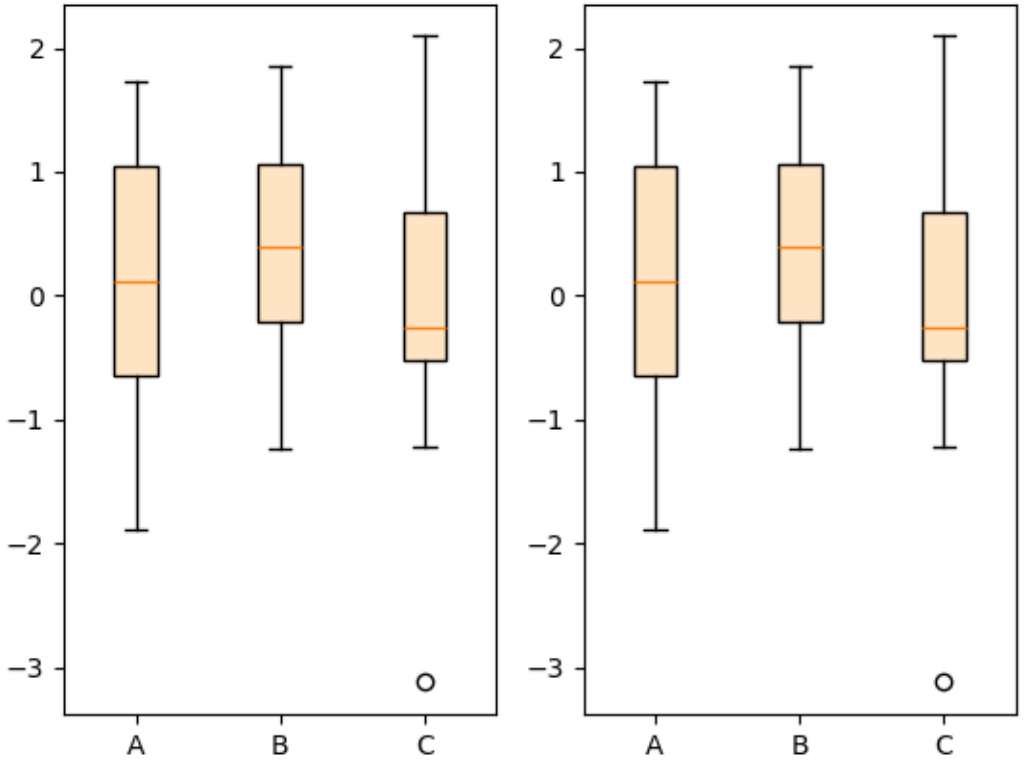
The following code demonstrates the equivalence between the two methods.

```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib import cbook
```

(continues on next page)

(continued from previous page)

```
np.random.seed(19680801)
data = np.random.randn(20, 3)

fig, (ax1, ax2) = plt.subplots(1, 2)

# single boxplot call
ax1.boxplot(data, tick_labels=['A', 'B', 'C'],
            patch_artist=True, boxprops={'facecolor': 'bisque'})

# separate calculation of statistics and plotting
stats = cbook.boxplot_stats(data, labels=['A', 'B', 'C'])
ax2.bxp(stats, patch_artist=True, boxprops={'facecolor': 'bisque'})
```




Using the separate functions allows to pre-calculate statistics, in case you need them explicitly for other purposes, or to reuse the statistics for multiple plots.

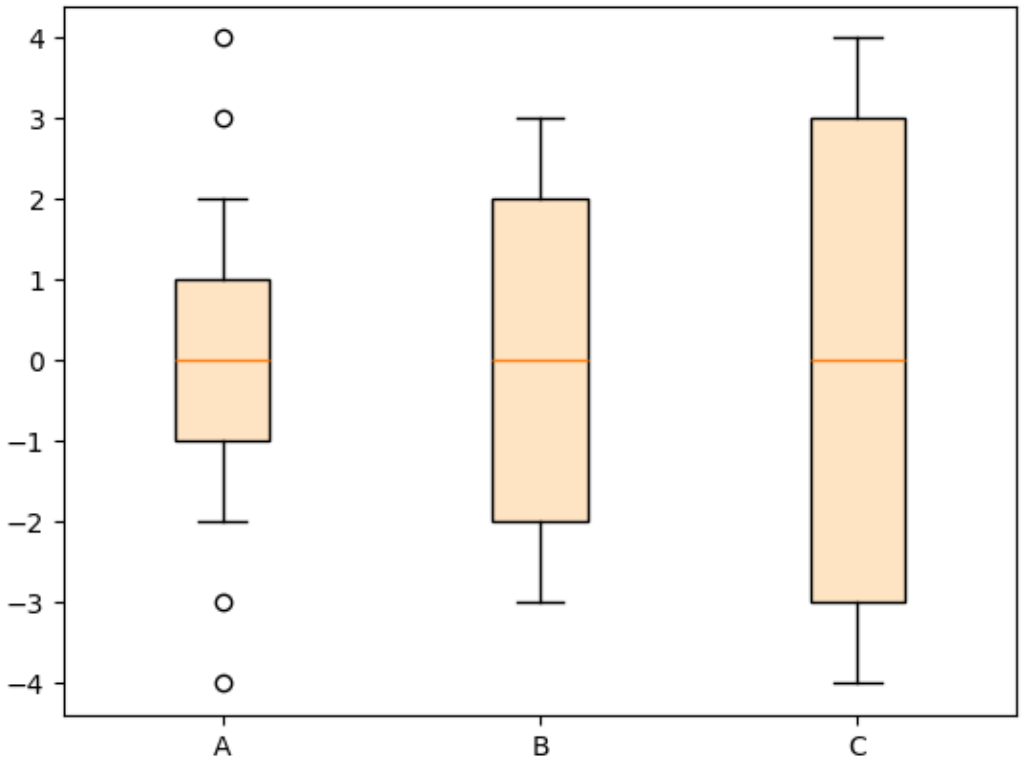
Conversely, you can also use the *bxp* function directly, if you already have the statistical parameters:

```
fig, ax = plt.subplots()

stats = [
    dict(med=0, q1=-1, q3=1, whislo=-2, whishi=2, fliers=[-4, -3, 3, 4],␣
```

(continues on next page)

(continued from previous page)

```
 ↪label='A'),
    dict(med=0, q1=-2, q3=2, whislo=-3, whishi=3, fliers=[], label='B'),
    dict(med=0, q1=-3, q3=3, whislo=-4, whishi=4, fliers=[], label='C'),
]

ax.bxp(stats, patch_artist=True, boxprops={'facecolor': 'bisque'})

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.bxp*
- *matplotlib.axes.Axes.boxplot*
- *matplotlib.cbook.boxplot_stats*

### Plot a confidence ellipse of a two-dimensional dataset

This example shows how to plot a confidence ellipse of a two-dimensional dataset, using its pearson correlation coefficient.

The approach that is used to obtain the correct geometry is explained and proved here:

https://carstenschelp.github.io/2018/09/14/Plot_Confidence_Ellipse_001.html

The method avoids the use of an iterative eigen decomposition algorithm and makes use of the fact that a normalized covariance matrix (composed of pearson correlation coefficients and ones) is particularly easy to handle.

```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.patches import Ellipse
import matplotlib.transforms as transforms
```

### The plotting function itself

This function plots the confidence ellipse of the covariance of the given array-like variables x and y. The ellipse is plotted into the given Axes object *ax*.

The radiuses of the ellipse can be controlled by n_std which is the number of standard deviations. The default value is 3 which makes the ellipse enclose 98.9% of the points if the data is normally distributed like in these examples (3 standard deviations in 1-D contain 99.7% of the data, which is 98.9% of the data in 2-D).

```
def confidence_ellipse(x, y, ax, n_std=3.0, facecolor='none', **kwargs):
    """
    Create a plot of the covariance confidence ellipse of *x* and *y*.

    Parameters
    ----------
    x, y : array-like, shape (n, )
        Input data.

    ax : matplotlib.axes.Axes
        The Axes object to draw the ellipse into.

    n_std : float
        The number of standard deviations to determine the ellipse's radiuses.

    **kwargs
        Forwarded to `~matplotlib.patches.Ellipse`

    Returns
    -------
    matplotlib.patches.Ellipse
    """
    if x.size != y.size:
```

(continues on next page)

(continued from previous page)

```
        raise ValueError("x and y must be the same size")

    cov = np.cov(x, y)
    pearson = cov[0, 1]/np.sqrt(cov[0, 0] * cov[1, 1])
    # Using a special case to obtain the eigenvalues of this
    # two-dimensional dataset.
    ell_radius_x = np.sqrt(1 + pearson)
    ell_radius_y = np.sqrt(1 - pearson)
    ellipse = Ellipse((0, 0), width=ell_radius_x * 2, height=ell_radius_y * 2,
                      facecolor=facecolor, **kwargs)

    # Calculating the standard deviation of x from
    # the squareroot of the variance and multiplying
    # with the given number of standard deviations.
    scale_x = np.sqrt(cov[0, 0]) * n_std
    mean_x = np.mean(x)

    # calculating the standard deviation of y ...
    scale_y = np.sqrt(cov[1, 1]) * n_std
    mean_y = np.mean(y)

    transf = transforms.Affine2D() \
        .rotate_deg(45) \
        .scale(scale_x, scale_y) \
        .translate(mean_x, mean_y)

    ellipse.set_transform(transf + ax.transData)
    return ax.add_patch(ellipse)
```

### A helper function to create a correlated dataset

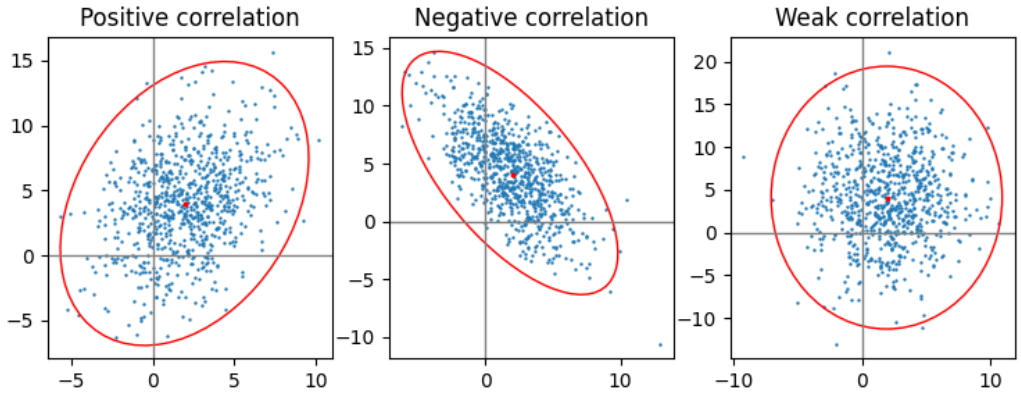
Creates a random two-dimensional dataset with the specified two-dimensional mean (mu) and dimensions (scale). The correlation can be controlled by the param 'dependency', a 2x2 matrix.

```
def get_correlated_dataset(n, dependency, mu, scale):
    latent = np.random.randn(n, 2)
    dependent = latent.dot(dependency)
    scaled = dependent * scale
    scaled_with_offset = scaled + mu
    # return x and y of the new, correlated dataset
    return scaled_with_offset[:, 0], scaled_with_offset[:, 1]
```

### Positive, negative and weak correlation

Note that the shape for the weak correlation (right) is an ellipse, not a circle because x and y are differently scaled. However, the fact that x and y are uncorrelated is shown by the axes of the ellipse being aligned with the x- and y-axis of the coordinate system.

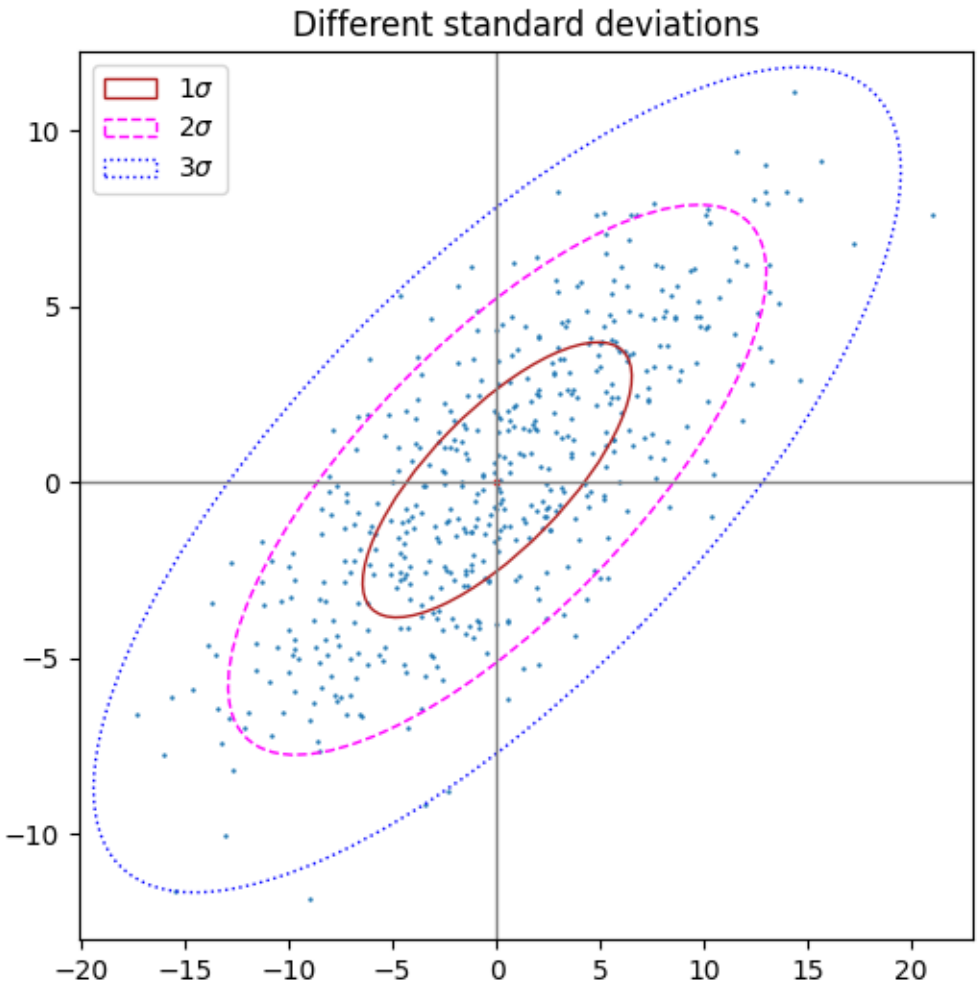
```
np.random.seed(0)

PARAMETERS = {
    'Positive correlation': [[0.85, 0.35],
                             [0.15, -0.65]],
    'Negative correlation': [[0.9, -0.4],
                             [0.1, -0.6]],
    'Weak correlation': [[1, 0],
                         [0, 1]],
}

mu = 2, 4
scale = 3, 5

fig, axs = plt.subplots(1, 3, figsize=(9, 3))
for ax, (title, dependency) in zip(axs, PARAMETERS.items()):
    x, y = get_correlated_dataset(800, dependency, mu, scale)
    ax.scatter(x, y, s=0.5)

    ax.axvline(c='grey', lw=1)
    ax.axhline(c='grey', lw=1)

    confidence_ellipse(x, y, ax, edgecolor='red')

    ax.scatter(mu[0], mu[1], c='red', s=3)
    ax.set_title(title)

plt.show()
```



### Different number of standard deviations

A plot with n_std = 3 (blue), 2 (purple) and 1 (red)

```
fig, ax_nstd = plt.subplots(figsize=(6, 6))

dependency_nstd = [[0.8, 0.75],
                   [-0.2, 0.35]]
mu = 0, 0
scale = 8, 5

ax_nstd.axvline(c='grey', lw=1)
ax_nstd.axhline(c='grey', lw=1)

x, y = get_correlated_dataset(500, dependency_nstd, mu, scale)
ax_nstd.scatter(x, y, s=0.5)

confidence_ellipse(x, y, ax_nstd, n_std=1,
                   label=r'$1\sigma$', edgecolor='firebrick')
confidence_ellipse(x, y, ax_nstd, n_std=2,
                   label=r'$2\sigma$', edgecolor='fuchsia', linestyle='--')
confidence_ellipse(x, y, ax_nstd, n_std=3,
                   label=r'$3\sigma$', edgecolor='blue', linestyle=':')

ax_nstd.scatter(mu[0], mu[1], c='red', s=3)
ax_nstd.set_title('Different standard deviations')
ax_nstd.legend()
plt.show()
```




### Using the keyword arguments

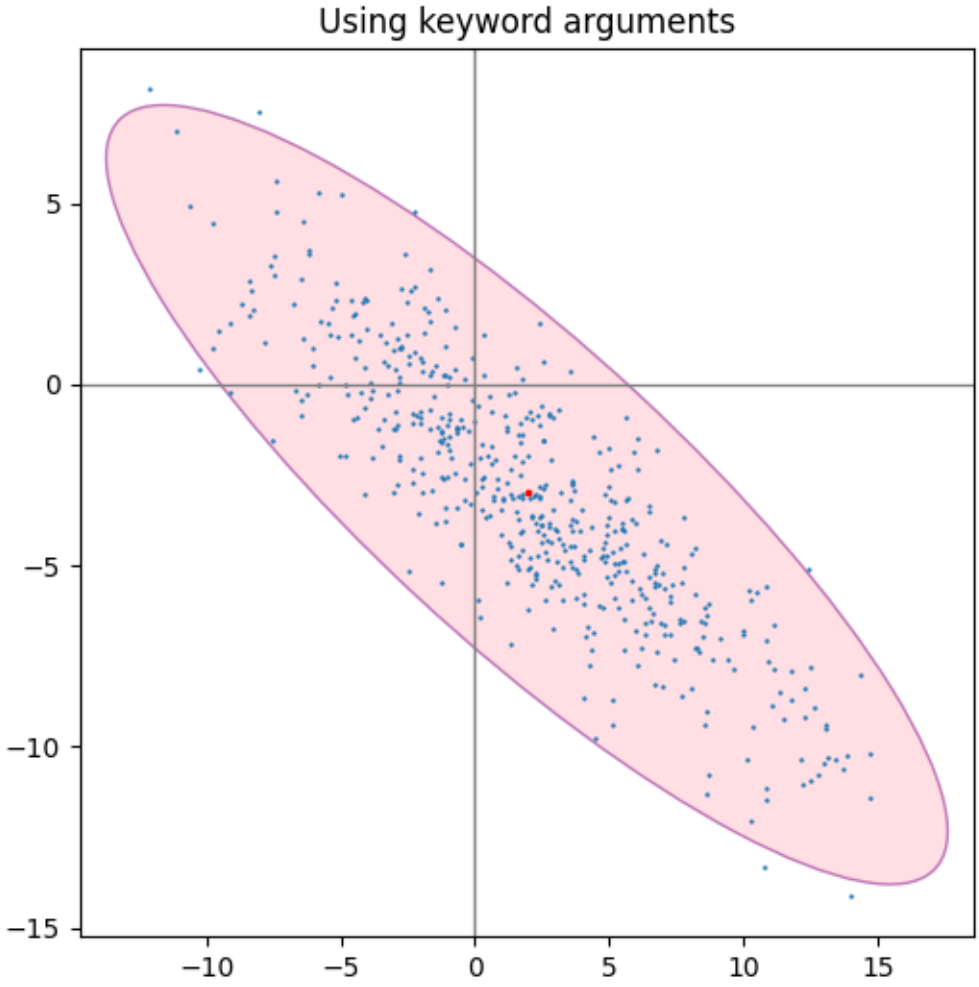
Use the keyword arguments specified for *matplotlib.patches.Patch* in order to have the ellipse rendered in different ways.

```
fig, ax_kwargs = plt.subplots(figsize=(6, 6))
dependency_kwargs = [[-0.8, 0.5],
                     [-0.2, 0.5]]
mu = 2, -3
scale = 6, 5

ax_kwargs.axvline(c='grey', lw=1)
ax_kwargs.axhline(c='grey', lw=1)
```

(continues on next page)

(continued from previous page)

```
x, y = get_correlated_dataset(500, dependency_kwargs, mu, scale)
# Plot the ellipse with zorder=0 in order to demonstrate
# its transparency (caused by the use of alpha).
confidence_ellipse(x, y, ax_kwargs,
                   alpha=0.5, facecolor='pink', edgecolor='purple', zorder=0)

ax_kwargs.scatter(x, y, s=0.5)
ax_kwargs.scatter(mu[0], mu[1], c='red', s=3)
ax_kwargs.set_title('Using keyword arguments')

fig.subplots_adjust(hspace=0.25)
plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.transforms.Affine2D*
- *matplotlib.patches.Ellipse*

**Total running time of the script:** (0 minutes 1.251 seconds)

### Violin plot customization

This example demonstrates how to fully customize violin plots. The first plot shows the default style by providing only the data. The second plot first limits what Matplotlib draws with additional keyword arguments. Then a simplified representation of a box plot is drawn on top. Lastly, the styles of the artists of the violins are modified.

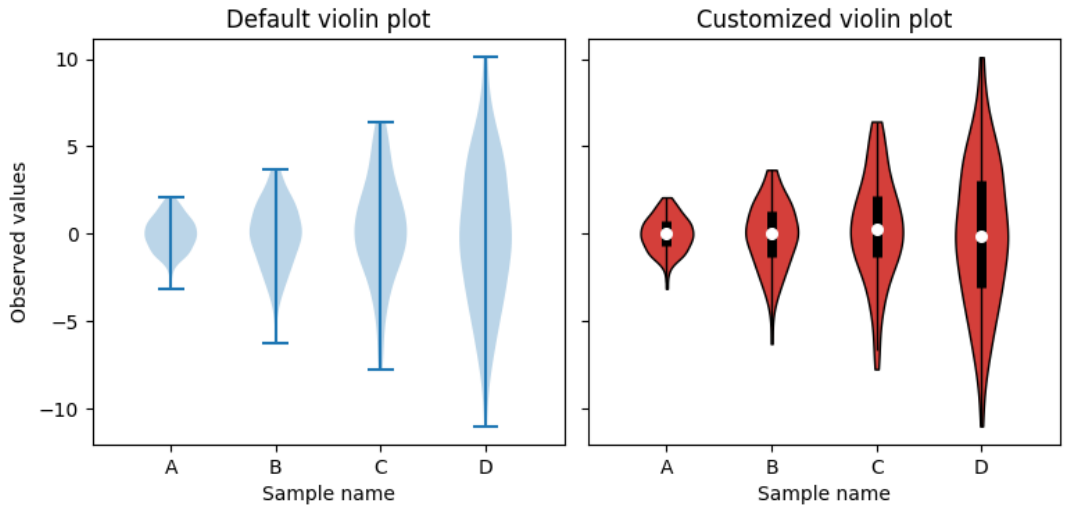
For more information on violin plots, the scikit-learn docs have a great section: https://scikit-learn.org/stable/modules/density.html

```
import matplotlib.pyplot as plt
import numpy as np


def adjacent_values(vals, q1, q3):
    upper_adjacent_value = q3 + (q3 - q1) * 1.5
    upper_adjacent_value = np.clip(upper_adjacent_value, q3, vals[-1])

    lower_adjacent_value = q1 - (q3 - q1) * 1.5
    lower_adjacent_value = np.clip(lower_adjacent_value, vals[0], q1)
    return lower_adjacent_value, upper_adjacent_value


def set_axis_style(ax, labels):
    ax.set_xticks(np.arange(1, len(labels) + 1), labels=labels)
    ax.set_xlim(0.25, len(labels) + 0.75)
    ax.set_xlabel('Sample name')


# create test data
np.random.seed(19680801)
data = [sorted(np.random.normal(0, std, 100)) for std in range(1, 5)]

fig, (ax1, ax2) = plt.subplots(nrows=1, ncols=2, figsize=(9, 4), sharey=True)

ax1.set_title('Default violin plot')
ax1.set_ylabel('Observed values')
ax1.violinplot(data)

ax2.set_title('Customized violin plot')
parts = ax2.violinplot(
        data, showmeans=False, showmedians=False,
        showextrema=False)
```

(continues on next page)

(continued from previous page)

```
for pc in parts['bodies']:
    pc.set_facecolor('#D43F3A')
    pc.set_edgecolor('black')
    pc.set_alpha(1)

quartile1, medians, quartile3 = np.percentile(data, [25, 50, 75], axis=1)
whiskers = np.array([
    adjacent_values(sorted_array, q1, q3)
    for sorted_array, q1, q3 in zip(data, quartile1, quartile3)])
whiskers_min, whiskers_max = whiskers[:, 0], whiskers[:, 1]

inds = np.arange(1, len(medians) + 1)
ax2.scatter(inds, medians, marker='o', color='white', s=30, zorder=3)
ax2.vlines(inds, quartile1, quartile3, color='k', linestyle='-', lw=5)
ax2.vlines(inds, whiskers_min, whiskers_max, color='k', linestyle='-', lw=1)

# set style for the axes
labels = ['A', 'B', 'C', 'D']
for ax in [ax1, ax2]:
    set_axis_style(ax, labels)

plt.subplots_adjust(bottom=0.15, wspace=0.05)
plt.show()
```




**References**

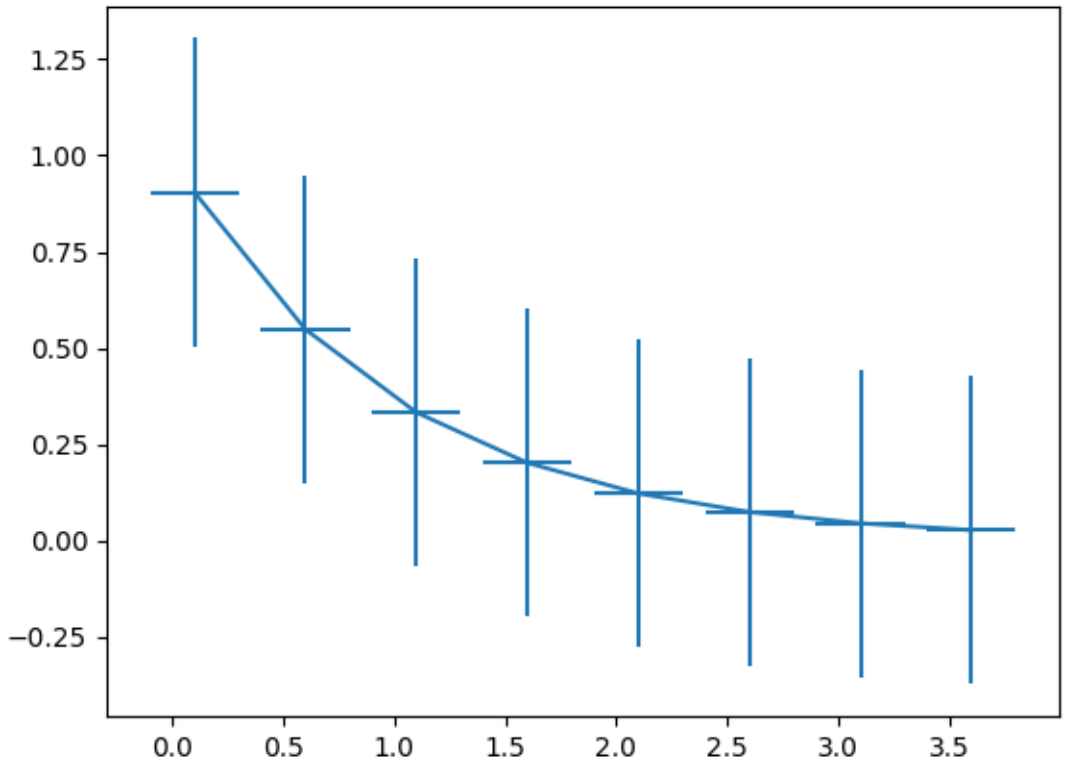
The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.violinplot* / *matplotlib.pyplot.violinplot*
- *matplotlib.axes.Axes.vlines* / *matplotlib.pyplot.vlines*

### Errorbar function

This exhibits the most basic use of the error bar method. In this case, constant values are provided for the error in both the x- and y-directions.

```
import matplotlib.pyplot as plt
import numpy as np

# example data
x = np.arange(0.1, 4, 0.5)
y = np.exp(-x)

fig, ax = plt.subplots()
ax.errorbar(x, y, xerr=0.2, yerr=0.4)
plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.errorbar* / *matplotlib.pyplot.errorbar*

### Different ways of specifying error bars

Errors can be specified as a constant value (as shown in *Errorbar function*). However, this example demonstrates how they vary by specifying arrays of error values.

If the raw `x` and `y` data have length N, there are two options:

**Array of shape (N,):**

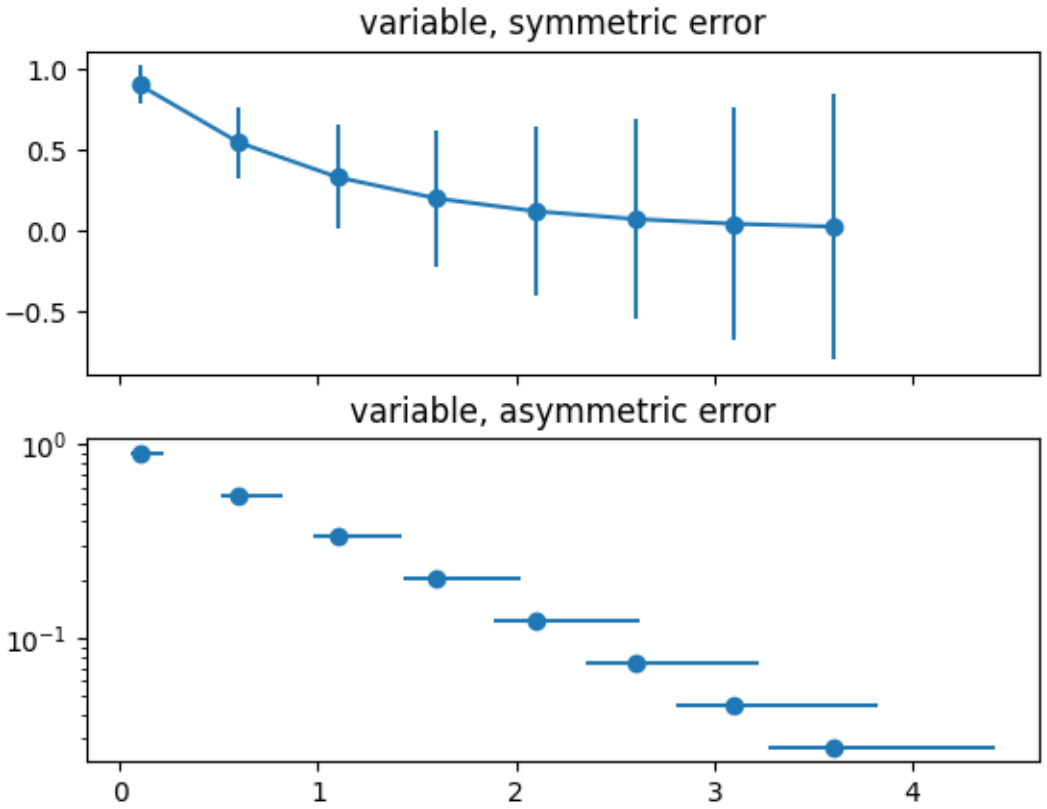
Error varies for each point, but the error values are symmetric (i.e. the lower and upper values are equal).

**Array of shape (2, N):**

Error varies for each point, and the lower and upper limits (in that order) are different (asymmetric case)

In addition, this example demonstrates how to use log scale with error bars.

```
import matplotlib.pyplot as plt
import numpy as np

# example data
x = np.arange(0.1, 4, 0.5)
y = np.exp(-x)

# example error bar values that vary with x-position
error = 0.1 + 0.2 * x

fig, (ax0, ax1) = plt.subplots(nrows=2, sharex=True)
ax0.errorbar(x, y, yerr=error, fmt='-o')
ax0.set_title('variable, symmetric error')

# error bar values w/ different -/+ errors that
# also vary with the x-position
lower_error = 0.4 * error
upper_error = error
asymmetric_error = [lower_error, upper_error]

ax1.errorbar(x, y, xerr=asymmetric_error, fmt='o')
ax1.set_title('variable, asymmetric error')
ax1.set_yscale('log')
plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.errorbar* / *matplotlib.pyplot.errorbar*

## Including upper and lower limits in error bars

In matplotlib, errors bars can have "limits". Applying limits to the error bars essentially makes the error unidirectional. Because of that, upper and lower limits can be applied in both the y- and x-directions via the `uplims`, `lolims`, `xuplims`, and `xlolims` parameters, respectively. These parameters can be scalar or boolean arrays.

For example, if `xlolims` is `True`, the x-error bars will only extend from the data towards increasing values. If `uplims` is an array filled with `False` except for the 4th and 7th values, all of the y-error bars will be bidirectional, except the 4th and 7th bars, which will extend from the data towards decreasing y-values.

```
import matplotlib.pyplot as plt
import numpy as np
```

(continues on next page)

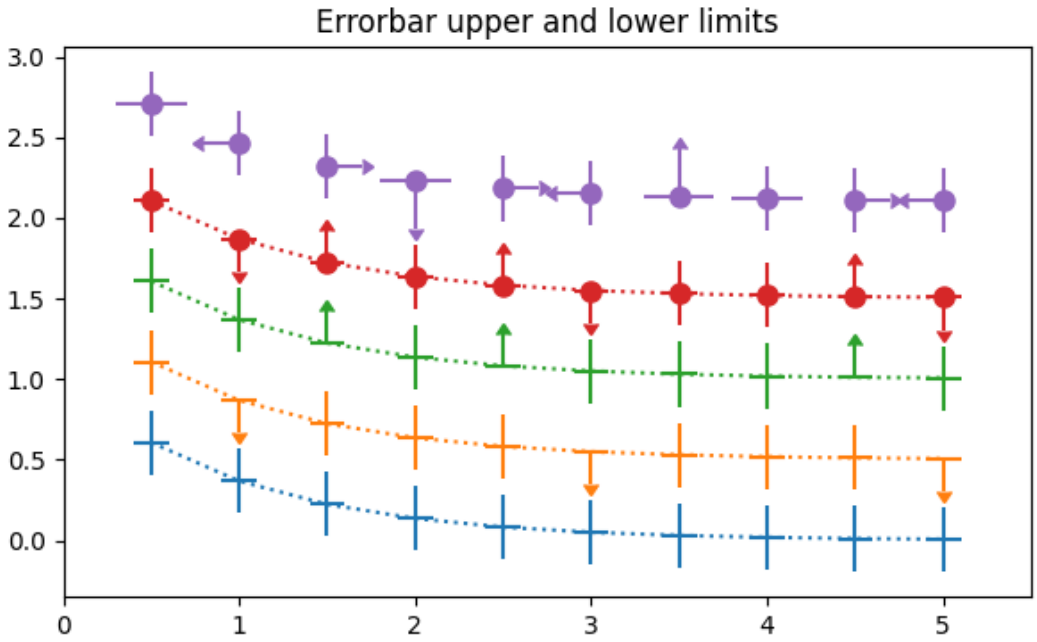
(continued from previous page)

```
# example data
x = np.array([0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0])
y = np.exp(-x)
xerr = 0.1
yerr = 0.2

# lower & upper limits of the error
lolims = np.array([0, 0, 1, 0, 1, 0, 0, 0, 1, 0], dtype=bool)
uplims = np.array([0, 1, 0, 0, 0, 1, 0, 0, 0, 1], dtype=bool)
ls = 'dotted'

fig, ax = plt.subplots(figsize=(7, 4))

# standard error bars
ax.errorbar(x, y, xerr=xerr, yerr=yerr, linestyle=ls)

# including upper limits
ax.errorbar(x, y + 0.5, xerr=xerr, yerr=yerr, uplims=uplims,
            linestyle=ls)

# including lower limits
ax.errorbar(x, y + 1.0, xerr=xerr, yerr=yerr, lolims=lolims,
            linestyle=ls)

# including upper and lower limits
ax.errorbar(x, y + 1.5, xerr=xerr, yerr=yerr,
            lolims=lolims, uplims=uplims,
            marker='o', markersize=8,
            linestyle=ls)

# Plot a series with lower and upper limits in both x & y
# constant x-error with varying y-error
xerr = 0.2
yerr = np.full_like(x, 0.2)
yerr[[3, 6]] = 0.3

# mock up some limits by modifying previous data
xlolims = lolims
xuplims = uplims
lolims = np.zeros_like(x)
uplims = np.zeros_like(x)
lolims[[6]] = True  # only limited at this index
uplims[[3]] = True  # only limited at this index

# do the plotting
ax.errorbar(x, y + 2.1, xerr=xerr, yerr=yerr,
            xlolims=xlolims, xuplims=xuplims,
            uplims=uplims, lolims=lolims,
            marker='o', markersize=8,
            linestyle='none')

# tidy up the figure
```

(continues on next page)

(continued from previous page)

```
ax.set_xlim((0, 5.5))
ax.set_title('Errorbar upper and lower limits')
plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

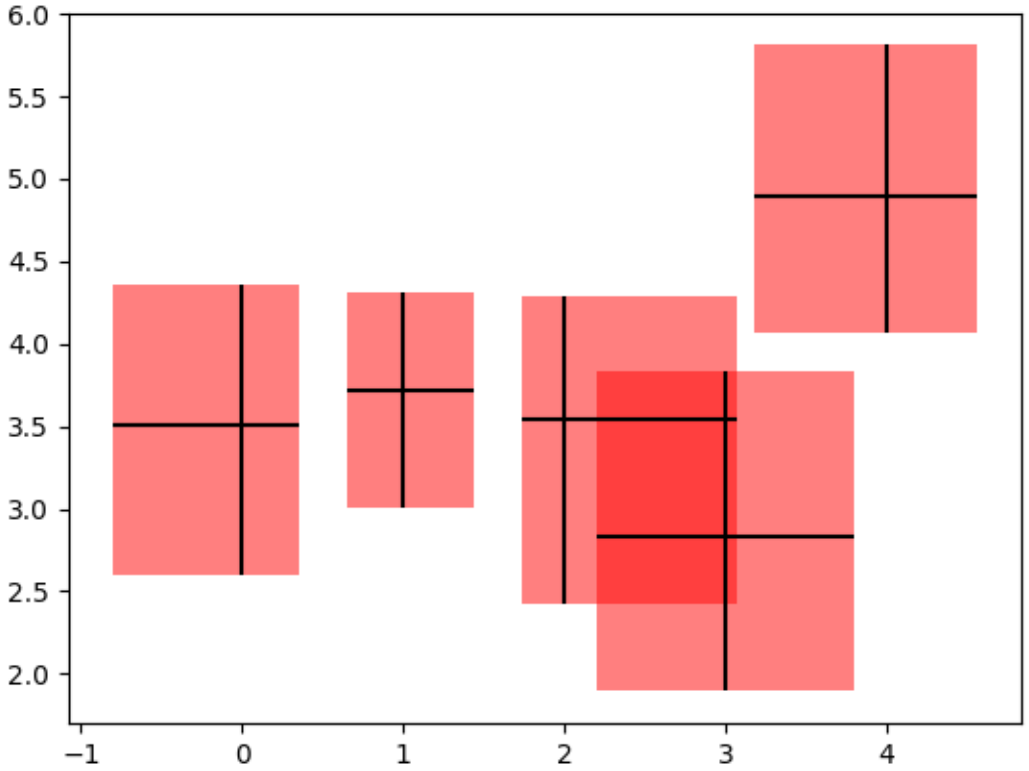
- *matplotlib.axes.Axes.errorbar* / *matplotlib.pyplot.errorbar*

## Creating boxes from error bars using PatchCollection

In this example, we snazz up a pretty standard error bar plot by adding a rectangle patch defined by the limits of the bars in both the x- and y- directions. To do this, we have to write our own custom function called `make_error_boxes`. Close inspection of this function will reveal the preferred pattern in writing functions for matplotlib:

1. an *Axes* object is passed directly to the function
2. the function operates on the `Axes` methods directly, not through the `pyplot` interface
3. plotting keyword arguments that could be abbreviated are spelled out for better code readability in the future (for example we use *facecolor* instead of *fc*)
4. the artists returned by the `Axes` plotting methods are then returned by the function so that, if desired, their styles can be modified later outside of the function (they are not modified in this example).

```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.collections import PatchCollection
from matplotlib.patches import Rectangle

# Number of data points
n = 5

# Dummy data
np.random.seed(19680801)
x = np.arange(0, n, 1)
y = np.random.rand(n) * 5.

# Dummy errors (above and below)
xerr = np.random.rand(2, n) + 0.1
yerr = np.random.rand(2, n) + 0.2


def make_error_boxes(ax, xdata, ydata, xerror, yerror, facecolor='r',
                     edgecolor='none', alpha=0.5):

    # Loop over data points; create box from errors at each point
    errorboxes = [Rectangle((x - xe[0], y - ye[0]), xe.sum(), ye.sum())
                  for x, y, xe, ye in zip(xdata, ydata, xerror.T, yerror.T)]

    # Create patch collection with specified colour/alpha
    pc = PatchCollection(errorboxes, facecolor=facecolor, alpha=alpha,
                         edgecolor=edgecolor)

    # Add collection to Axes
    ax.add_collection(pc)

    # Plot errorbars
    artists = ax.errorbar(xdata, ydata, xerr=xerror, yerr=yerror,
                          fmt='none', ecolor='k')

    return artists


# Create figure and Axes
fig, ax = plt.subplots(1)

# Call function to create error boxes
_ = make_error_boxes(ax, x, y, xerr, yerr)

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.errorbar* / *matplotlib.pyplot.errorbar*
- *matplotlib.axes.Axes.add_collection*
- *matplotlib.collections.PatchCollection*

## Hexagonal binned plot

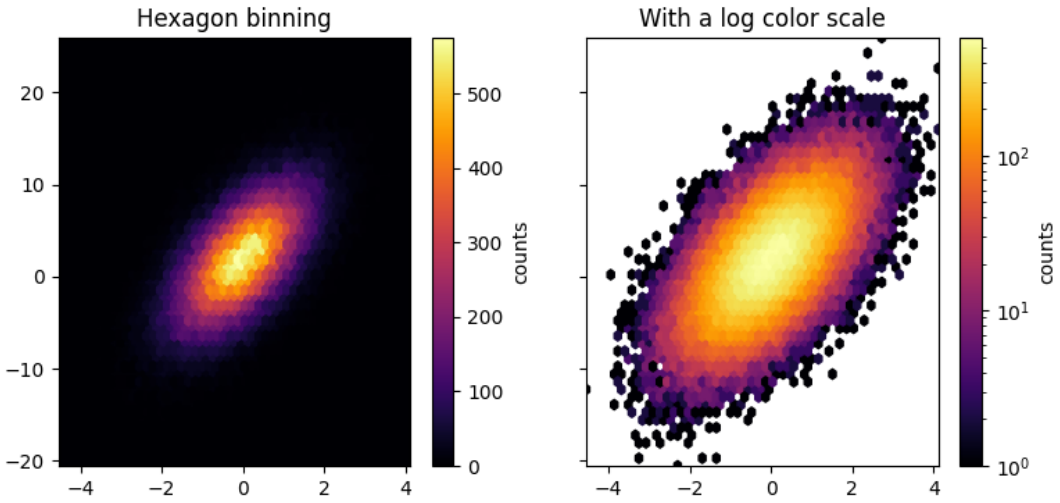
*hexbin* is a 2D histogram plot, in which the bins are hexagons and the color represents the number of data points within each bin.

```
import matplotlib.pyplot as plt
import numpy as np

# Fixing random state for reproducibility
np.random.seed(19680801)
```

(continues on next page)

(continued from previous page)

```
n = 100_000
x = np.random.standard_normal(n)
y = 2.0 + 3.0 * x + 4.0 * np.random.standard_normal(n)
xlim = x.min(), x.max()
ylim = y.min(), y.max()

fig, (ax0, ax1) = plt.subplots(ncols=2, sharey=True, figsize=(9, 4))

hb = ax0.hexbin(x, y, gridsize=50, cmap='inferno')
ax0.set(xlim=xlim, ylim=ylim)
ax0.set_title("Hexagon binning")
cb = fig.colorbar(hb, ax=ax0, label='counts')

hb = ax1.hexbin(x, y, gridsize=50, bins='log', cmap='inferno')
ax1.set(xlim=xlim, ylim=ylim)
ax1.set_title("With a log color scale")
cb = fig.colorbar(hb, ax=ax1, label='counts')

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.hexbin* / *matplotlib.pyplot.hexbin*

### Histograms

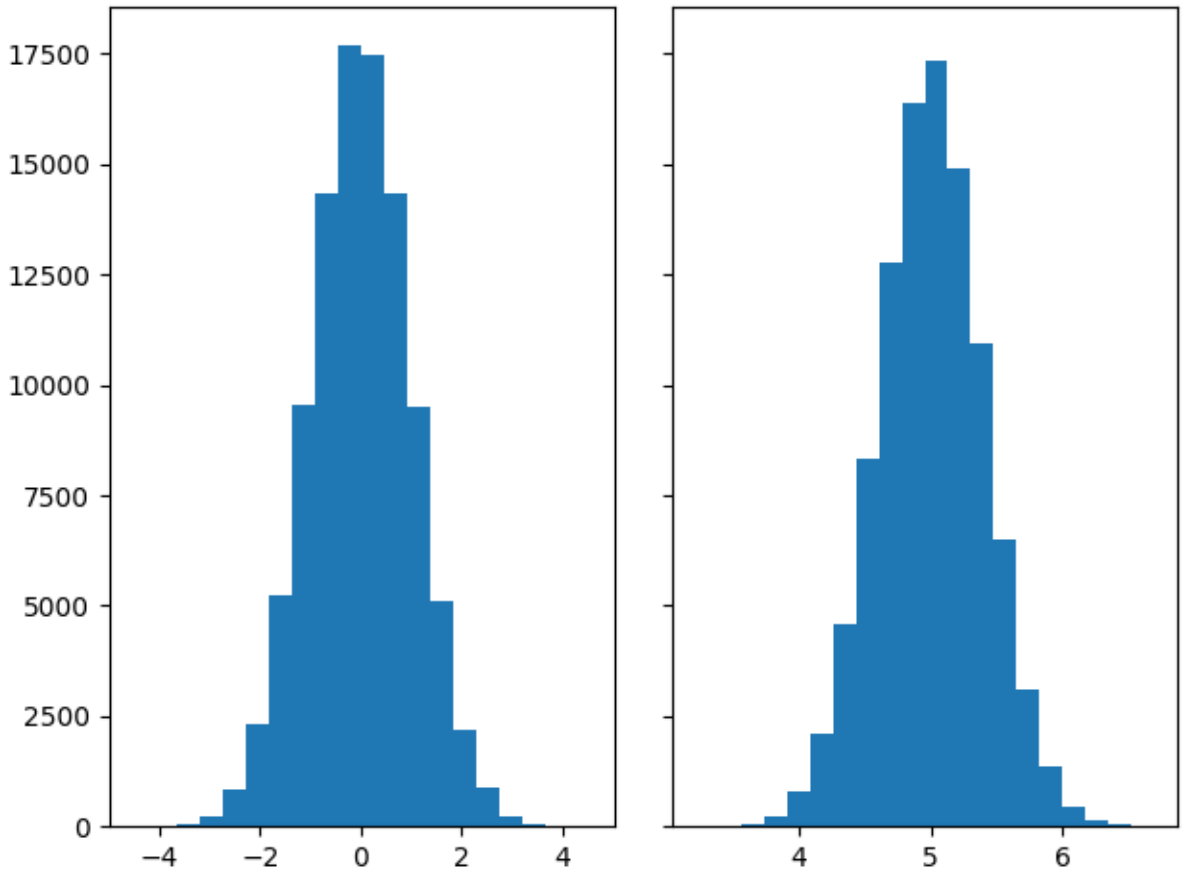
How to plot histograms with Matplotlib.

```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib import colors
from matplotlib.ticker import PercentFormatter

# Create a random number generator with a fixed seed for reproducibility
rng = np.random.default_rng(19680801)
```

### Generate data and plot a simple histogram

To generate a 1D histogram we only need a single vector of numbers. For a 2D histogram we'll need a second vector. We'll generate both below, and show the histogram for each vector.

```
N_points = 100000
n_bins = 20

# Generate two normal distributions
dist1 = rng.standard_normal(N_points)
dist2 = 0.4 * rng.standard_normal(N_points) + 5

fig, axs = plt.subplots(1, 2, sharey=True, tight_layout=True)

# We can set the number of bins with the *bins* keyword argument.
axs[0].hist(dist1, bins=n_bins)
axs[1].hist(dist2, bins=n_bins)

plt.show()
```




### Updating histogram colors

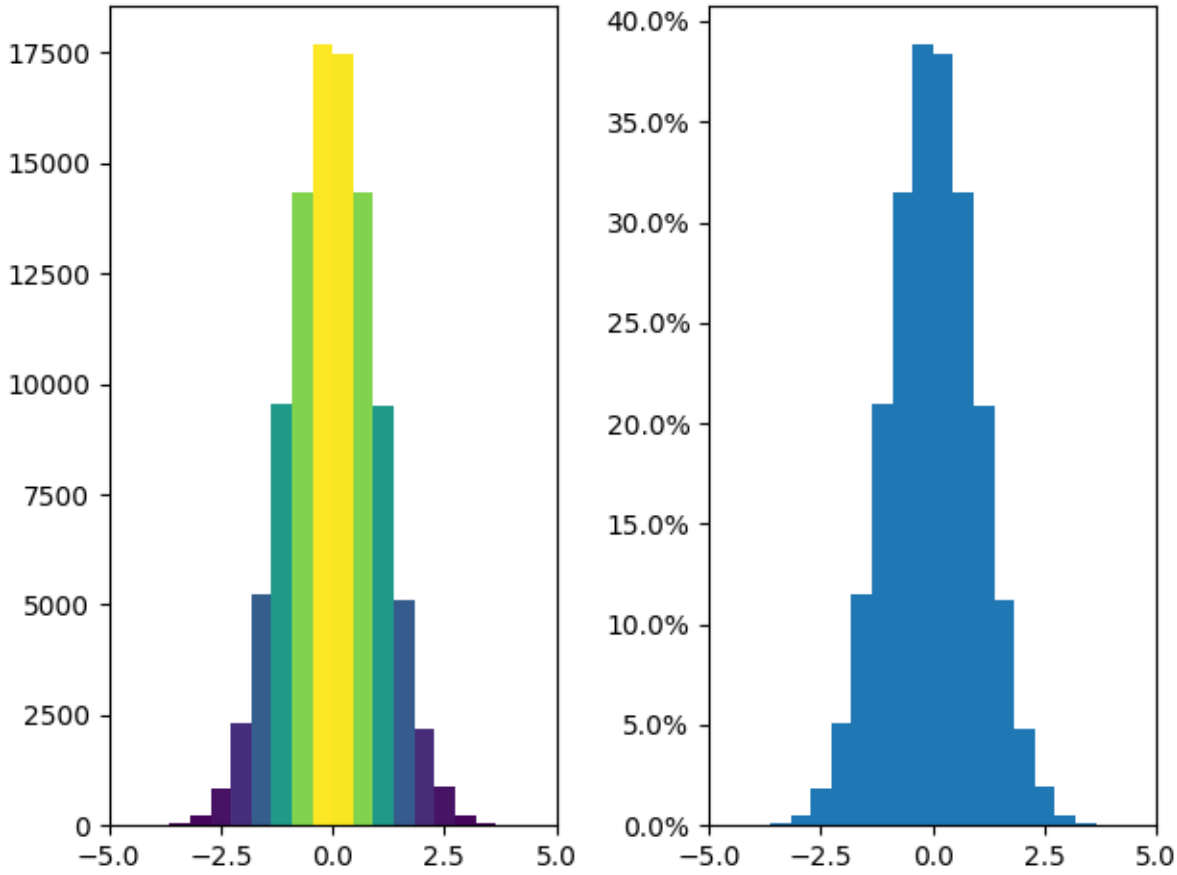
The histogram method returns (among other things) a `patches` object. This gives us access to the properties of the objects drawn. Using this, we can edit the histogram to our liking. Let's change the color of each bar based on its y value.

```
fig, axs = plt.subplots(1, 2, tight_layout=True)

# N is the count in each bin, bins is the lower-limit of the bin
N, bins, patches = axs[0].hist(dist1, bins=n_bins)

# We'll color code by height, but you could use any scalar
fracs = N / N.max()

# we need to normalize the data to 0..1 for the full range of the colormap
norm = colors.Normalize(fracs.min(), fracs.max())

# Now, we'll loop through our objects and set the color of each accordingly
for thisfrac, thispatch in zip(fracs, patches):
    color = plt.cm.viridis(norm(thisfrac))
    thispatch.set_facecolor(color)
```

(continues on next page)

(continued from previous page)

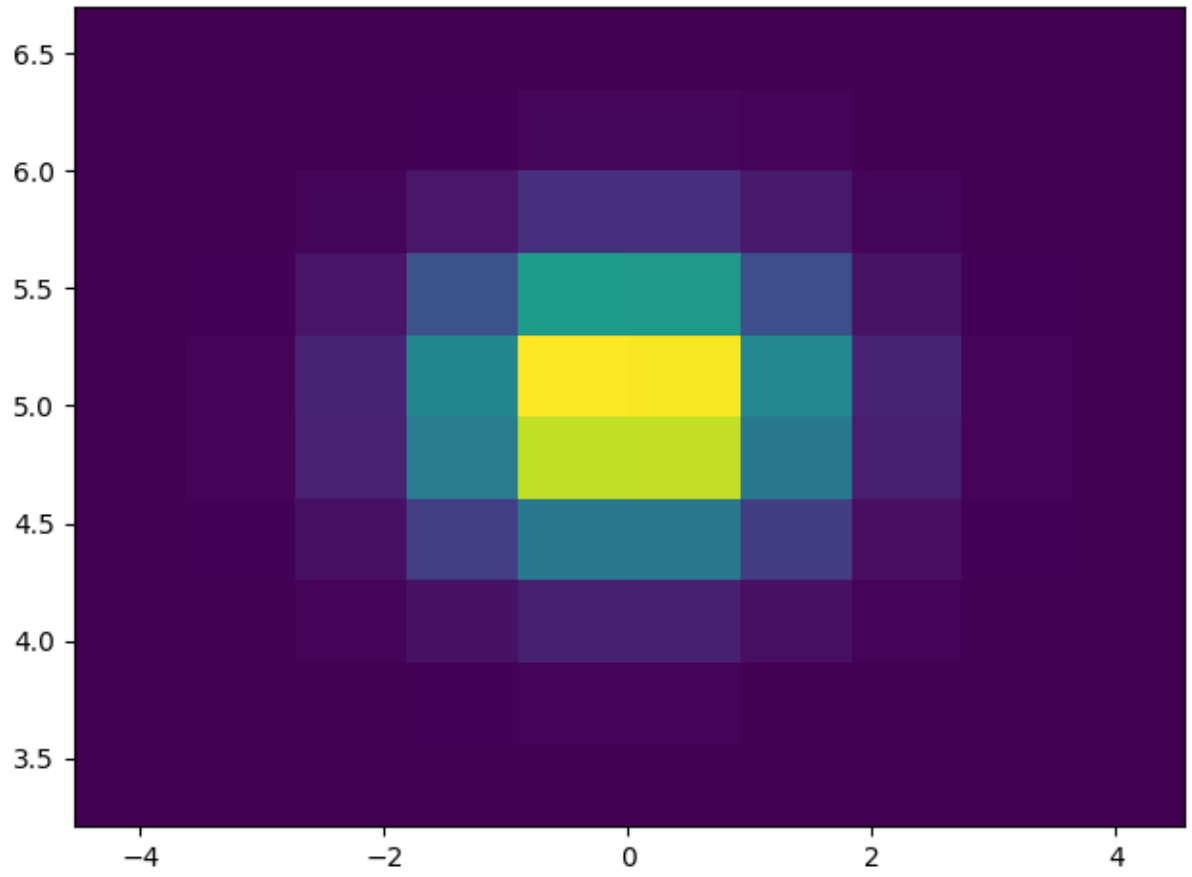
```
# We can also normalize our inputs by the total number of counts
axs[1].hist(dist1, bins=n_bins, density=True)

# Now we format the y-axis to display percentage
axs[1].yaxis.set_major_formatter(PercentFormatter(xmax=1))
```




### Plot a 2D histogram

To plot a 2D histogram, one only needs two vectors of the same length, corresponding to each axis of the histogram.

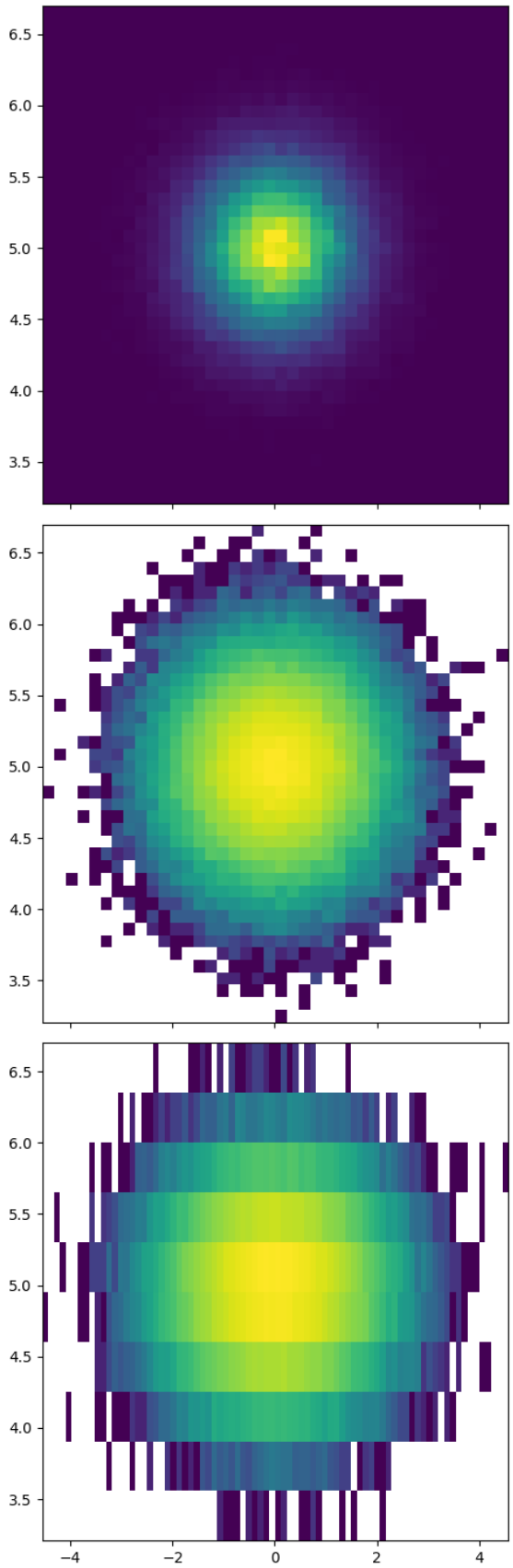
```
fig, ax = plt.subplots(tight_layout=True)
hist = ax.hist2d(dist1, dist2)
```




### Customizing your histogram

Customizing a 2D histogram is similar to the 1D case, you can control visual components such as the bin size or color normalization.

```
fig, axs = plt.subplots(3, 1, figsize=(5, 15), sharex=True, sharey=True,
                        tight_layout=True)

# We can increase the number of bins on each axis
axs[0].hist2d(dist1, dist2, bins=40)

# As well as define normalization of the colors
axs[1].hist2d(dist1, dist2, bins=40, norm=colors.LogNorm())

# We can also define custom numbers of bins for each axis
axs[2].hist2d(dist1, dist2, bins=(80, 10), norm=colors.LogNorm())
```


6.5
6.0
5.5
5.0
4.5
4.0
3.5
6.5
6.0
5.5
5.0
4.5
4.0
3.5
6.5
6.0
5.5
5.0
4.5
4.0
3.5
−4
−2
0
2
4

**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.hist* / *matplotlib.pyplot.hist*
- *matplotlib.pyplot.hist2d*
- *matplotlib.ticker.PercentFormatter*

**Total running time of the script:** (0 minutes 1.603 seconds)

## Bihistogram

How to plot a bihistogram with Matplotlib.

```
import matplotlib.pyplot as plt
import numpy as np

# Create a random number generator with a fixed seed for reproducibility
rng = np.random.default_rng(19680801)
```

## Generate data and plot a bihistogram

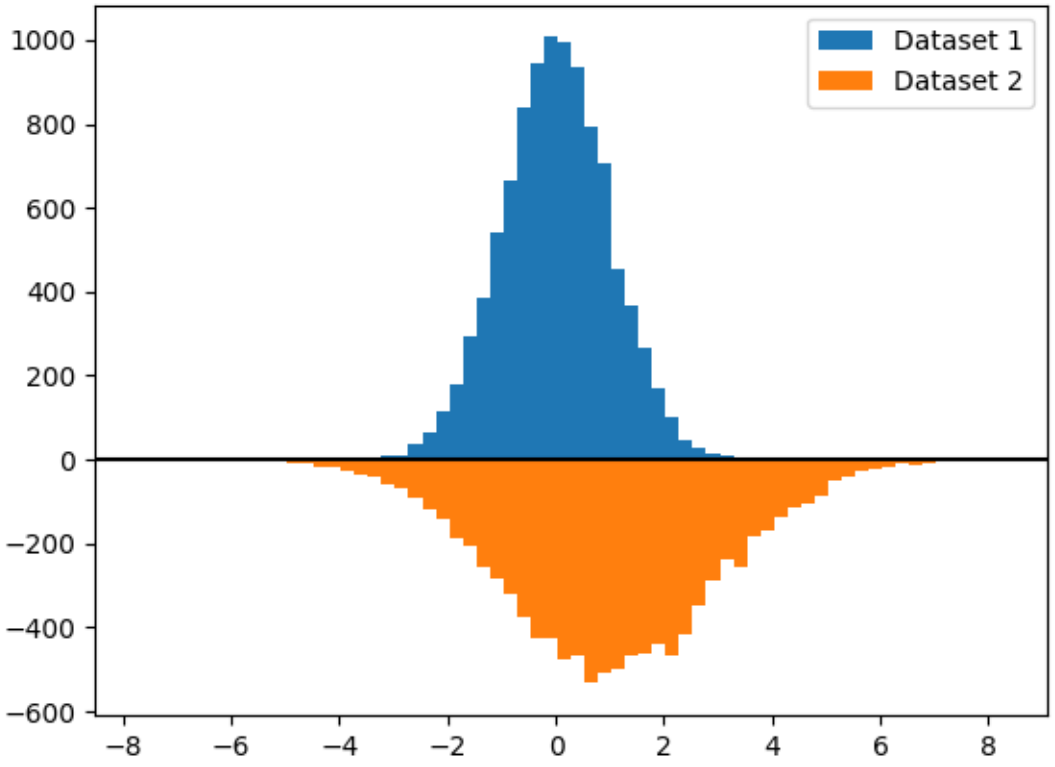
To generate a bihistogram we need two datasets (each being a vector of numbers). We will plot both histograms using plt.hist() and set the weights of the second one to be negative. We'll generate data below and plot the bihistogram.

```
N_points = 10_000

# Generate two normal distributions
dataset1 = np.random.normal(0, 1, size=N_points)
dataset2 = np.random.normal(1, 2, size=N_points)

# Use a constant bin width to make the two histograms easier to compare␣
 ↪visually
bin_width = 0.25
bins = np.arange(np.min([dataset1, dataset2]),
                 np.max([dataset1, dataset2]) + bin_width, bin_width)

fig, ax = plt.subplots()

# Plot the first histogram
ax.hist(dataset1, bins=bins, label="Dataset 1")

# Plot the second histogram
# (notice the negative weights, which flip the histogram upside down)
ax.hist(dataset2, weights=-np.ones_like(dataset2), bins=bins, label="Dataset 2
 ↪")
```

(continues on next page)

(continued from previous page)

```
ax.axhline(0, color="k")
ax.legend()

plt.show()
```




## Plotting cumulative distributions

This example shows how to plot the empirical cumulative distribution function (ECDF) of a sample. We also show the theoretical CDF.

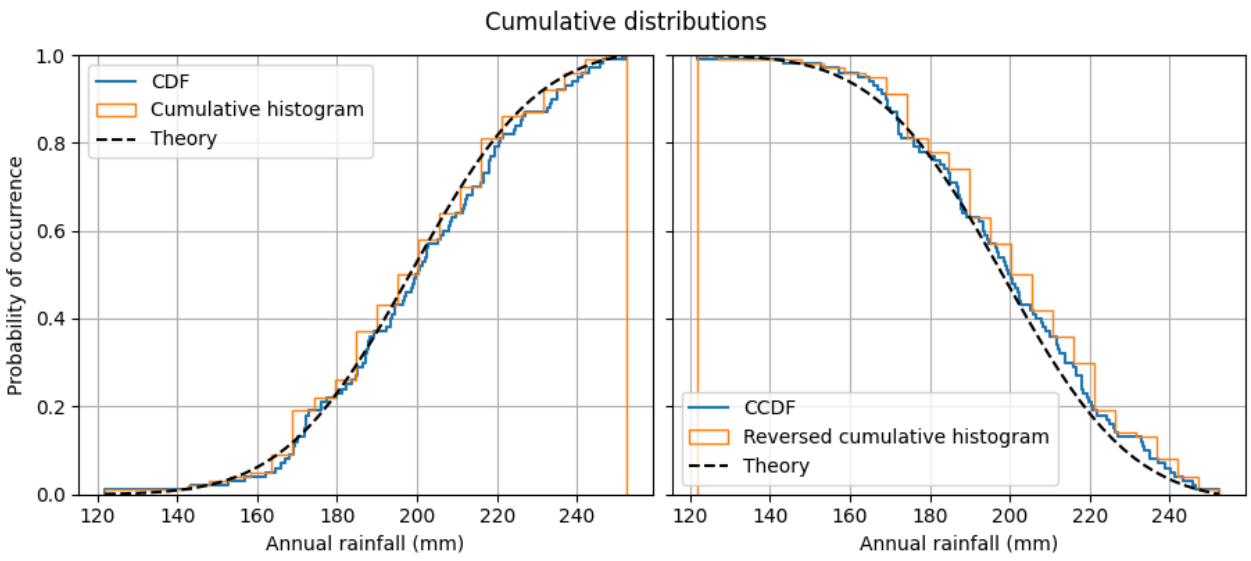
In engineering, ECDFs are sometimes called "non-exceedance" curves: the y-value for a given x-value gives probability that an observation from the sample is below that x-value. For example, the value of 220 on the x-axis corresponds to about 0.80 on the y-axis, so there is an 80% chance that an observation in the sample does not exceed 220. Conversely, the empirical *complementary* cumulative distribution function (the ECCDF, or "exceedance" curve) shows the probability y that an observation from the sample is above a value x.

A direct method to plot ECDFs is `Axes.ecdf`. Passing `complementary=True` results in an ECCDF instead.

Alternatively, one can use `ax.hist(data, density=True, cumulative=True)` to first bin the data, as if plotting a histogram, and then compute and plot the cumulative sums of the frequencies of en-

tries in each bin. Here, to plot the ECCDF, pass `cumulative=-1`. Note that this approach results in an approximation of the E(C)CDF, whereas *`Axes.ecdf`* is exact.

```
import matplotlib.pyplot as plt
import numpy as np

np.random.seed(19680801)

mu = 200
sigma = 25
n_bins = 25
data = np.random.normal(mu, sigma, size=100)

fig = plt.figure(figsize=(9, 4), layout="constrained")
axs = fig.subplots(1, 2, sharex=True, sharey=True)

# Cumulative distributions.
axs[0].ecdf(data, label="CDF")
n, bins, patches = axs[0].hist(data, n_bins, density=True, histtype="step",
                               cumulative=True, label="Cumulative histogram")
x = np.linspace(data.min(), data.max())
y = ((1 / (np.sqrt(2 * np.pi) * sigma)) *
     np.exp(-0.5 * (1 / sigma * (x - mu))**2))
y = y.cumsum()
y /= y[-1]
axs[0].plot(x, y, "k--", linewidth=1.5, label="Theory")

# Complementary cumulative distributions.
axs[1].ecdf(data, complementary=True, label="CCDF")
axs[1].hist(data, bins=bins, density=True, histtype="step", cumulative=-1,
            label="Reversed cumulative histogram")
axs[1].plot(x, 1 - y, "k--", linewidth=1.5, label="Theory")

# Label the figure.
fig.suptitle("Cumulative distributions")
for ax in axs:
    ax.grid(True)
    ax.legend()
    ax.set_xlabel("Annual rainfall (mm)")
    ax.set_ylabel("Probability of occurrence")
    ax.label_outer()

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.hist* / *matplotlib.pyplot.hist*
- *matplotlib.axes.Axes.ecdf* / *matplotlib.pyplot.ecdf*

## Demo of the histogram function's different `histtype` settings

- Histogram with step curve that has a color fill.
- Histogram with step curve with no fill.
- Histogram with custom and unequal bin widths.
- Two histograms with stacked bars.

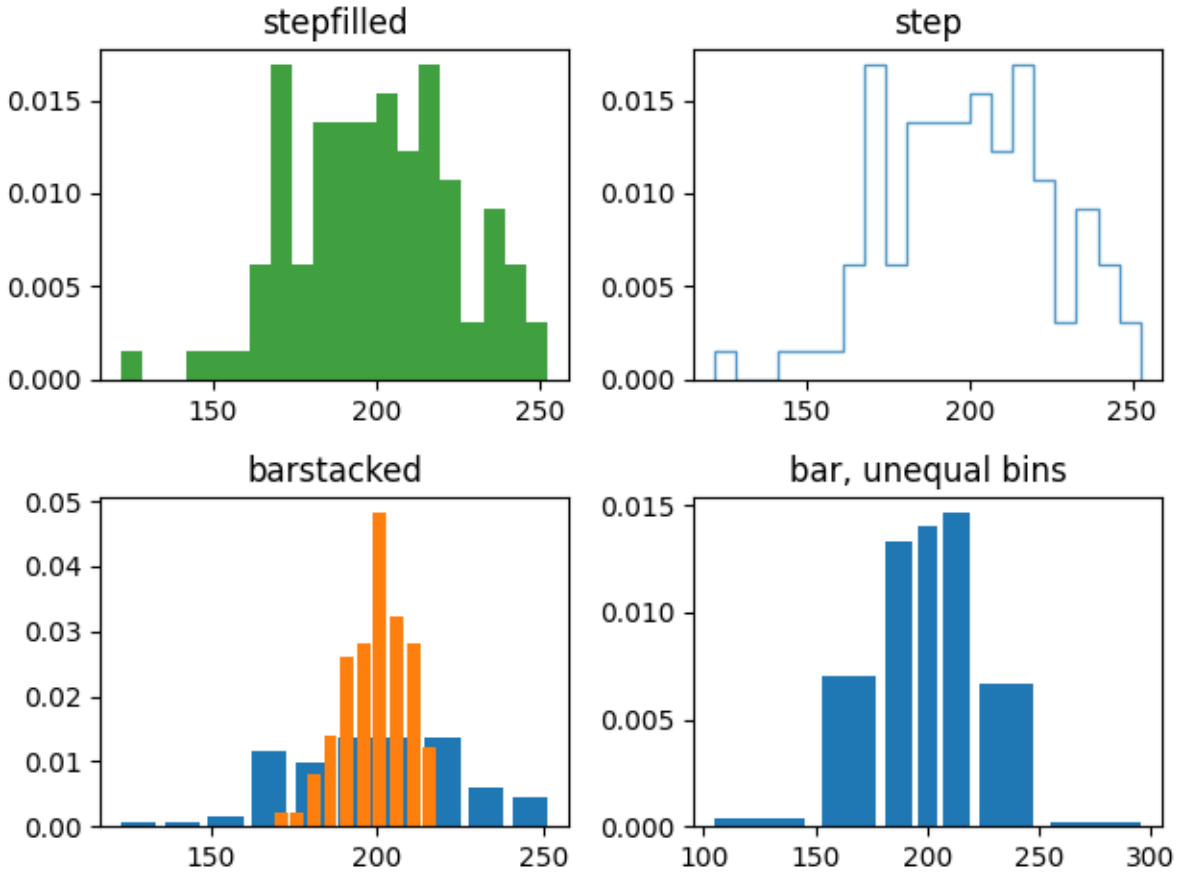
Selecting different bin counts and sizes can significantly affect the shape of a histogram. The Astropy docs have a great section on how to select these parameters: http://docs.astropy.org/en/stable/visualization/histogram.html

```
import matplotlib.pyplot as plt
import numpy as np

np.random.seed(19680801)

mu_x = 200
sigma_x = 25
x = np.random.normal(mu_x, sigma_x, size=100)

mu_w = 200
sigma_w = 10
```

(continues on next page)

(continued from previous page)

```
w = np.random.normal(mu_w, sigma_w, size=100)

fig, axs = plt.subplots(nrows=2, ncols=2)

axs[0, 0].hist(x, 20, density=True, histtype='stepfilled', facecolor='g',
               alpha=0.75)
axs[0, 0].set_title('stepfilled')

axs[0, 1].hist(x, 20, density=True, histtype='step', facecolor='g',
               alpha=0.75)
axs[0, 1].set_title('step')

axs[1, 0].hist(x, density=True, histtype='barstacked', rwidth=0.8)
axs[1, 0].hist(w, density=True, histtype='barstacked', rwidth=0.8)
axs[1, 0].set_title('barstacked')

# Create a histogram by providing the bin edges (unequally spaced).
bins = [100, 150, 180, 195, 205, 220, 250, 300]
axs[1, 1].hist(x, bins, density=True, histtype='bar', rwidth=0.8)
axs[1, 1].set_title('bar, unequal bins')

fig.tight_layout()
plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.hist* / *matplotlib.pyplot.hist*

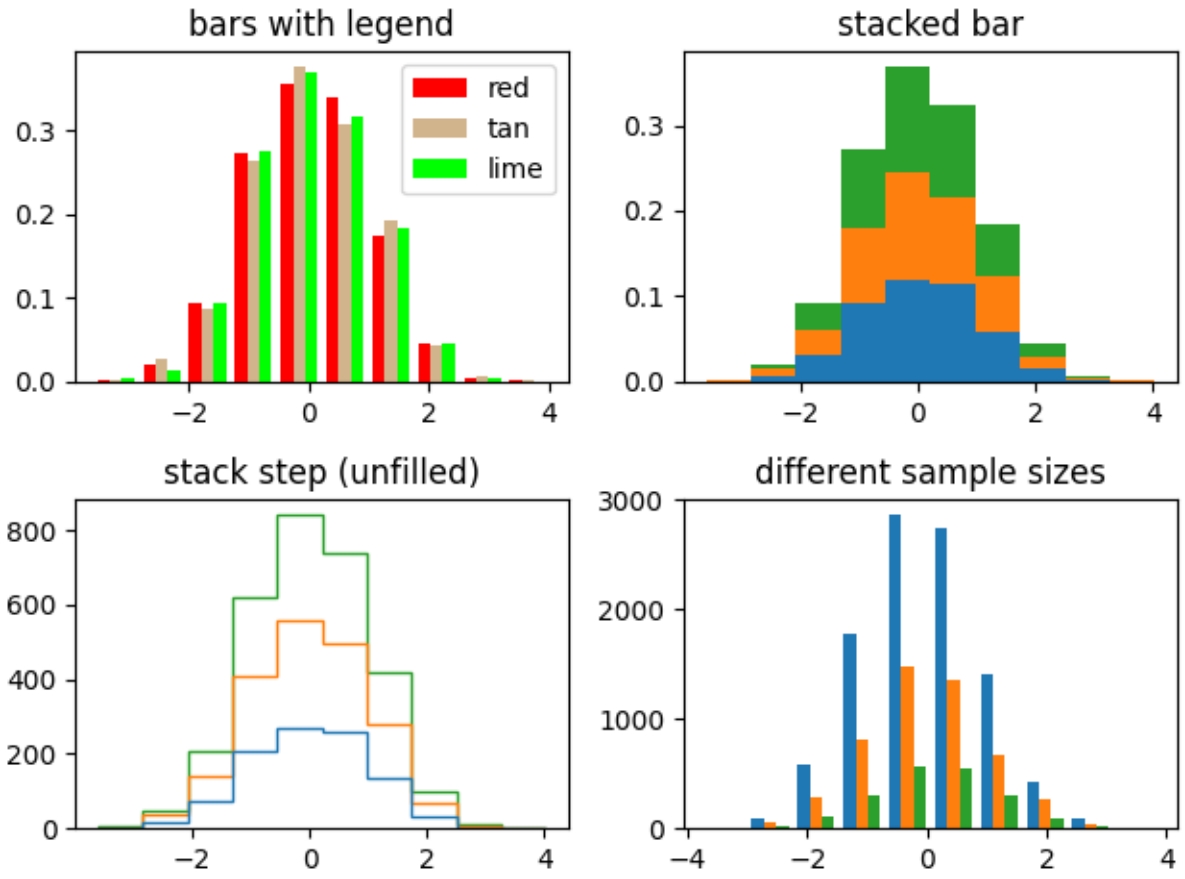
### The histogram (hist) function with multiple data sets

Plot histogram with multiple sample sets and demonstrate:

- Use of legend with multiple sample sets
- Stacked bars
- Step curve with no fill
- Data sets of different sample sizes

Selecting different bin counts and sizes can significantly affect the shape of a histogram. The Astropy docs have a great section on how to select these parameters: http://docs.astropy.org/en/stable/visualization/histogram.html

```
import matplotlib.pyplot as plt
import numpy as np

np.random.seed(19680801)

n_bins = 10
x = np.random.randn(1000, 3)

fig, ((ax0, ax1), (ax2, ax3)) = plt.subplots(nrows=2, ncols=2)

colors = ['red', 'tan', 'lime']
ax0.hist(x, n_bins, density=True, histtype='bar', color=colors, label=colors)
ax0.legend(prop={'size': 10})
ax0.set_title('bars with legend')

ax1.hist(x, n_bins, density=True, histtype='bar', stacked=True)
ax1.set_title('stacked bar')

ax2.hist(x, n_bins, histtype='step', stacked=True, fill=False)
ax2.set_title('stack step (unfilled)')

# Make a multiple-histogram of data-sets with different length.
x_multi = [np.random.randn(n) for n in [10000, 5000, 2000]]
ax3.hist(x_multi, n_bins, histtype='bar')
ax3.set_title('different sample sizes')

fig.tight_layout()
plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.hist* / *matplotlib.pyplot.hist*

### Histogram bins, density, and weight

The *Axes.hist* method can flexibly create histograms in a few different ways, which is flexible and helpful, but can also lead to confusion. In particular, you can:

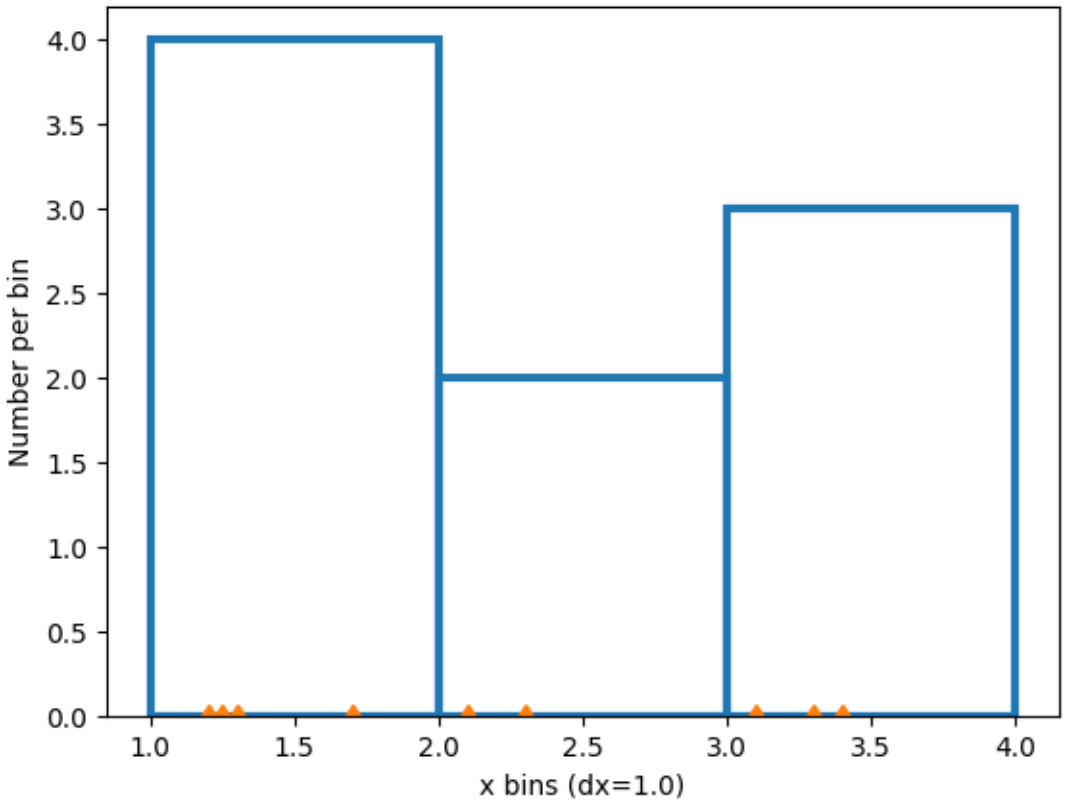
- bin the data as you want, either with an automatically chosen number of bins, or with fixed bin edges,
- normalize the histogram so that its integral is one,
- and assign weights to the data points, so that each data point affects the count in its bin differently.

The Matplotlib `hist` method calls `numpy.histogram` and plots the results, therefore users should consult the numpy documentation for a definitive guide.

Histograms are created by defining bin edges, and taking a dataset of values and sorting them into the bins, and counting or summing how much data is in each bin. In this simple example, 9 numbers between 1 and

4 are sorted into 3 bins:

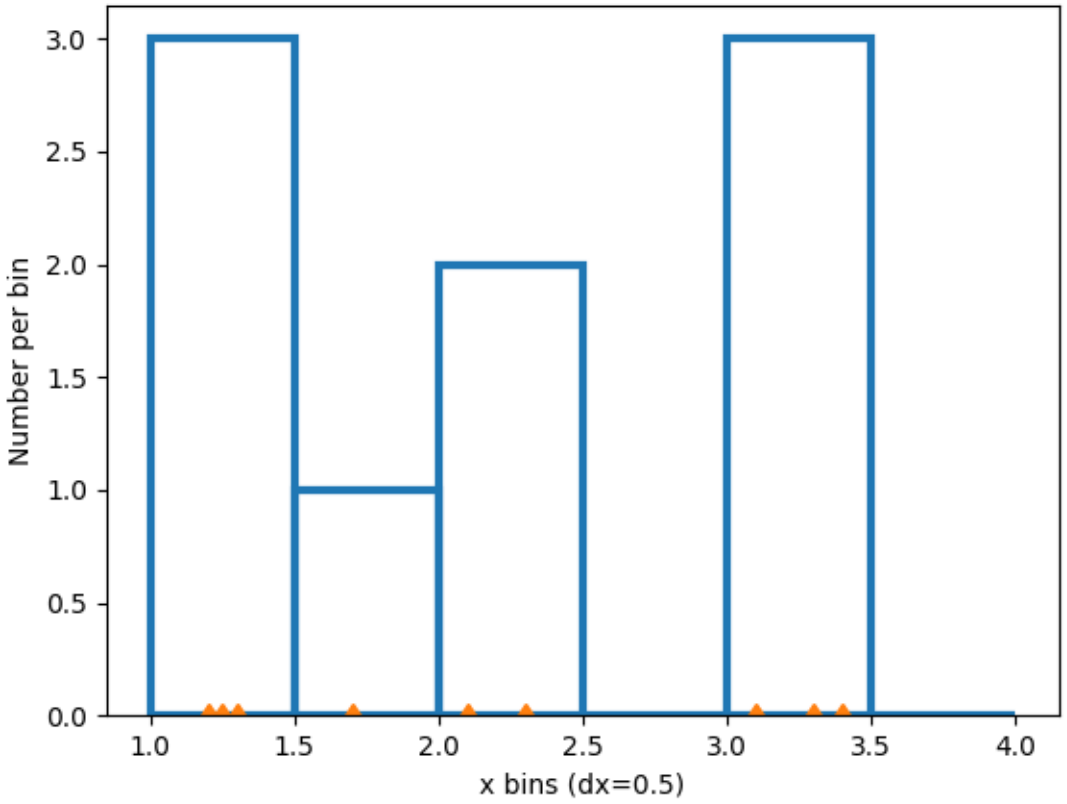
```
import matplotlib.pyplot as plt
import numpy as np

rng = np.random.default_rng(19680801)

xdata = np.array([1.2, 2.3, 3.3, 3.1, 1.7, 3.4, 2.1, 1.25, 1.3])
xbins = np.array([1, 2, 3, 4])

# changing the style of the histogram bars just to make it
# very clear where the boundaries of the bins are:
style = {'facecolor': 'none', 'edgecolor': 'C0', 'linewidth': 3}

fig, ax = plt.subplots()
ax.hist(xdata, bins=xbins, **style)

# plot the xdata locations on the x axis:
ax.plot(xdata, 0*xdata, 'd')
ax.set_ylabel('Number per bin')
ax.set_xlabel('x bins (dx=1.0)')
```



### Modifying bins

Changing the bin size changes the shape of this sparse histogram, so its a good idea to choose bins with some care with respect to your data. Here we make the bins half as wide.

```
xbins = np.arange(1, 4.5, 0.5)

fig, ax = plt.subplots()
ax.hist(xdata, bins=xbins, **style)
ax.plot(xdata, 0*xdata, 'd')
ax.set_ylabel('Number per bin')
ax.set_xlabel('x bins (dx=0.5)')
```



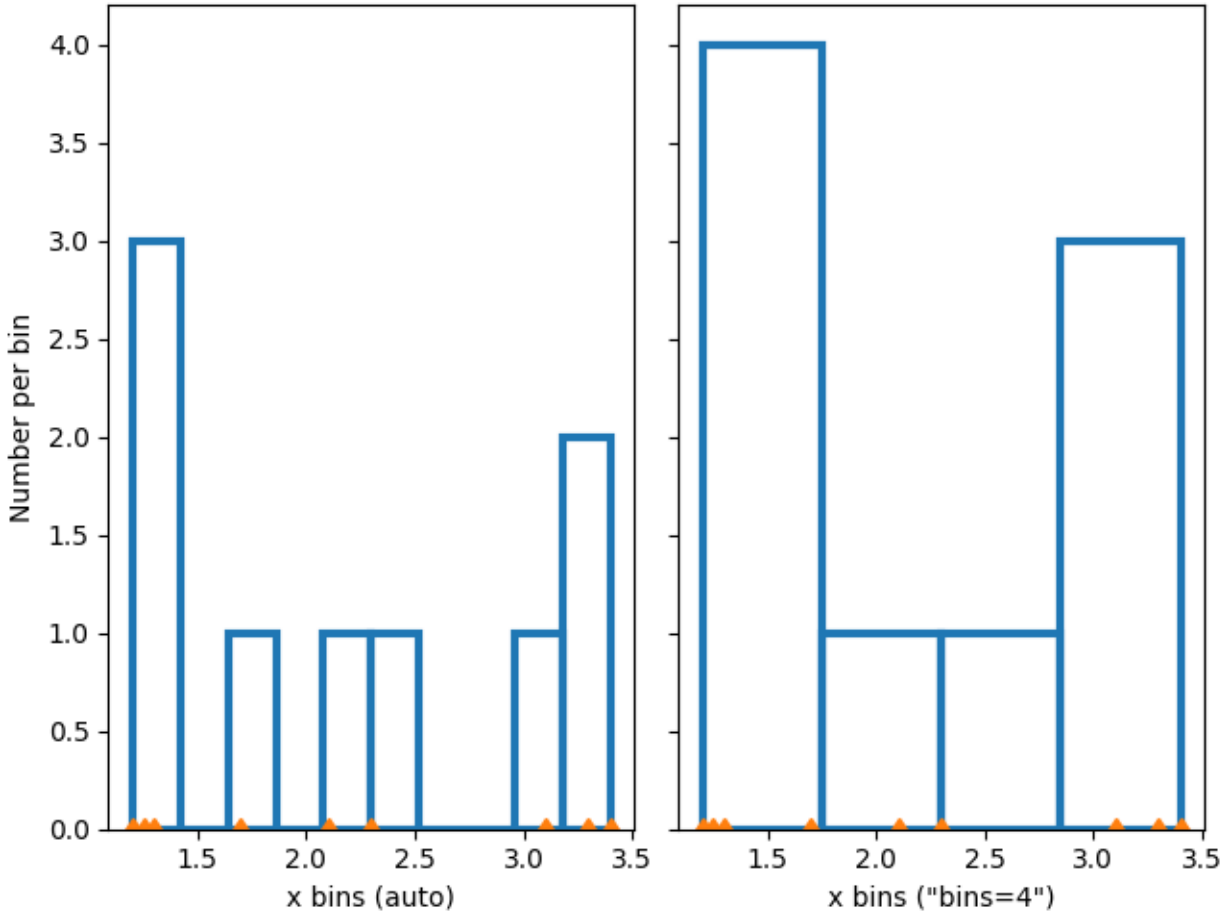

We can also let numpy (via Matplotlib) choose the bins automatically, or specify a number of bins to choose automatically:

```
fig, ax = plt.subplot_mosaic([['auto', 'n4']],
                             sharex=True, sharey=True, layout='constrained')

ax['auto'].hist(xdata, **style)
ax['auto'].plot(xdata, 0*xdata, 'd')
ax['auto'].set_ylabel('Number per bin')
ax['auto'].set_xlabel('x bins (auto)')
```

(continues on next page)

(continued from previous page)

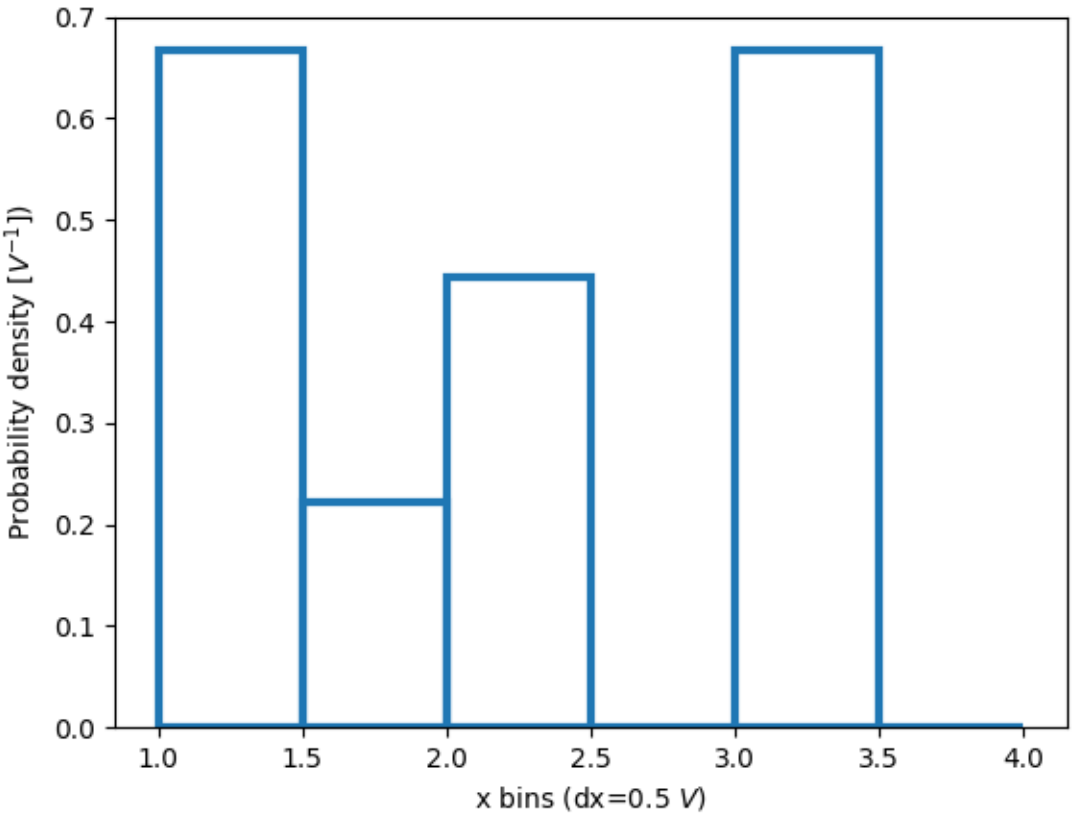
```
ax['n4'].hist(xdata, bins=4, **style)
ax['n4'].plot(xdata, 0*xdata, 'd')
ax['n4'].set_xlabel('x bins ("bins=4")')
```




### Normalizing histograms: density and weight

Counts-per-bin is the default length of each bar in the histogram. However, we can also normalize the bar lengths as a probability density function using the `density` parameter:

```
fig, ax = plt.subplots()
ax.hist(xdata, bins=xbins, density=True, **style)
ax.set_ylabel('Probability density [$V^{-1}$])')
ax.set_xlabel('x bins (dx=0.5 $V$)')
```




This normalization can be a little hard to interpret when just exploring the data. The value attached to each bar is divided by the total number of data points *and* the width of the bin, and thus the values _integrate_ to one when integrating across the full range of data. e.g.

```
density = counts / (sum(counts) * np.diff(bins))
np.sum(density * np.diff(bins)) == 1
```

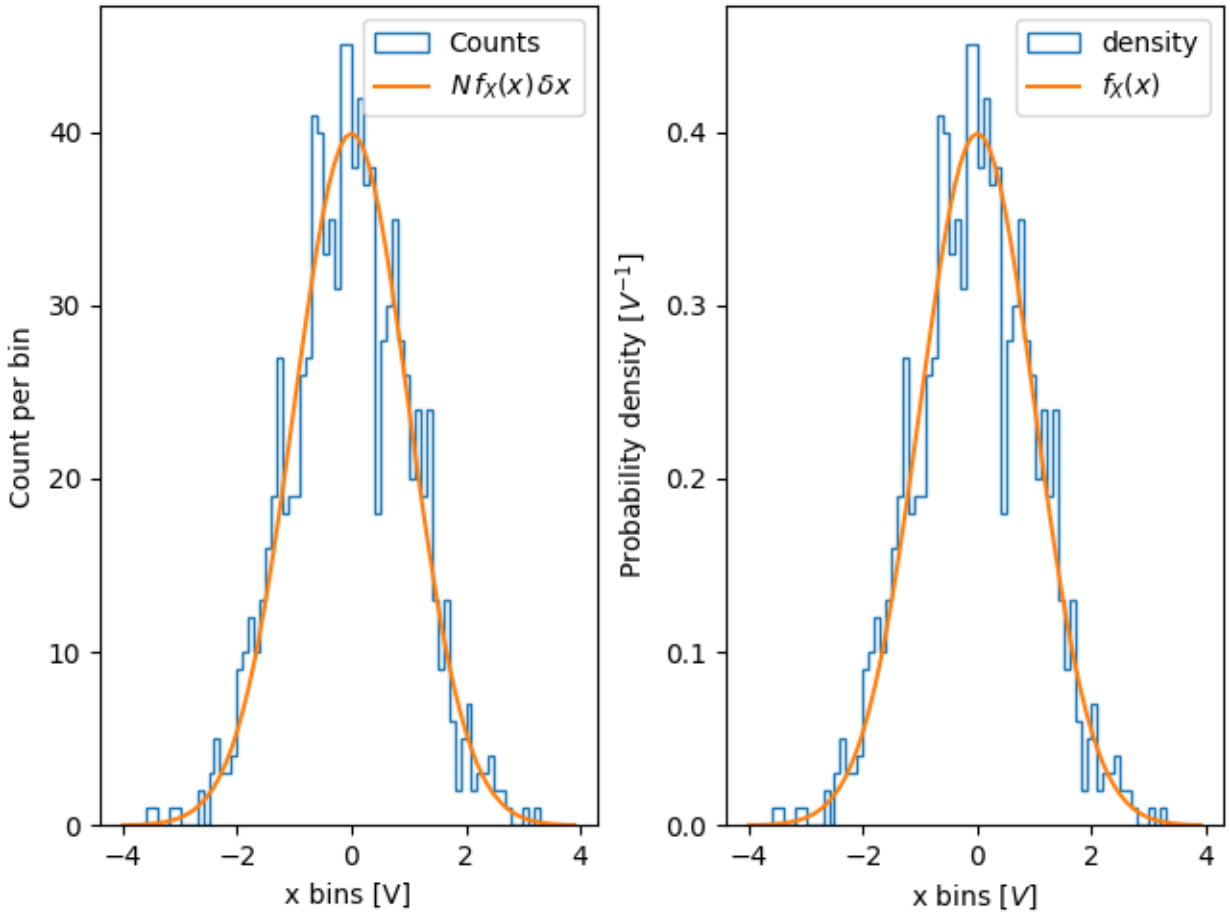
This normalization is how probability density functions are defined in statistics. If $X$ is a random variable on $x$, then $f_X$ is is the probability density function if $P[a < X < b] = \int_a^b f_X dx$. If the units of x are Volts, then the units of $f_X$ are $V^{-1}$ or probability per change in voltage.

The usefulness of this normalization is a little more clear when we draw from a known distribution and try to compare with theory. So, choose 1000 points from a normal distribution, and also calculate the known probability density function:

```
xdata = rng.normal(size=1000)
xpdf = np.arange(-4, 4, 0.1)
pdf = 1 / (np.sqrt(2 * np.pi)) * np.exp(-xpdf**2 / 2)
```

If we don't use `density=True`, we need to scale the expected probability distribution function by both the length of the data and the width of the bins:

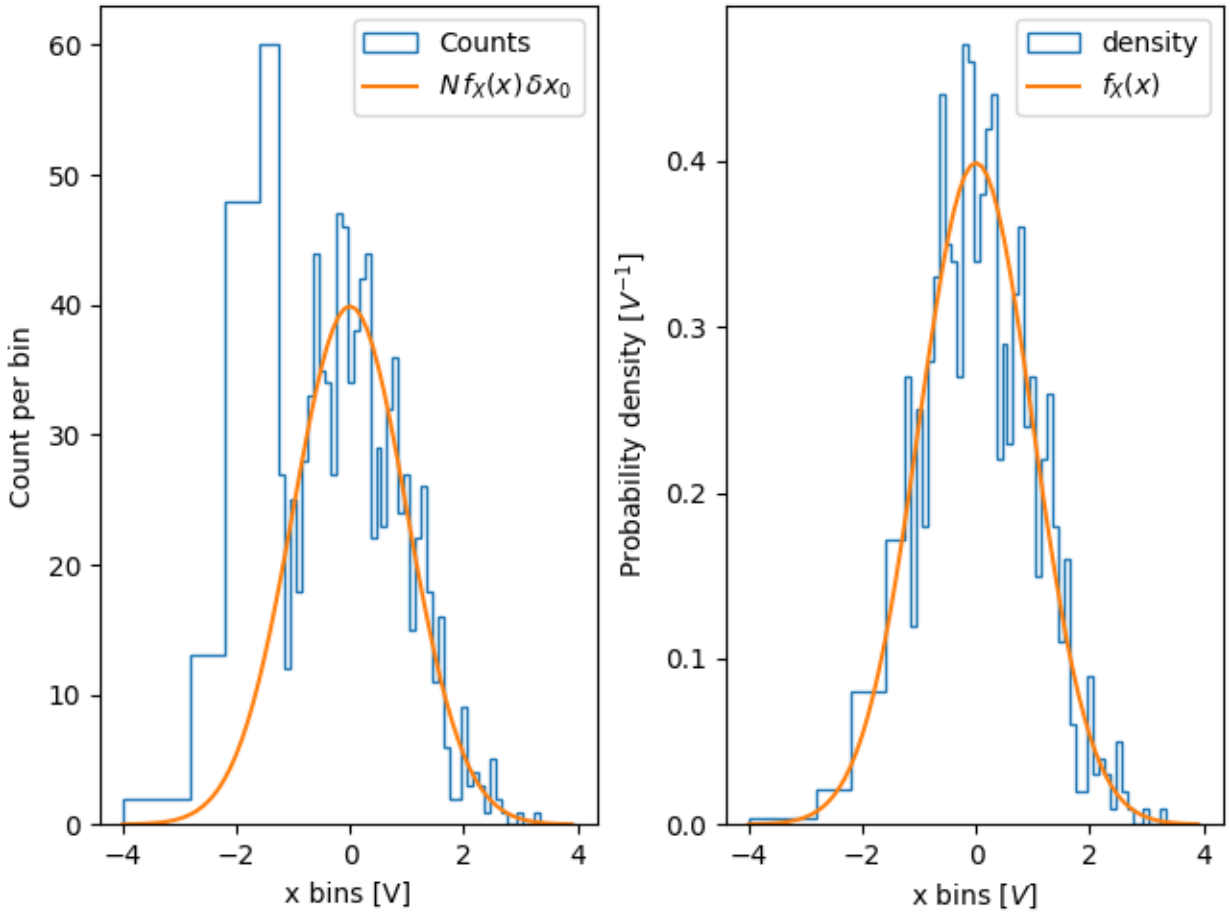
```
fig, ax = plt.subplot_mosaic([['False', 'True']], layout='constrained')
dx = 0.1
xbins = np.arange(-4, 4, dx)
ax['False'].hist(xdata, bins=xbins, density=False, histtype='step', label=
 ↪'Counts')

# scale and plot the expected pdf:
ax['False'].plot(xpdf, pdf * len(xdata) * dx, label=r'$N\,f_X(x)\,\delta x$')
ax['False'].set_ylabel('Count per bin')
ax['False'].set_xlabel('x bins [V]')
ax['False'].legend()

ax['True'].hist(xdata, bins=xbins, density=True, histtype='step', label=
 ↪'density')
ax['True'].plot(xpdf, pdf, label='$f_X(x)$')
ax['True'].set_ylabel('Probability density [$V^{-1}$]')
ax['True'].set_xlabel('x bins [$V$]')
ax['True'].legend()
```




One advantage of using the density is therefore that the shape and amplitude of the histogram does not depend on the size of the bins. Consider an extreme case where the bins do not have the same width. In this example, the bins below `x=-1.25` are six times wider than the rest of the bins. By normalizing by density, we preserve the shape of the distribution, whereas if we do not, then the wider bins have much higher counts

than the thinner bins:

```
fig, ax = plt.subplot_mosaic([['False', 'True']], layout='constrained')
dx = 0.1
xbins = np.hstack([np.arange(-4, -1.25, 6*dx), np.arange(-1.25, 4, dx)])
ax['False'].hist(xdata, bins=xbins, density=False, histtype='step', label=
 ↪'Counts')
ax['False'].plot(xpdf, pdf * len(xdata) * dx, label=r'$N\,f_X(x)\,\delta x_0$
 ↪')
ax['False'].set_ylabel('Count per bin')
ax['False'].set_xlabel('x bins [V]')
ax['False'].legend()

ax['True'].hist(xdata, bins=xbins, density=True, histtype='step', label=
 ↪'density')
ax['True'].plot(xpdf, pdf, label='$f_X(x)$')
ax['True'].set_ylabel('Probability density [$V^{-1}$]')
ax['True'].set_xlabel('x bins [$V$]')
ax['True'].legend()
```



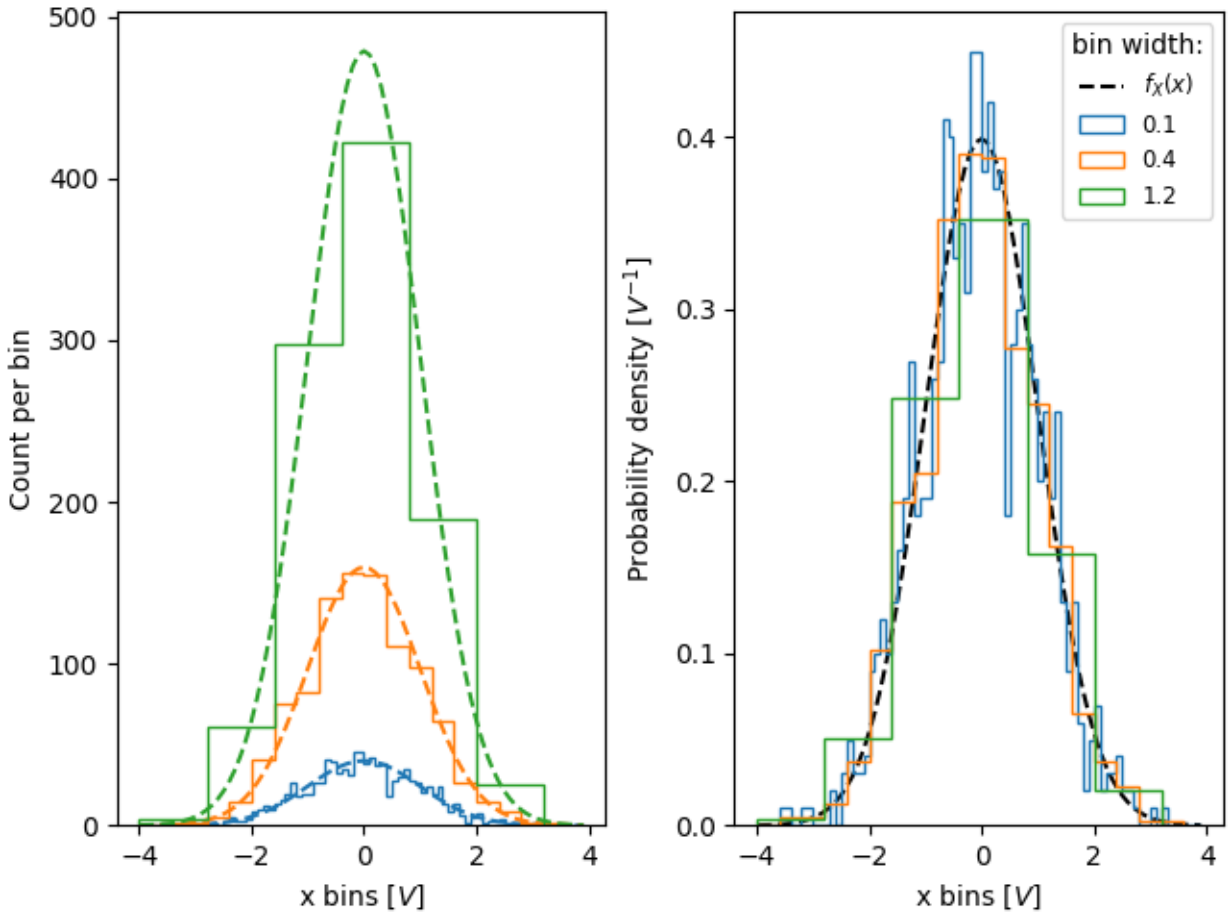

Similarly, if we want to compare histograms with different bin widths, we may want to use `density=True`:

```
fig, ax = plt.subplot_mosaic([['False', 'True']], layout='constrained')
```

(continues on next page)

(continued from previous page)

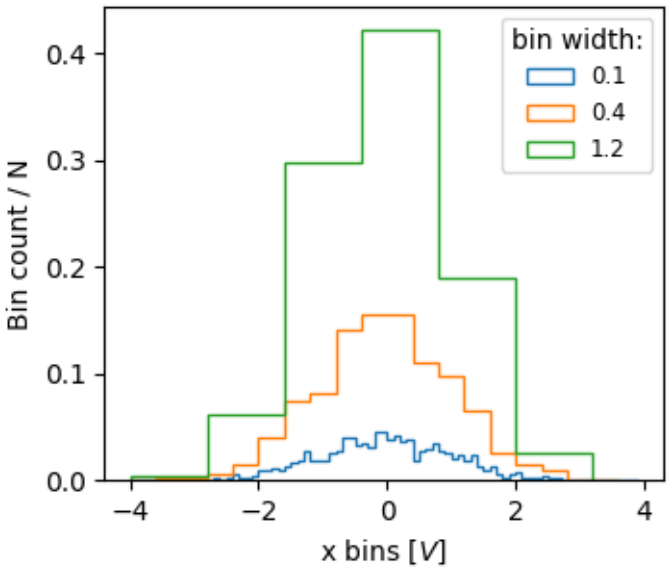
```
# expected PDF
ax['True'].plot(xpdf, pdf, '--', label='$f_X(x)$', color='k')

for nn, dx in enumerate([0.1, 0.4, 1.2]):
    xbins = np.arange(-4, 4, dx)
    # expected histogram:
    ax['False'].plot(xpdf, pdf*1000*dx, '--', color=f'C{nn}')
    ax['False'].hist(xdata, bins=xbins, density=False, histtype='step')

    ax['True'].hist(xdata, bins=xbins, density=True, histtype='step',␣
↪label=dx)

# Labels:
ax['False'].set_xlabel('x bins [$V$]')
ax['False'].set_ylabel('Count per bin')
ax['True'].set_ylabel('Probability density [$V^{-1}$]')
ax['True'].set_xlabel('x bins [$V$]')
ax['True'].legend(fontsize='small', title='bin width:')
```




Sometimes people want to normalize so that the sum of counts is one. This is analogous to a probability mass function for a discrete variable where the sum of probabilities for all the values equals one. Using `hist`, we

can get this normalization if we set the *weights* to 1/N. Note that the amplitude of this normalized histogram still depends on width and/or number of the bins:

```
fig, ax = plt.subplots(layout='constrained', figsize=(3.5, 3))

for nn, dx in enumerate([0.1, 0.4, 1.2]):
    xbins = np.arange(-4, 4, dx)
    ax.hist(xdata, bins=xbins, weights=1/len(xdata) * np.ones(len(xdata)),
                    histtype='step', label=f'{dx}')
ax.set_xlabel('x bins [$V$]')
ax.set_ylabel('Bin count / N')
ax.legend(fontsize='small', title='bin width:')
```



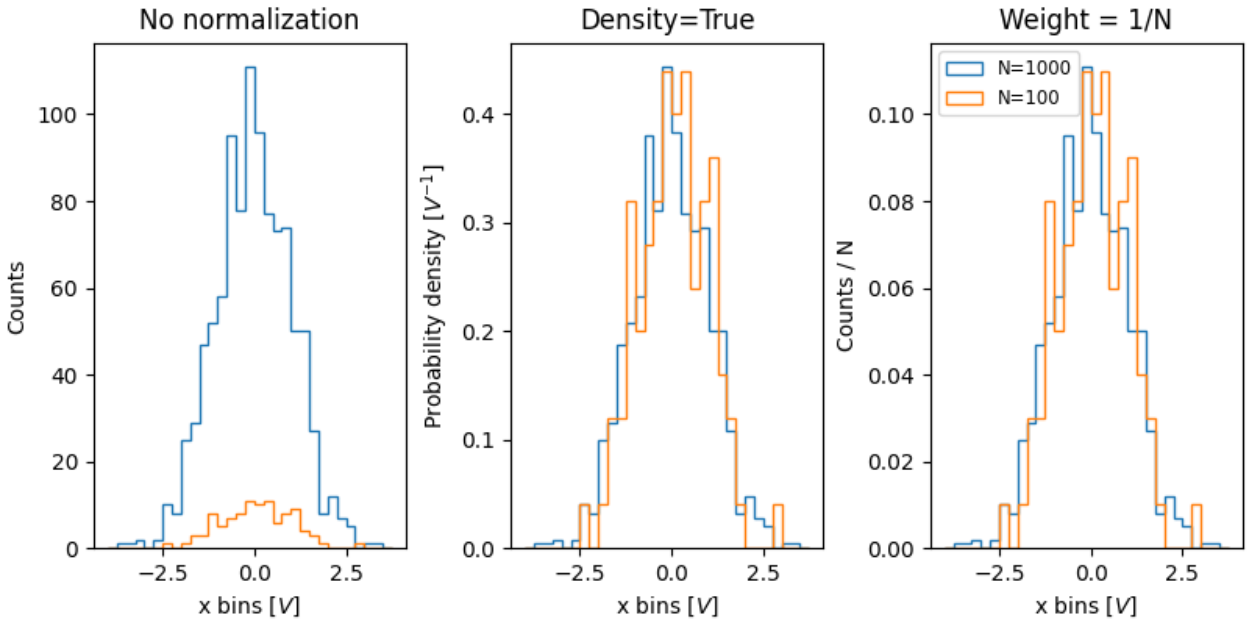

The value of normalizing histograms is comparing two distributions that have different sized populations. Here we compare the distribution of `xdata` with a population of 1000, and `xdata2` with 100 members.

```
xdata2 = rng.normal(size=100)

fig, ax = plt.subplot_mosaic([['no_norm', 'density', 'weight']],
                             layout='constrained', figsize=(8, 4))

xbins = np.arange(-4, 4, 0.25)

ax['no_norm'].hist(xdata, bins=xbins, histtype='step')
ax['no_norm'].hist(xdata2, bins=xbins, histtype='step')
ax['no_norm'].set_ylabel('Counts')
ax['no_norm'].set_xlabel('x bins [$V$]')
ax['no_norm'].set_title('No normalization')

ax['density'].hist(xdata, bins=xbins, histtype='step', density=True)
ax['density'].hist(xdata2, bins=xbins, histtype='step', density=True)
ax['density'].set_ylabel('Probability density [$V^{-1}$]')
ax['density'].set_title('Density=True')
ax['density'].set_xlabel('x bins [$V$]')
```

(continues on next page)

(continued from previous page)

```
ax['weight'].hist(xdata, bins=xbins, histtype='step',
                  weights=1 / len(xdata) * np.ones(len(xdata)),
                  label='N=1000')
ax['weight'].hist(xdata2, bins=xbins, histtype='step',
                  weights=1 / len(xdata2) * np.ones(len(xdata2)),
                  label='N=100')
ax['weight'].set_xlabel('x bins [$V$]')
ax['weight'].set_ylabel('Counts / N')
ax['weight'].legend(fontsize='small')
ax['weight'].set_title('Weight = 1/N')

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.hist* / *matplotlib.pyplot.hist*
- *matplotlib.axes.Axes.set_title*
- *matplotlib.axes.Axes.set_xlabel*
- *matplotlib.axes.Axes.set_ylabel*
- *matplotlib.axes.Axes.legend*

**Total running time of the script:** (0 minutes 3.700 seconds)

### Producing multiple histograms side by side

This example plots horizontal histograms of different samples along a categorical x-axis. Additionally, the histograms are plotted to be symmetrical about their x-position, thus making them very similar to violin plots.

To make this highly specialized plot, we can't use the standard `hist` method. Instead, we use `barh` to draw the horizontal bars directly. The vertical positions and lengths of the bars are computed via the `np.histogram` function. The histograms for all the samples are computed using the same range (min and max values) and number of bins, so that the bins for each sample are in the same vertical positions.

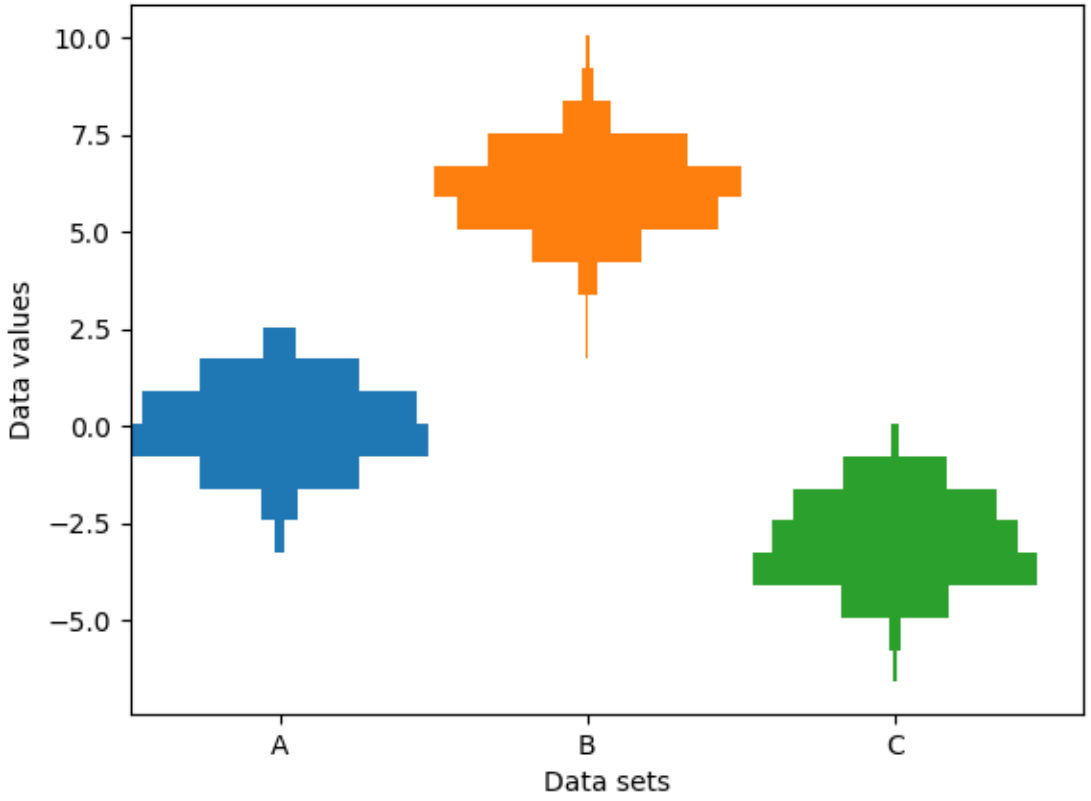
Selecting different bin counts and sizes can significantly affect the shape of a histogram. The Astropy docs have a great section on how to select these parameters: http://docs.astropy.org/en/stable/visualization/histogram.html

```
import matplotlib.pyplot as plt
import numpy as np

np.random.seed(19680801)
number_of_bins = 20

# An example of three data sets to compare
number_of_data_points = 387
labels = ["A", "B", "C"]
data_sets = [np.random.normal(0, 1, number_of_data_points),
             np.random.normal(6, 1, number_of_data_points),
             np.random.normal(-3, 1, number_of_data_points)]

# Computed quantities to aid plotting
hist_range = (np.min(data_sets), np.max(data_sets))
binned_data_sets = [
    np.histogram(d, range=hist_range, bins=number_of_bins)[0]
    for d in data_sets
]
binned_maximums = np.max(binned_data_sets, axis=1)
x_locations = np.arange(0, sum(binned_maximums), np.max(binned_maximums))

# The bin_edges are the same for all of the histograms
bin_edges = np.linspace(hist_range[0], hist_range[1], number_of_bins + 1)
heights = np.diff(bin_edges)
centers = bin_edges[:-1] + heights / 2

# Cycle through and plot each histogram
fig, ax = plt.subplots()
for x_loc, binned_data in zip(x_locations, binned_data_sets):
    lefts = x_loc - 0.5 * binned_data
    ax.barh(centers, binned_data, height=heights, left=lefts)

ax.set_xticks(x_locations, labels)

ax.set_ylabel("Data values")
ax.set_xlabel("Data sets")
```

(continues on next page)

(continued from previous page)

```
plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.barh* / *matplotlib.pyplot.barh*

### Time Series Histogram

This example demonstrates how to efficiently visualize large numbers of time series in a way that could potentially reveal hidden substructure and patterns that are not immediately obvious, and display them in a visually appealing way.

In this example, we generate multiple sinusoidal "signal" series that are buried under a larger number of random walk "noise/background" series. For an unbiased Gaussian random walk with standard deviation of σ, the RMS deviation from the origin after n steps is σ*sqrt(n). So in order to keep the sinusoids visible on the same scale as the random walks, we scale the amplitude by the random walk RMS. In addition, we also

introduce a small random offset `phi` to shift the sines left/right, and some additive random noise to shift individual data points up/down to make the signal a bit more "realistic" (you wouldn't expect a perfect sine wave to appear in your data).

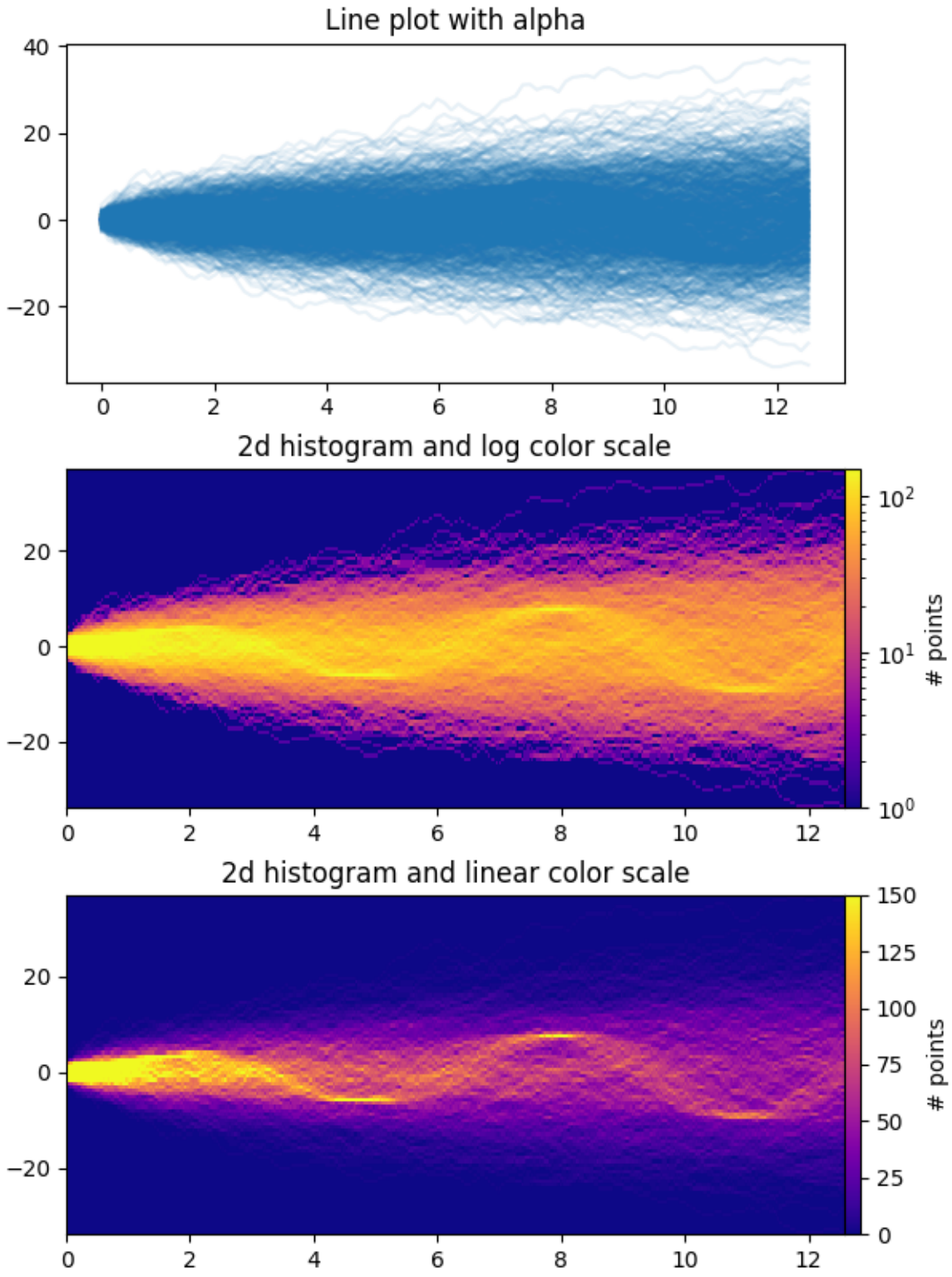
The first plot shows the typical way of visualizing multiple time series by overlaying them on top of each other with `plt.plot` and a small value of `alpha`. The second and third plots show how to reinterpret the data as a 2d histogram, with optional interpolation between data points, by using `np.histogram2d` and `plt.pcolormesh`.

```
import time

import matplotlib.pyplot as plt
import numpy as np

fig, axes = plt.subplots(nrows=3, figsize=(6, 8), layout='constrained')

# Fix random state for reproducibility
np.random.seed(19680801)
# Make some data; a 1D random walk + small fraction of sine waves
num_series = 1000
num_points = 100
SNR = 0.10  # Signal to Noise Ratio
x = np.linspace(0, 4 * np.pi, num_points)
# Generate unbiased Gaussian random walks
Y = np.cumsum(np.random.randn(num_series, num_points), axis=-1)
# Generate sinusoidal signals
num_signal = round(SNR * num_series)
phi = (np.pi / 8) * np.random.randn(num_signal, 1)  # small random offset
Y[-num_signal:] = (
    np.sqrt(np.arange(num_points))  # random walk RMS scaling factor
    * (np.sin(x - phi)
       + 0.05 * np.random.randn(num_signal, num_points))  # small random noise
)


# Plot series using `plot` and a small value of `alpha`. With this view it is
# very difficult to observe the sinusoidal behavior because of how many
# overlapping series there are. It also takes a bit of time to run because so
# many individual artists need to be generated.
tic = time.time()
axes[0].plot(x, Y.T, color="C0", alpha=0.1)
toc = time.time()
axes[0].set_title("Line plot with alpha")
print(f"{toc-tic:.3f} sec. elapsed")


# Now we will convert the multiple time series into a histogram. Not only will
# the hidden signal be more visible, but it is also a much quicker procedure.
tic = time.time()
# Linearly interpolate between the points in each time series
num_fine = 800
x_fine = np.linspace(x.min(), x.max(), num_fine)
y_fine = np.concatenate([np.interp(x_fine, x, y_row) for y_row in Y])
```

(continues on next page)

(continued from previous page)

```
x_fine = np.broadcast_to(x_fine, (num_series, num_fine)).ravel()


# Plot (x, y) points in 2d histogram with log colorscale
# It is pretty evident that there is some kind of structure under the noise
# You can tune vmax to make signal more visible
cmap = plt.colormaps["plasma"]
cmap = cmap.with_extremes(bad=cmap(0))
h, xedges, yedges = np.histogram2d(x_fine, y_fine, bins=[400, 100])
pcm = axes[1].pcolormesh(xedges, yedges, h.T, cmap=cmap,
                         norm="log", vmax=1.5e2, rasterized=True)
fig.colorbar(pcm, ax=axes[1], label="# points", pad=0)
axes[1].set_title("2d histogram and log color scale")

# Same data but on linear color scale
pcm = axes[2].pcolormesh(xedges, yedges, h.T, cmap=cmap,
                         vmax=1.5e2, rasterized=True)
fig.colorbar(pcm, ax=axes[2], label="# points", pad=0)
axes[2].set_title("2d histogram and linear color scale")

toc = time.time()
print(f"{toc-tic:.3f} sec. elapsed")
plt.show()
```




```
0.219 sec. elapsed
0.103 sec. elapsed
```

**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.pcolormesh* / *matplotlib.pyplot.pcolormesh*
- *matplotlib.figure.Figure.colorbar*

**Total running time of the script:** (0 minutes 1.817 seconds)

## Violin plot basics

Violin plots are similar to histograms and box plots in that they show an abstract representation of the probability distribution of the sample. Rather than showing counts of data points that fall into bins or order statistics, violin plots use kernel density estimation (KDE) to compute an empirical distribution of the sample. That computation is controlled by several parameters. This example demonstrates how to modify the number of points at which the KDE is evaluated (`points`) and how to modify the bandwidth of the KDE (`bw_method`).

For more information on violin plots and KDE, the scikit-learn docs have a great section: https://scikit-learn.org/stable/modules/density.html

```
import matplotlib.pyplot as plt
import numpy as np

# Fixing random state for reproducibility
np.random.seed(19680801)


# fake data
fs = 10  # fontsize
pos = [1, 2, 4, 5, 7, 8]
data = [np.random.normal(0, std, size=100) for std in pos]

fig, axs = plt.subplots(nrows=2, ncols=6, figsize=(10, 4))

axs[0, 0].violinplot(data, pos, points=20, widths=0.3,
                     showmeans=True, showextrema=True, showmedians=True)
axs[0, 0].set_title('Custom violin 1', fontsize=fs)

axs[0, 1].violinplot(data, pos, points=40, widths=0.5,
                     showmeans=True, showextrema=True, showmedians=True,
                     bw_method='silverman')
axs[0, 1].set_title('Custom violin 2', fontsize=fs)

axs[0, 2].violinplot(data, pos, points=60, widths=0.7, showmeans=True,
                     showextrema=True, showmedians=True, bw_method=0.5)
axs[0, 2].set_title('Custom violin 3', fontsize=fs)

axs[0, 3].violinplot(data, pos, points=60, widths=0.7, showmeans=True,
```

(continues on next page)

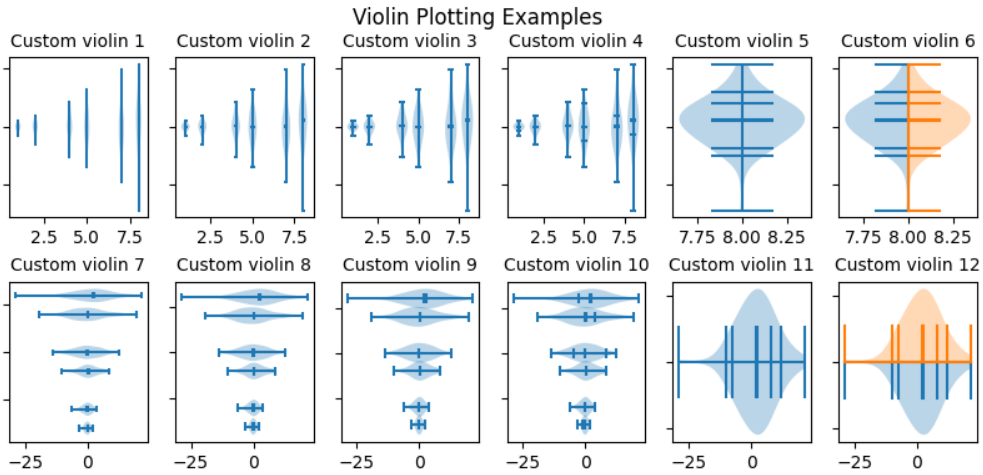
(continued from previous page)

```
                    showextrema=True, showmedians=True, bw_method=0.5,
                    quantiles=[[0.1], [], [], [0.175, 0.954], [0.75], [0.
 ↪25]])
axs[0, 3].set_title('Custom violin 4', fontsize=fs)

axs[0, 4].violinplot(data[-1:], pos[-1:], points=60, widths=0.7,
                    showmeans=True, showextrema=True, showmedians=True,
                    quantiles=[0.05, 0.1, 0.8, 0.9], bw_method=0.5)
axs[0, 4].set_title('Custom violin 5', fontsize=fs)

axs[0, 5].violinplot(data[-1:], pos[-1:], points=60, widths=0.7,
                    showmeans=True, showextrema=True, showmedians=True,
                    quantiles=[0.05, 0.1, 0.8, 0.9], bw_method=0.5, side='low
 ↪')

axs[0, 5].violinplot(data[-1:], pos[-1:], points=60, widths=0.7,
                    showmeans=True, showextrema=True, showmedians=True,
                    quantiles=[0.05, 0.1, 0.8, 0.9], bw_method=0.5, side=
 ↪'high')
axs[0, 5].set_title('Custom violin 6', fontsize=fs)

axs[1, 0].violinplot(data, pos, points=80, vert=False, widths=0.7,
                    showmeans=True, showextrema=True, showmedians=True)
axs[1, 0].set_title('Custom violin 7', fontsize=fs)

axs[1, 1].violinplot(data, pos, points=100, vert=False, widths=0.9,
                    showmeans=True, showextrema=True, showmedians=True,
                    bw_method='silverman')
axs[1, 1].set_title('Custom violin 8', fontsize=fs)

axs[1, 2].violinplot(data, pos, points=200, vert=False, widths=1.1,
                    showmeans=True, showextrema=True, showmedians=True,
                    bw_method=0.5)
axs[1, 2].set_title('Custom violin 9', fontsize=fs)

axs[1, 3].violinplot(data, pos, points=200, vert=False, widths=1.1,
                    showmeans=True, showextrema=True, showmedians=True,
                    quantiles=[[0.1], [], [], [0.175, 0.954], [0.75], [0.
 ↪25]],
                    bw_method=0.5)
axs[1, 3].set_title('Custom violin 10', fontsize=fs)

axs[1, 4].violinplot(data[-1:], pos[-1:], points=200, vert=False, widths=1.1,
                    showmeans=True, showextrema=True, showmedians=True,
                    quantiles=[0.05, 0.1, 0.8, 0.9], bw_method=0.5)
axs[1, 4].set_title('Custom violin 11', fontsize=fs)

axs[1, 5].violinplot(data[-1:], pos[-1:], points=200, vert=False, widths=1.1,
                    showmeans=True, showextrema=True, showmedians=True,
                    quantiles=[0.05, 0.1, 0.8, 0.9], bw_method=0.5, side='low
 ↪')
```

(continues on next page)

(continued from previous page)

```
axs[1, 5].violinplot(data[-1:], pos[-1:], points=200, vert=False, widths=1.1,
                     showmeans=True, showextrema=True, showmedians=True,
                     quantiles=[0.05, 0.1, 0.8, 0.9], bw_method=0.5, side=
 ↪'high')
axs[1, 5].set_title('Custom violin 12', fontsize=fs)


for ax in axs.flat:
    ax.set_yticklabels([])

fig.suptitle("Violin Plotting Examples")
fig.subplots_adjust(hspace=0.4)
plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.violinplot* / *matplotlib.pyplot.violinplot*

### 6.25.5 Pie and polar charts

#### Pie charts

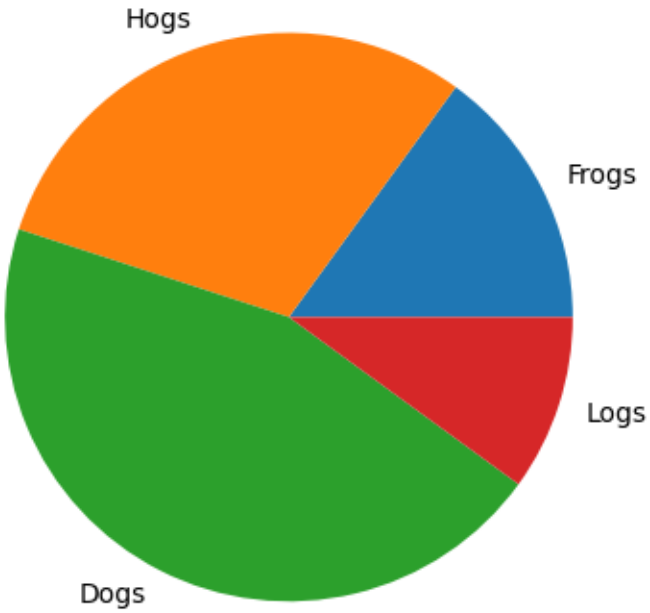
Demo of plotting a pie chart.

This example illustrates various parameters of *pie*.

### Label slices

Plot a pie chart of animals and label the slices. To add labels, pass a list of labels to the *labels* parameter

```
import matplotlib.pyplot as plt

labels = 'Frogs', 'Hogs', 'Dogs', 'Logs'
sizes = [15, 30, 45, 10]

fig, ax = plt.subplots()
ax.pie(sizes, labels=labels)
```

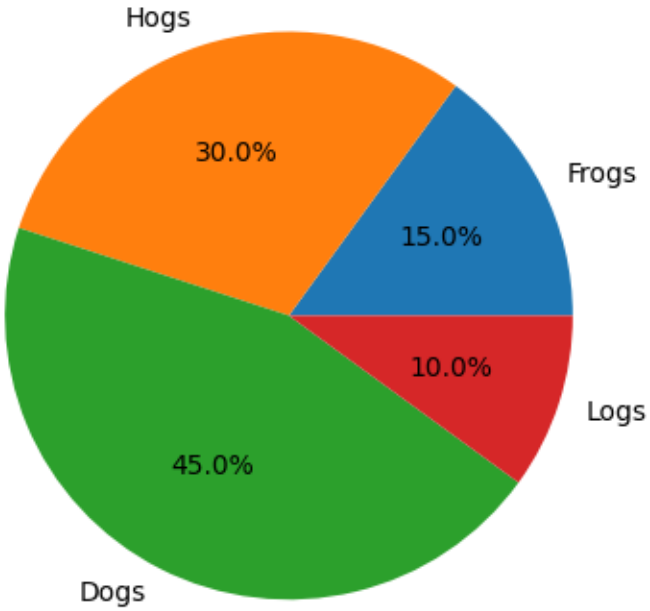



Each slice of the pie chart is a `patches.Wedge` object; therefore in addition to the customizations shown here, each wedge can be customized using the *wedgeprops* argument, as demonstrated in *Nested pie charts*.

### Auto-label slices

Pass a function or format string to *autopct* to label slices.

```
fig, ax = plt.subplots()
ax.pie(sizes, labels=labels, autopct='%1.1f%%')
```

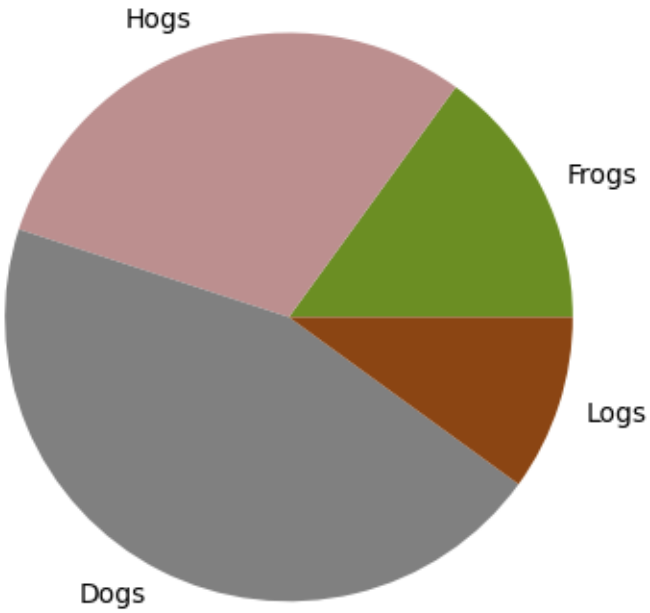



By default, the label values are obtained from the percent size of the slice.

### Color slices

Pass a list of colors to *colors* to set the color of each slice.

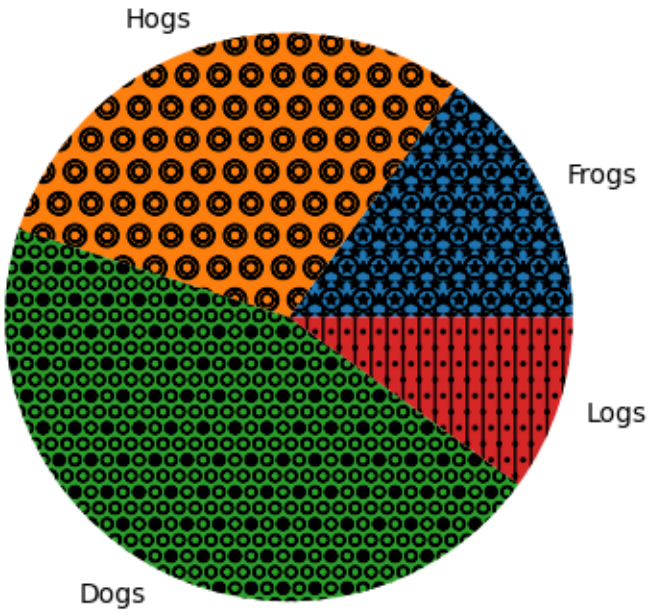
```
fig, ax = plt.subplots()
ax.pie(sizes, labels=labels,
       colors=['olivedrab', 'rosybrown', 'gray', 'saddlebrown'])
```




### Hatch slices

Pass a list of hatch patterns to *hatch* to set the pattern of each slice.

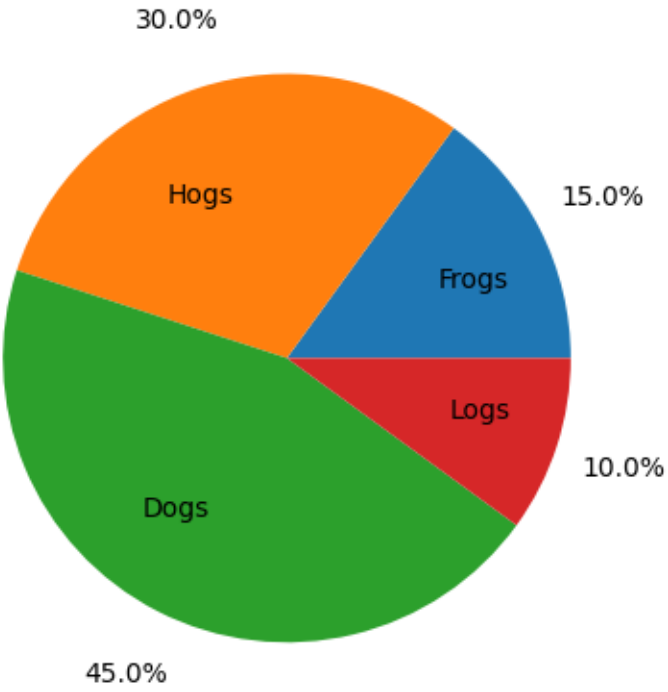
```
fig, ax = plt.subplots()
ax.pie(sizes, labels=labels, hatch=['**O', 'oO', 'O.O', '.||.'])
```




### Swap label and autopct text positions

Use the *labeldistance* and *pctdistance* parameters to position the *labels* and *autopct* text respectively.

```
fig, ax = plt.subplots()
ax.pie(sizes, labels=labels, autopct='%1.1f%%',
       pctdistance=1.25, labeldistance=.6)
```




*labeldistance* and *pctdistance* are ratios of the radius; therefore they vary between `0` for the center of the pie and `1` for the edge of the pie, and can be set to greater than `1` to place text outside the pie.

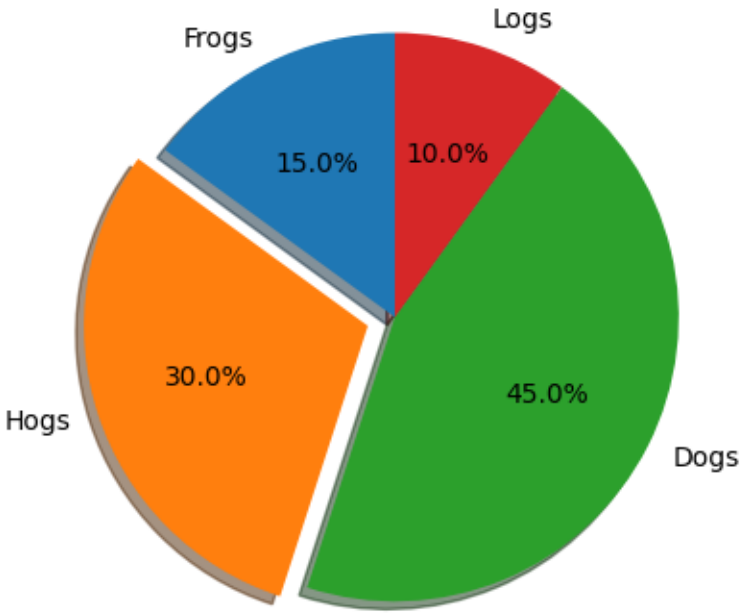
### Explode, shade, and rotate slices

In addition to the basic pie chart, this demo shows a few optional features:

- offsetting a slice using *explode*
- add a drop-shadow using *shadow*
- custom start angle using *startangle*

This example orders the slices, separates (explodes) them, and rotates them.

```
explode = (0, 0.1, 0, 0)  # only "explode" the 2nd slice (i.e. 'Hogs')

fig, ax = plt.subplots()
ax.pie(sizes, explode=explode, labels=labels, autopct='%1.1f%%',
       shadow=True, startangle=90)
plt.show()
```

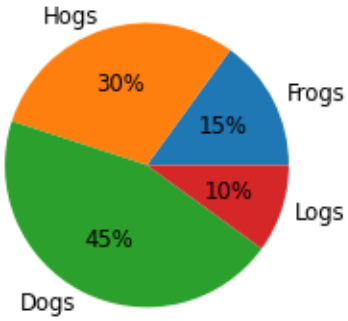



The default *startangle* is 0, which would start the first slice ("Frogs") on the positive x-axis. This example sets `startangle = 90` such that all the slices are rotated counter-clockwise by 90 degrees, and the frog slice starts on the positive y-axis.

### Controlling the size

By changing the *radius* parameter, and often the text size for better visual appearance, the pie chart can be scaled.

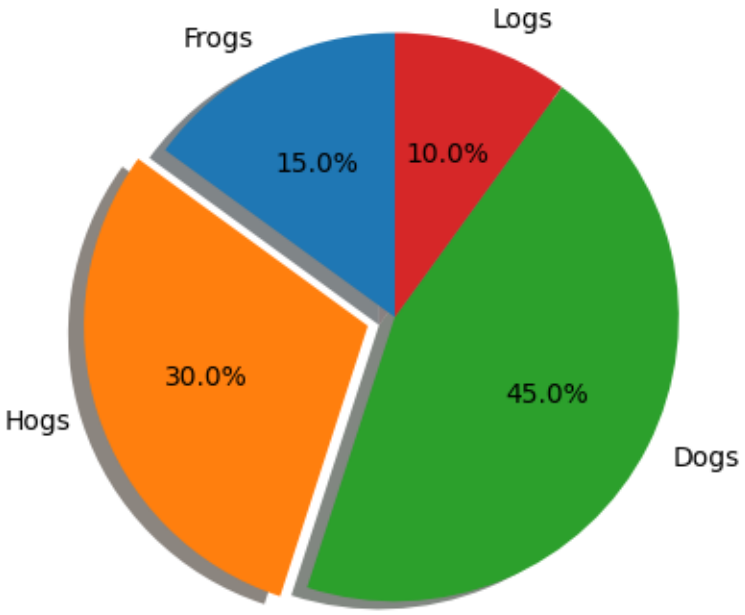
```
fig, ax = plt.subplots()

ax.pie(sizes, labels=labels, autopct='%.0f%%',
       textprops={'size': 'smaller'}, radius=0.5)
plt.show()
```




### Modifying the shadow

The *shadow* parameter may optionally take a dictionary with arguments to the *Shadow* patch. This can be used to modify the default shadow.

```
fig, ax = plt.subplots()
ax.pie(sizes, explode=explode, labels=labels, autopct='%1.1f%%',
       shadow={'ox': -0.04, 'edgecolor': 'none', 'shade': 0.9}, startangle=90)
plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.pie* / *matplotlib.pyplot.pie*

**Total running time of the script:** (0 minutes 1.970 seconds)

## Bar of pie

Make a "bar of pie" chart where the first slice of the pie is "exploded" into a bar chart with a further breakdown of said slice's characteristics. The example demonstrates using a figure with multiple sets of Axes and using the Axes patches list to add two ConnectionPatches to link the subplot charts.

```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.patches import ConnectionPatch

# make figure and assign axis objects
```

(continues on next page)

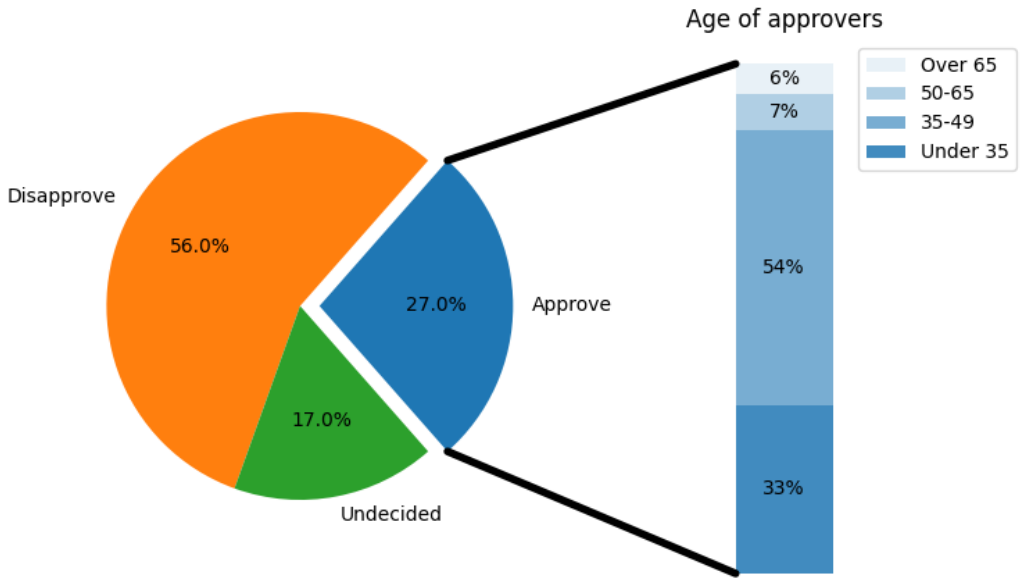
(continued from previous page)

```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(9, 5))
fig.subplots_adjust(wspace=0)

# pie chart parameters
overall_ratios = [.27, .56, .17]
labels = ['Approve', 'Disapprove', 'Undecided']
explode = [0.1, 0, 0]
# rotate so that first wedge is split by the x-axis
angle = -180 * overall_ratios[0]
wedges, *_ = ax1.pie(overall_ratios, autopct='%1.1f%%', startangle=angle,
                     labels=labels, explode=explode)

# bar chart parameters
age_ratios = [.33, .54, .07, .06]
age_labels = ['Under 35', '35-49', '50-65', 'Over 65']
bottom = 1
width = .2

# Adding from the top matches the legend.
for j, (height, label) in enumerate(reversed([*zip(age_ratios, age_labels)])):
    bottom -= height
    bc = ax2.bar(0, height, width, bottom=bottom, color='C0', label=label,
                 alpha=0.1 + 0.25 * j)
    ax2.bar_label(bc, labels=[f"{height:.0%}"], label_type='center')

ax2.set_title('Age of approvers')
ax2.legend()
ax2.axis('off')
ax2.set_xlim(- 2.5 * width, 2.5 * width)

# use ConnectionPatch to draw lines between the two plots
theta1, theta2 = wedges[0].theta1, wedges[0].theta2
center, r = wedges[0].center, wedges[0].r
bar_height = sum(age_ratios)

# draw top connecting line
x = r * np.cos(np.pi / 180 * theta2) + center[0]
y = r * np.sin(np.pi / 180 * theta2) + center[1]
con = ConnectionPatch(xyA=(-width / 2, bar_height), coordsA=ax2.transData,
                      xyB=(x, y), coordsB=ax1.transData)
con.set_color([0, 0, 0])
con.set_linewidth(4)
ax2.add_artist(con)

# draw bottom connecting line
x = r * np.cos(np.pi / 180 * theta1) + center[0]
y = r * np.sin(np.pi / 180 * theta1) + center[1]
con = ConnectionPatch(xyA=(-width / 2, 0), coordsA=ax2.transData,
                      xyB=(x, y), coordsB=ax1.transData)
con.set_color([0, 0, 0])
ax2.add_artist(con)
con.set_linewidth(4)
```

(continues on next page)

(continued from previous page)

```
plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.bar* / *matplotlib.pyplot.bar*
- *matplotlib.axes.Axes.pie* / *matplotlib.pyplot.pie*
- *matplotlib.patches.ConnectionPatch*

## Nested pie charts

The following examples show two ways to build a nested pie chart in Matplotlib. Such charts are often referred to as donut charts.

See also the *Left ventricle bullseye* example.

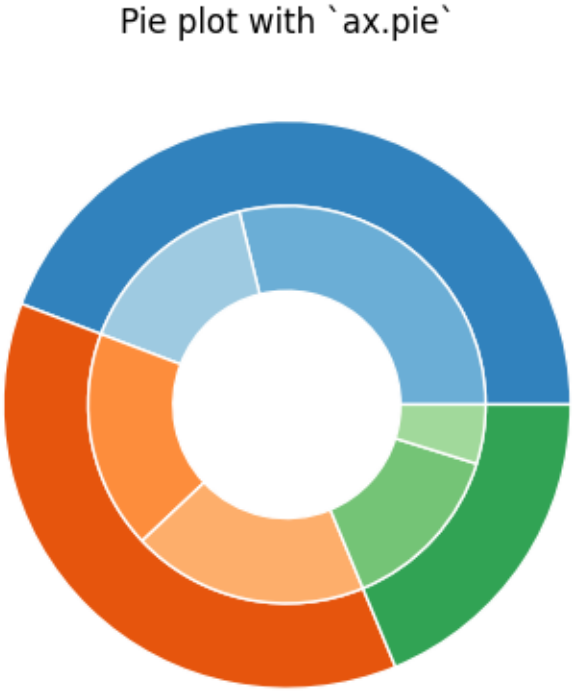
```
import matplotlib.pyplot as plt
import numpy as np
```

The most straightforward way to build a pie chart is to use the *pie* method.

In this case, pie takes values corresponding to counts in a group. We'll first generate some fake data, corresponding to three groups. In the inner circle, we'll treat each number as belonging to its own group. In the outer circle, we'll plot them as members of their original 3 groups.

The effect of the donut shape is achieved by setting a `width` to the pie's wedges through the *wedgeprops* argument.

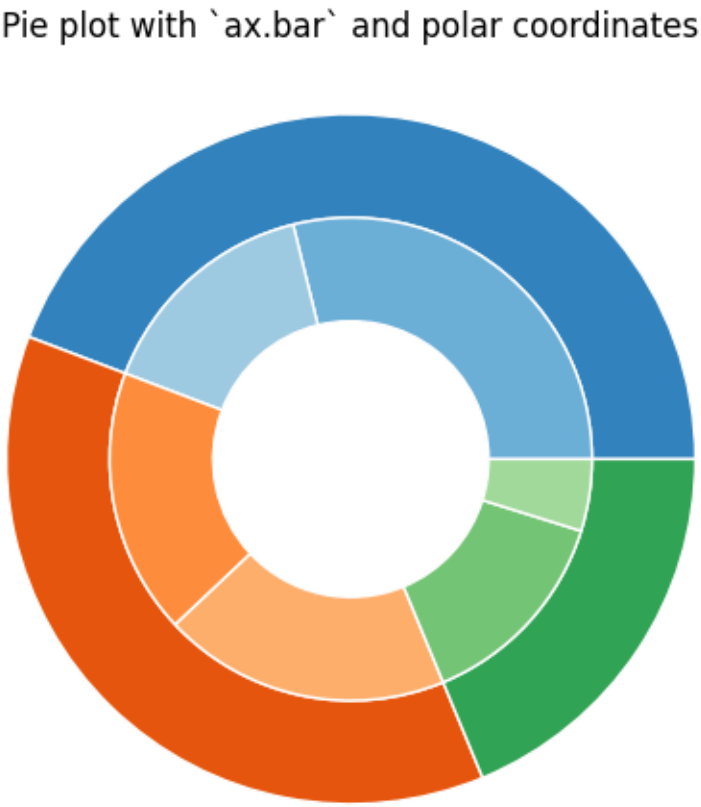
```
fig, ax = plt.subplots()

size = 0.3
vals = np.array([[60., 32.], [37., 40.], [29., 10.]])

cmap = plt.colormaps["tab20c"]
outer_colors = cmap(np.arange(3)*4)
inner_colors = cmap([1, 2, 5, 6, 9, 10])

ax.pie(vals.sum(axis=1), radius=1, colors=outer_colors,
       wedgeprops=dict(width=size, edgecolor='w'))

ax.pie(vals.flatten(), radius=1-size, colors=inner_colors,
       wedgeprops=dict(width=size, edgecolor='w'))

ax.set(aspect="equal", title='Pie plot with `ax.pie`')
plt.show()
```




However, you can accomplish the same output by using a bar plot on Axes with a polar coordinate system. This may give more flexibility on the exact design of the plot.

In this case, we need to map x-values of the bar chart onto radians of a circle. The cumulative sum of the values are used as the edges of the bars.

```
fig, ax = plt.subplots(subplot_kw=dict(projection="polar"))

size = 0.3
vals = np.array([[60., 32.], [37., 40.], [29., 10.]])
# Normalize vals to 2 pi
valsnorm = vals/np.sum(vals)*2*np.pi
# Obtain the ordinates of the bar edges
valsleft = np.cumsum(np.append(0, valsnorm.flatten()[:-1])).reshape(vals.
 ↪shape)

cmap = plt.colormaps["tab20c"]
outer_colors = cmap(np.arange(3)*4)
inner_colors = cmap([1, 2, 5, 6, 9, 10])

ax.bar(x=valsleft[:, 0],
       width=valsnorm.sum(axis=1), bottom=1-size, height=size,
       color=outer_colors, edgecolor='w', linewidth=1, align="edge")

ax.bar(x=valsleft.flatten(),
       width=valsnorm.flatten(), bottom=1-2*size, height=size,
       color=inner_colors, edgecolor='w', linewidth=1, align="edge")

ax.set(title="Pie plot with `ax.bar` and polar coordinates")
ax.set_axis_off()
plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.pie* / *matplotlib.pyplot.pie*
- *matplotlib.axes.Axes.bar* / *matplotlib.pyplot.bar*
- *matplotlib.projections.polar*
- `Axes.set` (*matplotlib.artist.Artist.set*)
- *matplotlib.axes.Axes.set_axis_off*

### Labeling a pie and a donut

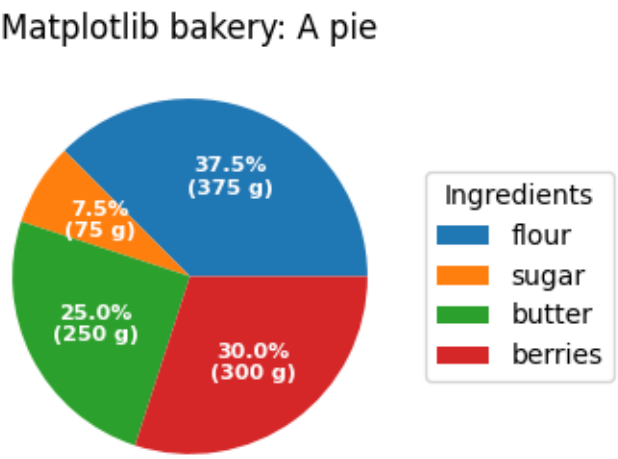
Welcome to the Matplotlib bakery. We will create a pie and a donut chart through the *pie method* and show how to label them with a *legend* as well as with *annotations*.

As usual we would start by defining the imports and create a figure with subplots. Now it's time for the pie. Starting with a pie recipe, we create the data and a list of labels from it.

We can provide a function to the `autopct` argument, which will expand automatic percentage labeling by showing absolute values; we calculate the latter back from relative data and the known sum of all values.

We then create the pie and store the returned objects for later. The first returned element of the returned tuple is a list of the wedges. Those are *matplotlib.patches.Wedge* patches, which can directly be used as the handles for a legend. We can use the legend's `bbox_to_anchor` argument to position the legend outside of the pie. Here we use the axes coordinates `(1, 0, 0.5, 1)` together with the location `"center left"`; i.e. the left central point of the legend will be at the left central point of the bounding box, spanning from `(1, 0)` to `(1.5, 1)` in axes coordinates.

```
import matplotlib.pyplot as plt
import numpy as np

fig, ax = plt.subplots(figsize=(6, 3), subplot_kw=dict(aspect="equal"))

recipe = ["375 g flour",
          "75 g sugar",
          "250 g butter",
          "300 g berries"]

data = [float(x.split()[0]) for x in recipe]
ingredients = [x.split()[-1] for x in recipe]


def func(pct, allvals):
    absolute = int(np.round(pct/100.*np.sum(allvals)))
    return f"{pct:.1f}%\n({absolute:d} g)"


wedges, texts, autotexts = ax.pie(data, autopct=lambda pct: func(pct, data),
                                  textprops=dict(color="w"))

ax.legend(wedges, ingredients,
          title="Ingredients",
          loc="center left",
          bbox_to_anchor=(1, 0, 0.5, 1))

plt.setp(autotexts, size=8, weight="bold")

ax.set_title("Matplotlib bakery: A pie")

plt.show()
```




Now it's time for the donut. Starting with a donut recipe, we transcribe the data to numbers (converting 1 egg to 50 g), and directly plot the pie. The pie? Wait... it's going to be donut, is it not? Well, as we see here, the donut is a pie, having a certain `width` set to the wedges, which is different from its radius. It's as easy as it gets. This is done via the `wedgeprops` argument.

We then want to label the wedges via *annotations*. We first create some dictionaries of common properties, which we can later pass as keyword argument. We then iterate over all wedges and for each

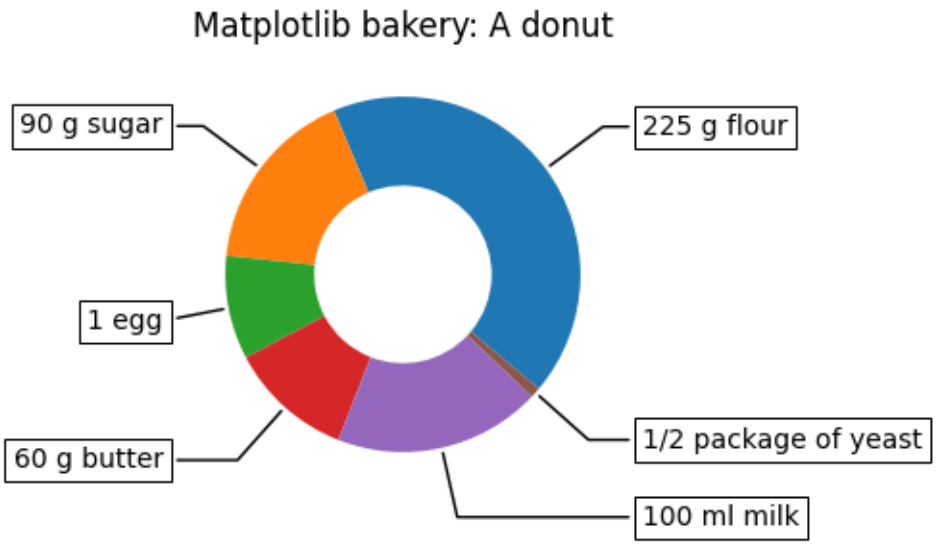
- calculate the angle of the wedge's center,
- from that obtain the coordinates of the point at that angle on the circumference,
- determine the horizontal alignment of the text, depending on which side of the circle the point lies,
- update the connection style with the obtained angle to have the annotation arrow point outwards from the donut,
- finally, create the annotation with all the previously determined parameters.

```
fig, ax = plt.subplots(figsize=(6, 3), subplot_kw=dict(aspect="equal"))

recipe = ["225 g flour",
          "90 g sugar",
          "1 egg",
          "60 g butter",
          "100 ml milk",
          "1/2 package of yeast"]

data = [225, 90, 50, 60, 100, 5]

wedges, texts = ax.pie(data, wedgeprops=dict(width=0.5), startangle=-40)

bbox_props = dict(boxstyle="square,pad=0.3", fc="w", ec="k", lw=0.72)
kw = dict(arrowprops=dict(arrowstyle="-"),
          bbox=bbox_props, zorder=0, va="center")
```

(continues on next page)

(continued from previous page)

```
for i, p in enumerate(wedges):
    ang = (p.theta2 - p.theta1)/2. + p.theta1
    y = np.sin(np.deg2rad(ang))
    x = np.cos(np.deg2rad(ang))
    horizontalalignment = {-1: "right", 1: "left"}[int(np.sign(x))]
    connectionstyle = f"angle,angleA=0,angleB={ang}"
    kw["arrowprops"].update({"connectionstyle": connectionstyle})
    ax.annotate(recipe[i], xy=(x, y), xytext=(1.35*np.sign(x), 1.4*y),
                horizontalalignment=horizontalalignment, **kw)

ax.set_title("Matplotlib bakery: A donut")

plt.show()
```




And here it is, the donut. Note however, that if we were to use this recipe, the ingredients would suffice for around 6 donuts - producing one huge donut is untested and might result in kitchen errors.

**References**

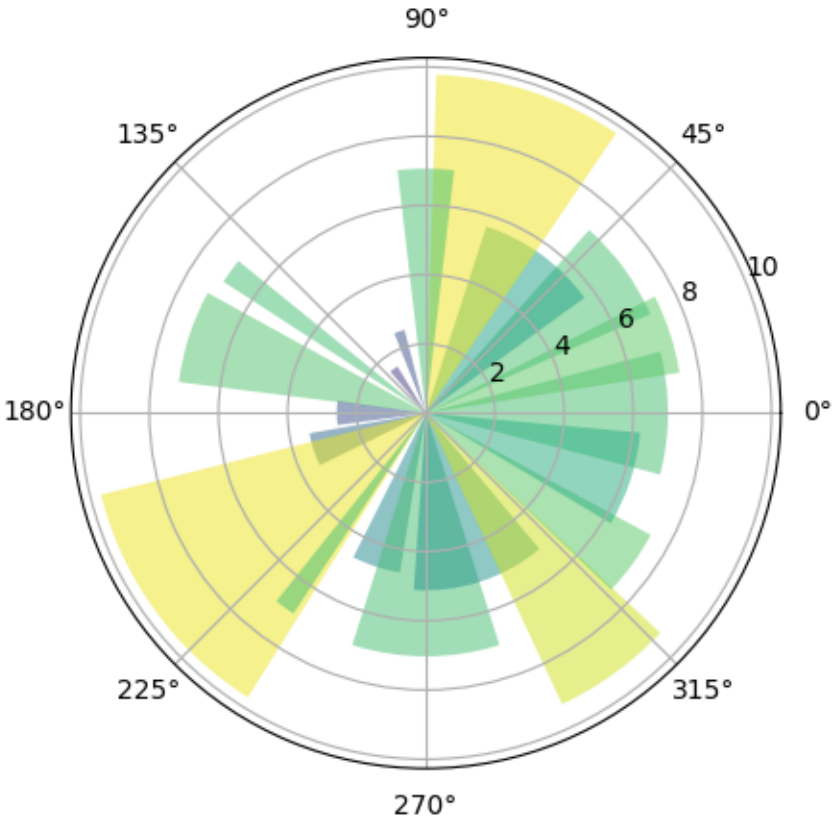
The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.pie* / *matplotlib.pyplot.pie*
- *matplotlib.axes.Axes.legend* / *matplotlib.pyplot.legend*

### Bar chart on polar axis

Demo of bar plot on a polar axis.

```
import matplotlib.pyplot as plt
import numpy as np

# Fixing random state for reproducibility
np.random.seed(19680801)

# Compute pie slices
N = 20
theta = np.linspace(0.0, 2 * np.pi, N, endpoint=False)
radii = 10 * np.random.rand(N)
width = np.pi / 4 * np.random.rand(N)
colors = plt.cm.viridis(radii / 10.)

ax = plt.subplot(projection='polar')
ax.bar(theta, radii, width=width, bottom=0.0, color=colors, alpha=0.5)

plt.show()
```




**References**

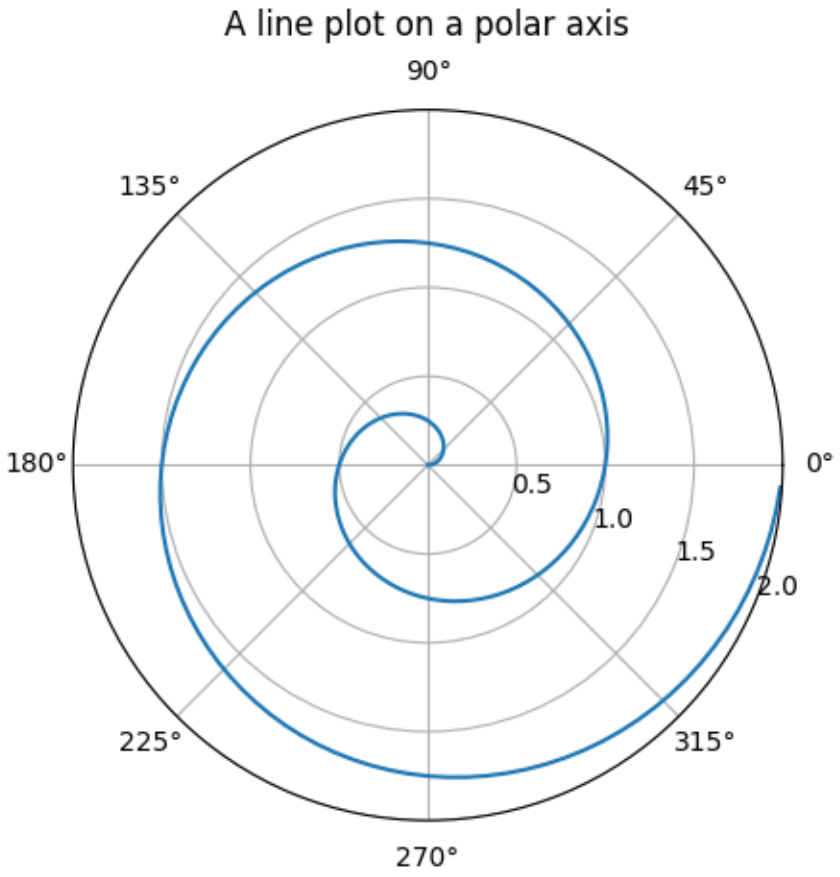
The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.bar* / *matplotlib.pyplot.bar*
- *matplotlib.projections.polar*

### Polar plot

Demo of a line plot on a polar axis.

```
import matplotlib.pyplot as plt
import numpy as np

r = np.arange(0, 2, 0.01)
theta = 2 * np.pi * r

fig, ax = plt.subplots(subplot_kw={'projection': 'polar'})
ax.plot(theta, r)
ax.set_rmax(2)
ax.set_rticks([0.5, 1, 1.5, 2])  # Less radial ticks
ax.set_rlabel_position(-22.5)  # Move radial labels away from plotted line
ax.grid(True)

ax.set_title("A line plot on a polar axis", va='bottom')
plt.show()
```




**References**

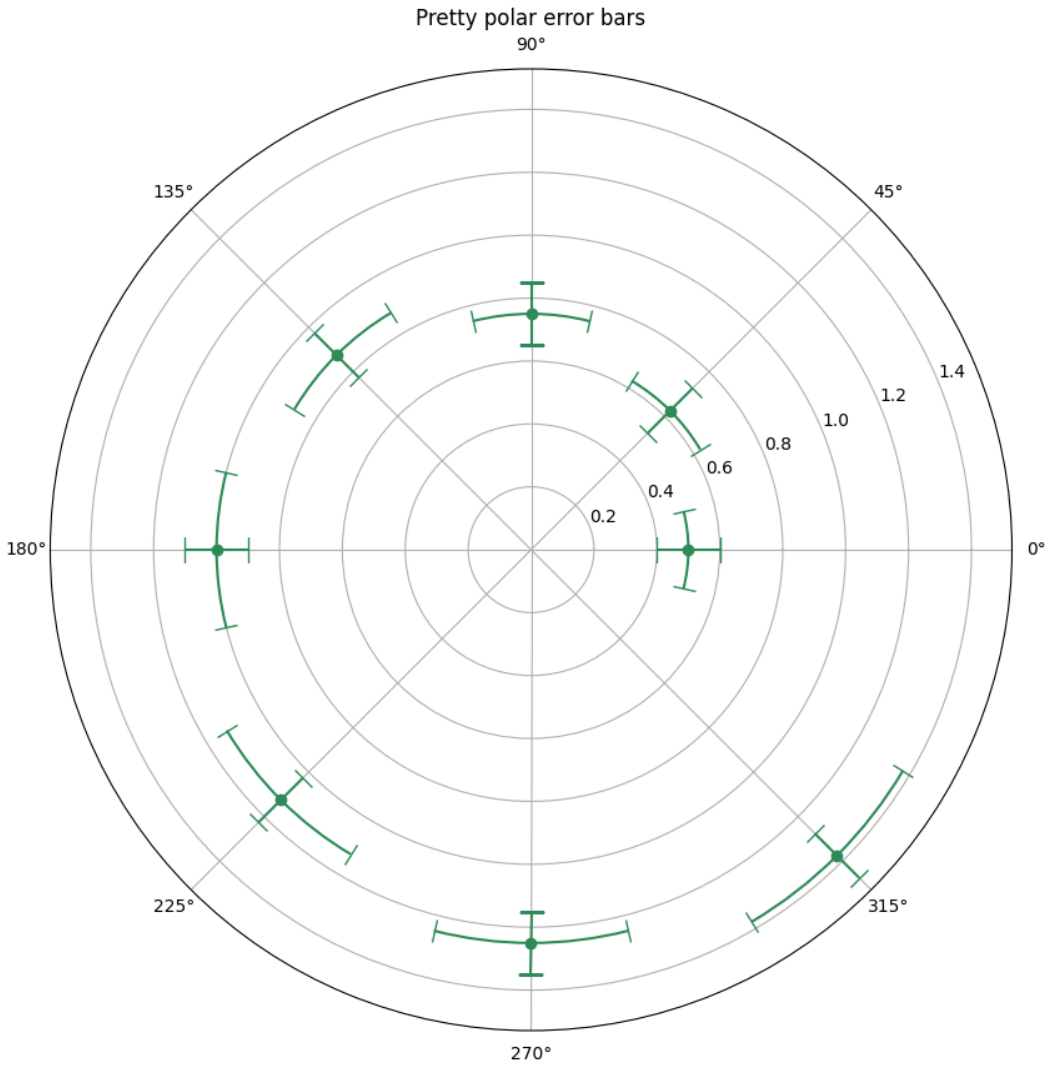
The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.plot* / *matplotlib.pyplot.plot*
- *matplotlib.projections.polar*
- *matplotlib.projections.polar.PolarAxes*
- *matplotlib.projections.polar.PolarAxes.set_rticks*
- *matplotlib.projections.polar.PolarAxes.set_rmax*
- *matplotlib.projections.polar.PolarAxes.set_rlabel_position*

### Error bar rendering on polar axis

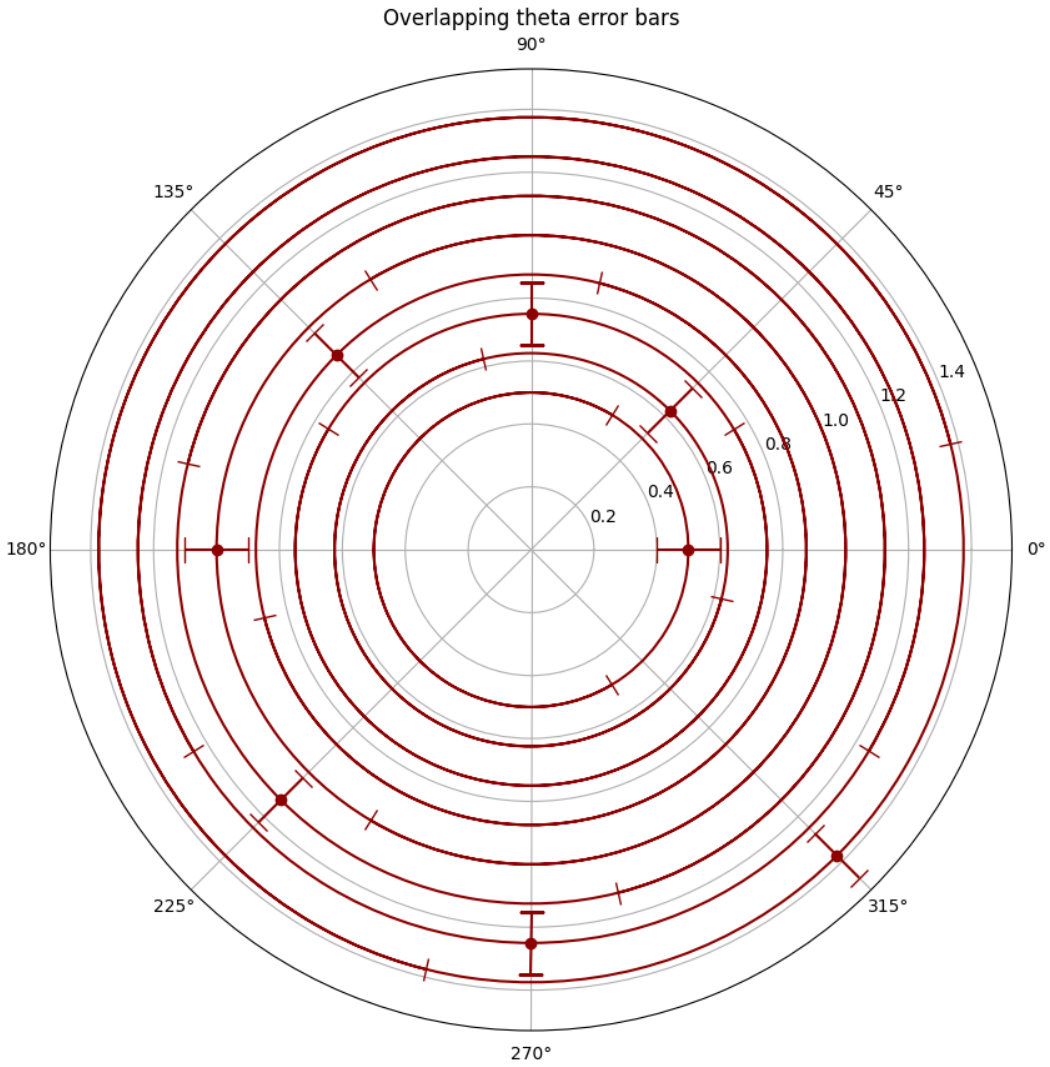
Demo of error bar plot in polar coordinates. Theta error bars are curved lines ended with caps oriented towards the center. Radius error bars are straight lines oriented towards center with perpendicular caps.

```
import matplotlib.pyplot as plt
import numpy as np

theta = np.arange(0, 2 * np.pi, np.pi / 4)
r = theta / np.pi / 2 + 0.5

fig = plt.figure(figsize=(10, 10))
ax = fig.add_subplot(projection='polar')
ax.errorbar(theta, r, xerr=0.25, yerr=0.1, capsize=7, fmt="o", c="seagreen")
ax.set_title("Pretty polar error bars")
plt.show()
```



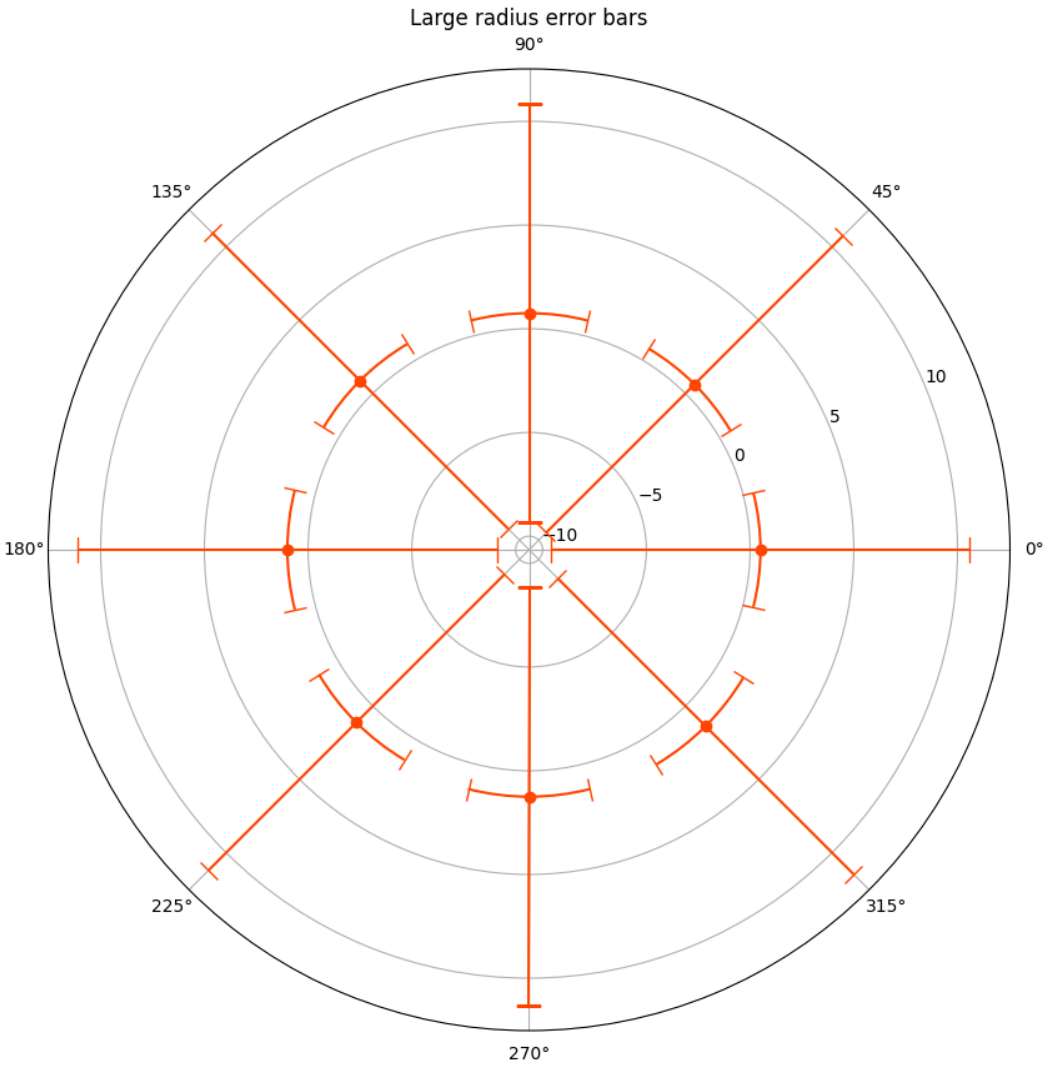

Please acknowledge that large theta error bars will be overlapping. This may reduce readability of the output plot. See example figure below:

```
fig = plt.figure(figsize=(10, 10))
ax = fig.add_subplot(projection='polar')
ax.errorbar(theta, r, xerr=5.25, yerr=0.1, capsize=7, fmt="o", c="darkred")
ax.set_title("Overlapping theta error bars")
plt.show()
```




On the other hand, large radius error bars will never overlap, they just lead to unwanted scale in the data, reducing the displayed range.

```
fig = plt.figure(figsize=(10, 10))
ax = fig.add_subplot(projection='polar')
ax.errorbar(theta, r, xerr=0.25, yerr=10.1, capsize=7, fmt="o", c="orangered")
ax.set_title("Large radius error bars")
plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

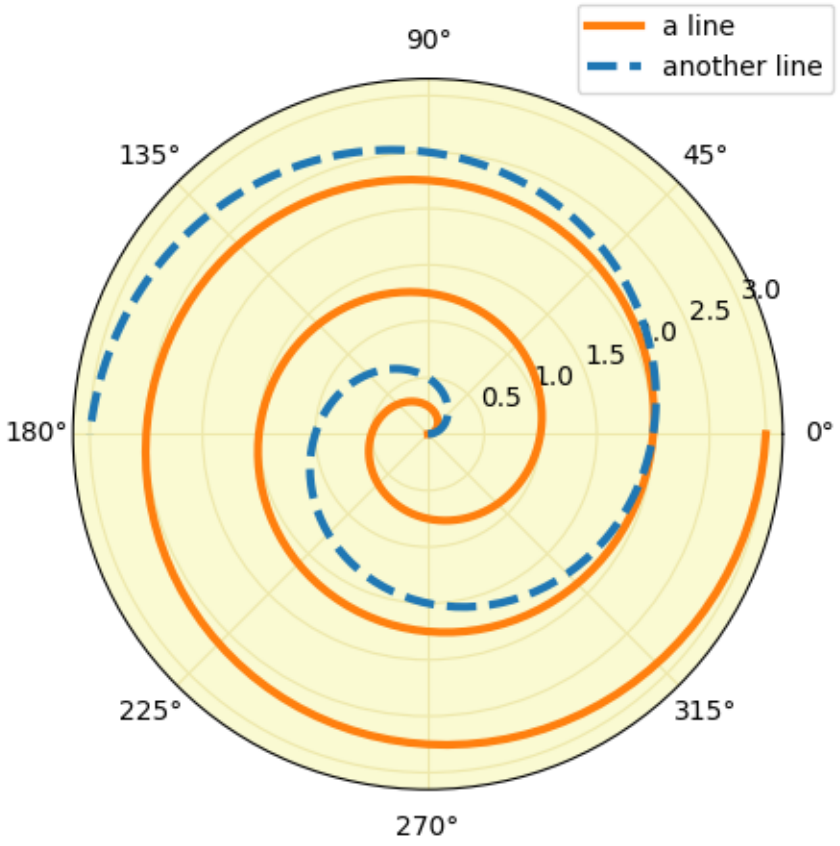
- *matplotlib.axes.Axes.errorbar* / *matplotlib.pyplot.errorbar*
- *matplotlib.projections.polar*

**Total running time of the script:** (0 minutes 3.117 seconds)

### Polar legend

Using a legend on a polar-axis plot.

```
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure()
ax = fig.add_subplot(projection="polar", facecolor="lightgoldenrodyellow")

r = np.linspace(0, 3, 301)
theta = 2 * np.pi * r
ax.plot(theta, r, color="tab:orange", lw=3, label="a line")
ax.plot(0.5 * theta, r, color="tab:blue", ls="--", lw=3, label="another line")
ax.tick_params(grid_color="palegoldenrod")
# For polar Axes, it may be useful to move the legend slightly away from the
# Axes center, to avoid overlap between the legend and the Axes.  The␣
 ↪following
# snippet places the legend's lower left corner just outside the polar Axes
# at an angle of 67.5 degrees in polar coordinates.
angle = np.deg2rad(67.5)
ax.legend(loc="lower left",
          bbox_to_anchor=(.5 + np.cos(angle)/2, .5 + np.sin(angle)/2))

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.plot* / *matplotlib.pyplot.plot*
- *matplotlib.axes.Axes.legend* / *matplotlib.pyplot.legend*
- *matplotlib.projections.polar*
- *matplotlib.projections.polar.PolarAxes*

## Scatter plot on polar axis

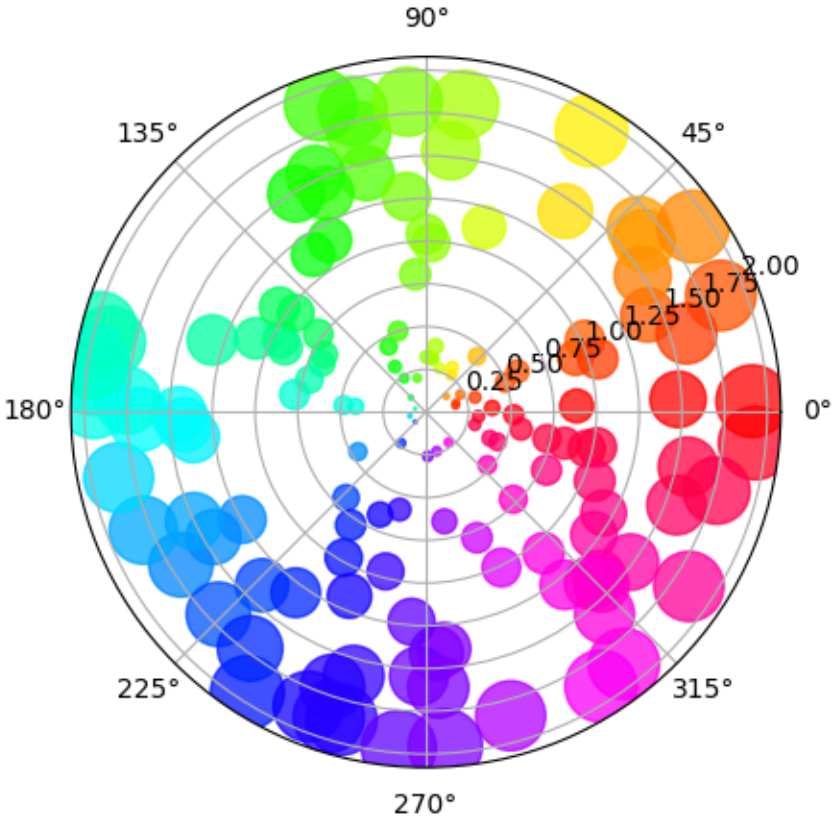
Size increases radially in this example and color increases with angle (just to verify the symbols are being scattered correctly).

```
import matplotlib.pyplot as plt
import numpy as np

# Fixing random state for reproducibility
```

(continues on next page)

(continued from previous page)

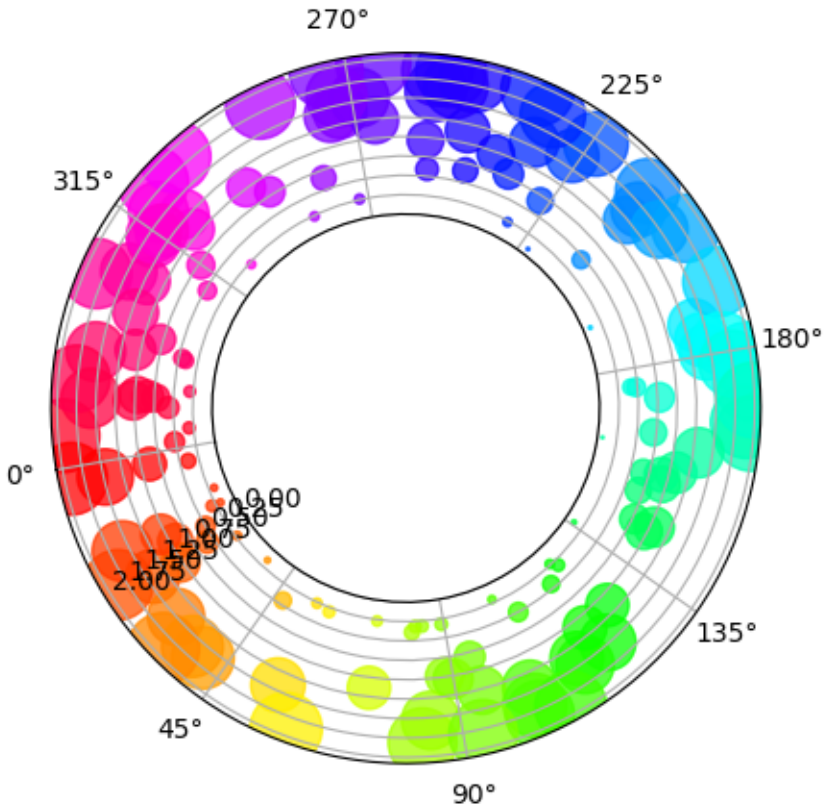
```
np.random.seed(19680801)

# Compute areas and colors
N = 150
r = 2 * np.random.rand(N)
theta = 2 * np.pi * np.random.rand(N)
area = 200 * r**2
colors = theta

fig = plt.figure()
ax = fig.add_subplot(projection='polar')
c = ax.scatter(theta, r, c=colors, s=area, cmap='hsv', alpha=0.75)
```



### Scatter plot on polar axis, with offset origin

The main difference with the previous plot is the configuration of the origin radius, producing an annulus. Additionally, the theta zero location is set to rotate the plot.

```
fig = plt.figure()
ax = fig.add_subplot(projection='polar')
c = ax.scatter(theta, r, c=colors, s=area, cmap='hsv', alpha=0.75)

ax.set_rorigin(-2.5)
ax.set_theta_zero_location('W', offset=10)
```




### Scatter plot on polar axis confined to a sector

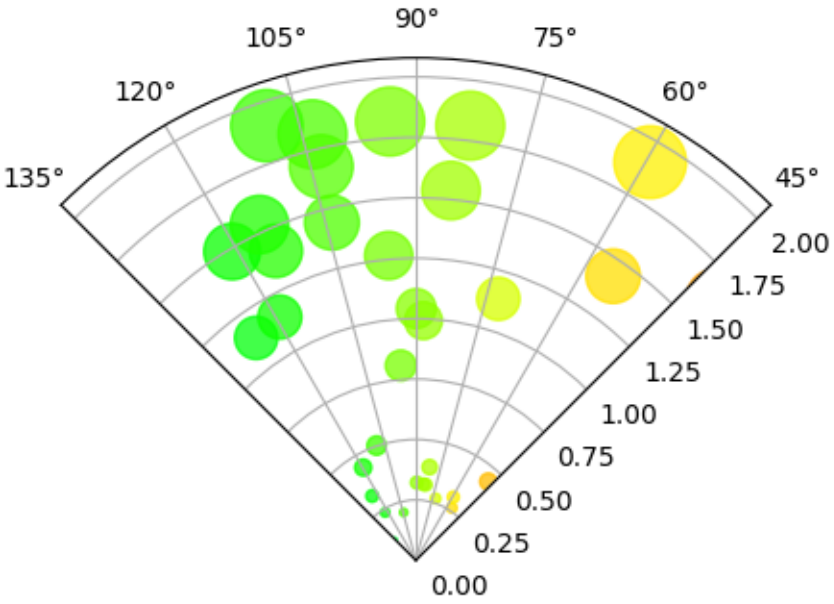
The main difference with the previous plots is the configuration of the theta start and end limits, producing a sector instead of a full circle.

```
fig = plt.figure()
ax = fig.add_subplot(projection='polar')
c = ax.scatter(theta, r, c=colors, s=area, cmap='hsv', alpha=0.75)
```

(continues on next page)

(continued from previous page)

```
ax.set_thetamin(45)
ax.set_thetamax(135)

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.scatter* / *matplotlib.pyplot.scatter*
- *matplotlib.projections.polar*
- *matplotlib.projections.polar.PolarAxes.set_rorigin*
- *matplotlib.projections.polar.PolarAxes.set_theta_zero_location*
- *matplotlib.projections.polar.PolarAxes.set_thetamin*
- *matplotlib.projections.polar.PolarAxes.set_thetamax*

**Total running time of the script:** (0 minutes 1.369 seconds)

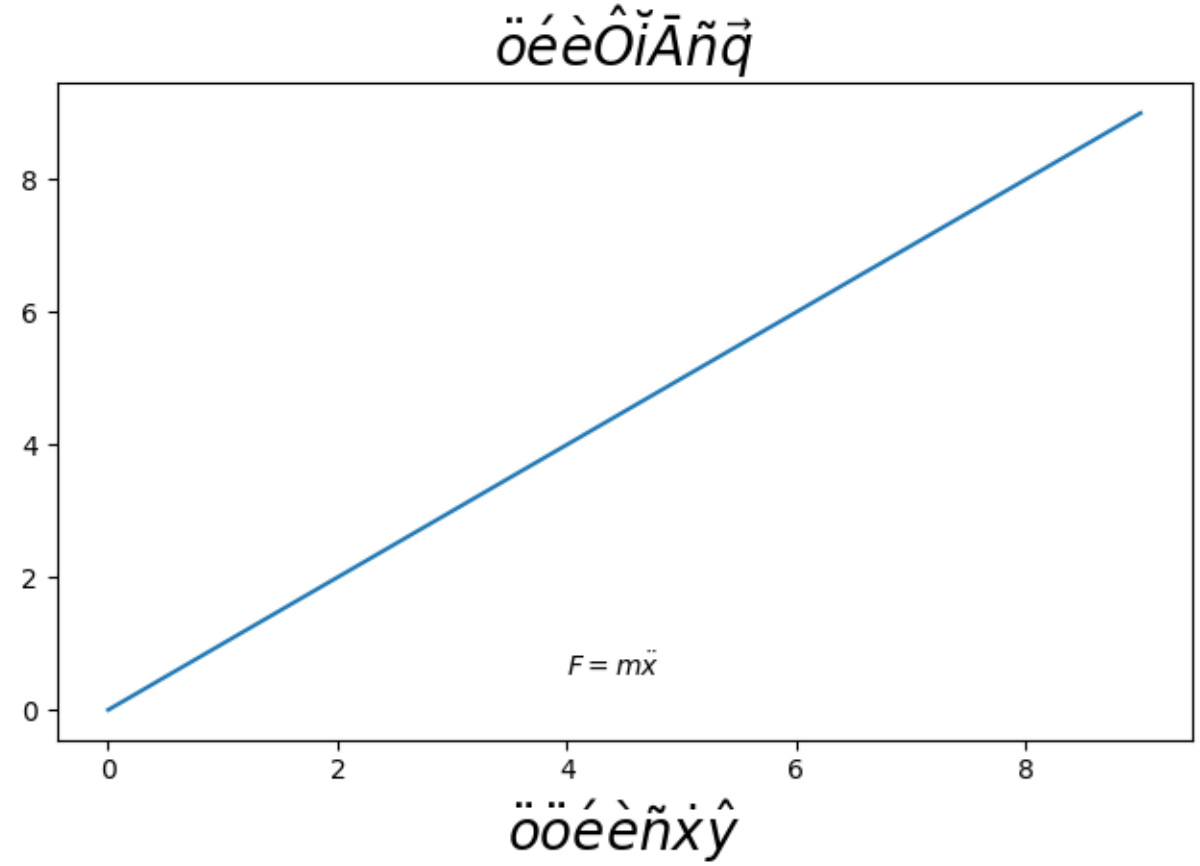
### 6.25.6 Text, labels and annotations

#### Accented text

Matplotlib supports accented characters via TeX mathtext or Unicode.

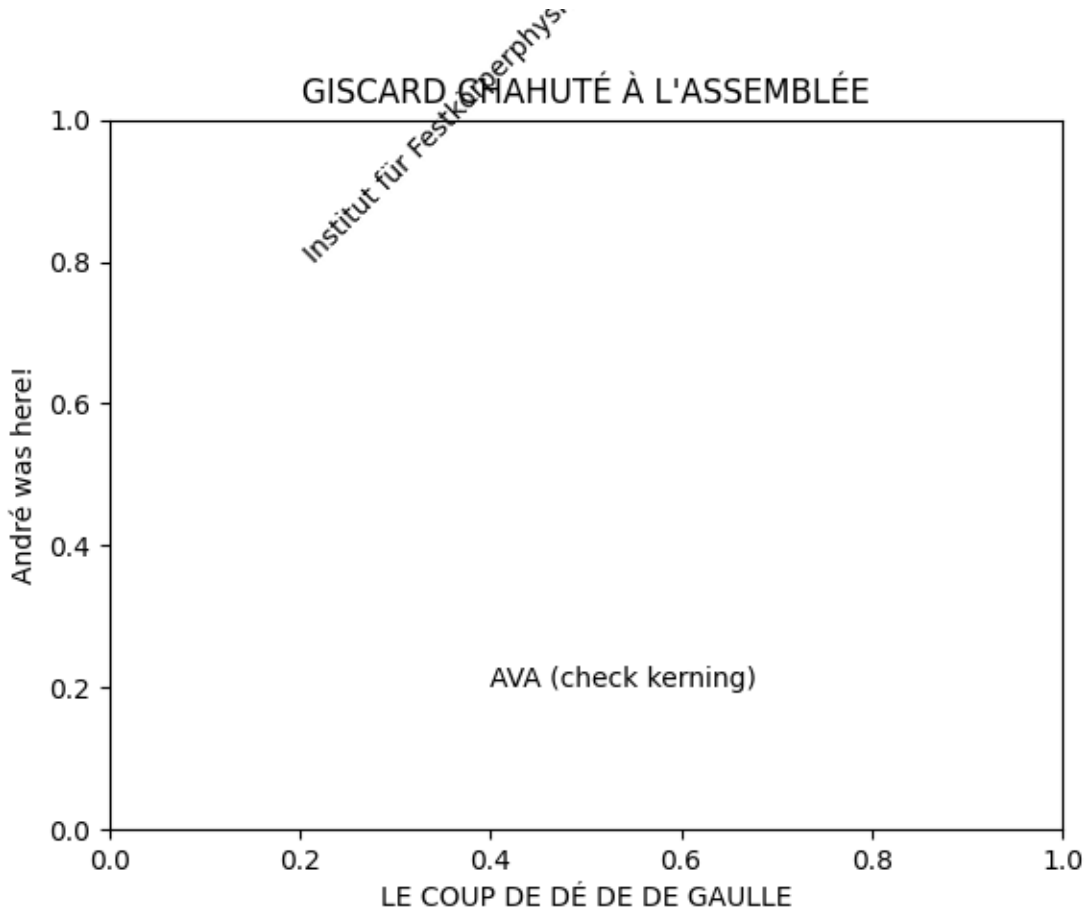
Using mathtext, the following accents are provided: \hat, \breve, \grave, \bar, \acute, \tilde, \vec, \dot, \ddot. All of them have the same syntax, e.g. \bar{o} yields "o overbar", \ddot{o} yields "o umlaut". Shortcuts such as \"o \'e \`e \~n \.x \^y are also supported.

```
import matplotlib.pyplot as plt

# Mathtext demo
fig, ax = plt.subplots()
ax.plot(range(10))
ax.set_title(r'$\ddot{o}\acute{e}\grave{e}\hat{O}'
             r'\breve{i}\bar{A}\tilde{n}\vec{q}$', fontsize=20)

# Shorthand is also supported and curly braces are optional
ax.set_xlabel(r"""$\"o\ddot o \'e\`e\~n\.x\^y$""", fontsize=20)
ax.text(4, 0.5, r"$F=m\ddot{x}$")
fig.tight_layout()
```



You can also use Unicode characters directly in strings.

```
fig, ax = plt.subplots()
ax.set_title("GISCARD CHAHUTÉ À L'ASSEMBLÉE")
ax.set_xlabel("LE COUP DE DÉ DE DE GAULLE")
ax.set_ylabel('André was here!')
ax.text(0.2, 0.8, 'Institut für Festkörperphysik', rotation=45)
ax.text(0.4, 0.2, 'AVA (check kerning)')

plt.show()
```




### Align y-labels

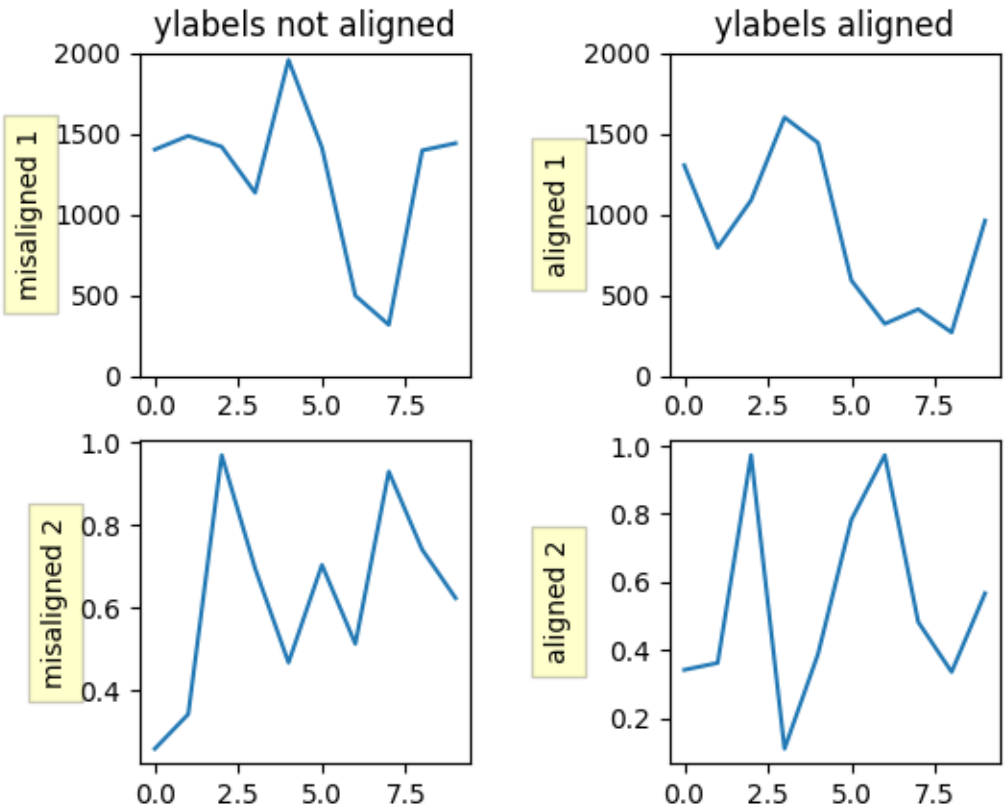
Two methods are shown here, one using a short call to *Figure.align_ylabels* and the second a manual way to align the labels.

```
import matplotlib.pyplot as plt
import numpy as np


def make_plot(axs):
    box = dict(facecolor='yellow', pad=5, alpha=0.2)
```

(continues on next page)

(continued from previous page)

```
    # Fixing random state for reproducibility
    np.random.seed(19680801)
    ax1 = axs[0, 0]
    ax1.plot(2000*np.random.rand(10))
    ax1.set_title('ylabels not aligned')
    ax1.set_ylabel('misaligned 1', bbox=box)
    ax1.set_ylim(0, 2000)

    ax3 = axs[1, 0]
    ax3.set_ylabel('misaligned 2', bbox=box)
    ax3.plot(np.random.rand(10))

    ax2 = axs[0, 1]
    ax2.set_title('ylabels aligned')
    ax2.plot(2000*np.random.rand(10))
    ax2.set_ylabel('aligned 1', bbox=box)
    ax2.set_ylim(0, 2000)

    ax4 = axs[1, 1]
    ax4.plot(np.random.rand(10))
    ax4.set_ylabel('aligned 2', bbox=box)


# Plot 1:
fig, axs = plt.subplots(2, 2)
fig.subplots_adjust(left=0.2, wspace=0.6)
make_plot(axs)

# just align the last column of Axes:
fig.align_ylabels(axs[:, 1])
plt.show()
```

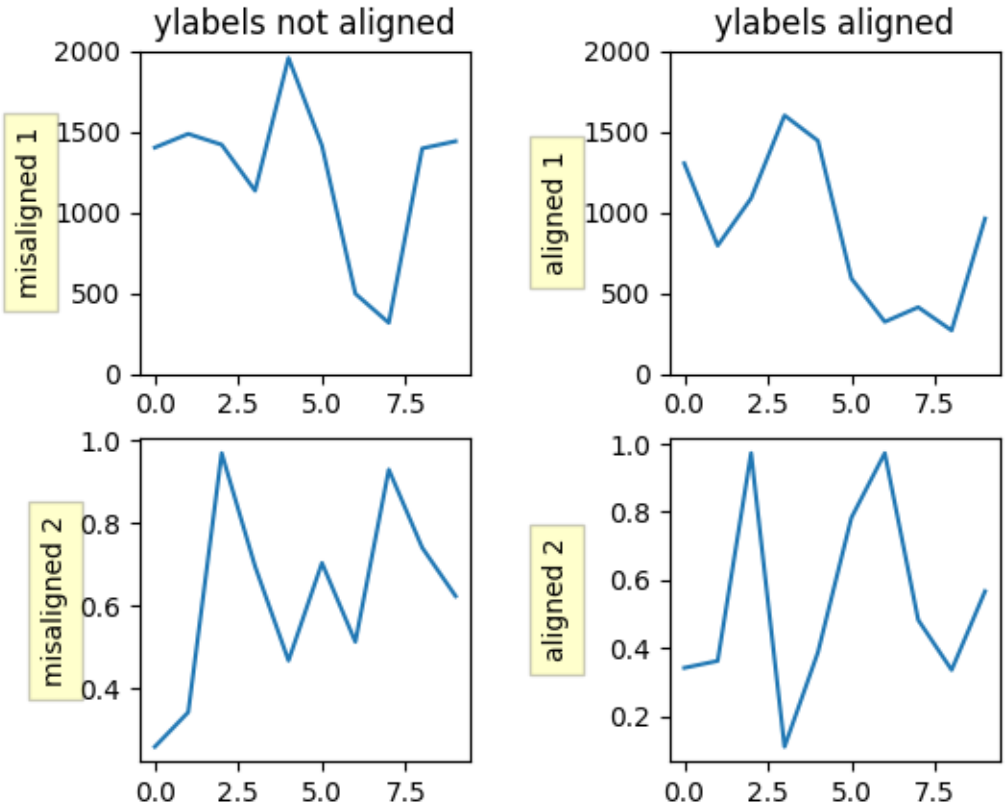



**See also:**

*Figure.align_ylabels* and *Figure.align_labels* for a direct method of doing the same thing. Also *Aligning Labels and Titles*

Or we can manually align the axis labels between subplots manually using the *set_label_coords* method of the y-axis object. Note this requires we know a good offset value which is hardcoded.

```
fig, axs = plt.subplots(2, 2)
fig.subplots_adjust(left=0.2, wspace=0.6)

make_plot(axs)

labelx = -0.3  # axes coords

for j in range(2):
    axs[j, 1].yaxis.set_label_coords(labelx, 0.5)

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.figure.Figure.align_ylabels*
- *matplotlib.axis.Axis.set_label_coords*
- *matplotlib.axes.Axes.plot* / *matplotlib.pyplot.plot*
- *matplotlib.axes.Axes.set_title*
- *matplotlib.axes.Axes.set_ylabel*
- *matplotlib.axes.Axes.set_ylim*

**Total running time of the script:** (0 minutes 1.322 seconds)

### Scale invariant angle label

This example shows how to create a scale invariant angle annotation. It is often useful to mark angles between lines or inside shapes with a circular arc. While Matplotlib provides an *Arc*, an inherent problem when directly using it for such purposes is that an arc being circular in data space is not necessarily circular in display space. Also, the arc's radius is often best defined in a coordinate system which is independent of the actual data coordinates - at least if you want to be able to freely zoom into your plot without the annotation growing to infinity.

This calls for a solution where the arc's center is defined in data space, but its radius in a physical unit like points or pixels, or as a ratio of the Axes dimension. The following `AngleAnnotation` class provides such solution.

The example below serves two purposes:

- It provides a ready-to-use solution for the problem of easily drawing angles in graphs.
- It shows how to subclass a Matplotlib artist to enhance its functionality, as well as giving a hands-on example on how to use Matplotlib's *transform system*.

If mainly interested in the former, you may copy the below class and jump to the *Usage* section.

### AngleAnnotation class

The essential idea here is to subclass *Arc* and set its transform to the *IdentityTransform*, making the parameters of the arc defined in pixel space. We then override the `Arc`'s attributes `_center`, `theta1`, `theta2`, `width` and `height` and make them properties, coupling to internal methods that calculate the respective parameters each time the attribute is accessed and thereby ensuring that the arc in pixel space stays synchronized with the input points and size. For example, each time the arc's drawing method would query its `_center` attribute, instead of receiving the same number all over again, it will instead receive the result of the `get_center_in_pixels` method we defined in the subclass. This method transforms the center in data coordinates to pixels via the Axes transform `ax.transData`. The size and the angles are calculated in a similar fashion, such that the arc changes its shape automatically when e.g. zooming or panning interactively.

The functionality of this class allows to annotate the arc with a text. This text is a *Annotation* stored in an attribute `text`. Since the arc's position and radius are defined only at draw time, we need to update the text's position accordingly. This is done by reimplementing the `Arc`'s `draw()` method to let it call an updating method for the text.

The arc and the text will be added to the provided Axes at instantiation: it is hence not strictly necessary to keep a reference to it.

```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.patches import Arc
from matplotlib.transforms import Bbox, IdentityTransform, TransformedBbox
```

(continues on next page)

(continued from previous page)

```
class AngleAnnotation(Arc):
    """
    Draws an arc between two vectors which appears circular in display space.
    """
    def __init__(self, xy, p1, p2, size=75, unit="points", ax=None,
                 text="", textposition="inside", text_kw=None, **kwargs):
        """
        Parameters
        ----------
        xy, p1, p2 : tuple or array of two floats
            Center position and two points. Angle annotation is drawn between
            the two vectors connecting *p1* and *p2* with *xy*, respectively.
            Units are data coordinates.

        size : float
            Diameter of the angle annotation in units specified by *unit*.

        unit : str
            One of the following strings to specify the unit of *size*:

            * "pixels": pixels
            * "points": points, use points instead of pixels to not have a
              dependence on the DPI
            * "axes width", "axes height": relative units of Axes width,␣
↪height
            * "axes min", "axes max": minimum or maximum of relative Axes
              width, height

        ax : `matplotlib.axes.Axes`
            The Axes to add the angle annotation to.

        text : str
            The text to mark the angle with.

        textposition : {"inside", "outside", "edge"}
            Whether to show the text in- or outside the arc. "edge" can be␣
↪used
            for custom positions anchored at the arc's edge.

        text_kw : dict
            Dictionary of arguments passed to the Annotation.

        **kwargs
            Further parameters are passed to `matplotlib.patches.Arc`. Use␣
↪this
            to specify, color, linewidth etc. of the arc.

        """
        self.ax = ax or plt.gca()
        self._xydata = xy  # in data coordinates
        self.vec1 = p1
        self.vec2 = p2
```

(continues on next page)

(continued from previous page)

```
        self.size = size
        self.unit = unit
        self.textposition = textposition

        super().__init__(self._xydata, size, size, angle=0.0,
                         theta1=self.theta1, theta2=self.theta2, **kwargs)

        self.set_transform(IdentityTransform())
        self.ax.add_patch(self)

        self.kw = dict(ha="center", va="center",
                       xycoords=IdentityTransform(),
                       xytext=(0, 0), textcoords="offset points",
                       annotation_clip=True)
        self.kw.update(text_kw or {})
        self.text = ax.annotate(text, xy=self._center, **self.kw)

    def get_size(self):
        factor = 1.
        if self.unit == "points":
            factor = self.ax.figure.dpi / 72.
        elif self.unit[:4] == "axes":
            b = TransformedBbox(Bbox.unit(), self.ax.transAxes)
            dic = {"max": max(b.width, b.height),
                   "min": min(b.width, b.height),
                   "width": b.width, "height": b.height}
            factor = dic[self.unit[5:]]
        return self.size * factor

    def set_size(self, size):
        self.size = size

    def get_center_in_pixels(self):
        """return center in pixels"""
        return self.ax.transData.transform(self._xydata)

    def set_center(self, xy):
        """set center in data coordinates"""
        self._xydata = xy

    def get_theta(self, vec):
        vec_in_pixels = self.ax.transData.transform(vec) - self._center
        return np.rad2deg(np.arctan2(vec_in_pixels[1], vec_in_pixels[0]))

    def get_theta1(self):
        return self.get_theta(self.vec1)

    def get_theta2(self):
        return self.get_theta(self.vec2)

    def set_theta(self, angle):
        pass
```

(continues on next page)

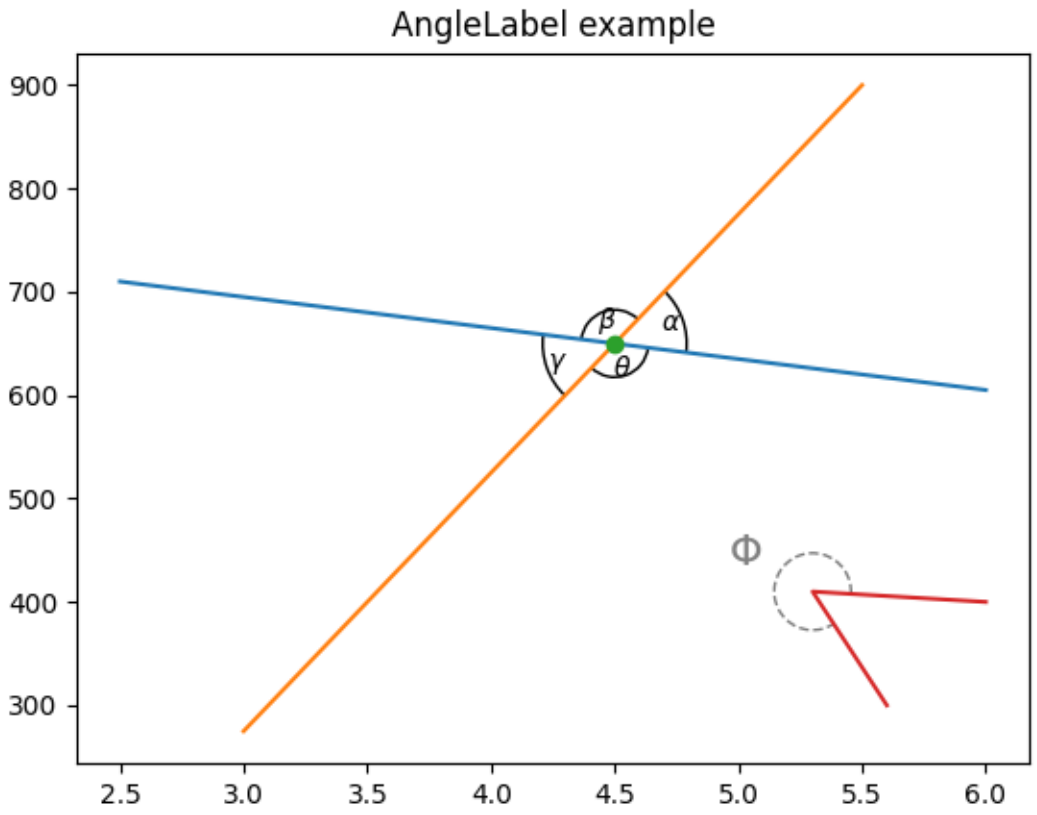
(continued from previous page)

```
    # Redefine attributes of the Arc to always give values in pixel space
    _center = property(get_center_in_pixels, set_center)
    theta1 = property(get_theta1, set_theta)
    theta2 = property(get_theta2, set_theta)
    width = property(get_size, set_size)
    height = property(get_size, set_size)

    # The following two methods are needed to update the text position.
    def draw(self, renderer):
        self.update_text()
        super().draw(renderer)

    def update_text(self):
        c = self._center
        s = self.get_size()
        angle_span = (self.theta2 - self.theta1) % 360
        angle = np.deg2rad(self.theta1 + angle_span / 2)
        r = s / 2
        if self.textposition == "inside":
            r = s / np.interp(angle_span, [60, 90, 135, 180],
                                          [3.3, 3.5, 3.8, 4])
        self.text.xy = c + r * np.array([np.cos(angle), np.sin(angle)])
        if self.textposition == "outside":
            def R90(a, r, w, h):
                if a < np.arctan(h/2/(r+w/2)):
                    return np.sqrt((r+w/2)**2 + (np.tan(a)*(r+w/2))**2)
                else:
                    c = np.sqrt((w/2)**2+(h/2)**2)
                    T = np.arcsin(c * np.cos(np.pi/2 - a + np.arcsin(h/2/c))/
↪r)
                    xy = r * np.array([np.cos(a + T), np.sin(a + T)])
                    xy += np.array([w/2, h/2])
                    return np.sqrt(np.sum(xy**2))

            def R(a, r, w, h):
                aa = (a % (np.pi/4))*((a % (np.pi/2)) <= np.pi/4) + \
                     (np.pi/4 - (a % (np.pi/4)))*((a % (np.pi/2)) >= np.pi/4)
                return R90(aa, r, *[w, h][::int(np.sign(np.cos(2*a)))])

            bbox = self.text.get_window_extent()
            X = R(angle, r, bbox.width, bbox.height)
            trans = self.ax.figure.dpi_scale_trans.inverted()
            offs = trans.transform(((X-s/2), 0))[0] * 72
            self.text.set_position([offs*np.cos(angle), offs*np.sin(angle)])
```

### Usage

Required arguments to `AngleAnnotation` are the center of the arc, *xy*, and two points, such that the arc spans between the two vectors connecting *p1* and *p2* with *xy*, respectively. Those are given in data coordinates. Further arguments are the *size* of the arc and its *unit*. Additionally, a *text* can be specified, that will be drawn either in- or outside of the arc, according to the value of *textposition*. Usage of those arguments is shown below.

```
fig, ax = plt.subplots()
fig.canvas.draw()  # Need to draw the figure to define renderer
ax.set_title("AngleLabel example")

# Plot two crossing lines and label each angle between them with the above
# ``AngleAnnotation`` tool.
center = (4.5, 650)
p1 = [(2.5, 710), (6.0, 605)]
p2 = [(3.0, 275), (5.5, 900)]
line1, = ax.plot(*zip(*p1))
line2, = ax.plot(*zip(*p2))
point, = ax.plot(*center, marker="o")

am1 = AngleAnnotation(center, p1[1], p2[1], ax=ax, size=75, text=r"$\alpha$")
am2 = AngleAnnotation(center, p2[1], p1[0], ax=ax, size=35, text=r"$\beta$")
am3 = AngleAnnotation(center, p1[0], p2[0], ax=ax, size=75, text=r"$\gamma$")
am4 = AngleAnnotation(center, p2[0], p1[1], ax=ax, size=35, text=r"$\theta$")


# Showcase some styling options for the angle arc, as well as the text.
p = [(6.0, 400), (5.3, 410), (5.6, 300)]
ax.plot(*zip(*p))
am5 = AngleAnnotation(p[1], p[0], p[2], ax=ax, size=40, text=r"$\Phi$",
                      linestyle="--", color="gray", textposition="outside",
                      text_kw=dict(fontsize=16, color="gray"))
```




### `AngleLabel` options

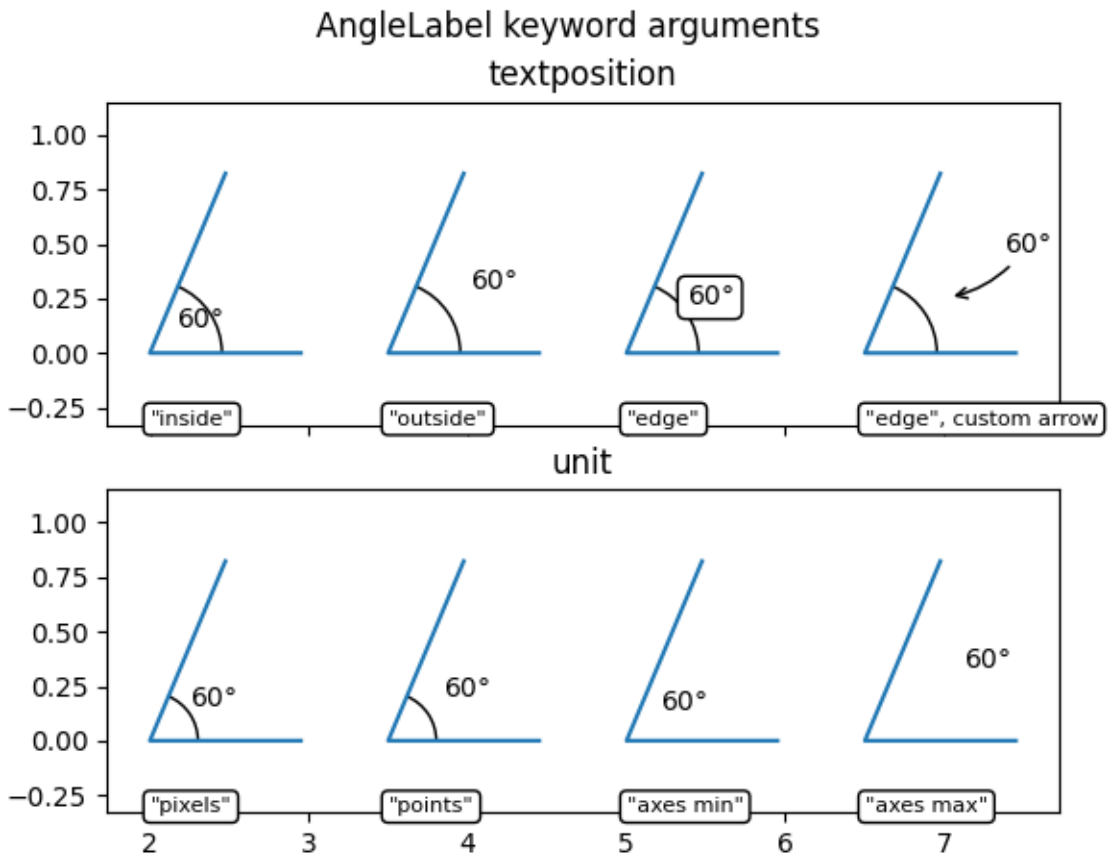
The *textposition* and *unit* keyword arguments may be used to modify the location of the text label, as shown below:

```
# Helper function to draw angle easily.
def plot_angle(ax, pos, angle, length=0.95, acol="C0", **kwargs):
    vec2 = np.array([np.cos(np.deg2rad(angle)), np.sin(np.deg2rad(angle))])
    xy = np.c_[[length, 0], [0, 0], vec2*length].T + np.array(pos)
    ax.plot(*xy.T, color=acol)
    return AngleAnnotation(pos, xy[0], xy[2], ax=ax, **kwargs)


fig, (ax1, ax2) = plt.subplots(nrows=2, sharex=True)
fig.suptitle("AngleLabel keyword arguments")
fig.canvas.draw()  # Need to draw the figure to define renderer

# Showcase different text positions.
ax1.margins(y=0.4)
ax1.set_title("textposition")
kw = dict(size=75, unit="points", text=r"$60°$")
```

(continues on next page)

(continued from previous page)

```
am6 = plot_angle(ax1, (2.0, 0), 60, textposition="inside", **kw)
am7 = plot_angle(ax1, (3.5, 0), 60, textposition="outside", **kw)
am8 = plot_angle(ax1, (5.0, 0), 60, textposition="edge",
                 text_kw=dict(bbox=dict(boxstyle="round", fc="w")), **kw)
am9 = plot_angle(ax1, (6.5, 0), 60, textposition="edge",
                 text_kw=dict(xytext=(30, 20), arrowprops=dict(arrowstyle="->
 ↪",
                                  connectionstyle="arc3,rad=-0.2")), **kw)

for x, text in zip([2.0, 3.5, 5.0, 6.5], ['"inside"', '"outside"', '"edge"',
                                          '"edge", custom arrow']):
    ax1.annotate(text, xy=(x, 0), xycoords=ax1.get_xaxis_transform(),
                 bbox=dict(boxstyle="round", fc="w"), ha="left", fontsize=8,
                 annotation_clip=True)

# Showcase different size units. The effect of this can best be observed
# by interactively changing the figure size
ax2.margins(y=0.4)
ax2.set_title("unit")
kw = dict(text=r"$60°$", textposition="outside")

am10 = plot_angle(ax2, (2.0, 0), 60, size=50, unit="pixels", **kw)
am11 = plot_angle(ax2, (3.5, 0), 60, size=50, unit="points", **kw)
am12 = plot_angle(ax2, (5.0, 0), 60, size=0.25, unit="axes min", **kw)
am13 = plot_angle(ax2, (6.5, 0), 60, size=0.25, unit="axes max", **kw)

for x, text in zip([2.0, 3.5, 5.0, 6.5], ['"pixels"', '"points"',
                                          '"axes min"', '"axes max"']):
    ax2.annotate(text, xy=(x, 0), xycoords=ax2.get_xaxis_transform(),
                 bbox=dict(boxstyle="round", fc="w"), ha="left", fontsize=8,
                 annotation_clip=True)

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.patches.Arc*
- *matplotlib.axes.Axes.annotate* / *matplotlib.pyplot.annotate*
- *matplotlib.text.Annotation*
- *matplotlib.transforms.IdentityTransform*
- *matplotlib.transforms.TransformedBbox*
- *matplotlib.transforms.Bbox*

### Angle annotations on bracket arrows

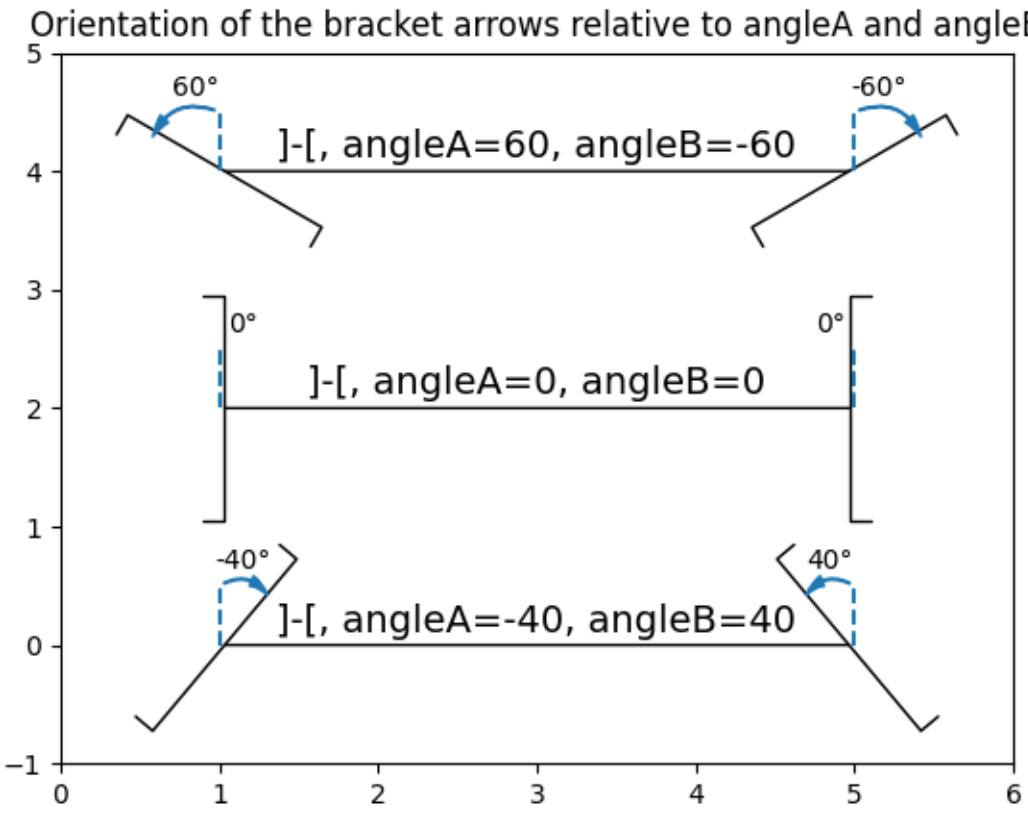
This example shows how to add angle annotations to bracket arrow styles created using *FancyArrowPatch*. *angleA* and *angleB* are measured from a vertical line as positive (to the left) or negative (to the right). Blue *FancyArrowPatch* arrows indicate the directions of *angleA* and *angleB* from the vertical and axes text annotate the angle sizes.

```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.patches import FancyArrowPatch


def get_point_of_rotated_vertical(origin, line_length, degrees):
    """Return xy coordinates of the vertical line end rotated by degrees."""
    rad = np.deg2rad(-degrees)
    return [origin[0] + line_length * np.sin(rad),
            origin[1] + line_length * np.cos(rad)]


fig, ax = plt.subplots()
ax.set(xlim=(0, 6), ylim=(-1, 5))
ax.set_title("Orientation of the bracket arrows relative to angleA and angleB
 ↪")

style = ']-['
for i, angle in enumerate([-40, 0, 60]):
    y = 2*i
    arrow_centers = ((1, y), (5, y))
    vlines = ((1, y + 0.5), (5, y + 0.5))
    anglesAB = (angle, -angle)
    bracketstyle = f"{style}, angleA={anglesAB[0]}, angleB={anglesAB[1]}"
    bracket = FancyArrowPatch(*arrow_centers, arrowstyle=bracketstyle,
                              mutation_scale=42)
    ax.add_patch(bracket)
    ax.text(3, y + 0.05, bracketstyle, ha="center", va="bottom", fontsize=14)
    ax.vlines([line[0] for line in vlines], [y, y], [line[1] for line in␣
 ↪vlines],
              linestyles="--", color="C0")
    # Get the top coordinates for the drawn patches at A and B
    patch_tops = [get_point_of_rotated_vertical(center, 0.5, angle)
                  for center, angle in zip(arrow_centers, anglesAB)]
    # Define the connection directions for the annotation arrows
    connection_dirs = (1, -1) if angle > 0 else (-1, 1)
    # Add arrows and annotation text
    arrowstyle = "Simple, tail_width=0.5, head_width=4, head_length=8"
    for vline, dir, patch_top, angle in zip(vlines, connection_dirs,
                                            patch_tops, anglesAB):
        kw = dict(connectionstyle=f"arc3,rad={dir * 0.5}",
                  arrowstyle=arrowstyle, color="C0")
        ax.add_patch(FancyArrowPatch(vline, patch_top, **kw))
        ax.text(vline[0] - dir * 0.15, y + 0.7, f'{angle}°', ha="center",
```

(continues on next page)

(continued from previous page)

```
                  va="center")

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.patches.ArrowStyle*

## Annotate Transform

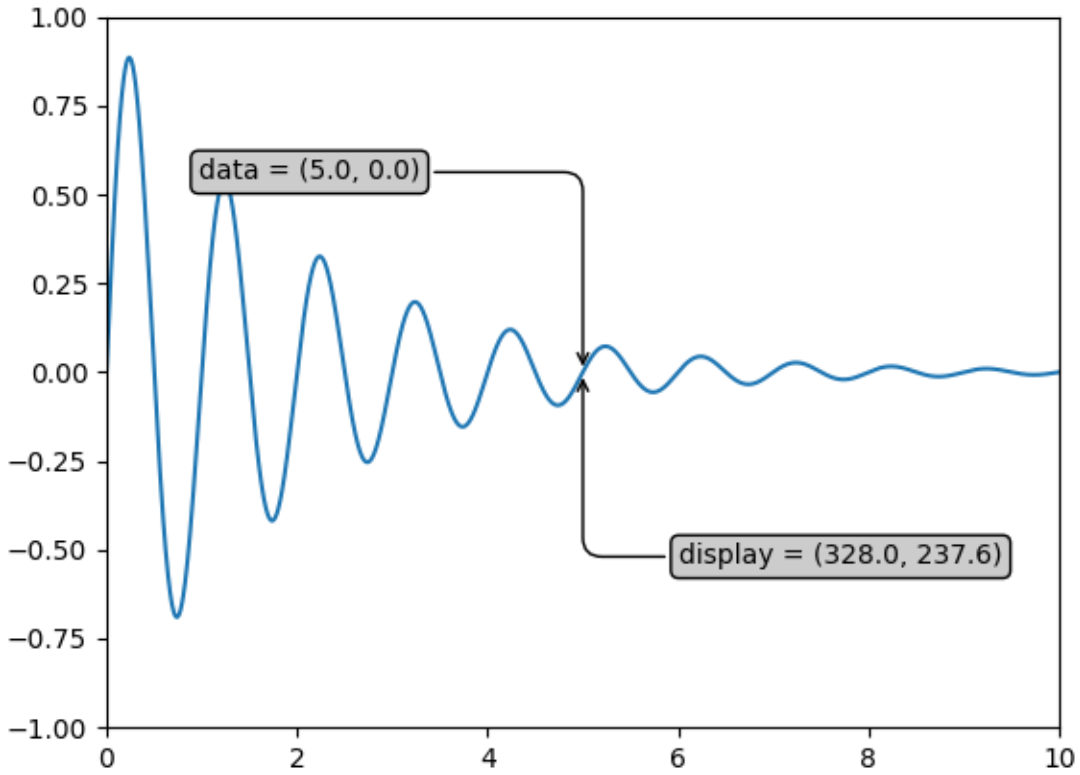
This example shows how to use different coordinate systems for annotations. For a complete overview of the annotation capabilities, also see the *annotation tutorial*.

```
import matplotlib.pyplot as plt
import numpy as np

x = np.arange(0, 10, 0.005)
```

(continues on next page)

(continued from previous page)

```
y = np.exp(-x/2.) * np.sin(2*np.pi*x)

fig, ax = plt.subplots()
ax.plot(x, y)
ax.set_xlim(0, 10)
ax.set_ylim(-1, 1)

xdata, ydata = 5, 0
xdisplay, ydisplay = ax.transData.transform((xdata, ydata))

bbox = dict(boxstyle="round", fc="0.8")
arrowprops = dict(
    arrowstyle="->",
    connectionstyle="angle,angleA=0,angleB=90,rad=10")

offset = 72
ax.annotate(
    f'data = ({xdata:.1f}, {ydata:.1f})',
    (xdata, ydata),
    xytext=(-2*offset, offset), textcoords='offset points',
    bbox=bbox, arrowprops=arrowprops)
ax.annotate(
    f'display = ({xdisplay:.1f}, {ydisplay:.1f})',
    xy=(xdisplay, ydisplay), xycoords='figure pixels',
    xytext=(0.5*offset, -offset), textcoords='offset points',
    bbox=bbox, arrowprops=arrowprops)

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.transforms.Transform.transform*
- *matplotlib.axes.Axes.annotate* / *matplotlib.pyplot.annotate*

## Annotating a plot

This example shows how to annotate a plot with an arrow pointing to provided coordinates. We modify the defaults of the arrow, to "shrink" it.

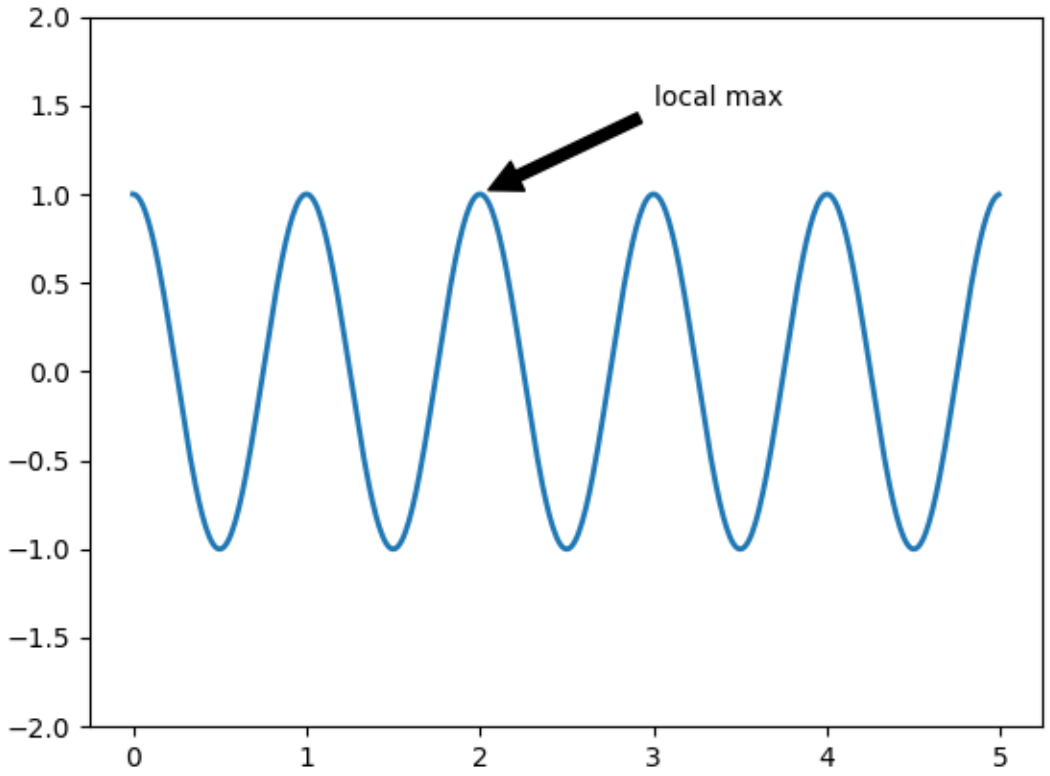
For a complete overview of the annotation capabilities, also see the *annotation tutorial*.

```
import matplotlib.pyplot as plt
import numpy as np

fig, ax = plt.subplots()

t = np.arange(0.0, 5.0, 0.01)
```

(continues on next page)

(continued from previous page)

```
s = np.cos(2*np.pi*t)
line, = ax.plot(t, s, lw=2)

ax.annotate('local max', xy=(2, 1), xytext=(3, 1.5),
            arrowprops=dict(facecolor='black', shrink=0.05),
            )
ax.set_ylim(-2, 2)
plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.annotate* / *matplotlib.pyplot.annotate*

### Annotating Plots

The following examples show ways to annotate plots in Matplotlib. This includes highlighting specific points of interest and using various visual tools to call attention to this point. For a more complete and in-depth description of the annotation and text tools in Matplotlib, see the *tutorial on annotation*.

```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.patches import Ellipse
from matplotlib.text import OffsetFrom
```

### Specifying text points and annotation points

You must specify an annotation point `xy=(x, y)` to annotate this point. Additionally, you may specify a text point `xytext=(x, y)` for the location of the text for this annotation. Optionally, you can specify the coordinate system of *xy* and *xytext* with one of the following strings for *xycoords* and *textcoords* (default is 'data'):

```
'figure points'   : points from the lower left corner of the figure
'figure pixels'   : pixels from the lower left corner of the figure
'figure fraction' : (0, 0) is lower left of figure and (1, 1) is upper right
'axes points'     : points from lower left corner of the Axes
'axes pixels'     : pixels from lower left corner of the Axes
'axes fraction'   : (0, 0) is lower left of Axes and (1, 1) is upper right
'offset points'   : Specify an offset (in points) from the xy value
'offset pixels'   : Specify an offset (in pixels) from the xy value
'data'            : use the Axes data coordinate system
```

Note: for physical coordinate systems (points or pixels) the origin is the (bottom, left) of the figure or Axes.

Optionally, you can specify arrow properties which draws and arrow from the text to the annotated point by giving a dictionary of arrow properties

Valid keys are:

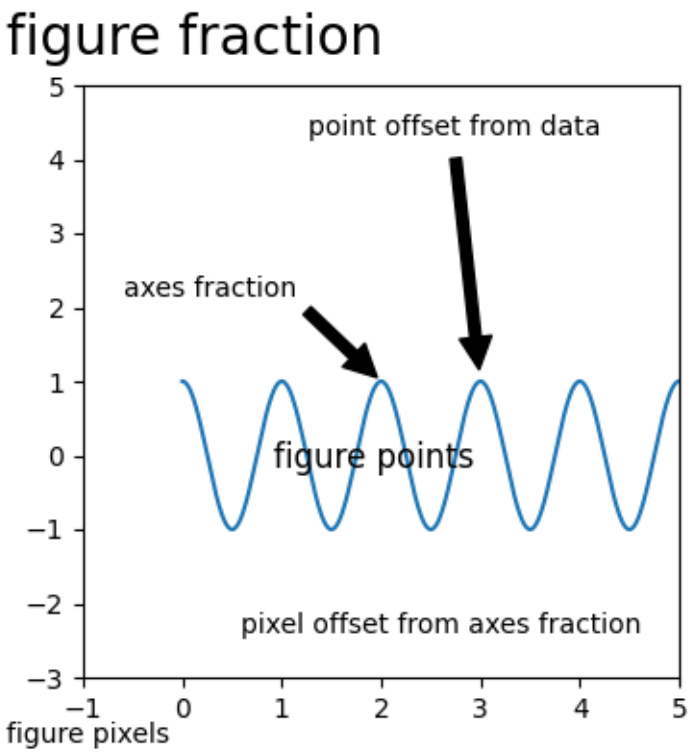
```
width : the width of the arrow in points
frac  : the fraction of the arrow length occupied by the head
headwidth : the width of the base of the arrow head in points
shrink : move the tip and base some percent away from the
         annotated point and text
any key for matplotlib.patches.polygon  (e.g., facecolor)
```

```
# Create our figure and data we'll use for plotting
fig, ax = plt.subplots(figsize=(4, 4))

t = np.arange(0.0, 5.0, 0.01)
s = np.cos(2*np.pi*t)

# Plot a line and add some simple annotations
```

(continues on next page)

(continued from previous page)

```
line, = ax.plot(t, s)
ax.annotate('figure pixels',
            xy=(10, 10), xycoords='figure pixels')
ax.annotate('figure points',
            xy=(107, 110), xycoords='figure points',
            fontsize=12)
ax.annotate('figure fraction',
            xy=(.025, .975), xycoords='figure fraction',
            horizontalalignment='left', verticalalignment='top',
            fontsize=20)

# The following examples show off how these arrows are drawn.

ax.annotate('point offset from data',
            xy=(3, 1), xycoords='data',
            xytext=(-10, 90), textcoords='offset points',
            arrowprops=dict(facecolor='black', shrink=0.05),
            horizontalalignment='center', verticalalignment='bottom')

ax.annotate('axes fraction',
            xy=(2, 1), xycoords='data',
            xytext=(0.36, 0.68), textcoords='axes fraction',
            arrowprops=dict(facecolor='black', shrink=0.05),
            horizontalalignment='right', verticalalignment='top')

# You may also use negative points or pixels to specify from (right, top).
# E.g., (-10, 10) is 10 points to the left of the right side of the Axes and␣
 ↪10
# points above the bottom

ax.annotate('pixel offset from axes fraction',
            xy=(1, 0), xycoords='axes fraction',
            xytext=(-20, 20), textcoords='offset pixels',
            horizontalalignment='right',
            verticalalignment='bottom')

ax.set(xlim=(-1, 5), ylim=(-3, 5))
```

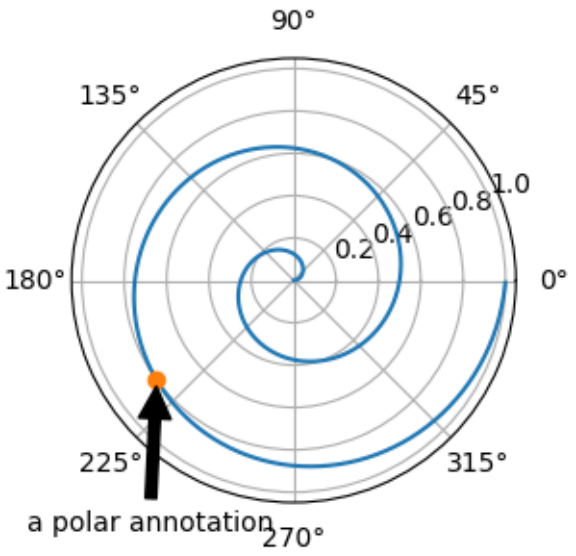



### Using multiple coordinate systems and axis types

You can specify the *xypoint* and the *xytext* in different positions and coordinate systems, and optionally turn on a connecting line and mark the point with a marker. Annotations work on polar Axes too.

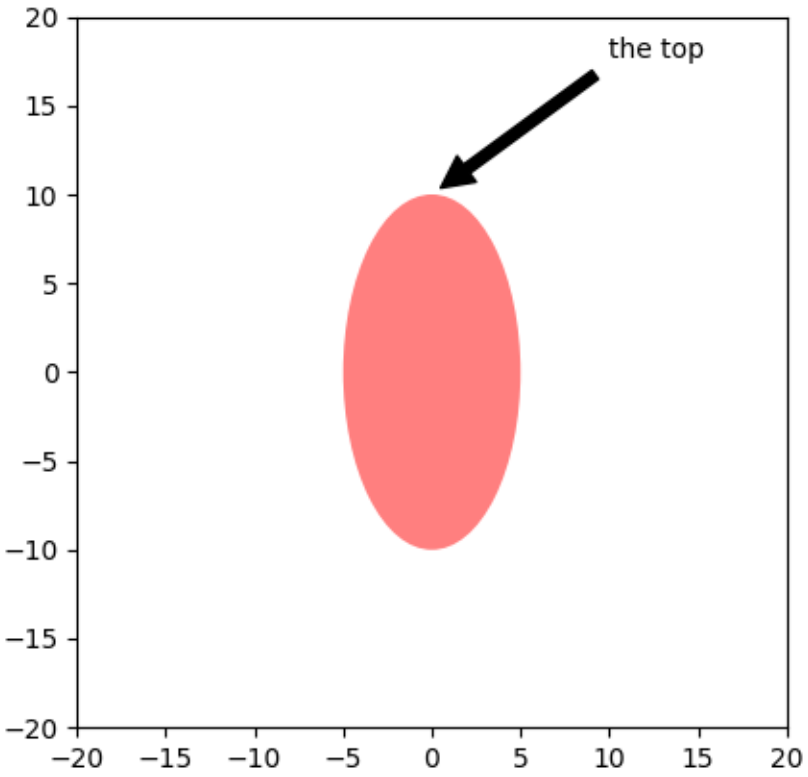
In the example below, the *xy* point is in native coordinates (*xycoords* defaults to 'data'). For a polar Axes, this is in (theta, radius) space. The text in the example is placed in the fractional figure coordinate system. Text keyword arguments like horizontal and vertical alignment are respected.

```
fig, ax = plt.subplots(subplot_kw=dict(projection='polar'), figsize=(3, 3))
r = np.arange(0, 1, 0.001)
theta = 2*2*np.pi*r
line, = ax.plot(theta, r)

ind = 800
thisr, thistheta = r[ind], theta[ind]
ax.plot([thistheta], [thisr], 'o')
ax.annotate('a polar annotation',
            xy=(thistheta, thisr),  # theta, radius
            xytext=(0.05, 0.05),    # fraction, fraction
            textcoords='figure fraction',
            arrowprops=dict(facecolor='black', shrink=0.05),
            horizontalalignment='left',
            verticalalignment='bottom')
```




You can also use polar notation on a cartesian Axes. Here the native coordinate system ('data') is cartesian, so you need to specify the xycoords and textcoords as 'polar' if you want to use (theta, radius).

```
el = Ellipse((0, 0), 10, 20, facecolor='r', alpha=0.5)

fig, ax = plt.subplots(subplot_kw=dict(aspect='equal'))
ax.add_artist(el)
el.set_clip_box(ax.bbox)
ax.annotate('the top',
            xy=(np.pi/2., 10.),      # theta, radius
            xytext=(np.pi/3, 20.),   # theta, radius
            xycoords='polar',
            textcoords='polar',
            arrowprops=dict(facecolor='black', shrink=0.05),
            horizontalalignment='left',
            verticalalignment='bottom',
            clip_on=True)  # clip to the Axes bounding box

ax.set(xlim=[-20, 20], ylim=[-20, 20])
```




### Customizing arrow and bubble styles

The arrow between *xytext* and the annotation point, as well as the bubble that covers the annotation text, are highly customizable. Below are a few parameter options as well as their resulting output.

```
fig, ax = plt.subplots(figsize=(8, 5))

t = np.arange(0.0, 5.0, 0.01)
s = np.cos(2*np.pi*t)
line, = ax.plot(t, s, lw=3)

ax.annotate(
    'straight',
    xy=(0, 1), xycoords='data',
    xytext=(-50, 30), textcoords='offset points',
    arrowprops=dict(arrowstyle="->"))
ax.annotate(
    'arc3,\nrad 0.2',
    xy=(0.5, -1), xycoords='data',
    xytext=(-80, -60), textcoords='offset points',
    arrowprops=dict(arrowstyle="->",
```

(continues on next page)

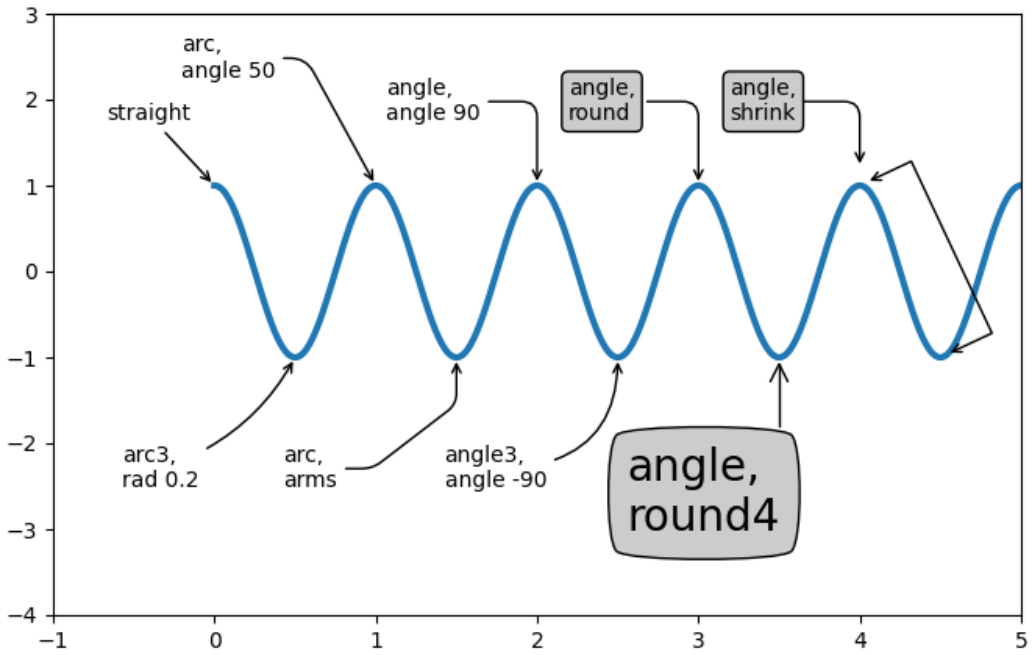
(continued from previous page)

```
                    connectionstyle="arc3,rad=.2"))
ax.annotate(
    'arc,\nangle 50',
    xy=(1., 1), xycoords='data',
    xytext=(-90, 50), textcoords='offset points',
    arrowprops=dict(arrowstyle="->",
                    connectionstyle="arc,angleA=0,armA=50,rad=10"))
ax.annotate(
    'arc,\narms',
    xy=(1.5, -1), xycoords='data',
    xytext=(-80, -60), textcoords='offset points',
    arrowprops=dict(
        arrowstyle="->",
        connectionstyle="arc,angleA=0,armA=40,angleB=-90,armB=30,rad=7"))
ax.annotate(
    'angle,\nangle 90',
    xy=(2., 1), xycoords='data',
    xytext=(-70, 30), textcoords='offset points',
    arrowprops=dict(arrowstyle="->",
                    connectionstyle="angle,angleA=0,angleB=90,rad=10"))
ax.annotate(
    'angle3,\nangle -90',
    xy=(2.5, -1), xycoords='data',
    xytext=(-80, -60), textcoords='offset points',
    arrowprops=dict(arrowstyle="->",
                    connectionstyle="angle3,angleA=0,angleB=-90"))
ax.annotate(
    'angle,\nround',
    xy=(3., 1), xycoords='data',
    xytext=(-60, 30), textcoords='offset points',
    bbox=dict(boxstyle="round", fc="0.8"),
    arrowprops=dict(arrowstyle="->",
                    connectionstyle="angle,angleA=0,angleB=90,rad=10"))
ax.annotate(
    'angle,\nround4',
    xy=(3.5, -1), xycoords='data',
    xytext=(-70, -80), textcoords='offset points',
    size=20,
    bbox=dict(boxstyle="round4,pad=.5", fc="0.8"),
    arrowprops=dict(arrowstyle="->",
                    connectionstyle="angle,angleA=0,angleB=-90,rad=10"))
ax.annotate(
    'angle,\nshrink',
    xy=(4., 1), xycoords='data',
    xytext=(-60, 30), textcoords='offset points',
    bbox=dict(boxstyle="round", fc="0.8"),
    arrowprops=dict(arrowstyle="->",
                    shrinkA=0, shrinkB=10,
                    connectionstyle="angle,angleA=0,angleB=90,rad=10"))
# You can pass an empty string to get only annotation arrows rendered
ax.annotate('', xy=(4., 1.), xycoords='data',
            xytext=(4.5, -1), textcoords='data',
```

(continues on next page)

(continued from previous page)

```
            arrowprops=dict(arrowstyle="<->",
                            connectionstyle="bar",
                            ec="k",
                            shrinkA=5, shrinkB=5))

ax.set(xlim=(-1, 5), ylim=(-4, 3))
```



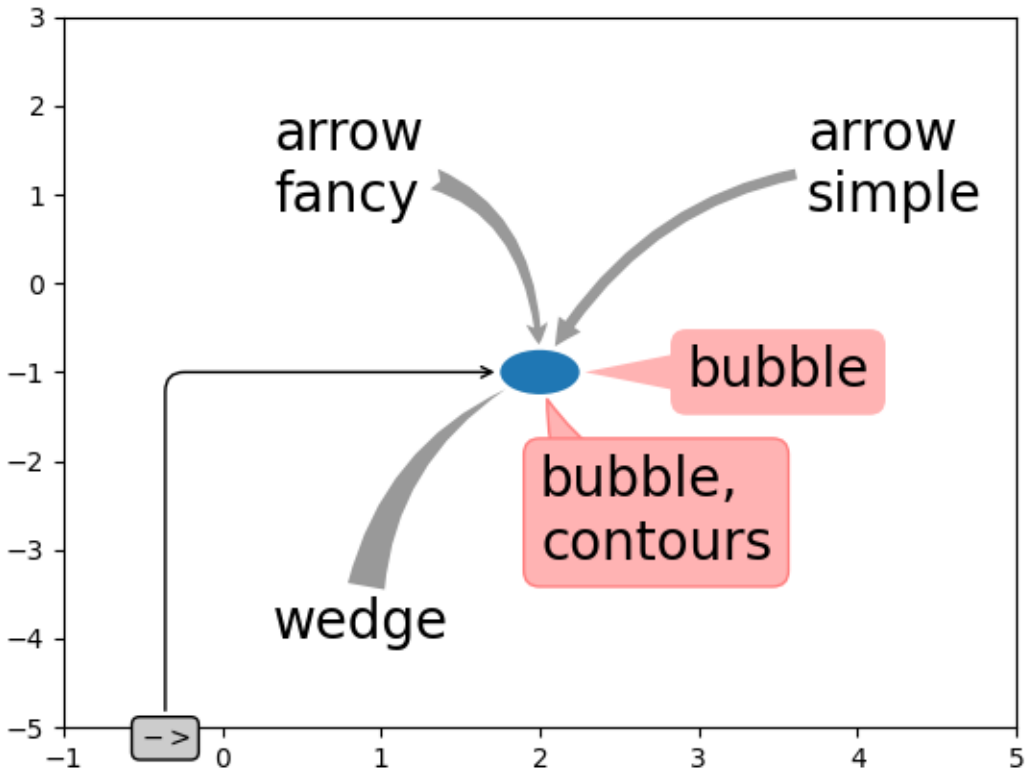

We'll create another figure so that it doesn't get too cluttered

```
fig, ax = plt.subplots()

el = Ellipse((2, -1), 0.5, 0.5)
ax.add_patch(el)

ax.annotate('$->$',
            xy=(2., -1), xycoords='data',
            xytext=(-150, -140), textcoords='offset points',
            bbox=dict(boxstyle="round", fc="0.8"),
            arrowprops=dict(arrowstyle="->",
                            patchB=el,
                            connectionstyle="angle,angleA=90,angleB=0,rad=10
 ↪"))
ax.annotate('arrow\nfancy',
            xy=(2., -1), xycoords='data',
            xytext=(-100, 60), textcoords='offset points',
            size=20,
```

(continues on next page)

(continued from previous page)

```
            arrowprops=dict(arrowstyle="fancy",
                            fc="0.6", ec="none",
                            patchB=el,
                            connectionstyle="angle3,angleA=0,angleB=-90"))
ax.annotate('arrow\nsimple',
            xy=(2., -1), xycoords='data',
            xytext=(100, 60), textcoords='offset points',
            size=20,
            arrowprops=dict(arrowstyle="simple",
                            fc="0.6", ec="none",
                            patchB=el,
                            connectionstyle="arc3,rad=0.3"))
ax.annotate('wedge',
            xy=(2., -1), xycoords='data',
            xytext=(-100, -100), textcoords='offset points',
            size=20,
            arrowprops=dict(arrowstyle="wedge,tail_width=0.7",
                            fc="0.6", ec="none",
                            patchB=el,
                            connectionstyle="arc3,rad=-0.3"))
ax.annotate('bubble,\ncontours',
            xy=(2., -1), xycoords='data',
            xytext=(0, -70), textcoords='offset points',
            size=20,
            bbox=dict(boxstyle="round",
                      fc=(1.0, 0.7, 0.7),
                      ec=(1., .5, .5)),
            arrowprops=dict(arrowstyle="wedge,tail_width=1.",
                            fc=(1.0, 0.7, 0.7), ec=(1., .5, .5),
                            patchA=None,
                            patchB=el,
                            relpos=(0.2, 0.8),
                            connectionstyle="arc3,rad=-0.1"))
ax.annotate('bubble',
            xy=(2., -1), xycoords='data',
            xytext=(55, 0), textcoords='offset points',
            size=20, va="center",
            bbox=dict(boxstyle="round", fc=(1.0, 0.7, 0.7), ec="none"),
            arrowprops=dict(arrowstyle="wedge,tail_width=1.",
                            fc=(1.0, 0.7, 0.7), ec="none",
                            patchA=None,
                            patchB=el,
                            relpos=(0.2, 0.5)))

ax.set(xlim=(-1, 5), ylim=(-5, 3))
```




### More examples of coordinate systems

Below we'll show a few more examples of coordinate systems and how the location of annotations may be specified.

```
fig, (ax1, ax2) = plt.subplots(1, 2)

bbox_args = dict(boxstyle="round", fc="0.8")
arrow_args = dict(arrowstyle="->")

# Here we'll demonstrate the extents of the coordinate system and how
# we place annotating text.

ax1.annotate('figure fraction : 0, 0', xy=(0, 0), xycoords='figure fraction',
             xytext=(20, 20), textcoords='offset points',
             ha="left", va="bottom",
             bbox=bbox_args,
             arrowprops=arrow_args)

ax1.annotate('figure fraction : 1, 1', xy=(1, 1), xycoords='figure fraction',
             xytext=(-20, -20), textcoords='offset points',
```

(continues on next page)

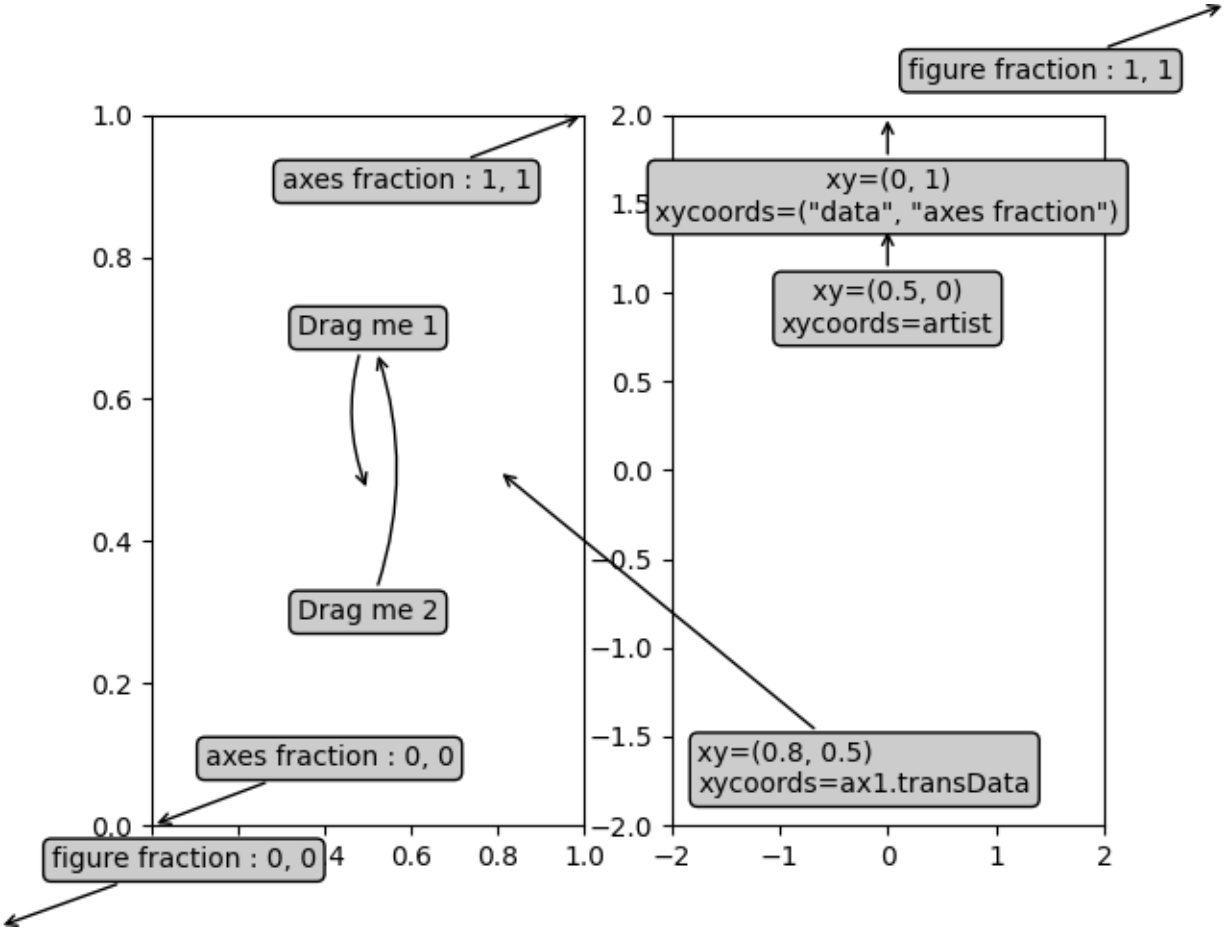
(continued from previous page)

```
            ha="right", va="top",
            bbox=bbox_args,
            arrowprops=arrow_args)

ax1.annotate('axes fraction : 0, 0', xy=(0, 0), xycoords='axes fraction',
            xytext=(20, 20), textcoords='offset points',
            ha="left", va="bottom",
            bbox=bbox_args,
            arrowprops=arrow_args)

ax1.annotate('axes fraction : 1, 1', xy=(1, 1), xycoords='axes fraction',
            xytext=(-20, -20), textcoords='offset points',
            ha="right", va="top",
            bbox=bbox_args,
            arrowprops=arrow_args)

# It is also possible to generate draggable annotations

an1 = ax1.annotate('Drag me 1', xy=(.5, .7), xycoords='data',
                   ha="center", va="center",
                   bbox=bbox_args)

an2 = ax1.annotate('Drag me 2', xy=(.5, .5), xycoords=an1,
                   xytext=(.5, .3), textcoords='axes fraction',
                   ha="center", va="center",
                   bbox=bbox_args,
                   arrowprops=dict(patchB=an1.get_bbox_patch(),
                                   connectionstyle="arc3,rad=0.2",
                                   **arrow_args))
an1.draggable()
an2.draggable()

an3 = ax1.annotate('', xy=(.5, .5), xycoords=an2,
                   xytext=(.5, .5), textcoords=an1,
                   ha="center", va="center",
                   bbox=bbox_args,
                   arrowprops=dict(patchA=an1.get_bbox_patch(),
                                   patchB=an2.get_bbox_patch(),
                                   connectionstyle="arc3,rad=0.2",
                                   **arrow_args))

# Finally we'll show off some more complex annotation and placement

text = ax2.annotate('xy=(0, 1)\nxycoords=("data", "axes fraction")',
                    xy=(0, 1), xycoords=("data", 'axes fraction'),
                    xytext=(0, -20), textcoords='offset points',
                    ha="center", va="top",
                    bbox=bbox_args,
                    arrowprops=arrow_args)

ax2.annotate('xy=(0.5, 0)\nxycoords=artist',
            xy=(0.5, 0.), xycoords=text,
```

(continues on next page)

(continued from previous page)

```
            xytext=(0, -20), textcoords='offset points',
            ha="center", va="top",
            bbox=bbox_args,
            arrowprops=arrow_args)

ax2.annotate('xy=(0.8, 0.5)\nxycoords=ax1.transData',
            xy=(0.8, 0.5), xycoords=ax1.transData,
            xytext=(10, 10),
            textcoords=OffsetFrom(ax2.bbox, (0, 0), "points"),
            ha="left", va="bottom",
            bbox=bbox_args,
            arrowprops=arrow_args)

ax2.set(xlim=[-2, 2], ylim=[-2, 2])
plt.show()
```



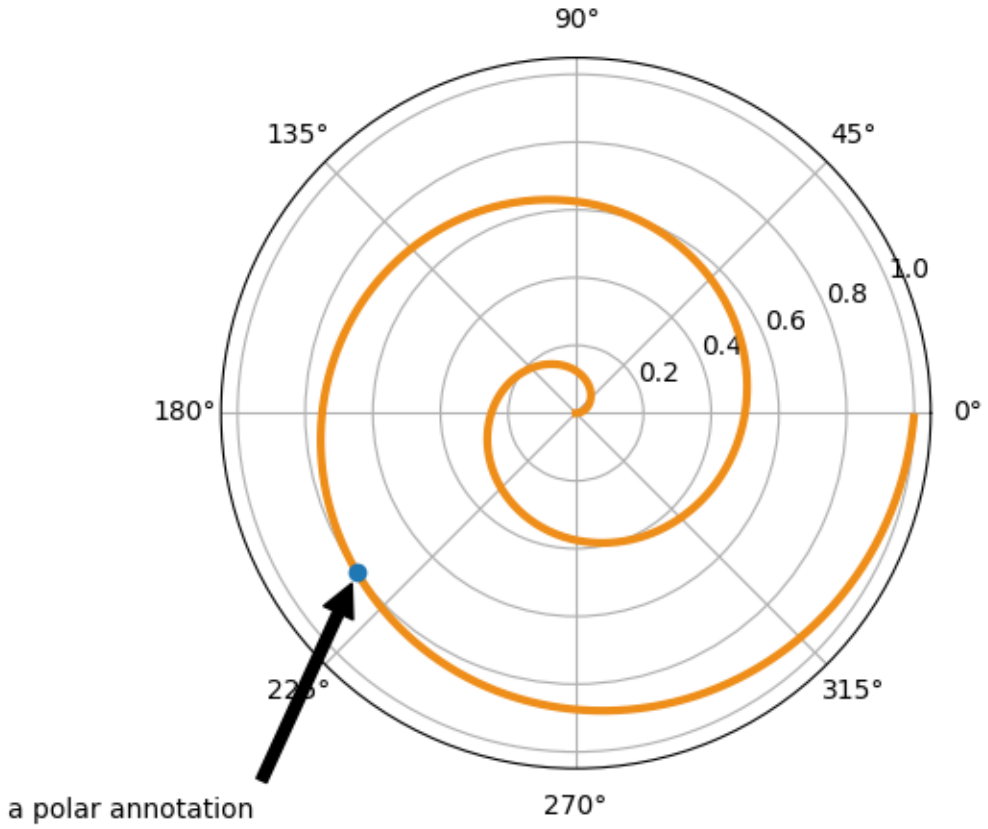

**Total running time of the script:** (0 minutes 1.939 seconds)

### Annotation Polar

This example shows how to create an annotation on a polar graph.

For a complete overview of the annotation capabilities, also see the *Annotations*.

```
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure()
ax = fig.add_subplot(projection='polar')
r = np.arange(0, 1, 0.001)
theta = 2 * 2*np.pi * r
line, = ax.plot(theta, r, color='#ee8d18', lw=3)

ind = 800
thisr, thistheta = r[ind], theta[ind]
ax.plot([thistheta], [thisr], 'o')
ax.annotate('a polar annotation',
            xy=(thistheta, thisr),  # theta, radius
            xytext=(0.05, 0.05),    # fraction, fraction
            textcoords='figure fraction',
            arrowprops=dict(facecolor='black', shrink=0.05),
            horizontalalignment='left',
            verticalalignment='bottom',
            )
plt.show()
```



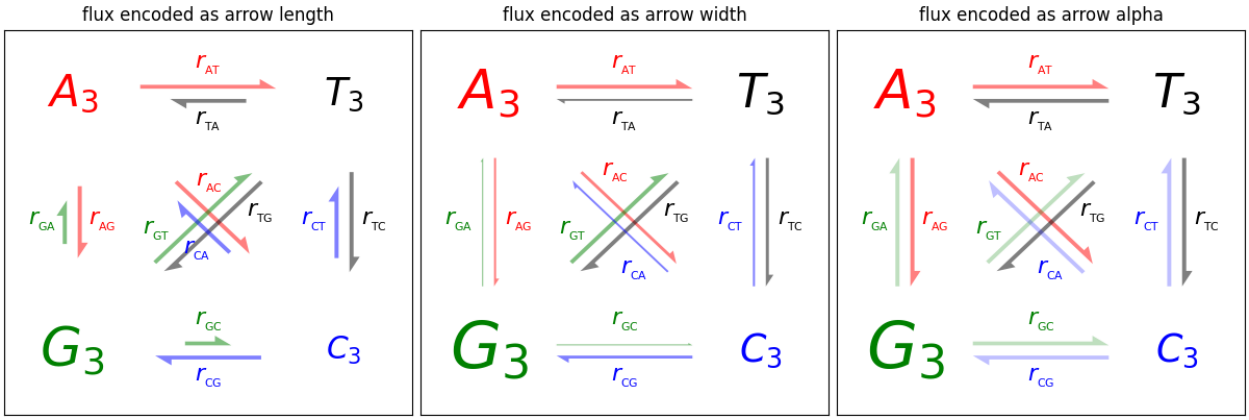

**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.projections.polar*
- *matplotlib.axes.Axes.annotate* / *matplotlib.pyplot.annotate*

### Arrow Demo

Three ways of drawing arrows to encode arrow "strength" (e.g., transition probabilities in a Markov model) using arrow length, width, or alpha (opacity).




```
import itertools

import matplotlib.pyplot as plt
import numpy as np


def make_arrow_graph(ax, data, size=4, display='length', shape='right',
                     max_arrow_width=0.03, arrow_sep=0.02, alpha=0.5,
                     normalize_data=False, ec=None, labelcolor=None,
                     **kwargs):
    """
    Makes an arrow plot.

    Parameters
    ----------
    ax
        The Axes where the graph is drawn.
    data
        Dict with probabilities for the bases and pair transitions.
    size
        Size of the plot, in inches.
    display : {'length', 'width', 'alpha'}
        The arrow property to change.
    shape : {'full', 'left', 'right'}
        For full or half arrows.
    max_arrow_width : float
        Maximum width of an arrow, in data coordinates.
    arrow_sep : float
        Separation between arrows in a pair, in data coordinates.
    alpha : float
        Maximum opacity of arrows.
    **kwargs
        `.FancyArrow` properties, e.g. *linewidth* or *edgecolor*.
    """
```

(continues on next page)

(continued from previous page)

```
    ax.set(xlim=(-0.25, 1.25), ylim=(-0.25, 1.25), xticks=[], yticks=[],
           title=f'flux encoded as arrow {display}')
    max_text_size = size * 12
    min_text_size = size
    label_text_size = size * 4

    bases = 'ATGC'
    coords = {
        'A': np.array([0, 1]),
        'T': np.array([1, 1]),
        'G': np.array([0, 0]),
        'C': np.array([1, 0]),
    }
    colors = {'A': 'r', 'T': 'k', 'G': 'g', 'C': 'b'}

    for base in bases:
        fontsize = np.clip(max_text_size * data[base]**(1/2),
                           min_text_size, max_text_size)
        ax.text(*coords[base], f'${base}_3$',
                color=colors[base], size=fontsize,
                horizontalalignment='center', verticalalignment='center',
                weight='bold')

    arrow_h_offset = 0.25  # data coordinates, empirically determined
    max_arrow_length = 1 - 2 * arrow_h_offset
    max_head_width = 2.5 * max_arrow_width
    max_head_length = 2 * max_arrow_width
    sf = 0.6  # max arrow size represents this in data coords

    if normalize_data:
        # find maximum value for rates, i.e. where keys are 2 chars long
        max_val = max((v for k, v in data.items() if len(k) == 2), default=0)
        # divide rates by max val, multiply by arrow scale factor
        for k, v in data.items():
            data[k] = v / max_val * sf

    # iterate over strings 'AT', 'TA', 'AG', 'GA', etc.
    for pair in map(''.join, itertools.permutations(bases, 2)):
        # set the length of the arrow
        if display == 'length':
            length = (max_head_length
                      + data[pair] / sf * (max_arrow_length - max_head_
↪length))
        else:
            length = max_arrow_length
        # set the transparency of the arrow
        if display == 'alpha':
            alpha = min(data[pair] / sf, alpha)
        # set the width of the arrow
        if display == 'width':
            scale = data[pair] / sf
```

(continues on next page)

(continued from previous page)

```
            width = max_arrow_width * scale
            head_width = max_head_width * scale
            head_length = max_head_length * scale
        else:
            width = max_arrow_width
            head_width = max_head_width
            head_length = max_head_length

        fc = colors[pair[0]]

        cp0 = coords[pair[0]]
        cp1 = coords[pair[1]]
        # unit vector in arrow direction
        delta = cos, sin = (cp1 - cp0) / np.hypot(*(cp1 - cp0))
        x_pos, y_pos = (
            (cp0 + cp1) / 2  # midpoint
            - delta * length / 2  # half the arrow length
            + np.array([-sin, cos]) * arrow_sep  # shift outwards by arrow_sep
        )
        ax.arrow(
            x_pos, y_pos, cos * length, sin * length,
            fc=fc, ec=ec or fc, alpha=alpha, width=width,
            head_width=head_width, head_length=head_length, shape=shape,
            length_includes_head=True,
            **kwargs
        )

        # figure out coordinates for text:
        # if drawing relative to base: x and y are same as for arrow
        # dx and dy are one arrow width left and up
        orig_positions = {
            'base': [3 * max_arrow_width, 3 * max_arrow_width],
            'center': [length / 2, 3 * max_arrow_width],
            'tip': [length - 3 * max_arrow_width, 3 * max_arrow_width],
        }
        # for diagonal arrows, put the label at the arrow base
        # for vertical or horizontal arrows, center the label
        where = 'base' if (cp0 != cp1).all() else 'center'
        # rotate based on direction of arrow (cos, sin)
        M = [[cos, -sin], [sin, cos]]
        x, y = np.dot(M, orig_positions[where]) + [x_pos, y_pos]
        label = r'$r_{_{\mathrm{%s}}}$' % (pair,)
        ax.text(x, y, label, size=label_text_size, ha='center', va='center',
                color=labelcolor or fc)


if __name__ == '__main__':
    data = {  # test data
        'A': 0.4, 'T': 0.3, 'G': 0.6, 'C': 0.2,
        'AT': 0.4, 'AC': 0.3, 'AG': 0.2,
        'TA': 0.2, 'TC': 0.3, 'TG': 0.4,
        'CT': 0.2, 'CG': 0.3, 'CA': 0.2,
```

(continues on next page)

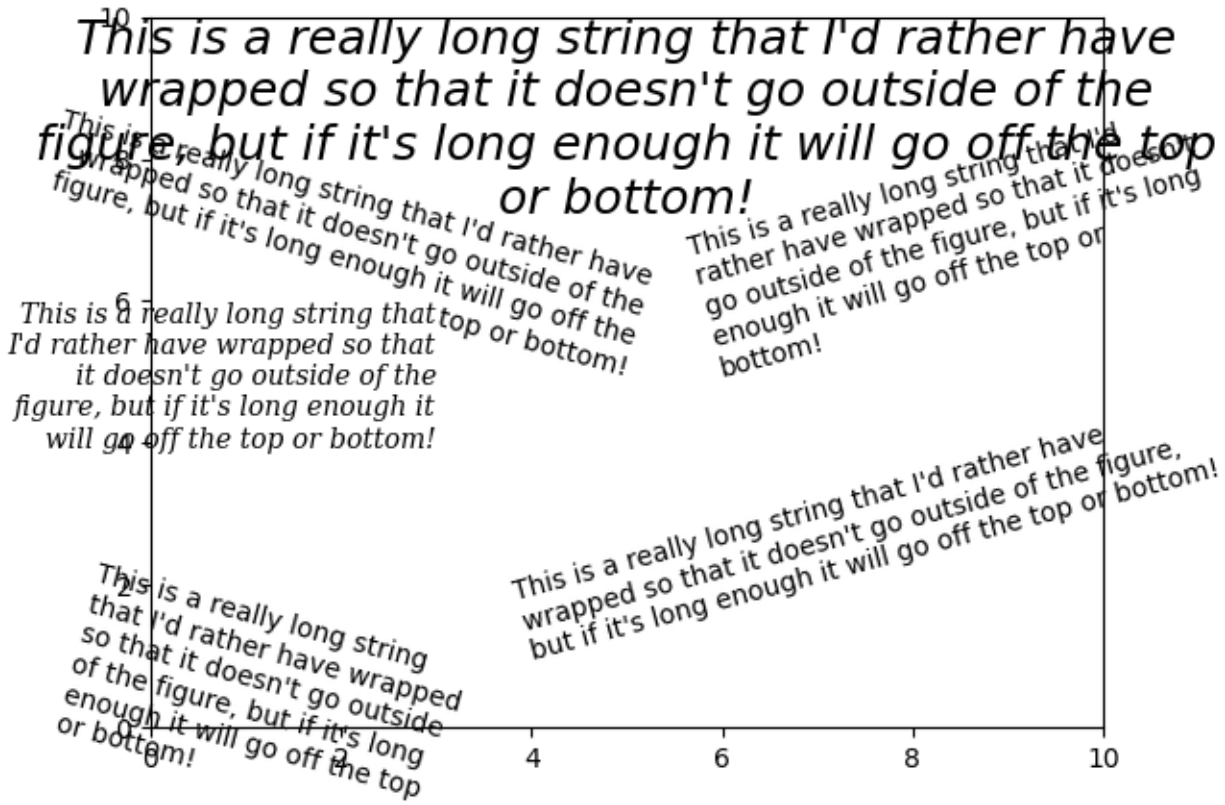
(continued from previous page)

```
        'GA': 0.1, 'GT': 0.4, 'GC': 0.1,
    }

    size = 4
    fig = plt.figure(figsize=(3 * size, size), layout="constrained")
    axs = fig.subplot_mosaic([["length", "width", "alpha"]])

    for display, ax in axs.items():
        make_arrow_graph(
            ax, data, display=display, linewidth=0.001, edgecolor=None,
            normalize_data=True, size=size)

    plt.show()
```

### Auto-wrapping text

Matplotlib can wrap text automatically, but if it's too long, the text will be displayed slightly outside of the boundaries of the axis anyways.

Note: Auto-wrapping does not work together with `savefig(..., bbox_inches='tight')`. The 'tight' setting rescales the canvas to accommodate all content and happens before wrapping. This affects `%matplotlib inline` in IPython and Jupyter notebooks where the inline setting uses `bbox_inches='tight'` by default when saving the image to embed.




```
import matplotlib.pyplot as plt

fig = plt.figure()
plt.axis((0, 10, 0, 10))
t = ("This is a really long string that I'd rather have wrapped so that it "
     "doesn't go outside of the figure, but if it's long enough it will go "
     "off the top or bottom!")
plt.text(4, 1, t, ha='left', rotation=15, wrap=True)
plt.text(6, 5, t, ha='left', rotation=15, wrap=True)
plt.text(5, 5, t, ha='right', rotation=-15, wrap=True)
plt.text(5, 10, t, fontsize=18, style='oblique', ha='center',
         va='top', wrap=True)
plt.text(3, 4, t, family='serif', style='italic', ha='right', wrap=True)
plt.text(-1, 0, t, ha='left', rotation=-15, wrap=True)

plt.show()
```

### Composing Custom Legends

Composing custom legends piece-by-piece.

---

**Note:** For more information on creating and customizing legends, see the following pages:

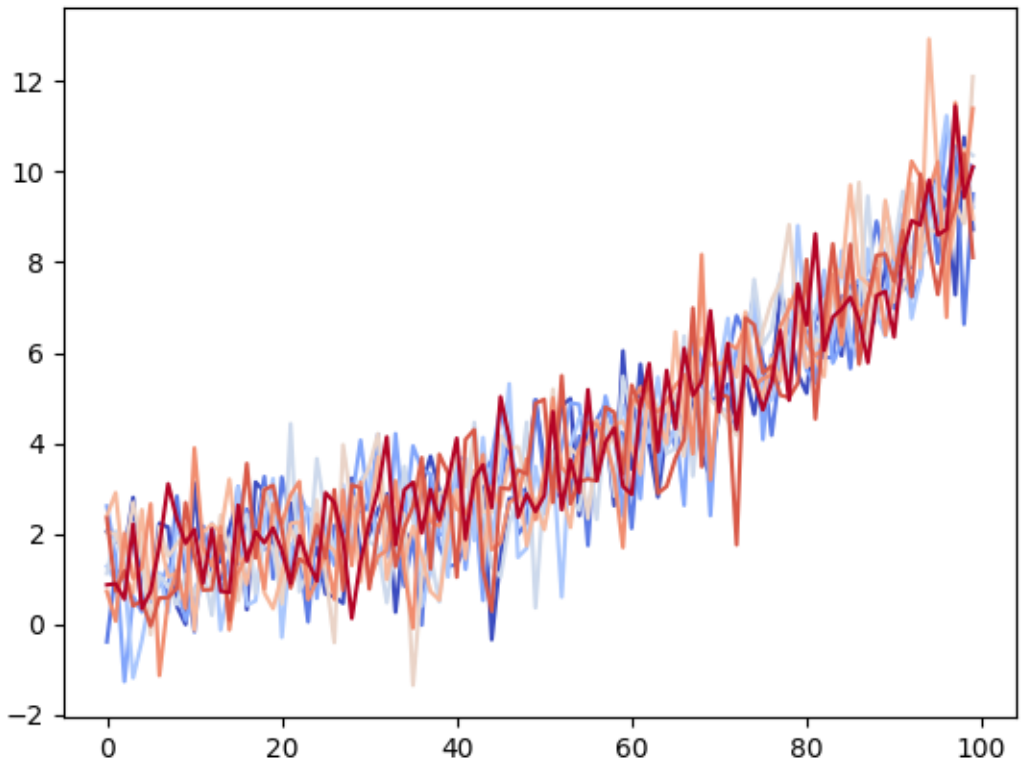
- *Legend guide*
- *Legend Demo*

---

Sometimes you don't want a legend that is explicitly tied to data that you have plotted. For example, say you have plotted 10 lines, but don't want a legend item to show up for each one. If you simply plot the lines and call `ax.legend()`, you will get the following:

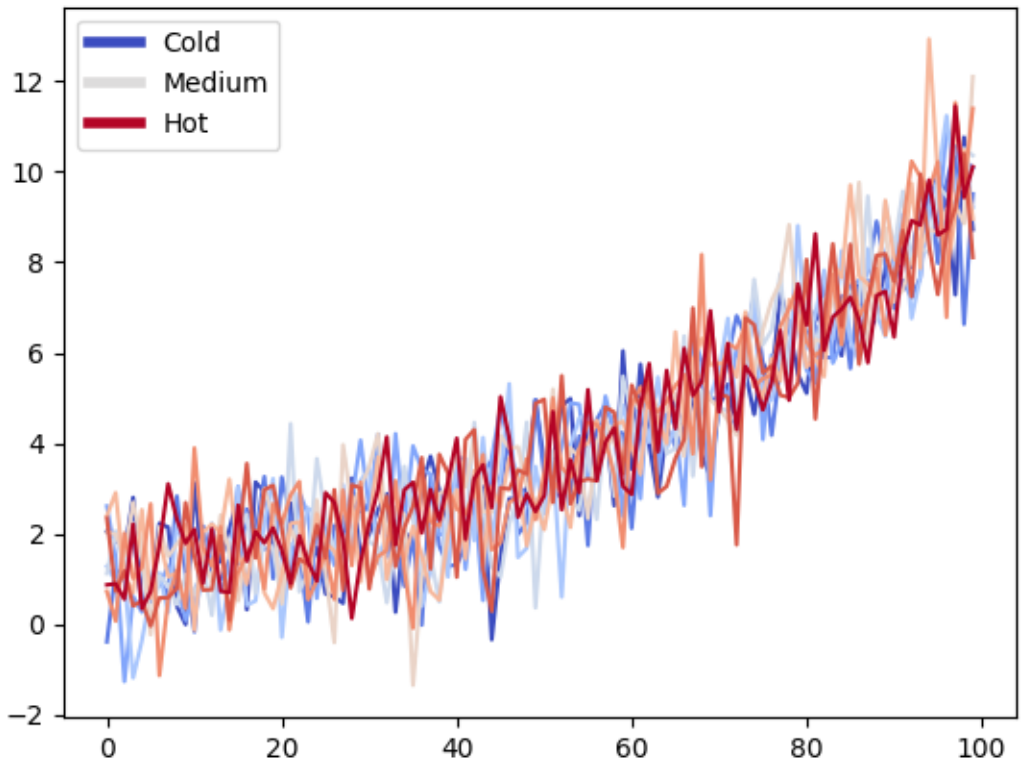
```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib as mpl
from matplotlib import cycler

# Fixing random state for reproducibility
np.random.seed(19680801)
```

```
N = 10
data = (np.geomspace(1, 10, 100) + np.random.randn(N, 100)).T
cmap = plt.cm.coolwarm
mpl.rcParams['axes.prop_cycle'] = cycler(color=cmap(np.linspace(0, 1, N)))

fig, ax = plt.subplots()
lines = ax.plot(data)
```



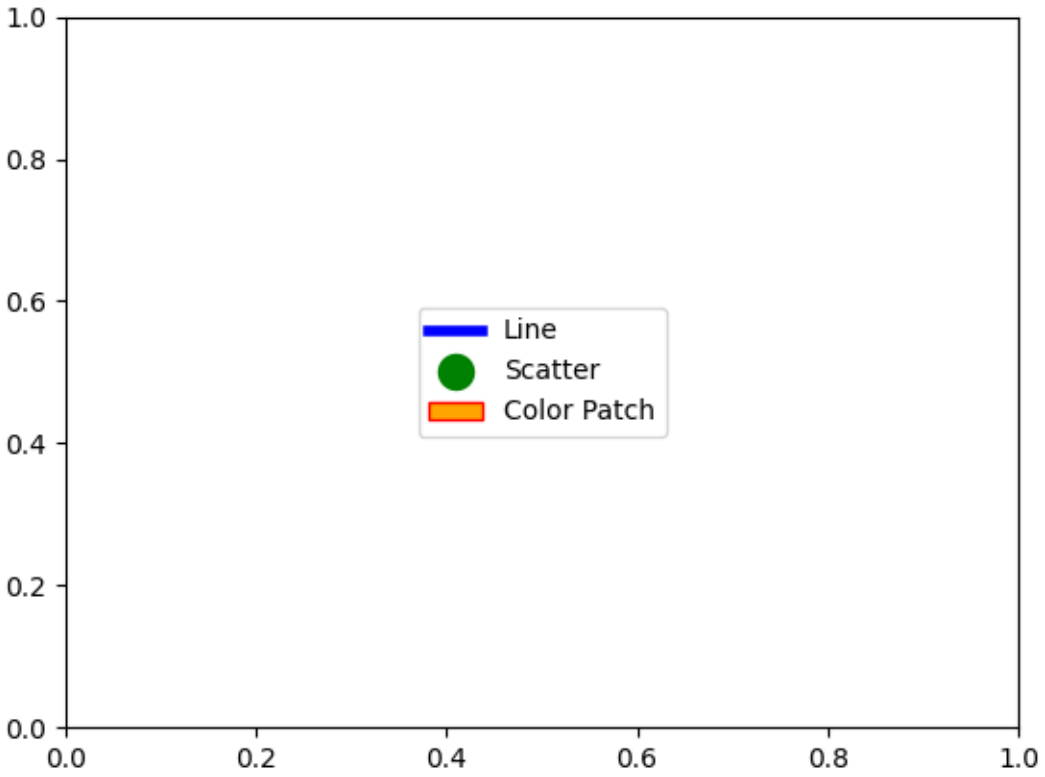

Since the data does not have any labels, creating a legend requires us to define the icons and labels. In this case, we can compose a legend using Matplotlib objects that aren't explicitly tied to the data that was plotted. For example:

```
from matplotlib.lines import Line2D

custom_lines = [Line2D([0], [0], color=cmap(0.), lw=4),
                Line2D([0], [0], color=cmap(.5), lw=4),
                Line2D([0], [0], color=cmap(1.), lw=4)]

fig, ax = plt.subplots()
lines = ax.plot(data)
ax.legend(custom_lines, ['Cold', 'Medium', 'Hot'])
```




There are many other Matplotlib objects that can be used in this way. In the code below we've listed a few common ones.

```
from matplotlib.lines import Line2D
from matplotlib.patches import Patch

legend_elements = [Line2D([0], [0], color='b', lw=4, label='Line'),
                   Line2D([0], [0], marker='o', color='w', label='Scatter',
                          markerfacecolor='g', markersize=15),
                   Patch(facecolor='orange', edgecolor='r',
                         label='Color Patch')]

# Create the figure
fig, ax = plt.subplots()
ax.legend(handles=legend_elements, loc='center')

plt.show()
```

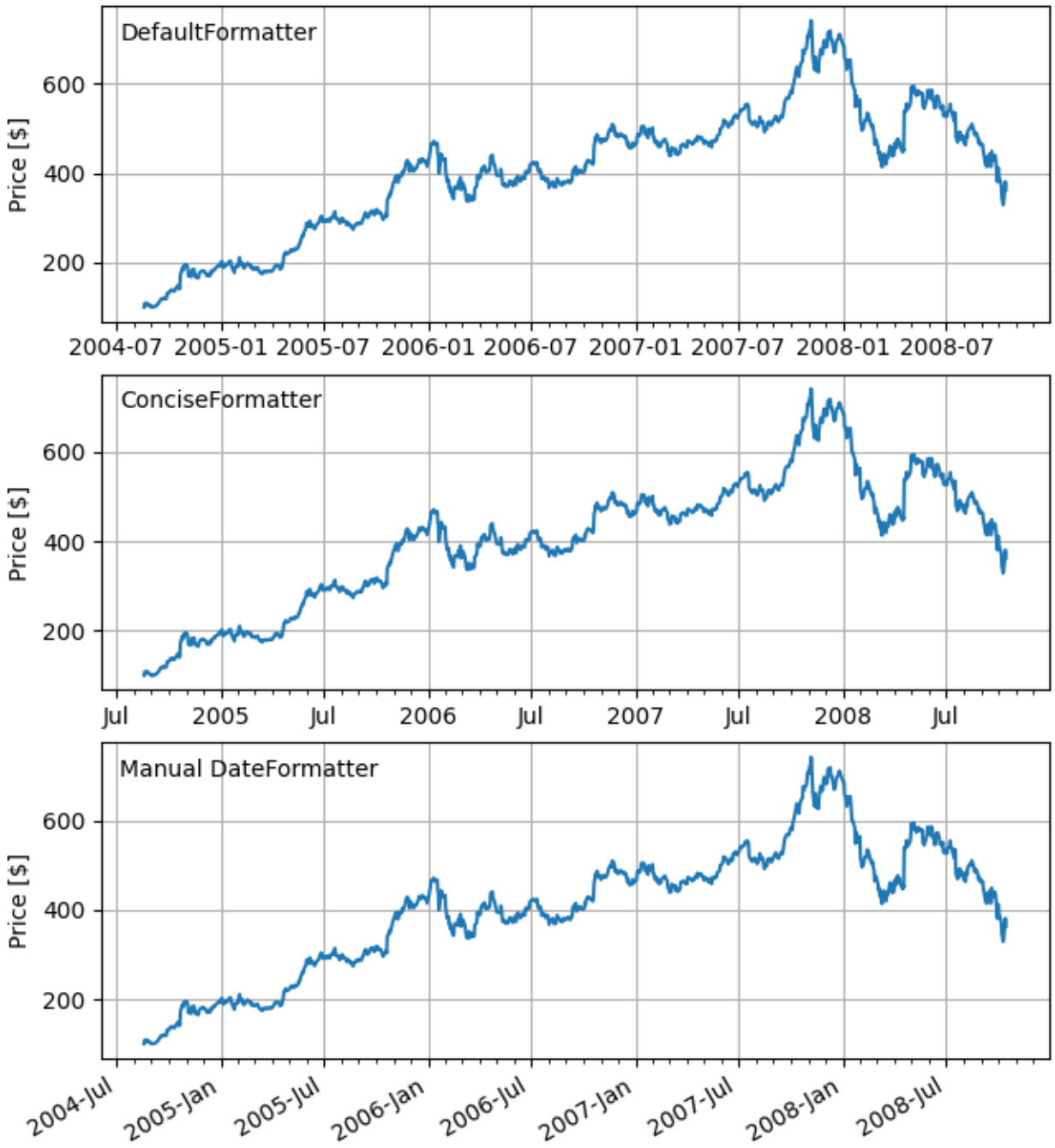



**Total running time of the script:** (0 minutes 1.351 seconds)

## Date tick labels

Matplotlib date plotting is done by converting date instances into days since an epoch (by default 1970-01-01T00:00:00). The *matplotlib.dates* module provides the converter functions *date2num* and *num2date* that convert `datetime.datetime` and `numpy.datetime64` objects to and from Matplotlib's internal representation. These data types are registered with the unit conversion mechanism described in *matplotlib.units*, so the conversion happens automatically for the user. The registration process also sets the default tick `locator` and `formatter` for the axis to be *AutoDateLocator* and *AutoDateFormatter*.

An alternative formatter is the *ConciseDateFormatter*, used in the second `Axes` below (see *Formatting date ticks using ConciseDateFormatter*), which often removes the need to rotate the tick labels. The last `Axes` formats the dates manually, using *DateFormatter* to format the dates using the format strings documented at `datetime.date.strftime`.




```
import matplotlib.pyplot as plt

import matplotlib.cbook as cbook
import matplotlib.dates as mdates

# Load a numpy record array from yahoo csv data with fields date, open, high,
# low, close, volume, adj_close from the mpl-data/sample_data directory. The
# record array stores the date as an np.datetime64 with a day unit ('D') in
# the date column.
data = cbook.get_sample_data('goog.npz')['price_data']
```

(continues on next page)

(continued from previous page)

```
fig, axs = plt.subplots(3, 1, figsize=(6.4, 7), layout='constrained')
# common to all three:
for ax in axs:
    ax.plot('date', 'adj_close', data=data)
    # Major ticks every half year, minor ticks every month,
    ax.xaxis.set_major_locator(mdates.MonthLocator(bymonth=(1, 7)))
    ax.xaxis.set_minor_locator(mdates.MonthLocator())
    ax.grid(True)
    ax.set_ylabel(r'Price [\$]')

# different formats:
ax = axs[0]
ax.set_title('DefaultFormatter', loc='left', y=0.85, x=0.02, fontsize='medium
 ↪')

ax = axs[1]
ax.set_title('ConciseFormatter', loc='left', y=0.85, x=0.02, fontsize='medium
 ↪')
ax.xaxis.set_major_formatter(
    mdates.ConciseDateFormatter(ax.xaxis.get_major_locator()))

ax = axs[2]
ax.set_title('Manual DateFormatter', loc='left', y=0.85, x=0.02,
             fontsize='medium')
# Text in the x-axis will be displayed in 'YYYY-mm' format.
ax.xaxis.set_major_formatter(mdates.DateFormatter('%Y-%b'))
# Rotates and right-aligns the x labels so they don't crowd each other.
for label in ax.get_xticklabels(which='major'):
    label.set(rotation=30, horizontalalignment='right')

plt.show()
```

**Total running time of the script:** (0 minutes 1.130 seconds)

### AnnotationBbox demo

*AnnotationBbox* creates an annotation using an *OffsetBox*, and provides more fine-grained control than *Axes.annotate*. This example demonstrates the use of AnnotationBbox together with three different OffsetBoxes: *TextArea*, *DrawingArea*, and *OffsetImage*.

```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.cbook import get_sample_data
from matplotlib.offsetbox import (AnnotationBbox, DrawingArea, OffsetImage,
                                  TextArea)
from matplotlib.patches import Circle

fig, ax = plt.subplots()
```

(continues on next page)

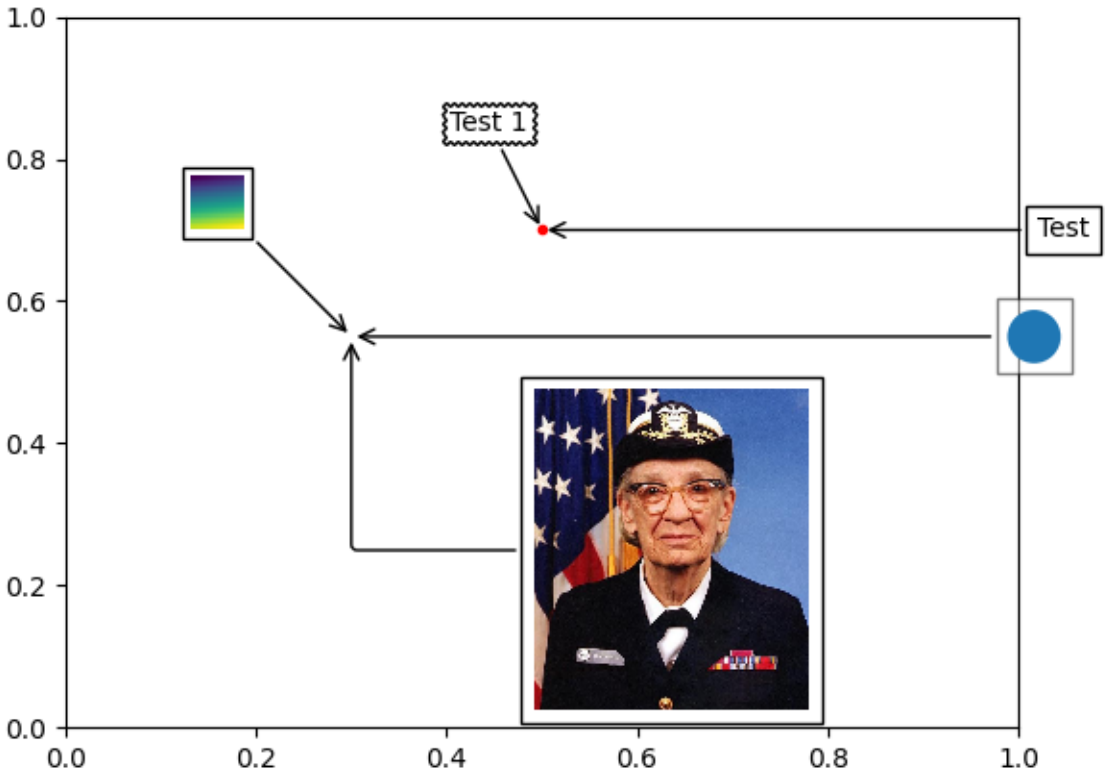
(continued from previous page)

```
# Define a 1st position to annotate (display it with a marker)
xy = (0.5, 0.7)
ax.plot(xy[0], xy[1], ".r")

# Annotate the 1st position with a text box ('Test 1')
offsetbox = TextArea("Test 1")

ab = AnnotationBbox(offsetbox, xy,
                    xybox=(-20, 40),
                    xycoords='data',
                    boxcoords="offset points",
                    arrowprops=dict(arrowstyle="->"),
                    bboxprops=dict(boxstyle="sawtooth"))
ax.add_artist(ab)

# Annotate the 1st position with another text box ('Test')
offsetbox = TextArea("Test")

ab = AnnotationBbox(offsetbox, xy,
                    xybox=(1.02, xy[1]),
                    xycoords='data',
                    boxcoords=("axes fraction", "data"),
                    box_alignment=(0., 0.5),
                    arrowprops=dict(arrowstyle="->"))
ax.add_artist(ab)

# Define a 2nd position to annotate (don't display with a marker this time)
xy = [0.3, 0.55]

# Annotate the 2nd position with a circle patch
da = DrawingArea(20, 20, 0, 0)
p = Circle((10, 10), 10)
da.add_artist(p)

ab = AnnotationBbox(da, xy,
                    xybox=(1., xy[1]),
                    xycoords='data',
                    boxcoords=("axes fraction", "data"),
                    box_alignment=(0.2, 0.5),
                    arrowprops=dict(arrowstyle="->"),
                    bboxprops=dict(alpha=0.5))

ax.add_artist(ab)

# Annotate the 2nd position with an image (a generated array of pixels)
arr = np.arange(100).reshape((10, 10))
im = OffsetImage(arr, zoom=2)
im.image.axes = ax

ab = AnnotationBbox(im, xy,
                    xybox=(-50., 50.),
```

(continues on next page)

(continued from previous page)

```
                    xycoords='data',
                    boxcoords="offset points",
                    pad=0.3,
                    arrowprops=dict(arrowstyle="->"))

ax.add_artist(ab)

# Annotate the 2nd position with another image (a Grace Hopper portrait)
with get_sample_data("grace_hopper.jpg") as file:
    arr_img = plt.imread(file)

imagebox = OffsetImage(arr_img, zoom=0.2)
imagebox.image.axes = ax

ab = AnnotationBbox(imagebox, xy,
                    xybox=(120., -80.),
                    xycoords='data',
                    boxcoords="offset points",
                    pad=0.5,
                    arrowprops=dict(
                        arrowstyle="->",
                        connectionstyle="angle,angleA=0,angleB=90,rad=3")
                    )

ax.add_artist(ab)

# Fix the display limits to see everything
ax.set_xlim(0, 1)
ax.set_ylim(0, 1)

plt.show()
```



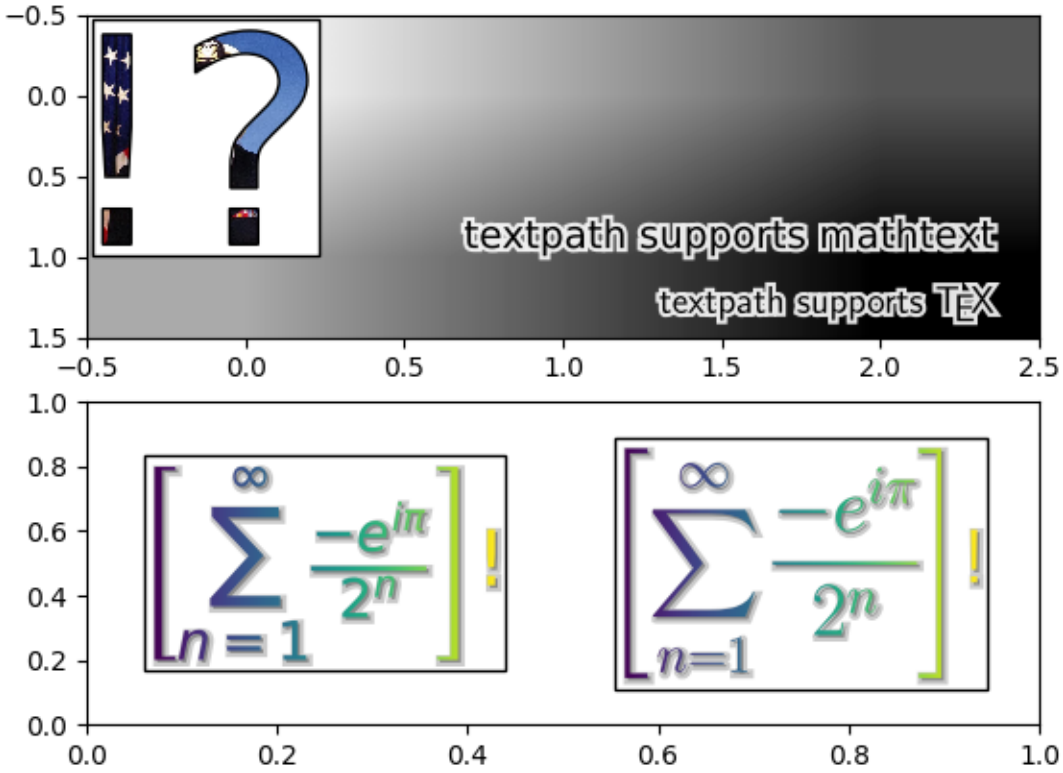

**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.patches.Circle*
- *matplotlib.offsetbox.TextArea*
- *matplotlib.offsetbox.DrawingArea*
- *matplotlib.offsetbox.OffsetImage*
- *matplotlib.offsetbox.AnnotationBbox*
- *matplotlib.cbook.get_sample_data*
- *matplotlib.pyplot.subplots*
- *matplotlib.pyplot.imread*

### Using a text as a Path

*TextPath* creates a *Path* that is the outline of the characters of a text. The resulting path can be employed e.g. as a clip path for an image.




```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.cbook import get_sample_data
from matplotlib.image import BboxImage
from matplotlib.offsetbox import (AnchoredOffsetbox, AnnotationBbox,
                                  AuxTransformBox)
from matplotlib.patches import PathPatch, Shadow
from matplotlib.text import TextPath
from matplotlib.transforms import IdentityTransform


class PathClippedImagePatch(PathPatch):
    """
    The given image is used to draw the face of the patch. Internally,
    it uses BboxImage whose clippath set to the path of the patch.

    FIXME : The result is currently dpi dependent.
```

(continues on next page)

(continued from previous page)

```
    """

    def __init__(self, path, bbox_image, **kwargs):
        super().__init__(path, **kwargs)
        self.bbox_image = BboxImage(
            self.get_window_extent, norm=None, origin=None)
        self.bbox_image.set_data(bbox_image)

    def set_facecolor(self, color):
        """Simply ignore facecolor."""
        super().set_facecolor("none")

    def draw(self, renderer=None):
        # the clip path must be updated every draw. any solution? -JJ
        self.bbox_image.set_clip_path(self._path, self.get_transform())
        self.bbox_image.draw(renderer)
        super().draw(renderer)


if __name__ == "__main__":

    fig, (ax1, ax2) = plt.subplots(2)

    # EXAMPLE 1

    arr = plt.imread(get_sample_data("grace_hopper.jpg"))

    text_path = TextPath((0, 0), "!?", size=150)
    p = PathClippedImagePatch(text_path, arr, ec="k")

    # make offset box
    offsetbox = AuxTransformBox(IdentityTransform())
    offsetbox.add_artist(p)

    # make anchored offset box
    ao = AnchoredOffsetbox(loc='upper left', child=offsetbox, frameon=True,
                           borderpad=0.2)
    ax1.add_artist(ao)

    # another text
    for usetex, ypos, string in [
            (False, 0.25, r"textpath supports mathtext"),
            (True, 0.05, r"textpath supports \TeX"),
    ]:
        text_path = TextPath((0, 0), string, size=20, usetex=usetex)

        p1 = PathPatch(text_path, ec="w", lw=3, fc="w", alpha=0.9)
        p2 = PathPatch(text_path, ec="none", fc="k")

        offsetbox2 = AuxTransformBox(IdentityTransform())
        offsetbox2.add_artist(p1)
        offsetbox2.add_artist(p2)
```

(continues on next page)

(continued from previous page)

```
        ab = AnnotationBbox(offsetbox2, (0.95, ypos),
                            xycoords='axes fraction',
                            boxcoords="offset points",
                            box_alignment=(1., 0.),
                            frameon=False,
                            )
        ax1.add_artist(ab)

    ax1.imshow([[0, 1, 2], [1, 2, 3]], cmap=plt.cm.gist_gray_r,
               interpolation="bilinear", aspect="auto")

    # EXAMPLE 2

    arr = np.arange(256).reshape(1, 256)

    for usetex, xpos, string in [
            (False, 0.25,
             r"$\left[\sum_{n=1}^\infty\frac{-e^{i\pi}}{2^n}\right]$!"),
            (True, 0.75,
             r"$\displaystyle\left[\sum_{n=1}^\infty"
             r"\frac{-e^{i\pi}}{2^n}\right]$!"),
    ]:
        text_path = TextPath((0, 0), string, size=40, usetex=usetex)
        text_patch = PathClippedImagePatch(text_path, arr, ec="none")
        shadow1 = Shadow(text_patch, 1, -1, fc="none", ec="0.6", lw=3)
        shadow2 = Shadow(text_patch, 1, -1, fc="0.3", ec="none")

        # make offset box
        offsetbox = AuxTransformBox(IdentityTransform())
        offsetbox.add_artist(shadow1)
        offsetbox.add_artist(shadow2)
        offsetbox.add_artist(text_patch)

        # place the anchored offset box using AnnotationBbox
        ab = AnnotationBbox(offsetbox, (xpos, 0.5), box_alignment=(0.5, 0.5))

        ax2.add_artist(ab)

    ax2.set_xlim(0, 1)
    ax2.set_ylim(0, 1)

    plt.show()
```

**Total running time of the script:** (0 minutes 1.670 seconds)

### Text Rotation Mode

This example illustrates the effect of `rotation_mode` on the positioning of rotated text.

Rotated *Text*s are created by passing the parameter `rotation` to the constructor or the Axes' method *text*.

The actual positioning depends on the additional parameters `horizontalalignment`, `verticalalignment` and `rotation_mode`. `rotation_mode` determines the order of rotation and alignment:

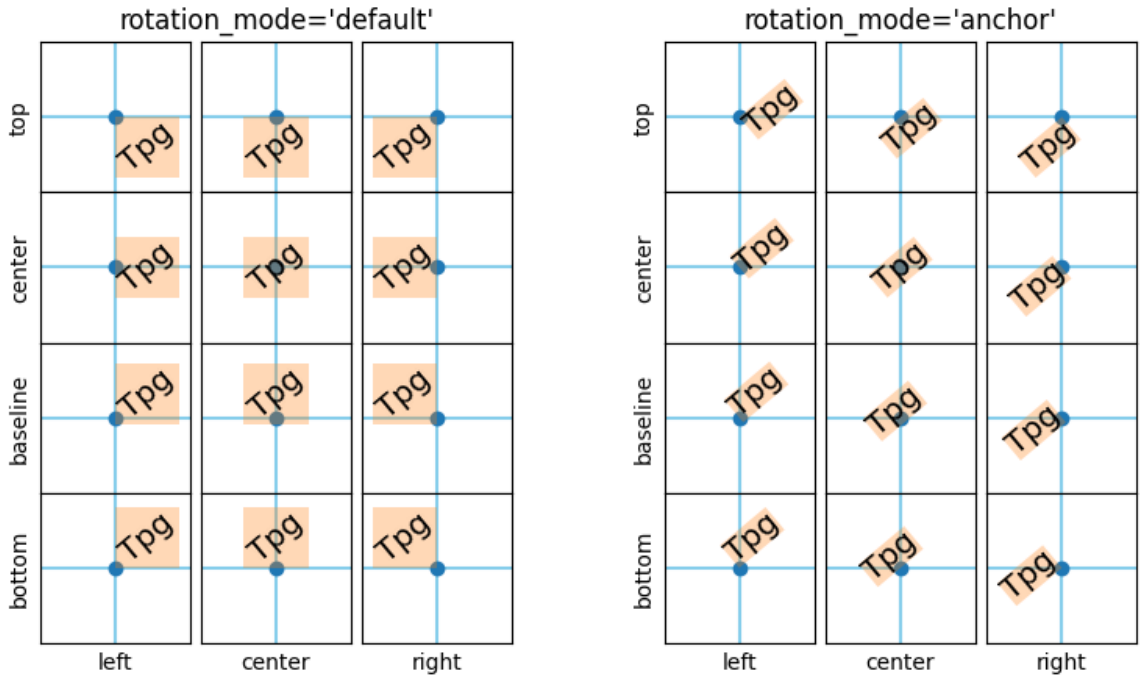
- `rotation_mode='default'` (or None) first rotates the text and then aligns the bounding box of the rotated text.
- `rotation_mode='anchor'` aligns the unrotated text and then rotates the text around the point of alignment.

```
import matplotlib.pyplot as plt


def test_rotation_mode(fig, mode):
    ha_list = ["left", "center", "right"]
    va_list = ["top", "center", "baseline", "bottom"]
    axs = fig.subplots(len(va_list), len(ha_list), sharex=True, sharey=True,
                       subplot_kw=dict(aspect=1),
                       gridspec_kw=dict(hspace=0, wspace=0))

    # labels and title
    for ha, ax in zip(ha_list, axs[-1, :]):
        ax.set_xlabel(ha)
    for va, ax in zip(va_list, axs[:, 0]):
        ax.set_ylabel(va)
    axs[0, 1].set_title(f"rotation_mode='{mode}'", size="large")

    kw = (
        {} if mode == "default" else
        {"bbox": dict(boxstyle="square,pad=0.", ec="none", fc="C1", alpha=0.
↪3)}
    )

    texts = {}

    # use a different text alignment in each Axes
    for i, va in enumerate(va_list):
        for j, ha in enumerate(ha_list):
            ax = axs[i, j]
            # prepare Axes layout
            ax.set(xticks=[], yticks=[])
            ax.axvline(0.5, color="skyblue", zorder=0)
            ax.axhline(0.5, color="skyblue", zorder=0)
            ax.plot(0.5, 0.5, color="C0", marker="o", zorder=1)
            # add text with rotation and alignment settings
            tx = ax.text(0.5, 0.5, "Tpg",
```

(continues on next page)

(continued from previous page)

```
                          size="x-large", rotation=40,
                          horizontalalignment=ha, verticalalignment=va,
                          rotation_mode=mode, **kw)
            texts[ax] = tx

    if mode == "default":
        # highlight bbox
        fig.canvas.draw()
        for ax, text in texts.items():
            bb = text.get_window_extent().transformed(ax.transData.inverted())
            rect = plt.Rectangle((bb.x0, bb.y0), bb.width, bb.height,
                                 facecolor="C1", alpha=0.3, zorder=2)
            ax.add_patch(rect)


fig = plt.figure(figsize=(8, 5))
subfigs = fig.subfigures(1, 2)
test_rotation_mode(subfigs[0], "default")
test_rotation_mode(subfigs[1], "anchor")
plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.text* / *matplotlib.pyplot.text*

### The difference between \dfrac and \frac

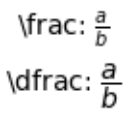
In this example, the differences between the \dfrac and \frac TeX macros are illustrated; in particular, the difference between display style and text style fractions when using Mathtex.

New in version 2.1.

---

**Note:** To use \dfrac with the LaTeX engine (text.usetex : True), you need to import the amsmath package with the text.latex.preamble rc, which is an unsupported feature; therefore, it is probably a better idea to just use the \displaystyle option before the \frac macro to get this behavior with the LaTeX engine.

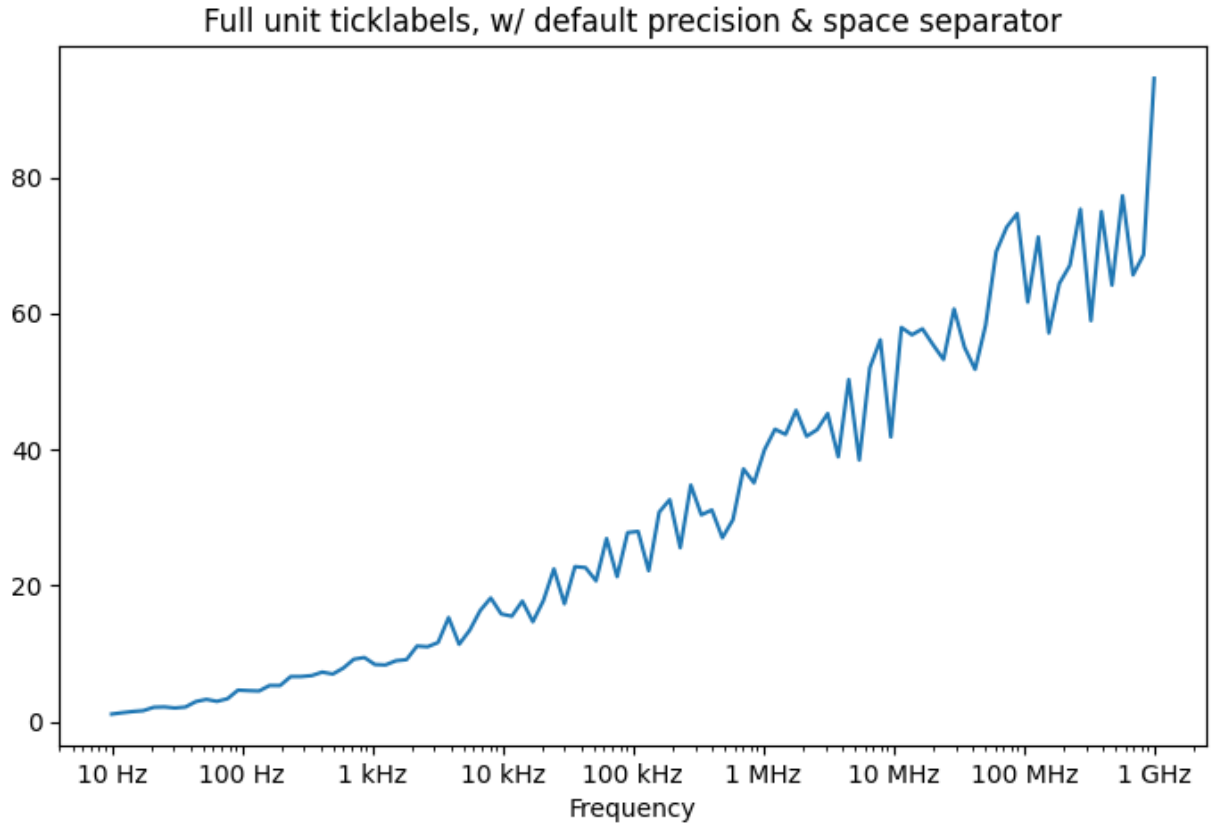
---

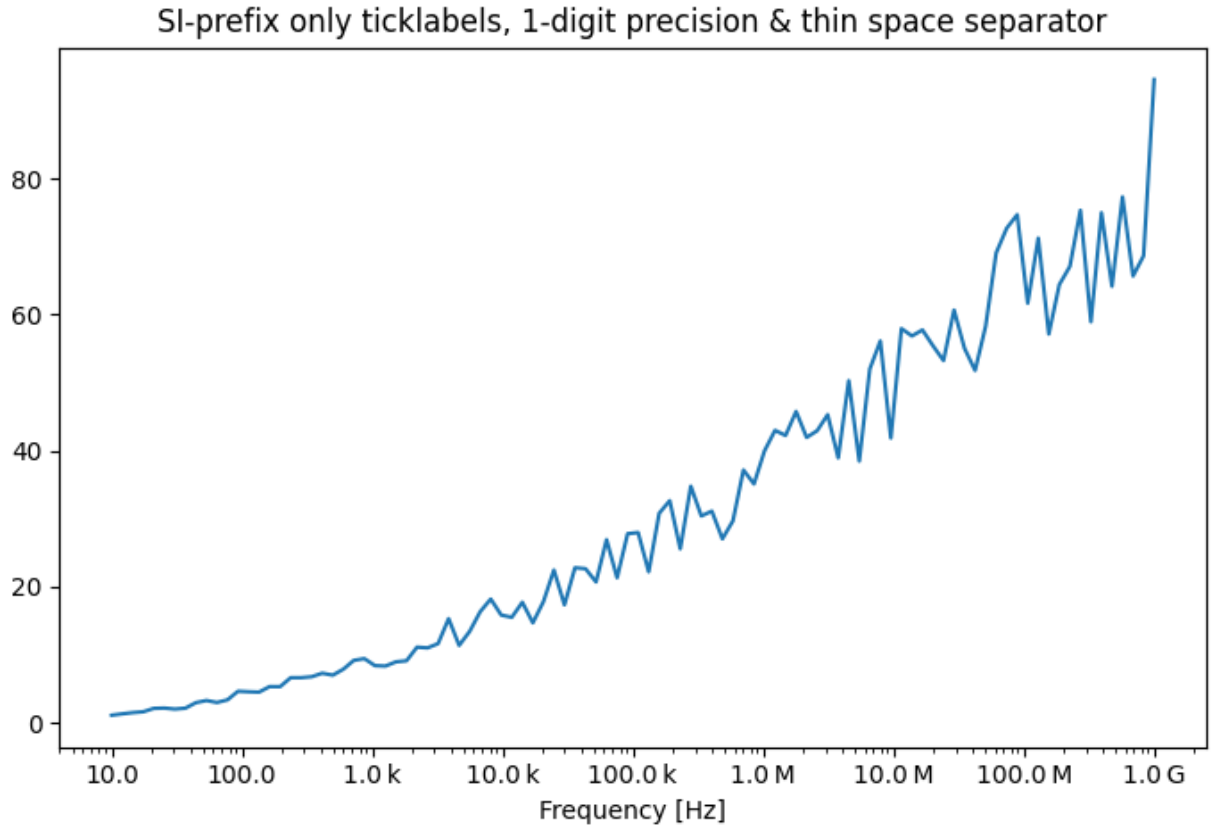



```
import matplotlib.pyplot as plt

fig = plt.figure(figsize=(5.25, 0.75))
fig.text(0.5, 0.3, r'\dfrac: $\dfrac{a}{b}$',
         horizontalalignment='center', verticalalignment='center')
fig.text(0.5, 0.7, r'\frac: $\frac{a}{b}$',
         horizontalalignment='center', verticalalignment='center')
plt.show()
```

### Labeling ticks using engineering notation

Use of the engineering Formatter.


Full unit ticklabels, w/ default precision & space separator
80
60
40
20
0
10 Hz
100 Hz
1 kHz
10 kHz
100 kHz
1 MHz
10 MHz
100 MHz
1 GHz
Frequency


SI-prefix only ticklabels, 1-digit precision & thin space separator
80
60
40
20
0
10.0
100.0
1.0 k
10.0 k
100.0 k
1.0 M
10.0 M
100.0 M
1.0 G
Frequency [Hz]

```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.ticker import EngFormatter

# Fixing random state for reproducibility
prng = np.random.RandomState(19680801)

# Create artificial data to plot.
# The x data span over several decades to demonstrate several SI prefixes.
xs = np.logspace(1, 9, 100)
ys = (0.8 + 0.4 * prng.uniform(size=100)) * np.log10(xs)**2

# Figure width is doubled (2*6.4) to display nicely 2 subplots side by side.
fig, (ax0, ax1) = plt.subplots(nrows=2, figsize=(7, 9.6))
for ax in (ax0, ax1):
    ax.set_xscale('log')

# Demo of the default settings, with a user-defined unit label.
ax0.set_title('Full unit ticklabels, w/ default precision & space separator')
formatter0 = EngFormatter(unit='Hz')
ax0.xaxis.set_major_formatter(formatter0)
ax0.plot(xs, ys)
ax0.set_xlabel('Frequency')

# Demo of the options `places` (number of digit after decimal point) and
# `sep` (separator between the number and the prefix/unit).
ax1.set_title('SI-prefix only ticklabels, 1-digit precision & '
              'thin space separator')
formatter1 = EngFormatter(places=1, sep="\N{THIN SPACE}")  # U+2009
ax1.xaxis.set_major_formatter(formatter1)
ax1.plot(xs, ys)
ax1.set_xlabel('Frequency [Hz]')

plt.tight_layout()
plt.show()
```

**Total running time of the script:** (0 minutes 1.097 seconds)

### Annotation arrow style reference

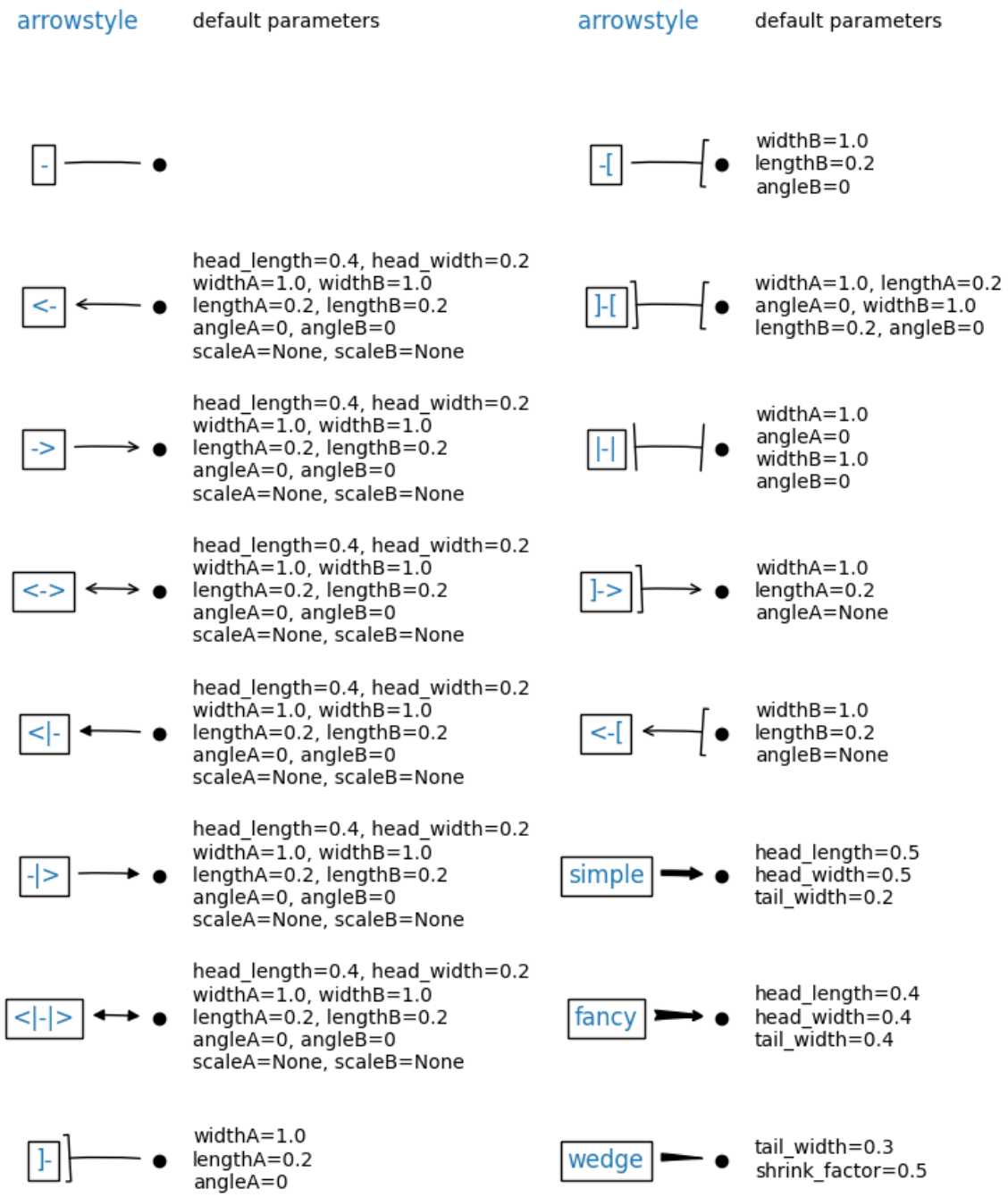
Overview of the arrow styles available in *annotate*.

```
import inspect
import itertools
import re

import matplotlib.pyplot as plt

import matplotlib.patches as mpatches
```

(continues on next page)

(continued from previous page)

```
styles = mpatches.ArrowStyle.get_styles()
ncol = 2
nrow = (len(styles) + 1) // ncol
axs = (plt.figure(figsize=(4 * ncol, 1 + nrow))
       .add_gridspec(1 + nrow, ncol,
                     wspace=.7, left=.1, right=.9, bottom=0, top=1).
 ↪subplots())
for ax in axs.flat:
    ax.set_axis_off()
for ax in axs[0, :]:
    ax.text(0, .5, "arrowstyle",
            transform=ax.transAxes, size="large", color="tab:blue",
            horizontalalignment="center", verticalalignment="center")
    ax.text(.35, .5, "default parameters",
            transform=ax.transAxes,
            horizontalalignment="left", verticalalignment="center")
for ax, (stylename, stylecls) in zip(axs[1:, :].T.flat, styles.items()):
    l, = ax.plot(.25, .5, "ok", transform=ax.transAxes)
    ax.annotate(stylename, (.25, .5), (-0.1, .5),
                xycoords="axes fraction", textcoords="axes fraction",
                size="large", color="tab:blue",
                horizontalalignment="center", verticalalignment="center",
                arrowprops=dict(
                    arrowstyle=stylename, connectionstyle="arc3,rad=-0.05",
                    color="k", shrinkA=5, shrinkB=5, patchB=l,
                ),
                bbox=dict(boxstyle="square", fc="w"))
    # wrap at every nth comma (n = 1 or 2, depending on text length)
    s = str(inspect.signature(stylecls))[1:-1]
    n = 2 if s.count(',') > 3 else 1
    ax.text(.35, .5,
            re.sub(', ', lambda m, c=itertools.count(1): m.group()
                   if next(c) % n else '\n', s),
            transform=ax.transAxes,
            horizontalalignment="left", verticalalignment="center")

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

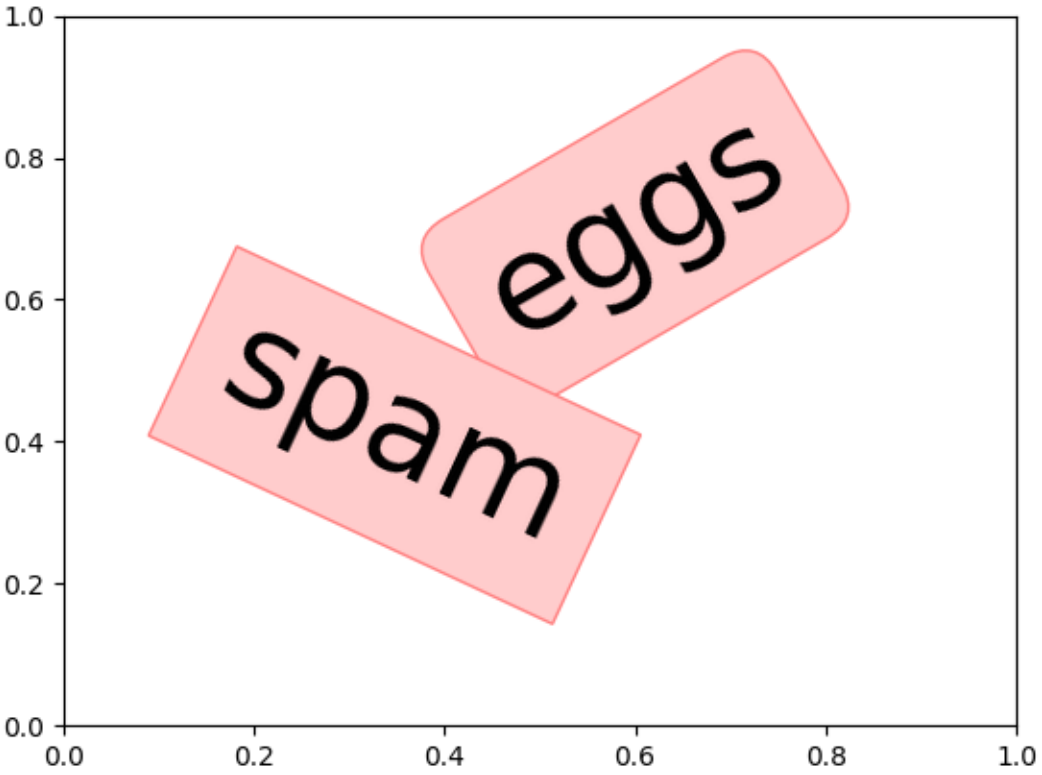
- *matplotlib.patches*
- *matplotlib.patches.ArrowStyle*
- matplotlib.patches.ArrowStyle.get_styles

- *matplotlib.axes.Axes.annotate*

**Total running time of the script:** (0 minutes 1.035 seconds)

### Styling text boxes

This example shows how to style text boxes using *bbox* parameters.




```
import matplotlib.pyplot as plt

plt.text(0.6, 0.7, "eggs", size=50, rotation=30.,
         ha="center", va="center",
         bbox=dict(boxstyle="round",
                   ec=(1., 0.5, 0.5),
                   fc=(1., 0.8, 0.8),
                   )
         )

plt.text(0.55, 0.6, "spam", size=50, rotation=-25.,
         ha="right", va="top",
         bbox=dict(boxstyle="square",
```

(continues on next page)

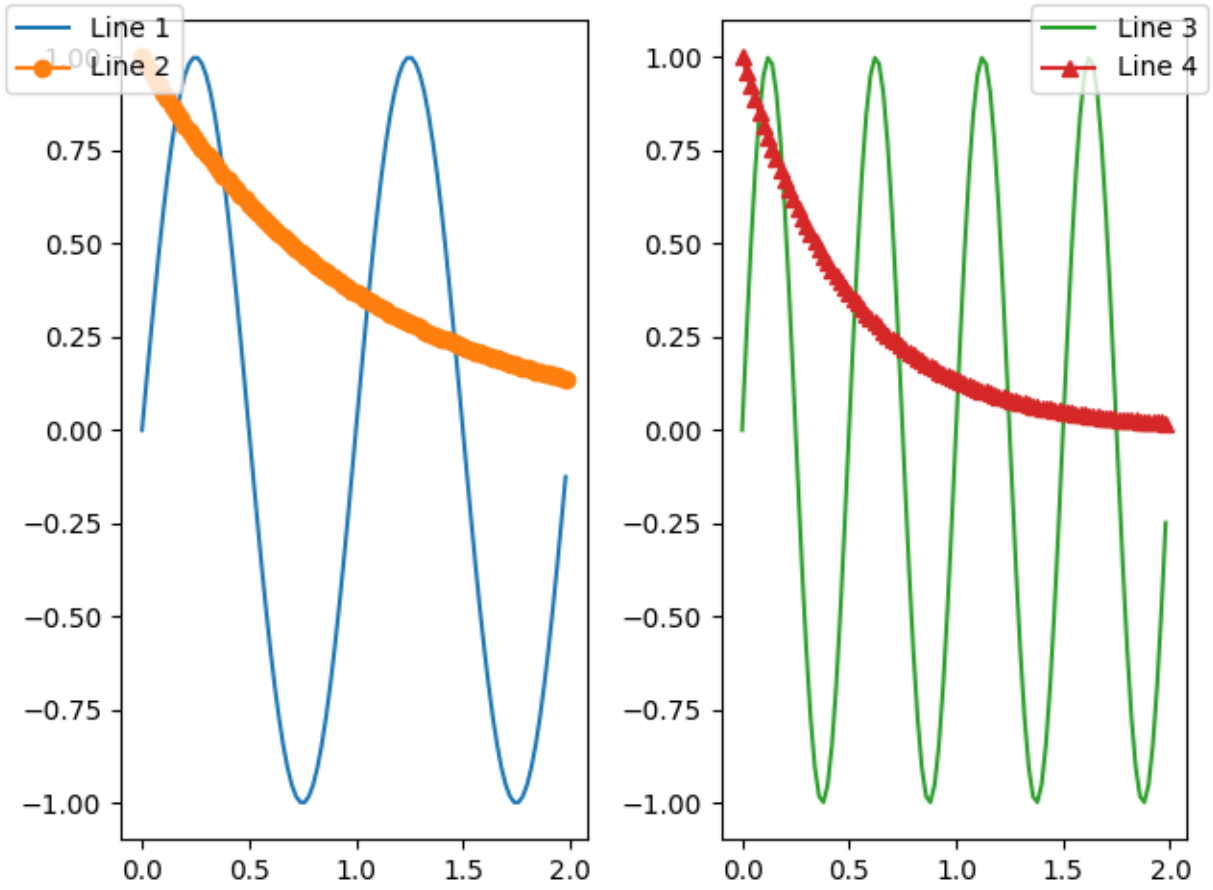
(continued from previous page)

```
                    ec=(1., 0.5, 0.5),
                    fc=(1., 0.8, 0.8),
                    )
        )

plt.show()
```

### Figure legend demo

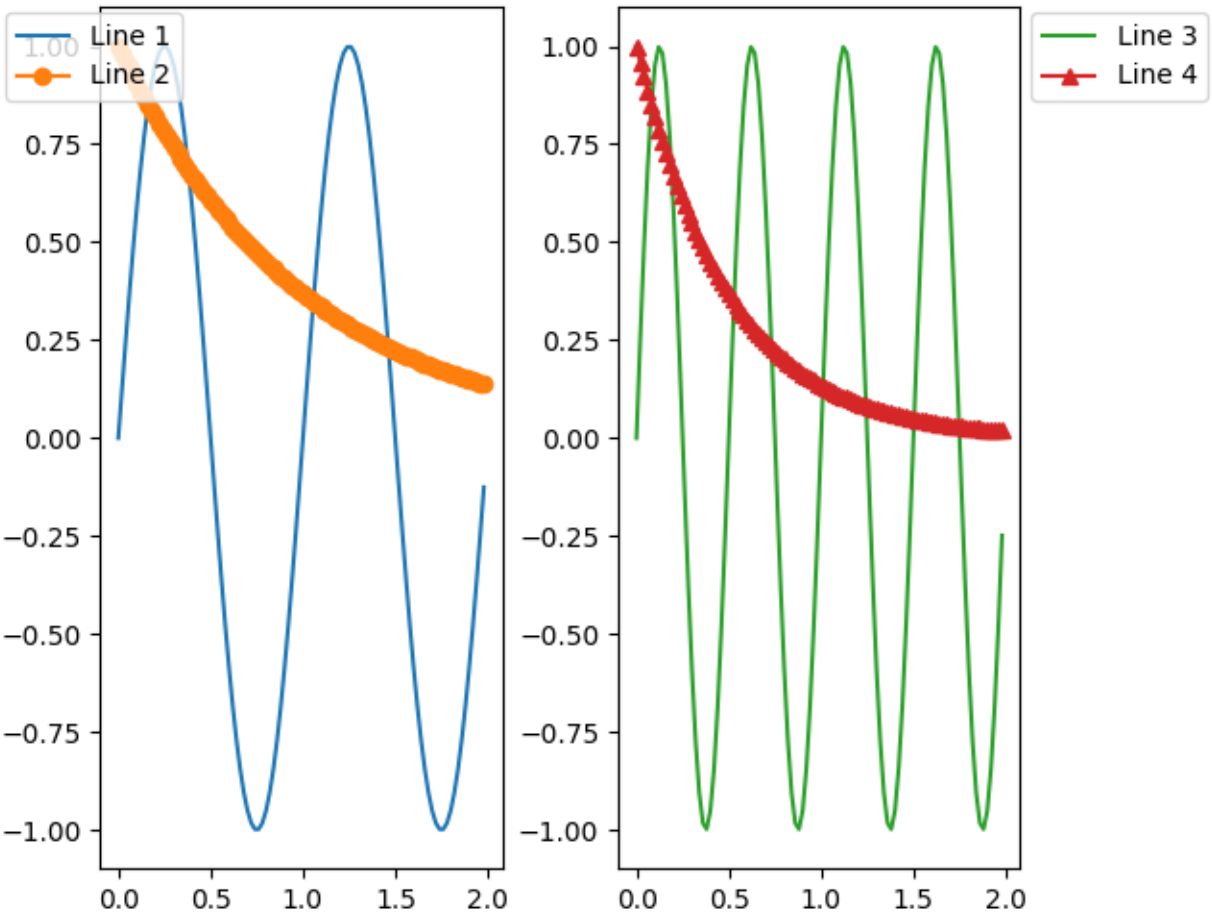
Instead of plotting a legend on each axis, a legend for all the artists on all the sub-axes of a figure can be plotted instead.

```
import matplotlib.pyplot as plt
import numpy as np

fig, axs = plt.subplots(1, 2)

x = np.arange(0.0, 2.0, 0.02)
y1 = np.sin(2 * np.pi * x)
y2 = np.exp(-x)
l1, = axs[0].plot(x, y1)
l2, = axs[0].plot(x, y2, marker='o')

y3 = np.sin(4 * np.pi * x)
y4 = np.exp(-2 * x)
l3, = axs[1].plot(x, y3, color='tab:green')
l4, = axs[1].plot(x, y4, color='tab:red', marker='^')

fig.legend((l1, l2), ('Line 1', 'Line 2'), loc='upper left')
fig.legend((l3, l4), ('Line 3', 'Line 4'), loc='upper right')

plt.tight_layout()
plt.show()
```




Sometimes we do not want the legend to overlap the Axes. If you use *constrained layout* you can specify "outside right upper", and *constrained layout* will make room for the legend.

```
fig, axs = plt.subplots(1, 2, layout='constrained')

x = np.arange(0.0, 2.0, 0.02)
y1 = np.sin(2 * np.pi * x)
y2 = np.exp(-x)
l1, = axs[0].plot(x, y1)
l2, = axs[0].plot(x, y2, marker='o')

y3 = np.sin(4 * np.pi * x)
y4 = np.exp(-2 * x)
l3, = axs[1].plot(x, y3, color='tab:green')
l4, = axs[1].plot(x, y4, color='tab:red', marker='^')

fig.legend((l1, l2), ('Line 1', 'Line 2'), loc='upper left')
fig.legend((l3, l4), ('Line 3', 'Line 4'), loc='outside right upper')

plt.show()
```




**Total running time of the script:** (0 minutes 1.081 seconds)

### Configuring the font family

You can explicitly set which font family is picked up, either by specifying family names of fonts installed on user's system, or generic-families (e.g., 'serif', 'sans-serif', 'monospace', 'fantasy' or 'cursive'), or a combination of both. (see *Text properties and layout*)

In the example below, we are overriding the default sans-serif generic family to include a specific (Tahoma) font. (Note that the best way to achieve this would simply be to prepend 'Tahoma' in 'font.family')

The default family is set with the font.family rcparam, e.g.

```
rcParams['font.family'] = 'sans-serif'
```

and for the font.family you set a list of font styles to try to find in order:

```
rcParams['font.sans-serif'] = ['Tahoma', 'DejaVu Sans',
                               'Lucida Grande', 'Verdana']
```

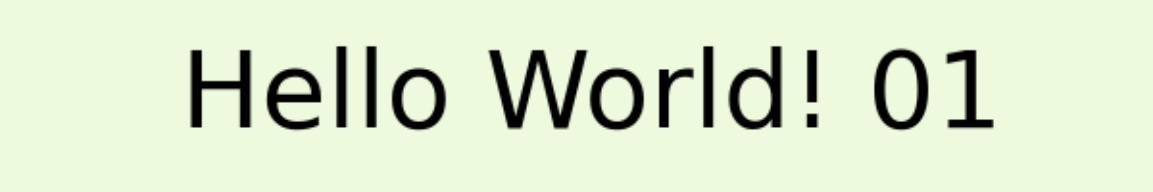
The `font.family` defaults are OS dependent and can be viewed with:

```
import matplotlib.pyplot as plt

print(plt.rcParams["font.sans-serif"][0])
print(plt.rcParams["font.monospace"][0])
```

```
DejaVu Sans
DejaVu Sans Mono
```

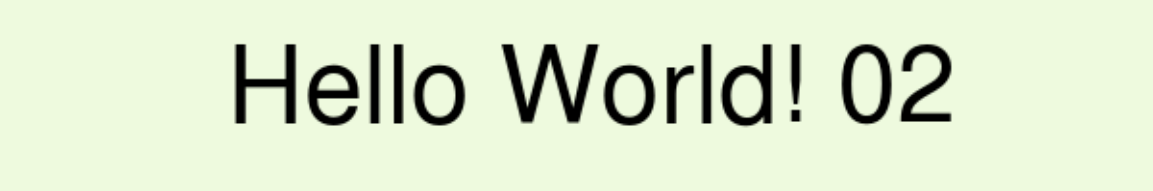
Choose default sans-serif font

```
def print_text(text):
    fig, ax = plt.subplots(figsize=(6, 1), facecolor="#eefade")
    ax.text(0.5, 0.5, text, ha='center', va='center', size=40)
    ax.axis("off")
    plt.show()


plt.rcParams["font.family"] = "sans-serif"
print_text("Hello World! 01")
```



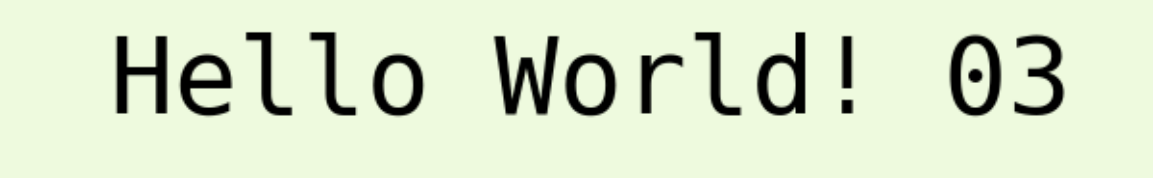

Choose sans-serif font and specify to it to "Nimbus Sans"

```
plt.rcParams["font.family"] = "sans-serif"
plt.rcParams["font.sans-serif"] = ["Nimbus Sans"]
print_text("Hello World! 02")
```



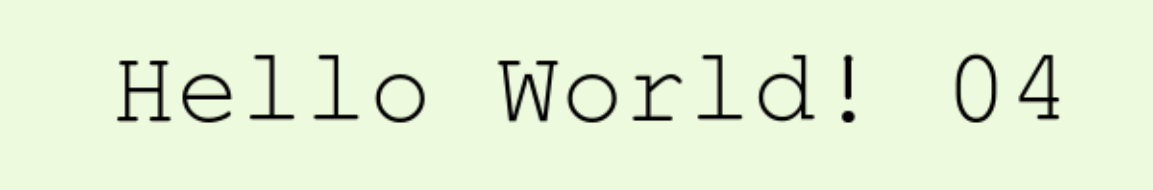

Choose default monospace font

```
plt.rcParams["font.family"] = "monospace"
print_text("Hello World! 03")
```




Choose monospace font and specify to it to "FreeMono"

```
plt.rcParams["font.family"] = "monospace"
plt.rcParams["font.monospace"] = ["FreeMono"]
print_text("Hello World! 04")
```



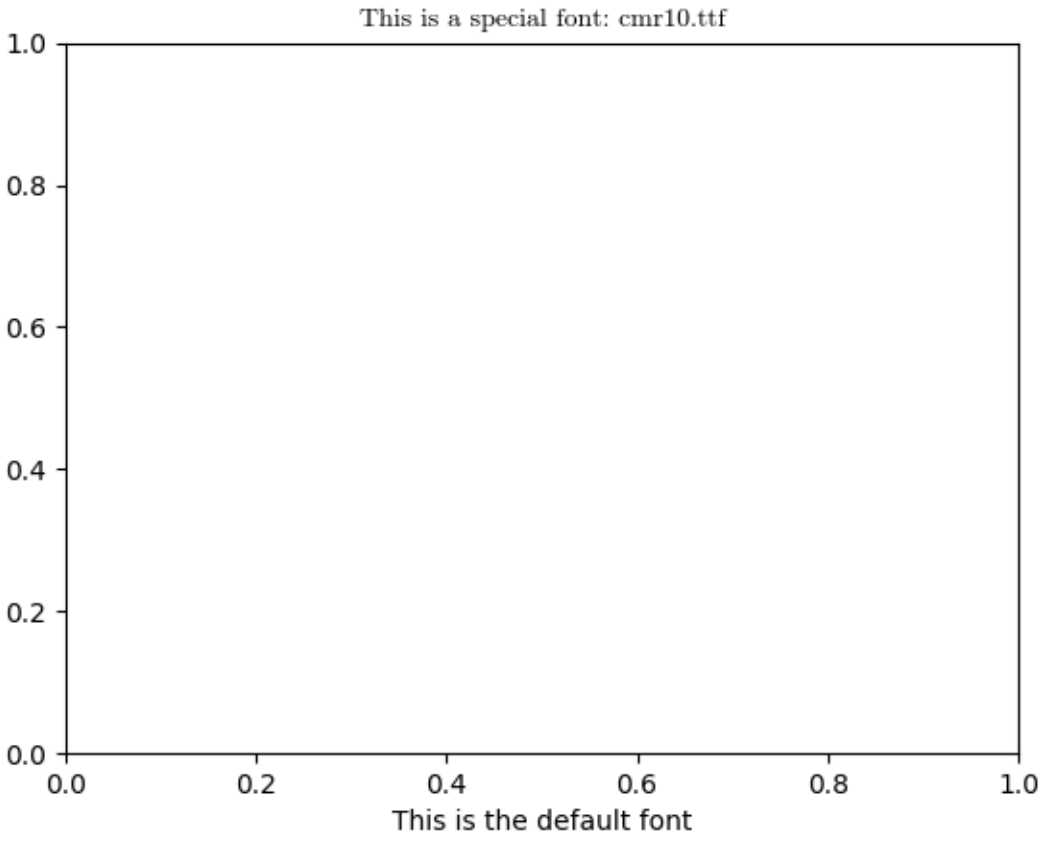

### Using ttf font files

Although it is usually not a good idea to explicitly point to a single ttf file for a font instance, you can do so by passing a `pathlib.Path` instance as the *font* parameter. Note that passing paths as `str`s is intentionally not supported, but you can simply wrap `str`s in `pathlib.Path`s as needed.

Here, we use the Computer Modern roman font (`cmr10`) shipped with Matplotlib.

For a more flexible solution, see *Configuring the font family* and *Fonts demo (object-oriented style)*.

```
from pathlib import Path

import matplotlib.pyplot as plt

import matplotlib as mpl

fig, ax = plt.subplots()

fpath = Path(mpl.get_data_path(), "fonts/ttf/cmr10.ttf")
ax.set_title(f'This is a special font: {fpath.name}', font=fpath)
ax.set_xlabel('This is the default font')

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.set_title*

## Font table

Matplotlib's font support is provided by the FreeType library.

Here, we use *table* to draw a table that shows the glyphs by Unicode codepoint. For brevity, the table only contains the first 256 glyphs.

The example is a full working script. You can download it and use it to investigate a font by running

```
python font_table.py /path/to/font/file
```

DejaVuSans.ttf

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 00 | | | | | | | | | | | | | | | | |
| 10 | | | | | | | | | | | | | | | | |
| 20 | | ! | " | # | $ | % | & | ' | ( | ) | * | + | , | - | . | / |
| 30 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | : | ; | < | = | > | ? |
| 40 | @ | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
| 50 | P | Q | R | S | T | U | V | W | X | Y | Z | [ | \ | ] | ^ | _ |
| 60 | ` | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o |
| 70 | p | q | r | s | t | u | v | w | x | y | z | { | \| | } | ~ | |
| 80 | | | | | | | | | | | | | | | | |
| 90 | | | | | | | | | | | | | | | | |
| A0 | | ¡ | ¢ | £ | ¤ | ¥ | ¦ | § | ¨ | © | ª | « | ¬ | - | ® | ¯ |
| B0 | ° | ± | ² | ³ | ´ | µ | ¶ | · | ¸ | ¹ | º | » | ¼ | ½ | ¾ | ¿ |
| C0 | À | Á | Â | Ã | Ä | Å | Æ | Ç | È | É | Ê | Ë | Ì | Í | Î | Ï |
| D0 | Ð | Ñ | Ò | Ó | Ô | Õ | Ö | × | Ø | Ù | Ú | Û | Ü | Ý | Þ | ß |
| E0 | à | á | â | ã | ä | å | æ | ç | è | é | ê | ë | ì | í | î | ï |
| F0 | ð | ñ | ò | ó | ô | õ | ö | ÷ | ø | ù | ú | û | ü | ý | þ | ÿ |

```
import os
from pathlib import Path
import unicodedata

import matplotlib.pyplot as plt

import matplotlib.font_manager as fm
from matplotlib.ft2font import FT2Font


def print_glyphs(path):
    """
    Print the all glyphs in the given font file to stdout.

    Parameters
    ----------
    path : str or None
        The path to the font file.  If None, use Matplotlib's default font.
    """
    if path is None:
        path = fm.findfont(fm.FontProperties())  # The default font.

    font = FT2Font(path)

    charmap = font.get_charmap()
    max_indices_len = len(str(max(charmap.values())))

    print("The font face contains the following glyphs:")
    for char_code, glyph_index in charmap.items():
        char = chr(char_code)
        name = unicodedata.name(
                char,
```

(continues on next page)

(continued from previous page)

```
                f"{char_code:#x} ({font.get_glyph_name(glyph_index)})")
        print(f"{glyph_index:>{max_indices_len}} {char} {name}")


def draw_font_table(path):
    """
    Draw a font table of the first 255 chars of the given font.

    Parameters
    ----------
    path : str or None
        The path to the font file.  If None, use Matplotlib's default font.
    """
    if path is None:
        path = fm.findfont(fm.FontProperties())  # The default font.

    font = FT2Font(path)
    # A charmap is a mapping of "character codes" (in the sense of a character
    # encoding, e.g. latin-1) to glyph indices (i.e. the internal storage␣
↪table
    # of the font face).
    # In FreeType>=2.1, a Unicode charmap (i.e. mapping Unicode codepoints)
    # is selected by default.  Moreover, recent versions of FreeType will
    # automatically synthesize such a charmap if the font does not include one
    # (this behavior depends on the font format; for example it is present
    # since FreeType 2.0 for Type 1 fonts but only since FreeType 2.8 for
    # TrueType (actually, SFNT) fonts).
    # The code below (specifically, the ``chr(char_code)`` call) assumes that
    # we have indeed selected a Unicode charmap.
    codes = font.get_charmap().items()

    labelc = [f"{i:X}" for i in range(16)]
    labelr = [f"{i:02X}" for i in range(0, 16*16, 16)]
    chars = [["" for c in range(16)] for r in range(16)]

    for char_code, glyph_index in codes:
        if char_code >= 256:
            continue
        row, col = divmod(char_code, 16)
        chars[row][col] = chr(char_code)

    fig, ax = plt.subplots(figsize=(8, 4))
    ax.set_title(os.path.basename(path))
    ax.set_axis_off()

    table = ax.table(
        cellText=chars,
        rowLabels=labelr,
        colLabels=labelc,
        rowColours=["palegreen"] * 16,
        colColours=["palegreen"] * 16,
        cellColours=[[".95" for c in range(16)] for r in range(16)],
```

(continues on next page)

(continued from previous page)

```
        cellLoc='center',
        loc='upper left',
    )
    for key, cell in table.get_celld().items():
        row, col = key
        if row > 0 and col > -1:  # Beware of table's idiosyncratic indexing..
 ↪.
            cell.set_text_props(font=Path(path))

    fig.tight_layout()
    plt.show()


if __name__ == "__main__":
    from argparse import ArgumentParser

    parser = ArgumentParser(description="Display a font table.")
    parser.add_argument("path", nargs="?", help="Path to the font file.")
    parser.add_argument("--print-all", action="store_true",
                        help="Additionally, print all chars to stdout.")
    args = parser.parse_args()

    if args.print_all:
        print_glyphs(args.path)
    draw_font_table(args.path)
```

**Total running time of the script:** (0 minutes 1.061 seconds)

### Fonts demo (object-oriented style)

Set font properties using setters.

See *Fonts demo (keyword arguments)* to achieve the same effect using keyword arguments.

```
import matplotlib.pyplot as plt

from matplotlib.font_manager import FontProperties

fig = plt.figure()
alignment = {'horizontalalignment': 'center', 'verticalalignment': 'baseline'}
yp = [0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2]
heading_font = FontProperties(size='large')

# Show family options
fig.text(0.1, 0.9, 'family', fontproperties=heading_font, **alignment)
families = ['serif', 'sans-serif', 'cursive', 'fantasy', 'monospace']
for k, family in enumerate(families):
    font = FontProperties()
    font.set_family(family)
    fig.text(0.1, yp[k], family, fontproperties=font, **alignment)

# Show style options
styles = ['normal', 'italic', 'oblique']
fig.text(0.3, 0.9, 'style', fontproperties=heading_font, **alignment)
for k, style in enumerate(styles):
    font = FontProperties()
    font.set_family('sans-serif')
```

(continues on next page)

(continued from previous page)

```
    font.set_style(style)
    fig.text(0.3, yp[k], style, fontproperties=font, **alignment)

# Show variant options
variants = ['normal', 'small-caps']
fig.text(0.5, 0.9, 'variant', fontproperties=heading_font, **alignment)
for k, variant in enumerate(variants):
    font = FontProperties()
    font.set_family('serif')
    font.set_variant(variant)
    fig.text(0.5, yp[k], variant, fontproperties=font, **alignment)

# Show weight options
weights = ['light', 'normal', 'medium', 'semibold', 'bold', 'heavy', 'black']
fig.text(0.7, 0.9, 'weight', fontproperties=heading_font, **alignment)
for k, weight in enumerate(weights):
    font = FontProperties()
    font.set_weight(weight)
    fig.text(0.7, yp[k], weight, fontproperties=font, **alignment)

# Show size options
sizes = [
    'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large']
fig.text(0.9, 0.9, 'size', fontproperties=heading_font, **alignment)
for k, size in enumerate(sizes):
    font = FontProperties()
    font.set_size(size)
    fig.text(0.9, yp[k], size, fontproperties=font, **alignment)

# Show bold italic
font = FontProperties(style='italic', weight='bold', size='x-small')
fig.text(0.3, 0.1, 'bold italic', fontproperties=font, **alignment)
font = FontProperties(style='italic', weight='bold', size='medium')
fig.text(0.3, 0.2, 'bold italic', fontproperties=font, **alignment)
font = FontProperties(style='italic', weight='bold', size='x-large')
fig.text(0.3, 0.3, 'bold italic', fontproperties=font, **alignment)

plt.show()
```

### Fonts demo (keyword arguments)

Set font properties using keyword arguments.

See *Fonts demo (object-oriented style)* to achieve the same effect using setters.

```
import matplotlib.pyplot as plt

fig = plt.figure()
alignment = {'horizontalalignment': 'center', 'verticalalignment': 'baseline'}
yp = [0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2]

# Show family options
fig.text(0.1, 0.9, 'family', size='large', **alignment)
families = ['serif', 'sans-serif', 'cursive', 'fantasy', 'monospace']
for k, family in enumerate(families):
    fig.text(0.1, yp[k], family, family=family, **alignment)

# Show style options
fig.text(0.3, 0.9, 'style', **alignment)
styles = ['normal', 'italic', 'oblique']
for k, style in enumerate(styles):
    fig.text(0.3, yp[k], style, family='sans-serif', style=style, **alignment)

# Show variant options
fig.text(0.5, 0.9, 'variant', **alignment)
variants = ['normal', 'small-caps']
for k, variant in enumerate(variants):
    fig.text(0.5, yp[k], variant, family='serif', variant=variant,␣
```

(continues on next page)

(continued from previous page)

```
 ↪**alignment)

# Show weight options
fig.text(0.7, 0.9, 'weight', **alignment)
weights = ['light', 'normal', 'medium', 'semibold', 'bold', 'heavy', 'black']
for k, weight in enumerate(weights):
    fig.text(0.7, yp[k], weight, weight=weight, **alignment)

# Show size options
fig.text(0.9, 0.9, 'size', **alignment)
sizes = [
    'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large']
for k, size in enumerate(sizes):
    fig.text(0.9, yp[k], size, size=size, **alignment)

# Show bold italic
fig.text(0.3, 0.1, 'bold italic',
         style='italic', weight='bold', size='x-small', **alignment)
fig.text(0.3, 0.2, 'bold italic',
         style='italic', weight='bold', size='medium', **alignment)
fig.text(0.3, 0.3, 'bold italic',
         style='italic', weight='bold', size='x-large', **alignment)

plt.show()
```

## Labelling subplots

Labelling subplots is relatively straightforward, and varies, so Matplotlib does not have a general method for doing this.

We showcase two methods to position text at a given physical offset (in fontsize units or in points) away from a corner of the Axes: one using *annotate*, and one using *ScaledTranslation*.

For convenience, this example uses *pyplot.subplot_mosaic* and subplot labels as keys for the subplots. However, the approach also works with *pyplot.subplots* or keys that are different from what you want to label the subplot with.

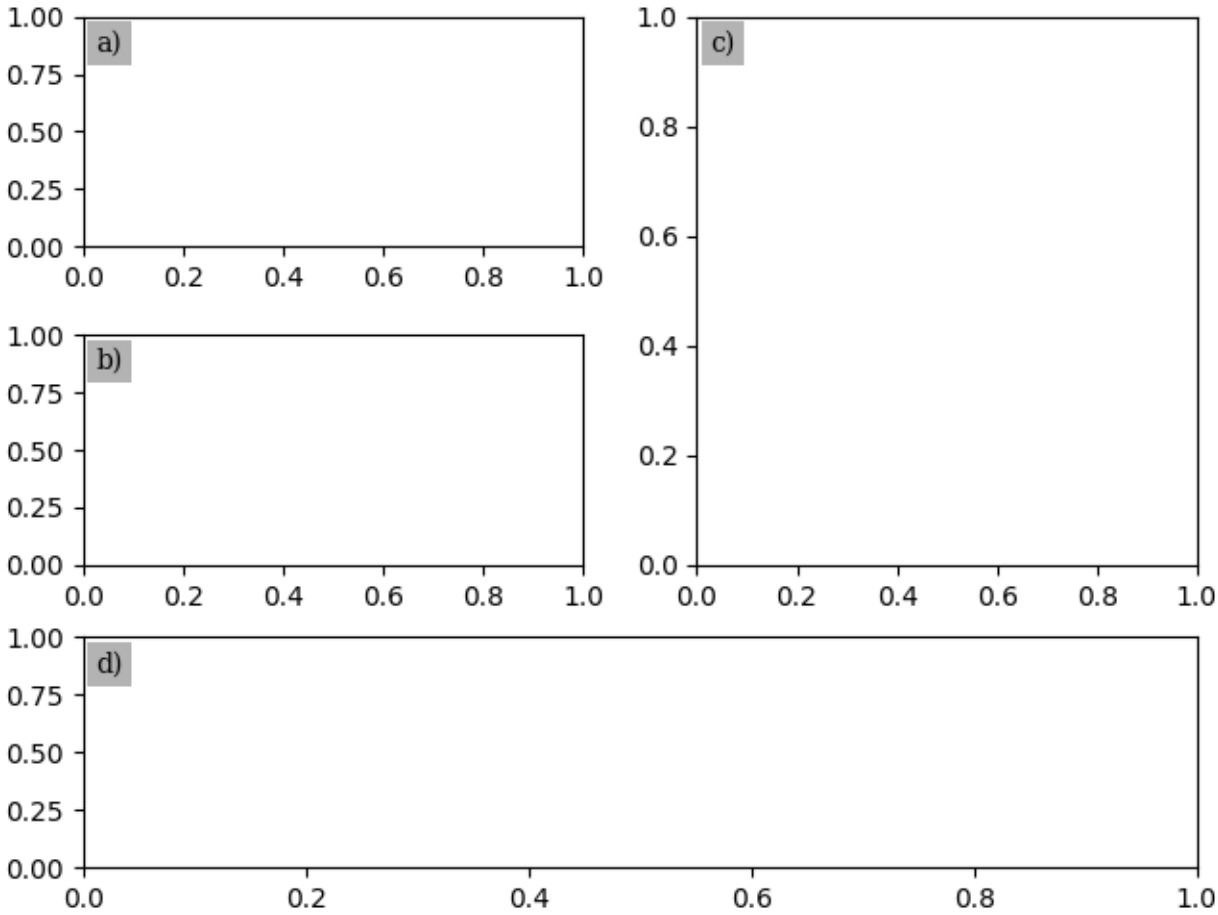
```
import matplotlib.pyplot as plt

from matplotlib.transforms import ScaledTranslation
```

```
fig, axs = plt.subplot_mosaic([['a)', 'c)'], ['b)', 'c)'], ['d)', 'd)']],
                              layout='constrained')
for label, ax in axs.items():
    # Use Axes.annotate to put the label
    # - at the top left corner (axes fraction (0, 1)),
    # - offset half-a-fontsize right and half-a-fontsize down
    #   (offset fontsize (+0.5, -0.5)),
    # i.e. just inside the axes.
    ax.annotate(
```

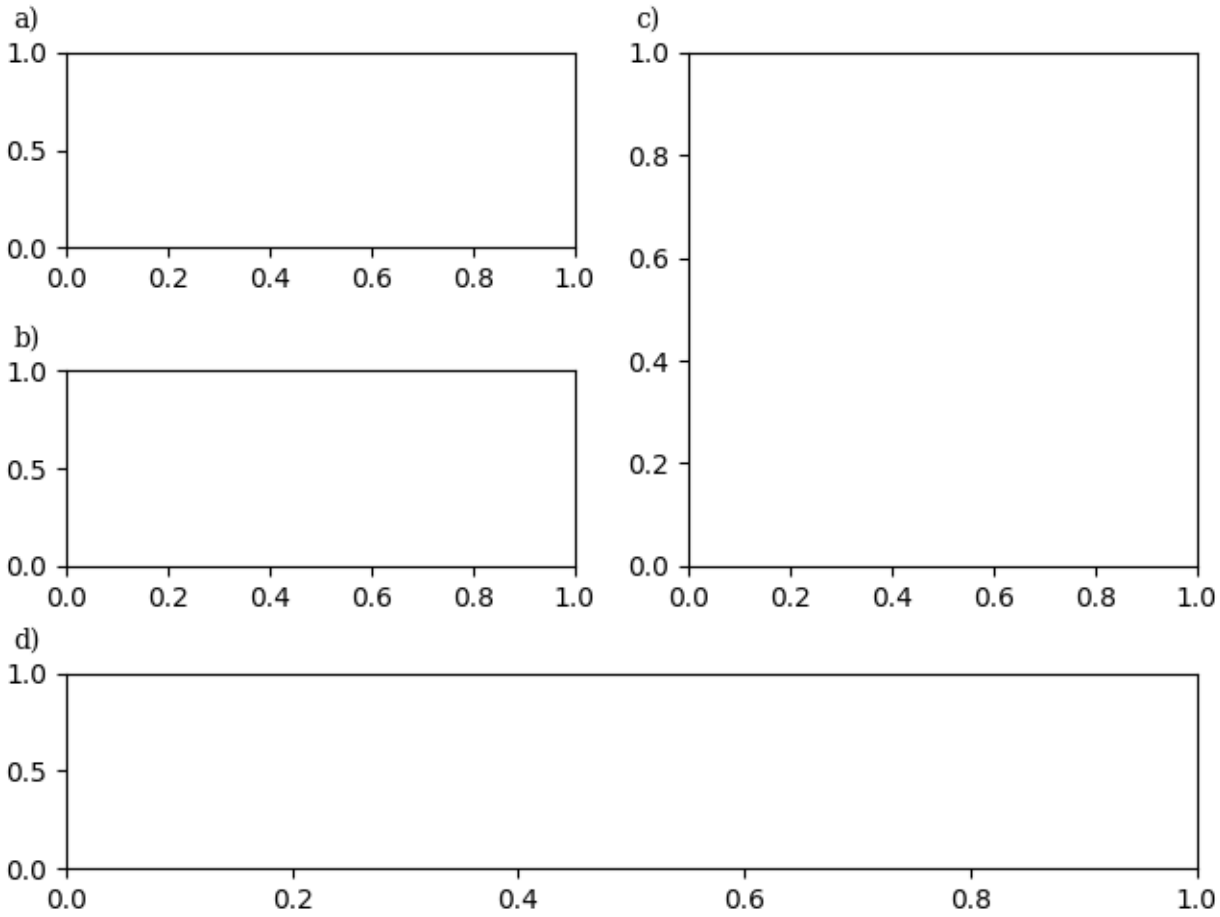
(continues on next page)

(continued from previous page)

```
        label,
        xy=(0, 1), xycoords='axes fraction',
        xytext=(+0.5, -0.5), textcoords='offset fontsize',
        fontsize='medium', verticalalignment='top', fontfamily='serif',
        bbox=dict(facecolor='0.7', edgecolor='none', pad=3.0))
```

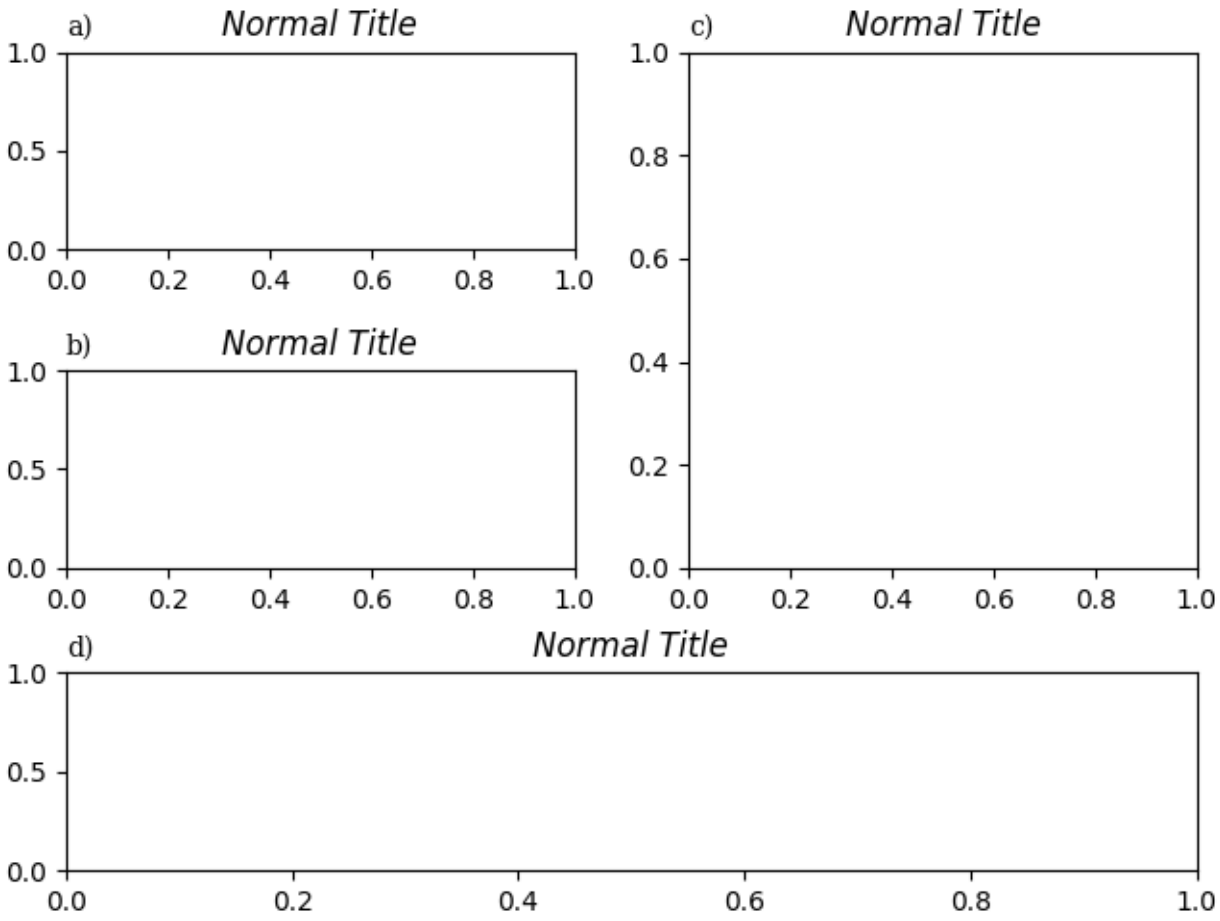



```
fig, axs = plt.subplot_mosaic([['a)', 'c)'], ['b)', 'c)'], ['d)', 'd)']],
                              layout='constrained')
for label, ax in axs.items():
    # Use ScaledTranslation to put the label
    # - at the top left corner (axes fraction (0, 1)),
    # - offset 20 pixels left and 7 pixels up (offset points (-20, +7)),
    # i.e. just outside the axes.
    ax.text(
        0.0, 1.0, label, transform=(
            ax.transAxes + ScaledTranslation(-20/72, +7/72, fig.dpi_scale_
↪trans)),
        fontsize='medium', va='bottom', fontfamily='serif')
```




If we want it aligned with the title, either incorporate in the title or use the *loc* keyword argument:

```
fig, axs = plt.subplot_mosaic([['a)', 'c)'], ['b)', 'c)'], ['d)', 'd)']],
                              layout='constrained')
for label, ax in axs.items():
    ax.set_title('Normal Title', fontstyle='italic')
    ax.set_title(label, fontfamily='serif', loc='left', fontsize='medium')

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

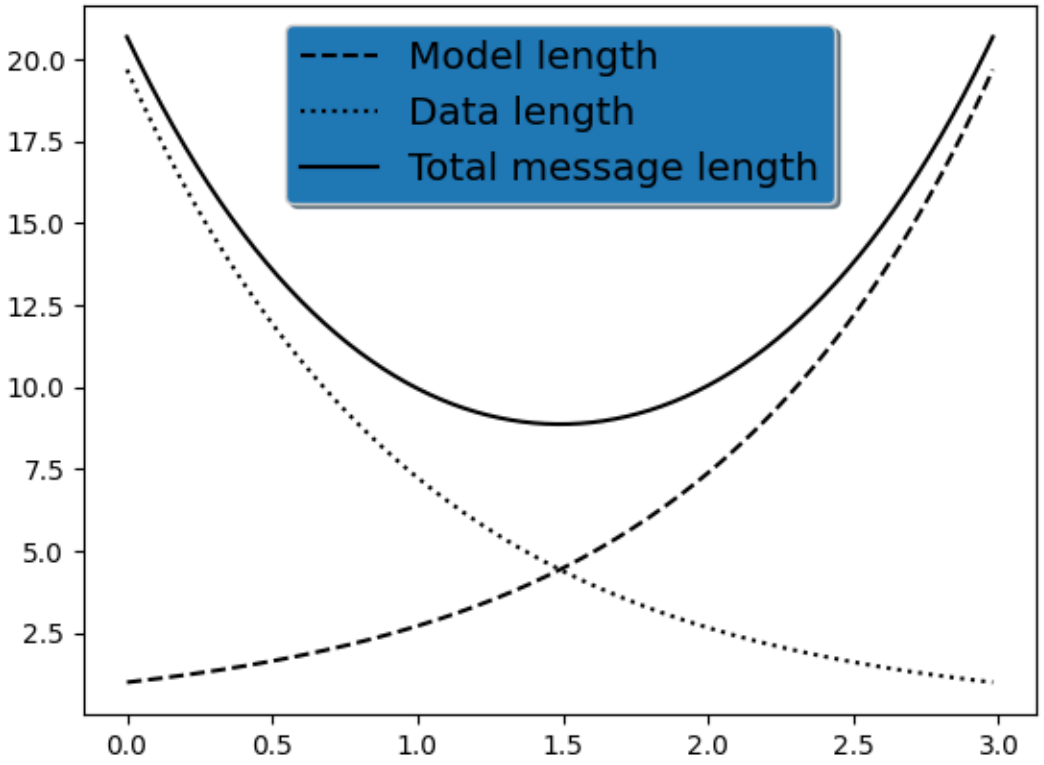
- *matplotlib.figure.Figure.subplot_mosaic* / *matplotlib.pyplot.subplot_mosaic*
- *matplotlib.axes.Axes.set_title*
- *matplotlib.axes.Axes.annotate*

**Total running time of the script:** (0 minutes 1.299 seconds)

### Legend using pre-defined labels

Defining legend labels with plots.

```
import matplotlib.pyplot as plt
import numpy as np

# Make some fake data.
a = b = np.arange(0, 3, .02)
c = np.exp(a)
d = c[::-1]

# Create plots with pre-defined labels.
fig, ax = plt.subplots()
ax.plot(a, c, 'k--', label='Model length')
ax.plot(a, d, 'k:', label='Data length')
ax.plot(a, c + d, 'k', label='Total message length')

legend = ax.legend(loc='upper center', shadow=True, fontsize='x-large')

# Put a nicer background color on the legend.
legend.get_frame().set_facecolor('C0')

plt.show()
```



**References**

The use of the following functions, methods, classes and modules is shown in this example:

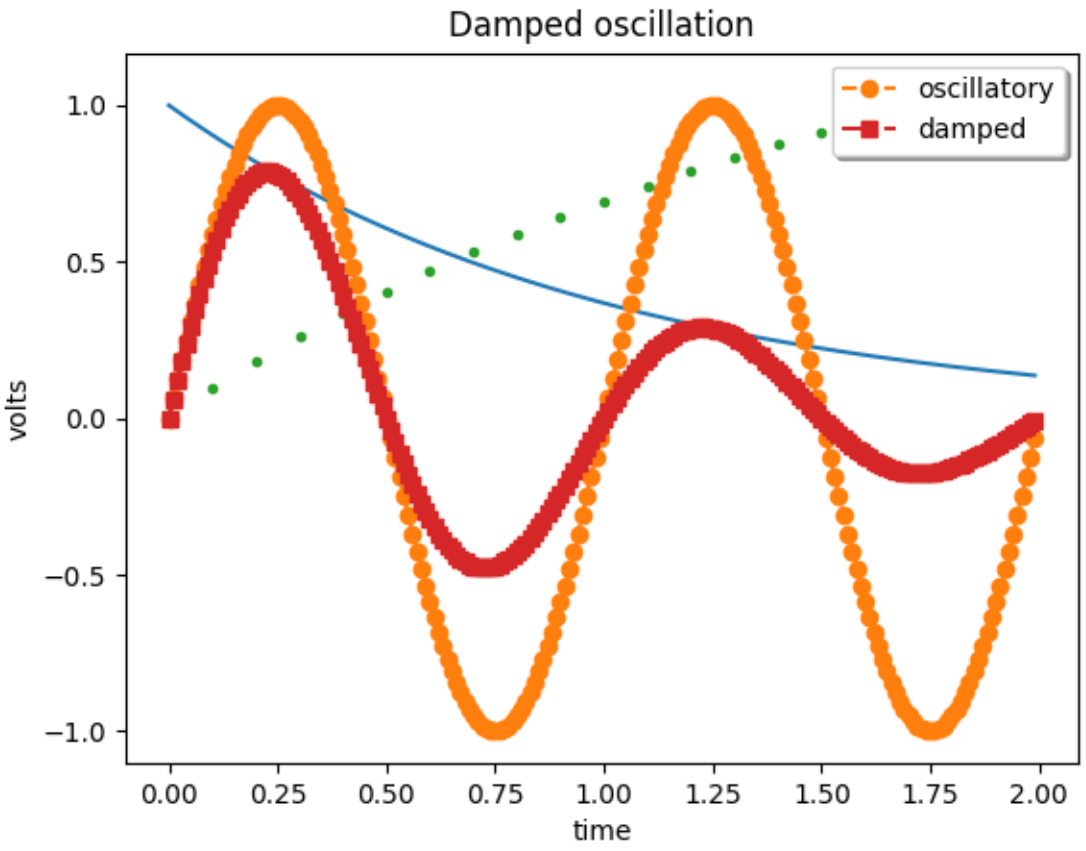
- *matplotlib.axes.Axes.plot* / *matplotlib.pyplot.plot*
- *matplotlib.axes.Axes.legend* / *matplotlib.pyplot.legend*

## Legend Demo

There are many ways to create and customize legends in Matplotlib. Below we'll show a few examples for how to do so.

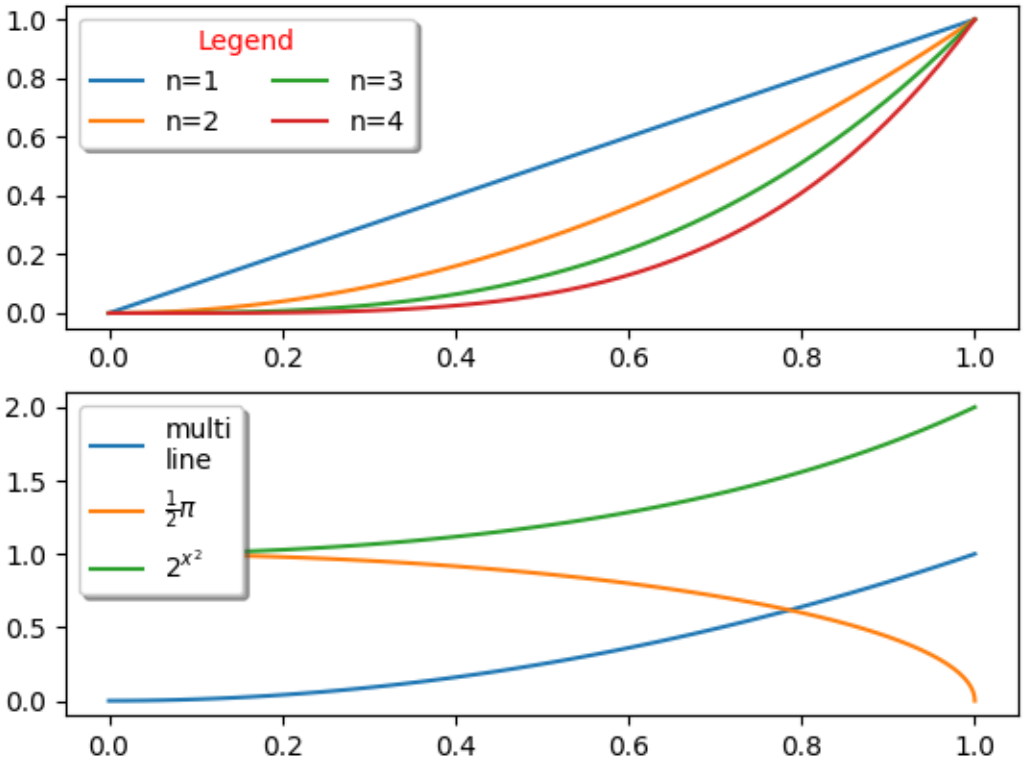
First we'll show off how to make a legend for specific lines.

```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib.collections as mcol
from matplotlib.legend_handler import HandlerLineCollection, HandlerTuple
from matplotlib.lines import Line2D

t1 = np.arange(0.0, 2.0, 0.1)
t2 = np.arange(0.0, 2.0, 0.01)

fig, ax = plt.subplots()

# note that plot returns a list of lines.  The "l1, = plot" usage
# extracts the first element of the list into l1 using tuple
# unpacking.  So l1 is a Line2D instance, not a sequence of lines
l1, = ax.plot(t2, np.exp(-t2))
l2, l3 = ax.plot(t2, np.sin(2 * np.pi * t2), '--o', t1, np.log(1 + t1), '.')
l4, = ax.plot(t2, np.exp(-t2) * np.sin(2 * np.pi * t2), 's-.')

ax.legend((l2, l4), ('oscillatory', 'damped'), loc='upper right', shadow=True)
ax.set_xlabel('time')
ax.set_ylabel('volts')
ax.set_title('Damped oscillation')
plt.show()
```



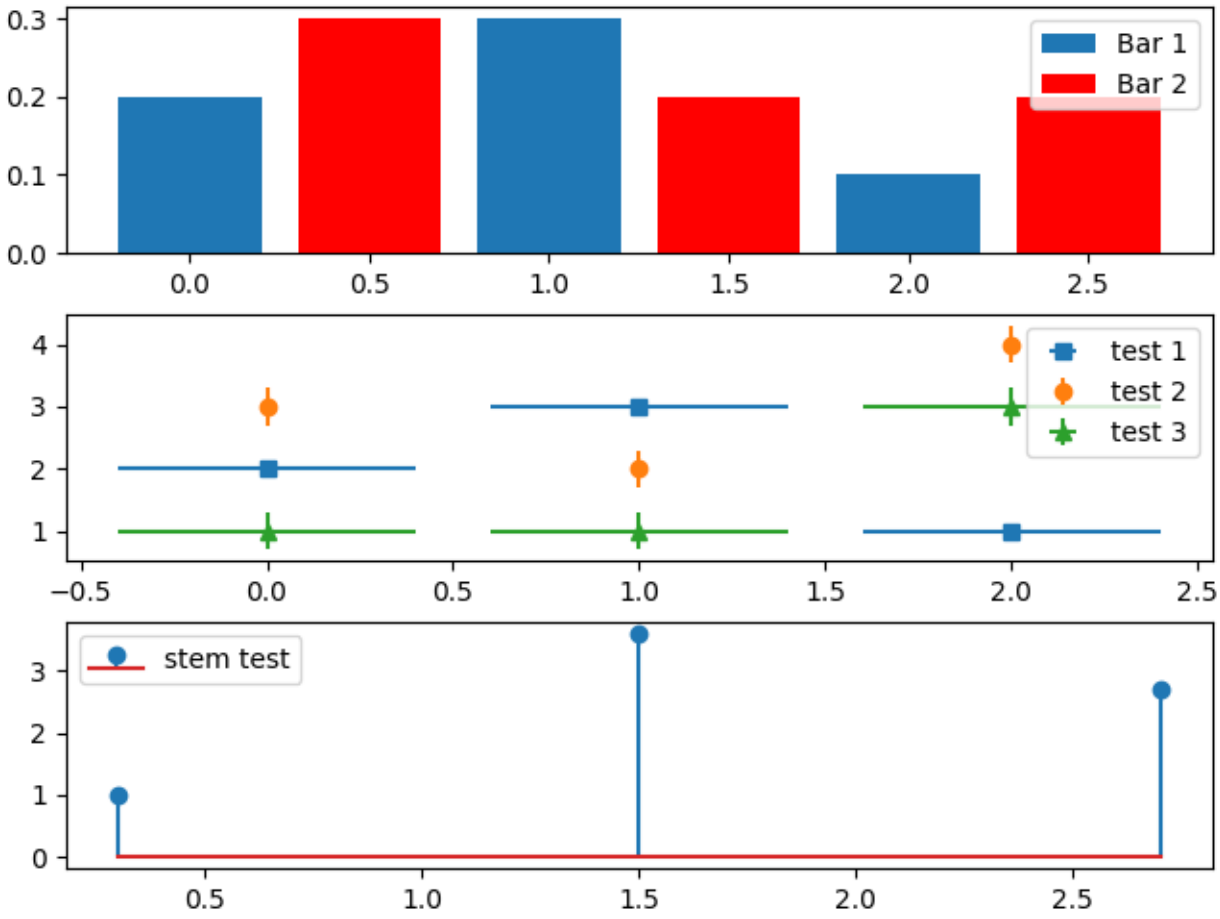

Next we'll demonstrate plotting more complex labels.

```
x = np.linspace(0, 1)

fig, (ax0, ax1) = plt.subplots(2, 1)

# Plot the lines y=x**n for n=1..4.
for n in range(1, 5):
    ax0.plot(x, x**n, label=f"{n=}")
leg = ax0.legend(loc="upper left", bbox_to_anchor=[0, 1],
                 ncols=2, shadow=True, title="Legend", fancybox=True)
leg.get_title().set_color("red")

# Demonstrate some more complex labels.
ax1.plot(x, x**2, label="multi\nline")
half_pi = np.linspace(0, np.pi / 2)
ax1.plot(np.sin(half_pi), np.cos(half_pi), label=r"$\frac{1}{2}\pi$")
ax1.plot(x, 2**(x**2), label="$2^{x^2}$")
ax1.legend(shadow=True, fancybox=True)

plt.show()
```




Here we attach legends to more complex plots.

```
fig, axs = plt.subplots(3, 1, layout="constrained")
top_ax, middle_ax, bottom_ax = axs

top_ax.bar([0, 1, 2], [0.2, 0.3, 0.1], width=0.4, label="Bar 1",
           align="center")
top_ax.bar([0.5, 1.5, 2.5], [0.3, 0.2, 0.2], color="red", width=0.4,
           label="Bar 2", align="center")
top_ax.legend()

middle_ax.errorbar([0, 1, 2], [2, 3, 1], xerr=0.4, fmt="s", label="test 1")
middle_ax.errorbar([0, 1, 2], [3, 2, 4], yerr=0.3, fmt="o", label="test 2")
middle_ax.errorbar([0, 1, 2], [1, 1, 3], xerr=0.4, yerr=0.3, fmt="^",
                   label="test 3")
middle_ax.legend()

bottom_ax.stem([0.3, 1.5, 2.7], [1, 3.6, 2.7], label="stem test")
bottom_ax.legend()

plt.show()
```



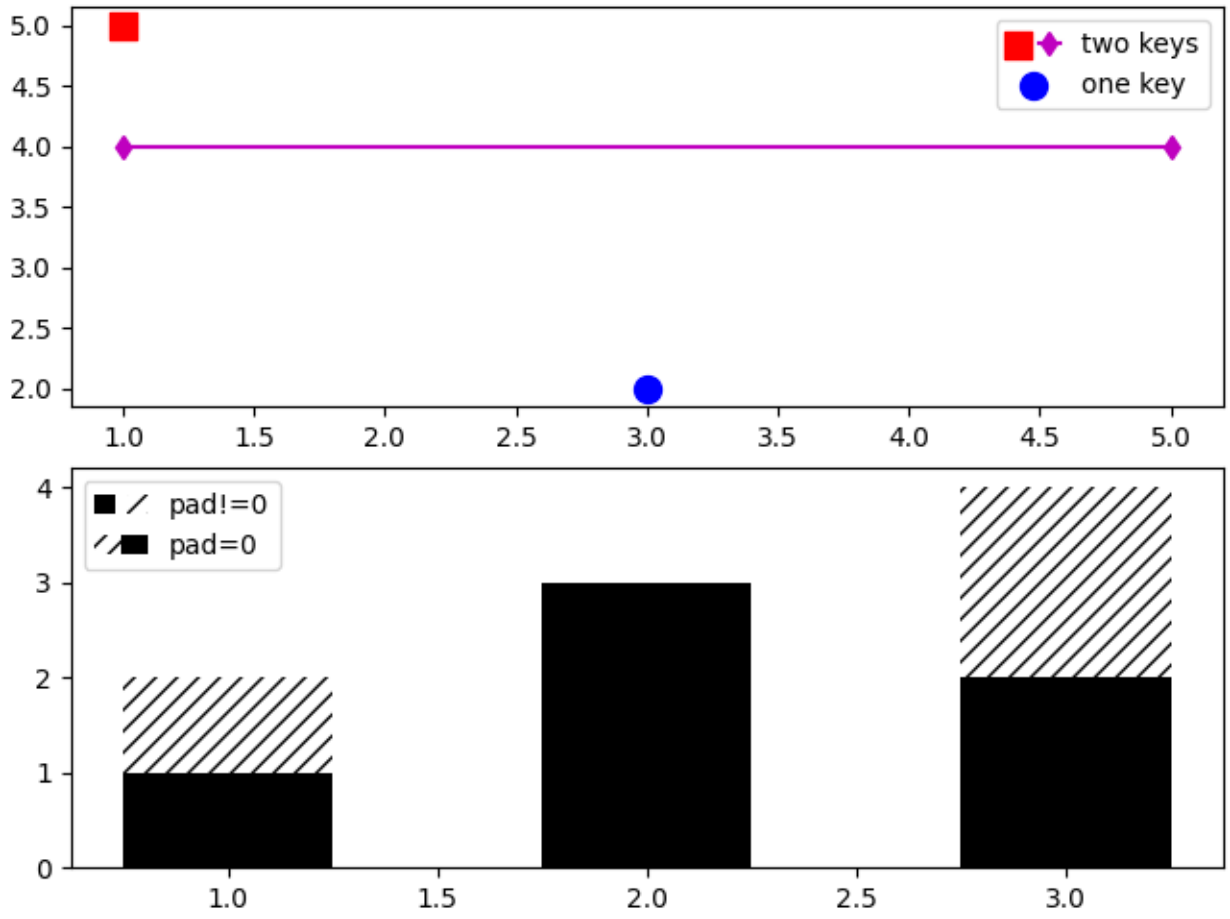

Now we'll showcase legend entries with more than one legend key.

```
fig, (ax1, ax2) = plt.subplots(2, 1, layout='constrained')

# First plot: two legend keys for a single entry
p1 = ax1.scatter([1], [5], c='r', marker='s', s=100)
p2 = ax1.scatter([3], [2], c='b', marker='o', s=100)
# `plot` returns a list, but we want the handle - thus the comma on the left
p3, = ax1.plot([1, 5], [4, 4], 'm-d')

# Assign two of the handles to the same legend entry by putting them in a␣
 ↪tuple
# and using a generic handler map (which would be used for any additional
# tuples of handles like (p1, p3)).
l = ax1.legend([(p1, p3), p2], ['two keys', 'one key'], scatterpoints=1,
               numpoints=1, handler_map={tuple: HandlerTuple(ndivide=None)})

# Second plot: plot two bar charts on top of each other and change the padding
# between the legend keys
x_left = [1, 2, 3]
y_pos = [1, 3, 2]
y_neg = [2, 1, 4]
```

(continues on next page)

(continued from previous page)

```
rneg = ax2.bar(x_left, y_neg, width=0.5, color='w', hatch='///', label='-1')
rpos = ax2.bar(x_left, y_pos, width=0.5, color='k', label='+1')

# Treat each legend entry differently by using specific `HandlerTuple`s
l = ax2.legend([(rpos, rneg), (rneg, rpos)], ['pad!=0', 'pad=0'],
               handler_map={(rpos, rneg): HandlerTuple(ndivide=None),
                            (rneg, rpos): HandlerTuple(ndivide=None, pad=0.)})
plt.show()
```



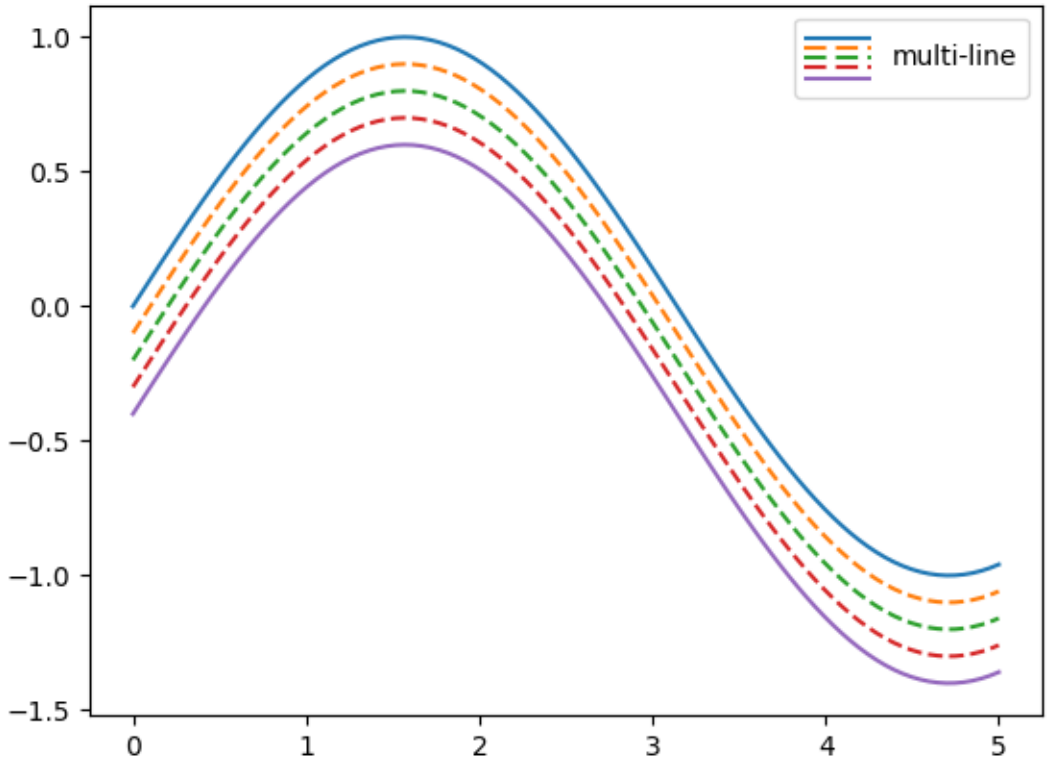

Finally, it is also possible to write custom classes that define how to stylize legends.

```
class HandlerDashedLines(HandlerLineCollection):
    """
    Custom Handler for LineCollection instances.
    """
    def create_artists(self, legend, orig_handle,
                       xdescent, ydescent, width, height, fontsize, trans):
        # figure out how many lines there are
        numlines = len(orig_handle.get_segments())
        xdata, xdata_marker = self.get_xdata(legend, xdescent, ydescent,
                                             width, height, fontsize)
        leglines = []
```

(continues on next page)

(continued from previous page)

```
        # divide the vertical space where the lines will go
        # into equal parts based on the number of lines
        ydata = np.full_like(xdata, height / (numlines + 1))
        # for each line, create the line at the proper location
        # and set the dash pattern
        for i in range(numlines):
            legline = Line2D(xdata, ydata * (numlines - i) - ydescent)
            self.update_prop(legline, orig_handle, legend)
            # set color, dash pattern, and linewidth to that
            # of the lines in linecollection
            try:
                color = orig_handle.get_colors()[i]
            except IndexError:
                color = orig_handle.get_colors()[0]
            try:
                dashes = orig_handle.get_dashes()[i]
            except IndexError:
                dashes = orig_handle.get_dashes()[0]
            try:
                lw = orig_handle.get_linewidths()[i]
            except IndexError:
                lw = orig_handle.get_linewidths()[0]
            if dashes[1] is not None:
                legline.set_dashes(dashes[1])
            legline.set_color(color)
            legline.set_transform(trans)
            legline.set_linewidth(lw)
            leglines.append(legline)
        return leglines

x = np.linspace(0, 5, 100)

fig, ax = plt.subplots()
colors = plt.rcParams['axes.prop_cycle'].by_key()['color'][:5]
styles = ['solid', 'dashed', 'dashed', 'dashed', 'solid']
for i, color, style in zip(range(5), colors, styles):
    ax.plot(x, np.sin(x) - .1 * i, c=color, ls=style)

# make proxy artists
# make list of one line -- doesn't matter what the coordinates are
line = [[(0, 0)]]
# set up the proxy artist
lc = mcol.LineCollection(5 * line, linestyles=styles, colors=colors)
# create the legend
ax.legend([lc], ['multi-line'], handler_map={type(lc): HandlerDashedLines()},
          handlelength=2.5, handleheight=3)

plt.show()
```




**Total running time of the script:** (0 minutes 2.031 seconds)

### Artist within an artist

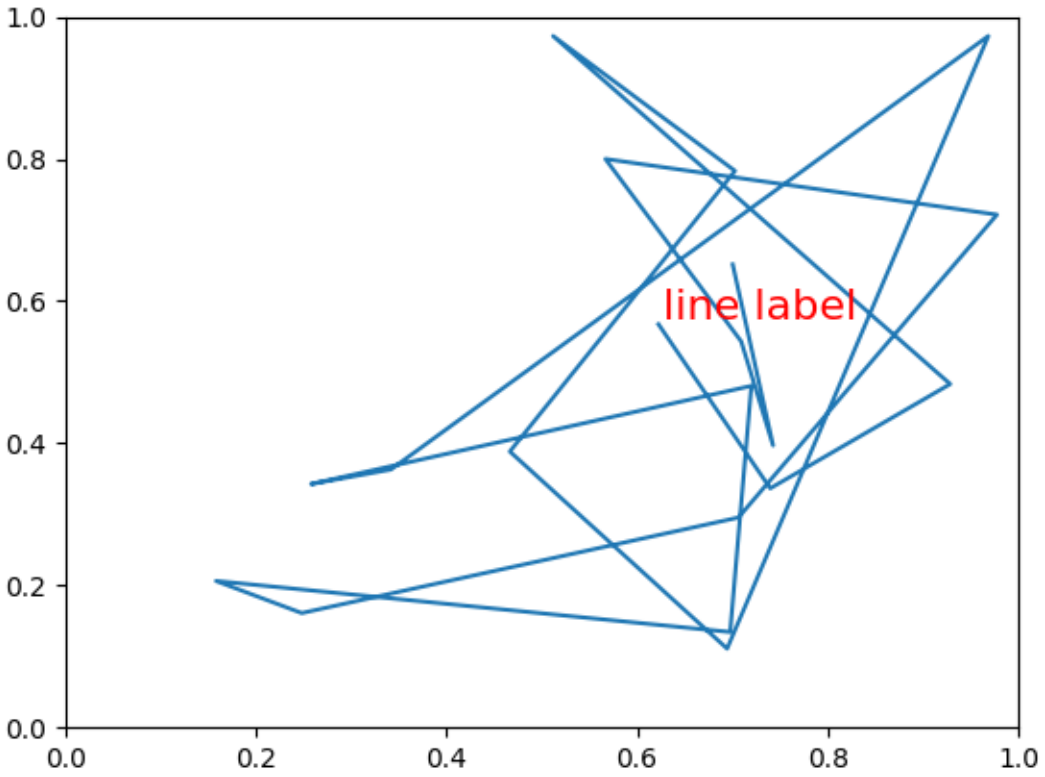
Override basic methods so an artist can contain another artist. In this case, the line contains a Text instance to label it.

```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib.lines as lines
import matplotlib.text as mtext
import matplotlib.transforms as mtransforms


class MyLine(lines.Line2D):
    def __init__(self, *args, **kwargs):
        # we'll update the position when the line data is set
        self.text = mtext.Text(0, 0, '')
        super().__init__(*args, **kwargs)

        # we can't access the label attr until *after* the line is
```

(continues on next page)

(continued from previous page)

```
        # initiated
        self.text.set_text(self.get_label())

    def set_figure(self, figure):
        self.text.set_figure(figure)
        super().set_figure(figure)

    # Override the Axes property setter to set Axes on our children as well.
    @lines.Line2D.axes.setter
    def axes(self, new_axes):
        self.text.axes = new_axes
        lines.Line2D.axes.fset(self, new_axes)  # Call the superclass␣
↪property setter.

    def set_transform(self, transform):
        # 2 pixel offset
        texttrans = transform + mtransforms.Affine2D().translate(2, 2)
        self.text.set_transform(texttrans)
        super().set_transform(transform)

    def set_data(self, x, y):
        if len(x):
            self.text.set_position((x[-1], y[-1]))

        super().set_data(x, y)

    def draw(self, renderer):
        # draw my label at the end of the line with 2 pixel offset
        super().draw(renderer)
        self.text.draw(renderer)

# Fixing random state for reproducibility
np.random.seed(19680801)


fig, ax = plt.subplots()
x, y = np.random.rand(2, 20)
line = MyLine(x, y, mfc='red', ms=12, label='line label')
line.text.set_color('red')
line.text.set_fontsize(16)

ax.add_line(line)

plt.show()
```




**References**

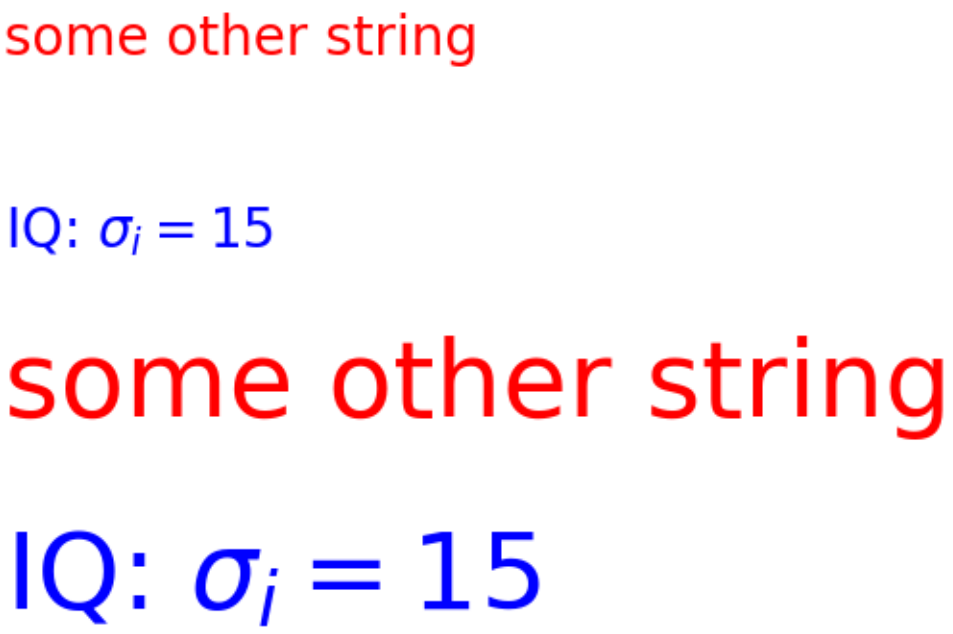
The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.lines*
- *matplotlib.lines.Line2D*
- *matplotlib.lines.Line2D.set_data*
- *matplotlib.artist*
- *matplotlib.artist.Artist*
- *matplotlib.artist.Artist.draw*
- *matplotlib.artist.Artist.set_transform*
- *matplotlib.text*
- *matplotlib.text.Text*
- *matplotlib.text.Text.set_color*
- *matplotlib.text.Text.set_fontsize*
- *matplotlib.text.Text.set_position*

- *matplotlib.axes.Axes.add_line*
- *matplotlib.transforms*
- *matplotlib.transforms.Affine2D*

### Convert texts to images

```
from io import BytesIO

import matplotlib.pyplot as plt

from matplotlib.figure import Figure
from matplotlib.transforms import IdentityTransform


def text_to_rgba(s, *, dpi, **kwargs):
    # To convert a text string to an image, we can:
    # - draw it on an empty and transparent figure;
    # - save the figure to a temporary buffer using ``bbox_inches="tight",
    #   pad_inches=0`` which will pick the correct area to save;
    # - load the buffer using ``plt.imread``.
    #
    # (If desired, one can also directly save the image to the filesystem.)
    fig = Figure(facecolor="none")
    fig.text(0, 0, s, **kwargs)
    with BytesIO() as buf:
        fig.savefig(buf, dpi=dpi, format="png", bbox_inches="tight",
                    pad_inches=0)
        buf.seek(0)
        rgba = plt.imread(buf)
    return rgba


fig = plt.figure()
rgba1 = text_to_rgba(r"IQ: $\sigma_i=15$", color="blue", fontsize=20, dpi=200)
rgba2 = text_to_rgba(r"some other string", color="red", fontsize=20, dpi=200)
# One can then draw such text images to a Figure using `.Figure.figimage`.
fig.figimage(rgba1, 100, 50)
fig.figimage(rgba2, 100, 150)

# One can also directly draw texts to a figure with positioning
# in pixel coordinates by using `.Figure.text` together with
# `.transforms.IdentityTransform`.
fig.text(100, 250, r"IQ: $\sigma_i=15$", color="blue", fontsize=20,
         transform=IdentityTransform())
fig.text(100, 350, r"some other string", color="red", fontsize=20,
         transform=IdentityTransform())

plt.show()
```



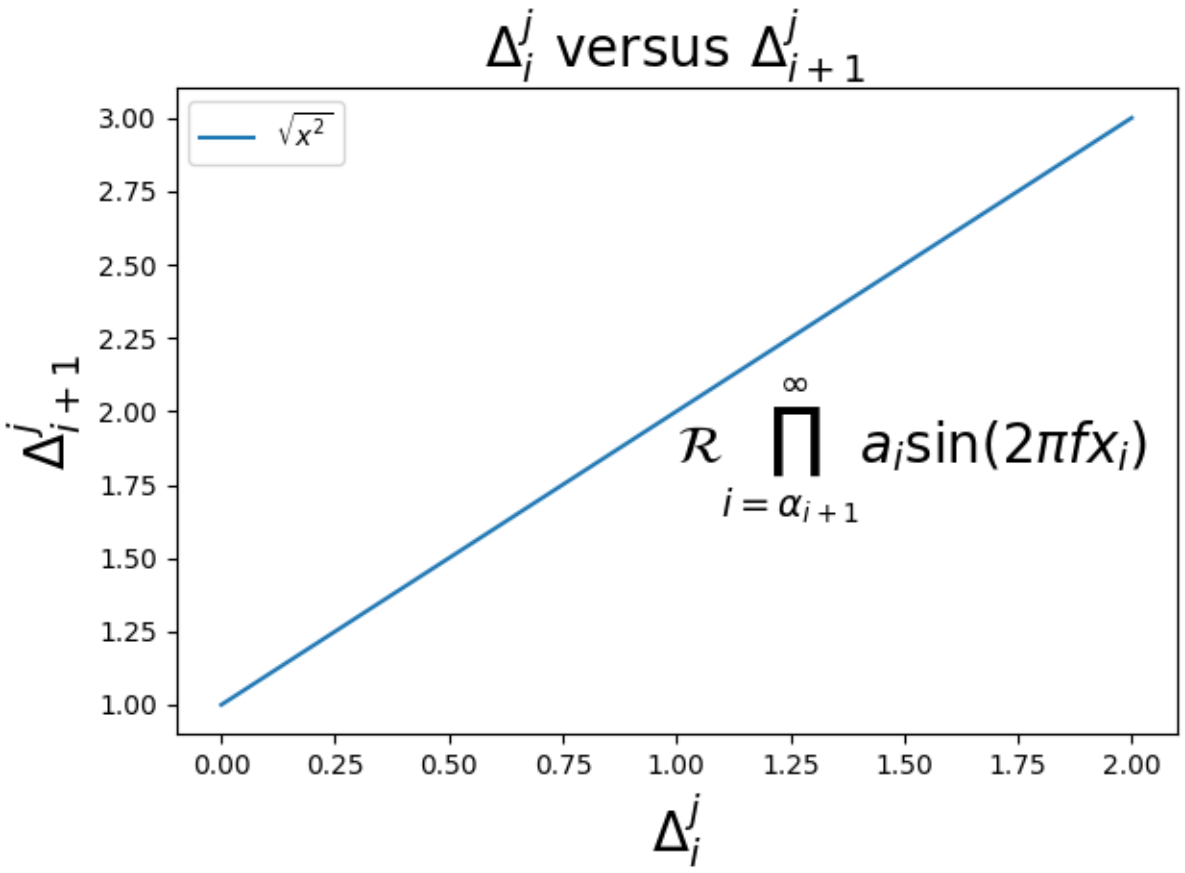

**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.figure.Figure.figimage*
- *matplotlib.figure.Figure.text*
- *matplotlib.transforms.IdentityTransform*
- *matplotlib.image.imread*

### Mathtext

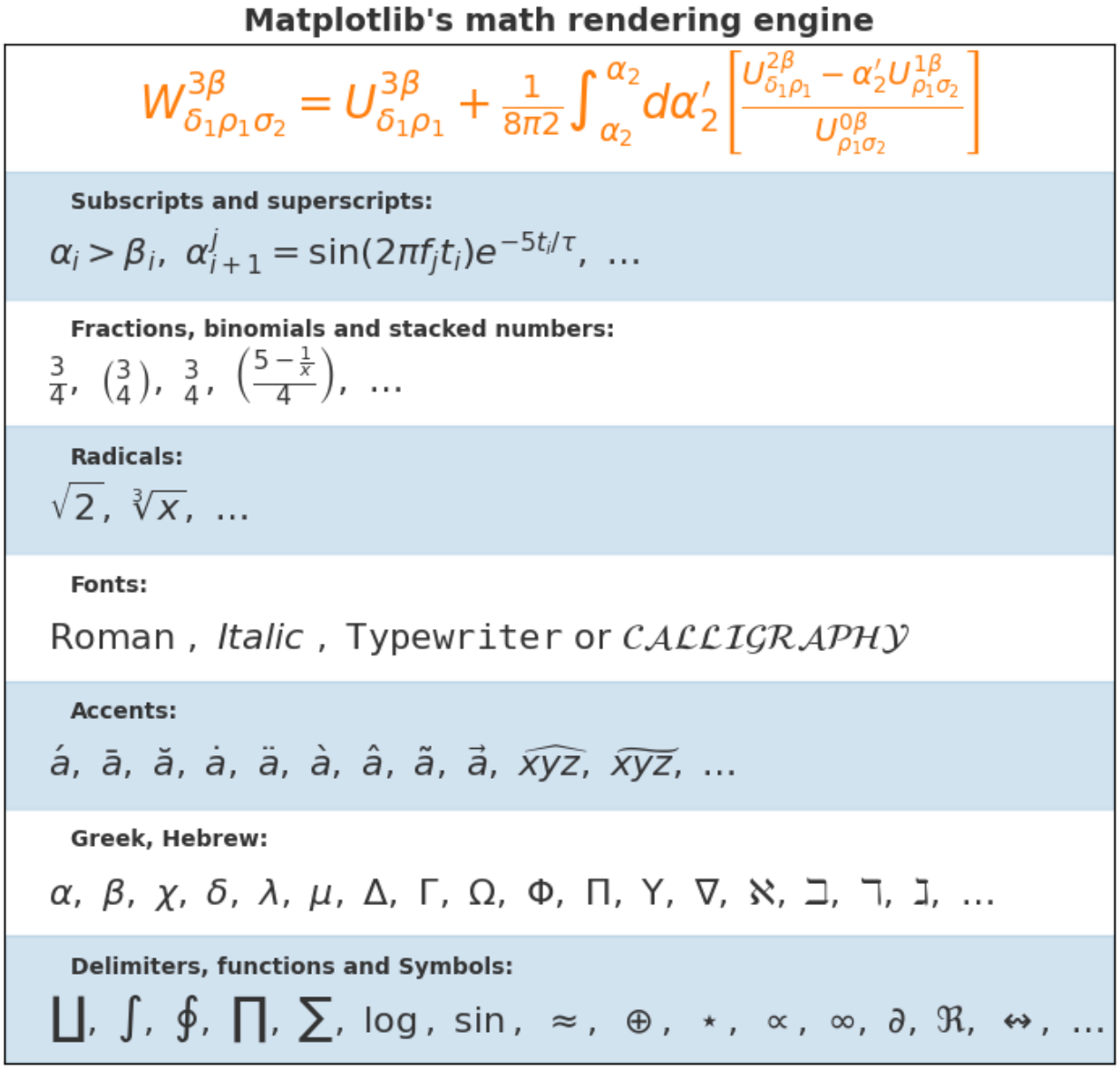
Use Matplotlib's internal LaTeX parser and layout engine. For true LaTeX rendering, see the text.usetex option.




```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()

ax.plot([1, 2, 3], label=r'$\sqrt{x^2}$')
ax.legend()

ax.set_xlabel(r'$\Delta_i^j$', fontsize=20)
ax.set_ylabel(r'$\Delta_{i+1}^j$', fontsize=20)
ax.set_title(r'$\Delta_i^j \hspace{0.4} \mathrm{versus} \hspace{0.4} '
             r'\Delta_{i+1}^j$', fontsize=20)

tex = r'$\mathcal{R}\prod_{i=\alpha_{i+1}}^\infty a_i\sin(2 \pi f x_i)$'
ax.text(1, 1.6, tex, fontsize=20, va='bottom')

fig.tight_layout()
plt.show()
```

### Mathtext Examples

Selected features of Matplotlib's math rendering engine.




```
0 $W^{3\beta}_{\delta_1 \rho_1 \sigma_2} = U^{3\beta}_{\delta_1 \rho_1} + \
 ↪frac{1}{8 \pi 2} \int^{\alpha_2}_{\alpha_2} d \alpha^\prime_2 \left[\frac{␣
 ↪U^{2\beta}_{\delta_1 \rho_1} - \alpha^\prime_2U^{1\beta}_{\rho_1 \sigma_2} }
 ↪{U^{0\beta}_{\rho_1 \sigma_2}}\right]$
1 $\alpha_i > \beta_i,\ \alpha_{i+1}^j = {\rm sin}(2\pi f_j t_i) e^{-5 t_i/\
 ↪tau},\ \ldots$
2 $\frac{3}{4},\ \binom{3}{4},\ \genfrac{}{}{0}{}{3}{4},\ \left(\frac{5 - \
 ↪frac{1}{x}}{4}\right),\ \ldots$
3 $\sqrt{2},\ \sqrt[3]{x},\ \ldots$
```

(continues on next page)

(continued from previous page)

```
4 $\mathrm{Roman}\ , \ \mathit{Italic}\ , \ \mathtt{Typewriter} \ \mathrm{or}\
 ↪ \mathcal{CALLIGRAPHY}$
5 $\acute a,\ \bar a,\ \breve a,\ \dot a,\ \ddot a, \ \grave a, \ \hat a,\ \
 ↪tilde a,\ \vec a,\ \widehat{xyz},\ \widetilde{xyz},\ \ldots$
6 $\alpha,\ \beta,\ \chi,\ \delta,\ \lambda,\ \mu,\ \Delta,\ \Gamma,\ \Omega,\
 ↪ \Phi,\ \Pi,\ \Upsilon,\ \nabla,\ \aleph,\ \beth,\ \daleth,\ \gimel,\ \ldots
 ↪$
7 $\coprod,\ \int,\ \oint,\ \prod,\ \sum,\ \log,\ \sin,\ \approx,\ \oplus,\ \
 ↪star,\ \varpropto,\ \infty,\ \partial,\ \Re,\ \leftrightsquigarrow, \ \ldots
 ↪$
```

```
import re
import subprocess
import sys

import matplotlib.pyplot as plt

# Selection of features following "Writing mathematical expressions" tutorial,
# with randomly picked examples.
mathtext_demos = {
    "Header demo":
        r"$W^{3\beta}_{\delta_1 \rho_1 \sigma_2} = "
        r"U^{3\beta}_{\delta_1 \rho_1} + \frac{1}{8 \pi 2} "
        r"\int^{\alpha_2}_{\alpha_2} d \alpha^\prime_2 \left[\frac{ "
        r"U^{2\beta}_{\delta_1 \rho_1} - \alpha^\prime_2U^{1\beta}_"
        r"{\rho_1 \sigma_2} }{U^{0\beta}_{\rho_1 \sigma_2}}\right]$",

    "Subscripts and superscripts":
        r"$\alpha_i > \beta_i,\ "
        r"\alpha_{i+1}^j = {\rm sin}(2\pi f_j t_i) e^{-5 t_i/\tau},\ "
        r"\ldots$",

    "Fractions, binomials and stacked numbers":
        r"$\frac{3}{4},\ \binom{3}{4},\ \genfrac{}{}{0}{}{3}{4},\ "
        r"\left(\frac{5 - \frac{1}{x}}{4}\right),\ \ldots$",

    "Radicals":
        r"$\sqrt{2},\ \sqrt[3]{x},\ \ldots$",

    "Fonts":
        r"$\mathrm{Roman}\ , \ \mathit{Italic}\ , \ \mathtt{Typewriter} \ "
        r"\mathrm{or}\ \mathcal{CALLIGRAPHY}$",

    "Accents":
        r"$\acute a,\ \bar a,\ \breve a,\ \dot a,\ \ddot a, \ \grave a, \ "
        r"\hat a,\ \tilde a,\ \vec a,\ \widehat{xyz},\ \widetilde{xyz},\ "
```

(continues on next page)

(continued from previous page)

```
        r"\ldots$",

    "Greek, Hebrew":
        r"$\alpha,\ \beta,\ \chi,\ \delta,\ \lambda,\ \mu,\ "
        r"\Delta,\ \Gamma,\ \Omega,\ \Phi,\ \Pi,\ \Upsilon,\ \nabla,\ "
        r"\aleph,\ \beth,\ \daleth,\ \gimel,\ \ldots$",

    "Delimiters, functions and Symbols":
        r"$\coprod,\ \int,\ \oint,\ \prod,\ \sum,\ "
        r"\log,\ \sin,\ \approx,\ \oplus,\ \star,\ \varpropto,\ "
        r"\infty,\ \partial,\ \Re,\ \leftrightsquigarrow, \ \ldots$",
}
n_lines = len(mathtext_demos)


def doall():
    # Colors used in Matplotlib online documentation.
    mpl_grey_rgb = (51 / 255, 51 / 255, 51 / 255)

    # Creating figure and axis.
    fig = plt.figure(figsize=(7, 7))
    ax = fig.add_axes([0.01, 0.01, 0.98, 0.90],
                      facecolor="white", frameon=True)
    ax.set_xlim(0, 1)
    ax.set_ylim(0, 1)
    ax.set_title("Matplotlib's math rendering engine",
                 color=mpl_grey_rgb, fontsize=14, weight='bold')
    ax.set_xticks([])
    ax.set_yticks([])

    # Gap between lines in axes coords
    line_axesfrac = 1 / n_lines

    # Plot header demonstration formula.
    full_demo = mathtext_demos['Header demo']
    ax.annotate(full_demo,
                xy=(0.5, 1. - 0.59 * line_axesfrac),
                color='tab:orange', ha='center', fontsize=20)

    # Plot feature demonstration formulae.
    for i_line, (title, demo) in enumerate(mathtext_demos.items()):
        print(i_line, demo)
        if i_line == 0:
            continue

        baseline = 1 - i_line * line_axesfrac
        baseline_next = baseline - line_axesfrac
        fill_color = ['white', 'tab:blue'][i_line % 2]
        ax.axhspan(baseline, baseline_next, color=fill_color, alpha=0.2)
        ax.annotate(f'{title}:',
                    xy=(0.06, baseline - 0.3 * line_axesfrac),
                    color=mpl_grey_rgb, weight='bold')
```

(continues on next page)

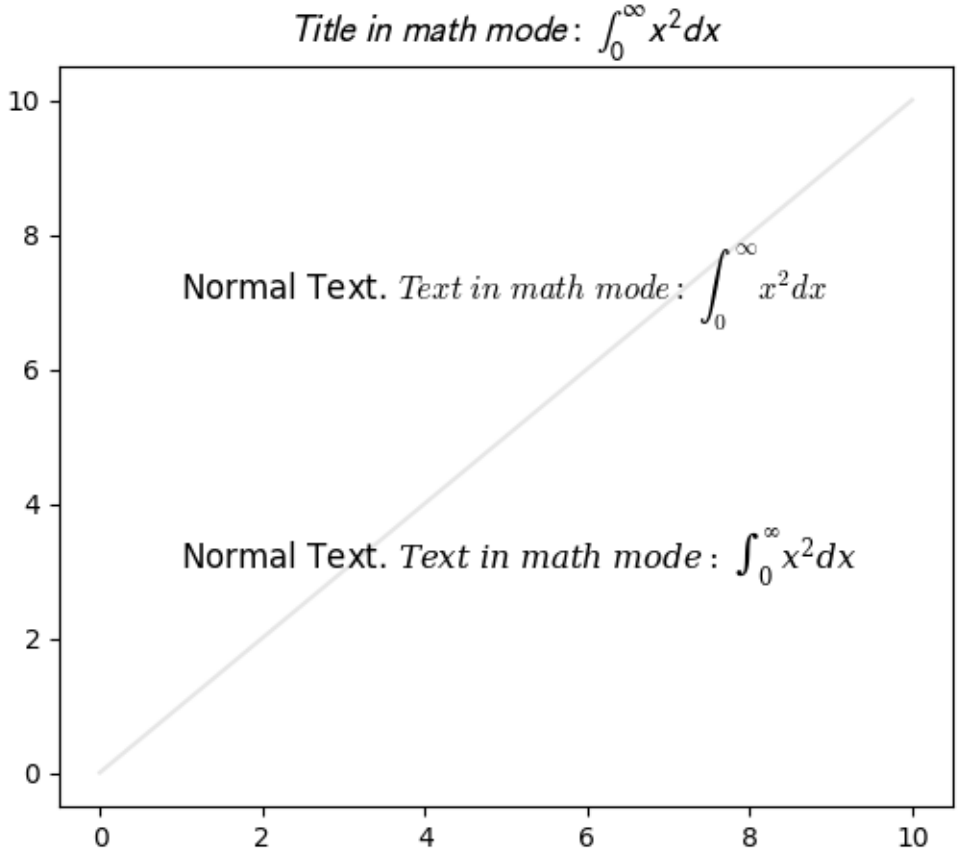
(continued from previous page)

```
        ax.annotate(demo,
                    xy=(0.04, baseline - 0.75 * line_axesfrac),
                    color=mpl_grey_rgb, fontsize=16)

    plt.show()


if '--latex' in sys.argv:
    # Run: python mathtext_examples.py --latex
    # Need amsmath and amssymb packages.
    with open("mathtext_examples.ltx", "w") as fd:
        fd.write("\\documentclass{article}\n")
        fd.write("\\usepackage{amsmath, amssymb}\n")
        fd.write("\\begin{document}\n")
        fd.write("\\begin{enumerate}\n")

        for s in mathtext_demos.values():
            s = re.sub(r"(?<!\\)\$", "$$", s)
            fd.write("\\item %s\n" % s)

        fd.write("\\end{enumerate}\n")
        fd.write("\\end{document}\n")

    subprocess.call(["pdflatex", "mathtext_examples.ltx"])
else:
    doall()
```

### Math fontfamily

A simple example showcasing the new *math_fontfamily* parameter that can be used to change the family of fonts for each individual text element in a plot.

If no parameter is set, the global value *rcParams["mathtext.fontset"]* (default: `'dejavusans'`) will be used.



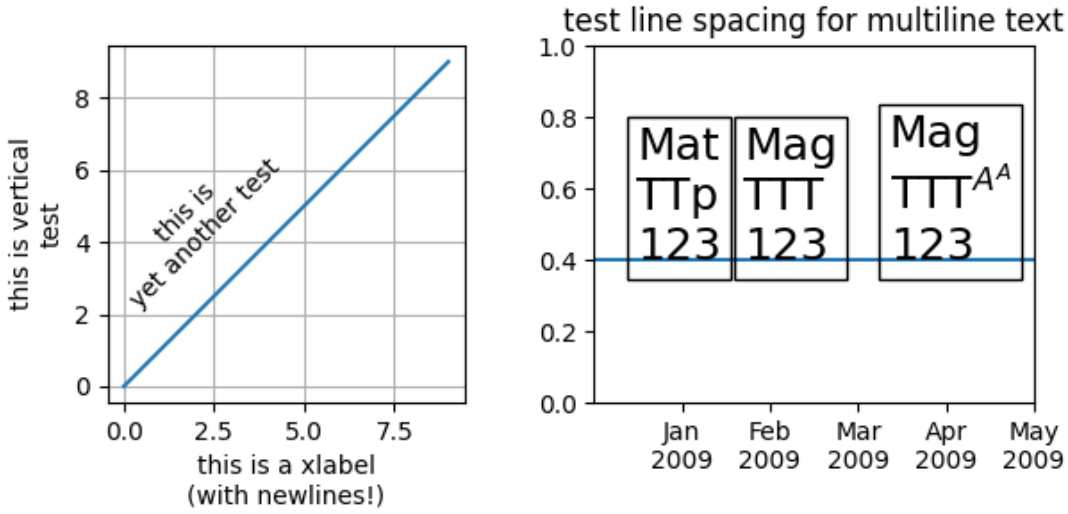

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(figsize=(6, 5))

# A simple plot for the background.
ax.plot(range(11), color="0.9")

# A text mixing normal text and math text.
msg = (r"Normal Text. $Text\ in\ math\ mode:\ "
       r"\int_{0}^{\infty } x^2 dx$")

# Set the text in the plot.
ax.text(1, 7, msg, size=12, math_fontfamily='cm')

# Set another font for the next text.
ax.text(1, 3, msg, size=12, math_fontfamily='dejavuserif')

# *math_fontfamily* can be used in most places where there is text,
# like in the title:
ax.set_title(r"$Title\ in\ math\ mode:\ \int_{0}^{\infty } x^2 dx$",
             math_fontfamily='stixsans', size=14)
```

(continues on next page)

(continued from previous page)

```
# Note that the normal text is not changed by *math_fontfamily*.
plt.show()
```

### Multiline




```
import matplotlib.pyplot as plt
import numpy as np

fig, (ax0, ax1) = plt.subplots(ncols=2, figsize=(7, 4))

ax0.set_aspect(1)
ax0.plot(np.arange(10))
ax0.set_xlabel('this is a xlabel\n(with newlines!)')
ax0.set_ylabel('this is vertical\ntest', multialignment='center')
ax0.text(2, 7, 'this is\nyet another test',
         rotation=45,
         horizontalalignment='center',
         verticalalignment='top',
         multialignment='center')

ax0.grid()


ax1.text(0.29, 0.4, "Mat\nTTp\n123", size=18,
         va="baseline", ha="right", multialignment="left",
         bbox=dict(fc="none"))
```

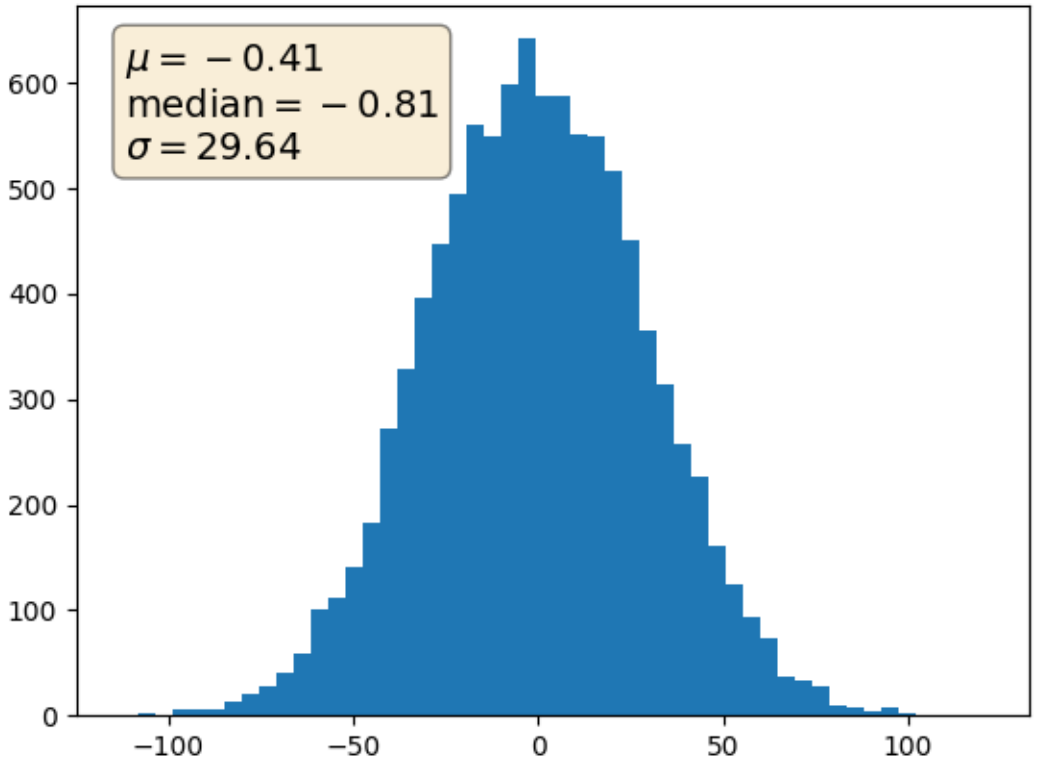
(continues on next page)

(continued from previous page)

```
ax1.text(0.34, 0.4, "Mag\nTTT\n123", size=18,
         va="baseline", ha="left", multialignment="left",
         bbox=dict(fc="none"))

ax1.text(0.95, 0.4, "Mag\nTTT$^{A^A}$\n123", size=18,
         va="baseline", ha="right", multialignment="left",
         bbox=dict(fc="none"))

ax1.set_xticks([0.2, 0.4, 0.6, 0.8, 1.],
               labels=["Jan\n2009", "Feb\n2009", "Mar\n2009", "Apr\n2009",
                       "May\n2009"])

ax1.axhline(0.4)
ax1.set_title("test line spacing for multiline text")

fig.subplots_adjust(bottom=0.25, top=0.75)
plt.show()
```

### Placing text boxes

When decorating Axes with text boxes, two useful tricks are to place the text in axes coordinates (see *Transformations Tutorial*), so the text doesn't move around with changes in x or y limits. You can also use the `bbox` property of text to surround the text with a *`Patch`* instance -- the `bbox` keyword argument takes a dictionary with keys that are Patch properties.




```
import matplotlib.pyplot as plt
import numpy as np

np.random.seed(19680801)

fig, ax = plt.subplots()
x = 30*np.random.randn(10000)
mu = x.mean()
median = np.median(x)
sigma = x.std()
textstr = '\n'.join((
    r'$\mu=%.2f$' % (mu, ),
    r'$\mathrm{median}=%.2f$' % (median, ),
    r'$\sigma=%.2f$' % (sigma, )))

ax.hist(x, 50)
# these are matplotlib.patch.Patch properties
props = dict(boxstyle='round', facecolor='wheat', alpha=0.5)

# place a text box in upper left in axes coords
ax.text(0.05, 0.95, textstr, transform=ax.transAxes, fontsize=14,
        verticalalignment='top', bbox=props)
```

(continues on next page)

(continued from previous page)

```
plt.show()
```

### Concatenating text objects with different properties

The example strings together several Text objects with different properties (e.g., color or font), positioning each one after the other. The first Text is created directly using *text*; all subsequent ones are created with *annotate*, which allows positioning the Text's lower left corner at the lower right corner (`xy=(1, 0)`) of the previous one (`xycoords=text`).

```
import matplotlib.pyplot as plt

plt.rcParams["font.size"] = 20
ax = plt.figure().add_subplot(xticks=[], yticks=[])

# The first word, created with text().
text = ax.text(.1, .5, "Matplotlib", color="red")
# Subsequent words, positioned with annotate(), relative to the preceding one.
text = ax.annotate(
    " says,", xycoords=text, xy=(1, 0), verticalalignment="bottom",
    color="gold", weight="bold")  # custom properties
```

(continues on next page)

(continued from previous page)

```
text = ax.annotate(
    " hello", xycoords=text, xy=(1, 0), verticalalignment="bottom",
    color="green", style="italic")  # custom properties
text = ax.annotate(
    " world!", xycoords=text, xy=(1, 0), verticalalignment="bottom",
    color="blue", family="serif")  # custom properties

plt.show()
```

### STIX Fonts

Demonstration of STIX Fonts used in LaTeX rendering.

①②③ ①②③ ①②③

SansΩ SansΩ SansΩ

Monospace

*CALLIGRAPHIC*

*Blackboard π*

Blackboard π

**Blackboard π**

Fraktur **Fraktur**

*Script*

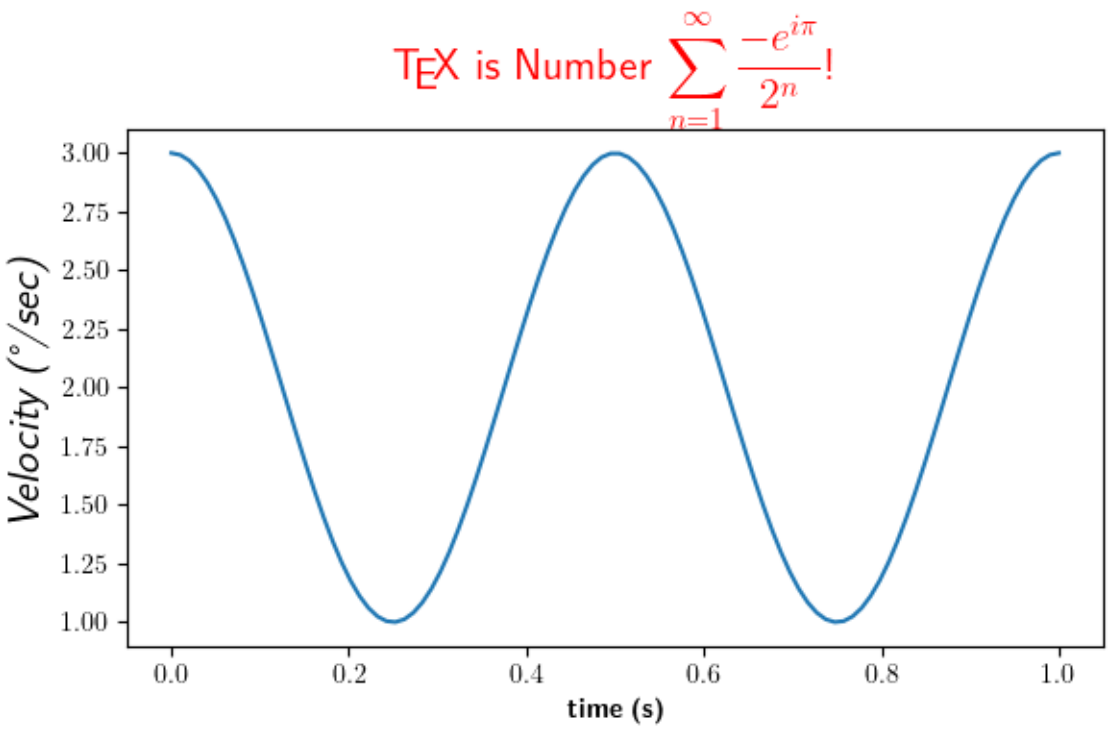
```
import matplotlib.pyplot as plt

circle123 = "\N{CIRCLED DIGIT ONE}\N{CIRCLED DIGIT TWO}\N{CIRCLED DIGIT THREE}
 ↪"

tests = [
    r'$%s\;\mathrm{%s}\;\mathbf{%s}$' % ((circle123,) * 3),
    r'$\mathsf{Sans \Omega}\;\mathrm{\mathsf{Sans \Omega}}\;'
    r'\mathbf{\mathsf{Sans \Omega}}$',
    r'$\mathtt{Monospace}$',
    r'$\mathcal{CALLIGRAPHIC}$',
    r'$\mathbb{Blackboard\;\pi}$',
    r'$\mathrm{\mathbb{Blackboard\;\pi}}$',
    r'$\mathbf{\mathbb{Blackboard\;\pi}}$',
    r'$\mathfrak{Fraktur}\;\mathbf{\mathfrak{Fraktur}}$',
    r'$\mathscr{Script}$',
]

fig = plt.figure(figsize=(8, len(tests) + 2))
for i, s in enumerate(tests[::-1]):
    fig.text(0, (i + .5) / len(tests), s, fontsize=32)

plt.show()
```

### Rendering math equations using TeX

You can use TeX to render all of your Matplotlib text by setting *rcParams["text.usetex"]* (default: `False`) to True. This requires that you have TeX and the other dependencies described in the *Text rendering with LaTeX* tutorial properly installed on your system. Matplotlib caches processed TeX expressions, so that only the first occurrence of an expression triggers a TeX compilation. Later occurrences reuse the rendered image from the cache and are thus faster.

Unicode input is supported, e.g. for the y-axis label in this example.

```
import matplotlib.pyplot as plt
import numpy as np

plt.rcParams['text.usetex'] = True


t = np.linspace(0.0, 1.0, 100)
s = np.cos(4 * np.pi * t) + 2

fig, ax = plt.subplots(figsize=(6, 4), tight_layout=True)
ax.plot(t, s)

ax.set_xlabel(r'\textbf{time (s)}')
ax.set_ylabel('\\textit{Velocity (\N{DEGREE SIGN}/sec)}', fontsize=16)
ax.set_title(r'\TeX\ is Number $\displaystyle\sum_{n=1}^\infty'
             r'\frac{-e^{i\pi}}{2^n}$!', fontsize=16, color='r')
```



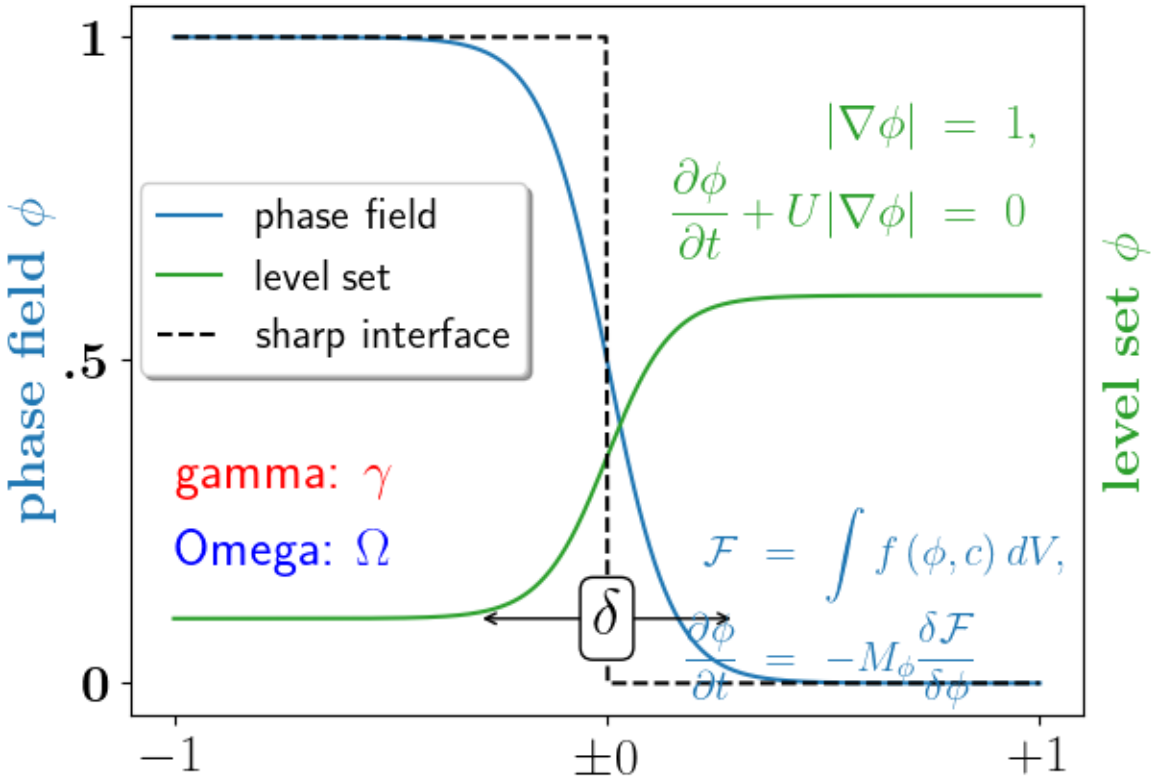

A more complex example.

```
fig, ax = plt.subplots()
# interface tracking profiles
N = 500
delta = 0.6
X = np.linspace(-1, 1, N)
ax.plot(X, (1 - np.tanh(4 * X / delta)) / 2,     # phase field tanh profiles
        X, (1.4 + np.tanh(4 * X / delta)) / 4, "C2",   # composition profile
        X, X < 0, "k--")                          # sharp interface

# legend
ax.legend(("phase field", "level set", "sharp interface"),
          shadow=True, loc=(0.01, 0.48), handlelength=1.5, fontsize=16)

# the arrow
ax.annotate("", xy=(-delta / 2., 0.1), xytext=(delta / 2., 0.1),
            arrowprops=dict(arrowstyle="<->", connectionstyle="arc3"))
ax.text(0, 0.1, r"$\delta$",
        color="black", fontsize=24,
        horizontalalignment="center", verticalalignment="center",
        bbox=dict(boxstyle="round", fc="white", ec="black", pad=0.2))

# Use tex in labels
ax.set_xticks([-1, 0, 1])
ax.set_xticklabels(["$-1$", r"$\pm 0$", "$+1$"], color="k", size=20)

# Left Y-axis labels, combine math mode and text mode
```

(continues on next page)

(continued from previous page)

```
ax.set_ylabel(r"\bf{phase field} $\phi$", color="C0", fontsize=20)
ax.set_yticks([0, 0.5, 1])
ax.set_yticklabels([r"\bf{0}", r"\bf{.5}", r"\bf{1}"], color="k", size=20)

# Right Y-axis labels
ax.text(1.02, 0.5, r"\bf{level set} $\phi$",
        color="C2", fontsize=20, rotation=90,
        horizontalalignment="left", verticalalignment="center",
        clip_on=False, transform=ax.transAxes)

# Use multiline environment inside a `text`.
# level set equations
eq1 = (r"\begin{eqnarray*}"
       r"|\nabla\phi| &=& 1,\\"
       r"\frac{\partial \phi}{\partial t} + U|\nabla \phi| &=& 0 "
       r"\end{eqnarray*}")
ax.text(1, 0.9, eq1, color="C2", fontsize=18,
        horizontalalignment="right", verticalalignment="top")

# phase field equations
eq2 = (r"\begin{eqnarray*}"
       r"\mathcal{F} &=& \int f\left( \phi, c \right) dV, \\ "
       r"\frac{ \partial \phi } { \partial t } &=& -M_{ \phi } "
       r"\frac{ \delta \mathcal{F} } { \delta \phi }"
       r"\end{eqnarray*}")
ax.text(0.18, 0.18, eq2, color="C0", fontsize=16)

ax.text(-1, .30, r"gamma: $\gamma$", color="r", fontsize=20)
ax.text(-1, .18, r"Omega: $\Omega$", color="b", fontsize=20)

plt.show()
```




### Text alignment

Texts are aligned relative to their anchor point depending on the properties `horizontalalignment` (default: `left`) and `verticalalignment` (default: `baseline`.)

The following plot uses this to align text relative to a plotted rectangle.

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()

# Build a rectangle in axes coords
left, width = .25, .5
bottom, height = .25, .5
right = left + width
top = bottom + height
p = plt.Rectangle((left, bottom), width, height, fill=False)
p.set_transform(ax.transAxes)
p.set_clip_on(False)
ax.add_patch(p)
```

(continues on next page)

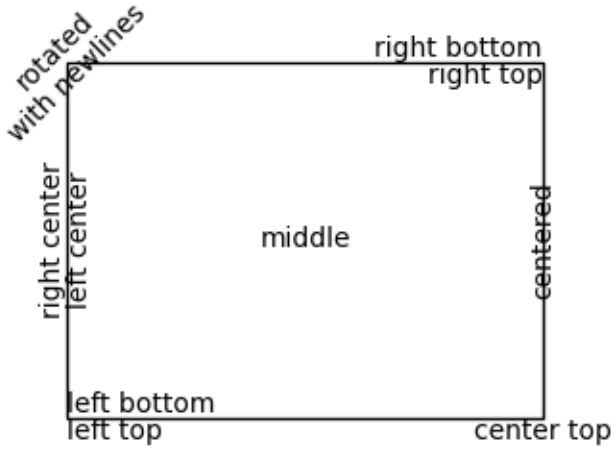
(continued from previous page)

```
ax.text(left, bottom, 'left top',
        horizontalalignment='left',
        verticalalignment='top',
        transform=ax.transAxes)

ax.text(left, bottom, 'left bottom',
        horizontalalignment='left',
        verticalalignment='bottom',
        transform=ax.transAxes)

ax.text(right, top, 'right bottom',
        horizontalalignment='right',
        verticalalignment='bottom',
        transform=ax.transAxes)

ax.text(right, top, 'right top',
        horizontalalignment='right',
        verticalalignment='top',
        transform=ax.transAxes)

ax.text(right, bottom, 'center top',
        horizontalalignment='center',
        verticalalignment='top',
        transform=ax.transAxes)

ax.text(left, 0.5 * (bottom + top), 'right center',
        horizontalalignment='right',
        verticalalignment='center',
        rotation='vertical',
        transform=ax.transAxes)

ax.text(left, 0.5 * (bottom + top), 'left center',
        horizontalalignment='left',
        verticalalignment='center',
        rotation='vertical',
        transform=ax.transAxes)

ax.text(0.5 * (left + right), 0.5 * (bottom + top), 'middle',
        horizontalalignment='center',
        verticalalignment='center',
        transform=ax.transAxes)

ax.text(right, 0.5 * (bottom + top), 'centered',
        horizontalalignment='center',
        verticalalignment='center',
        rotation='vertical',
        transform=ax.transAxes)

ax.text(left, top, 'rotated\nwith newlines',
        horizontalalignment='center',
        verticalalignment='center',
        rotation=45,
```

(continues on next page)

(continued from previous page)

```
        transform=ax.transAxes)

ax.set_axis_off()

plt.show()
```




## Text Commands

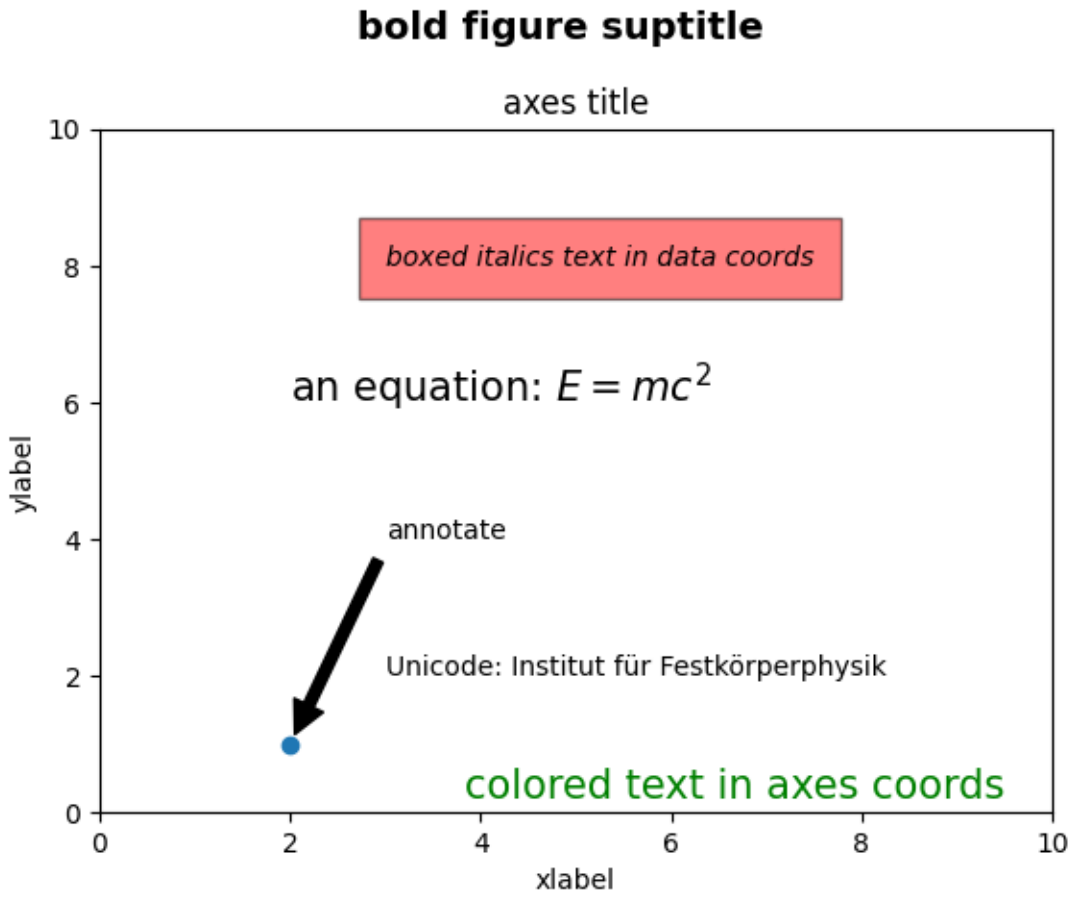
Plotting text of many different kinds.

```
import matplotlib.pyplot as plt

fig = plt.figure()
fig.suptitle('bold figure suptitle', fontsize=14, fontweight='bold')

ax = fig.add_subplot()
fig.subplots_adjust(top=0.85)
ax.set_title('axes title')

ax.set_xlabel('xlabel')
```

(continues on next page)

(continued from previous page)

```
ax.set_ylabel('ylabel')

ax.text(3, 8, 'boxed italics text in data coords', style='italic',
        bbox={'facecolor': 'red', 'alpha': 0.5, 'pad': 10})

ax.text(2, 6, r'an equation: $E=mc^2$', fontsize=15)

ax.text(3, 2, 'Unicode: Institut f\374r Festk\366rperphysik')

ax.text(0.95, 0.01, 'colored text in axes coords',
        verticalalignment='bottom', horizontalalignment='right',
        transform=ax.transAxes,
        color='green', fontsize=15)


ax.plot([2], [1], 'o')
ax.annotate('annotate', xy=(2, 1), xytext=(3, 4),
            arrowprops=dict(facecolor='black', shrink=0.05))

ax.set(xlim=(0, 10), ylim=(0, 10))

plt.show()
```



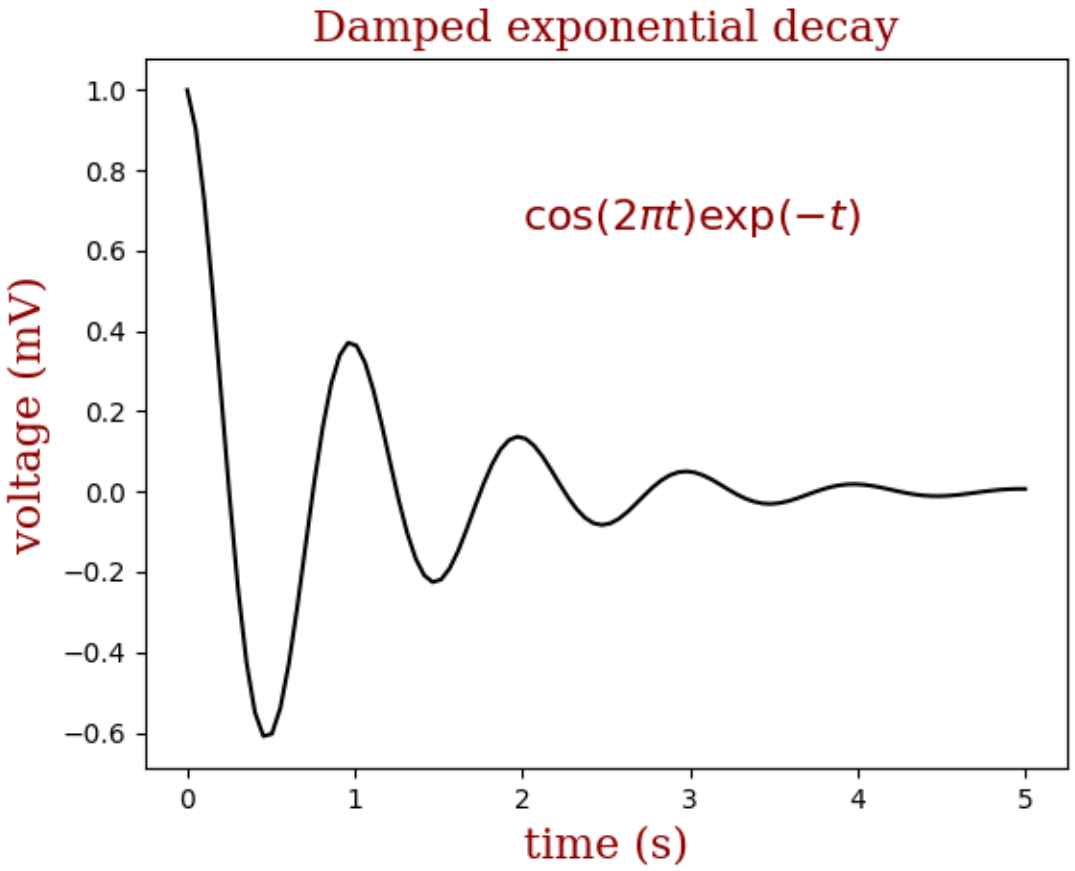
**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.figure.Figure.suptitle*
- *matplotlib.figure.Figure.add_subplot*
- *matplotlib.figure.Figure.subplots_adjust*
- *matplotlib.axes.Axes.set_title*
- *matplotlib.axes.Axes.set_xlabel*
- *matplotlib.axes.Axes.set_ylabel*
- *matplotlib.axes.Axes.text*
- *matplotlib.axes.Axes.annotate*

### Controlling style of text and labels using a dictionary

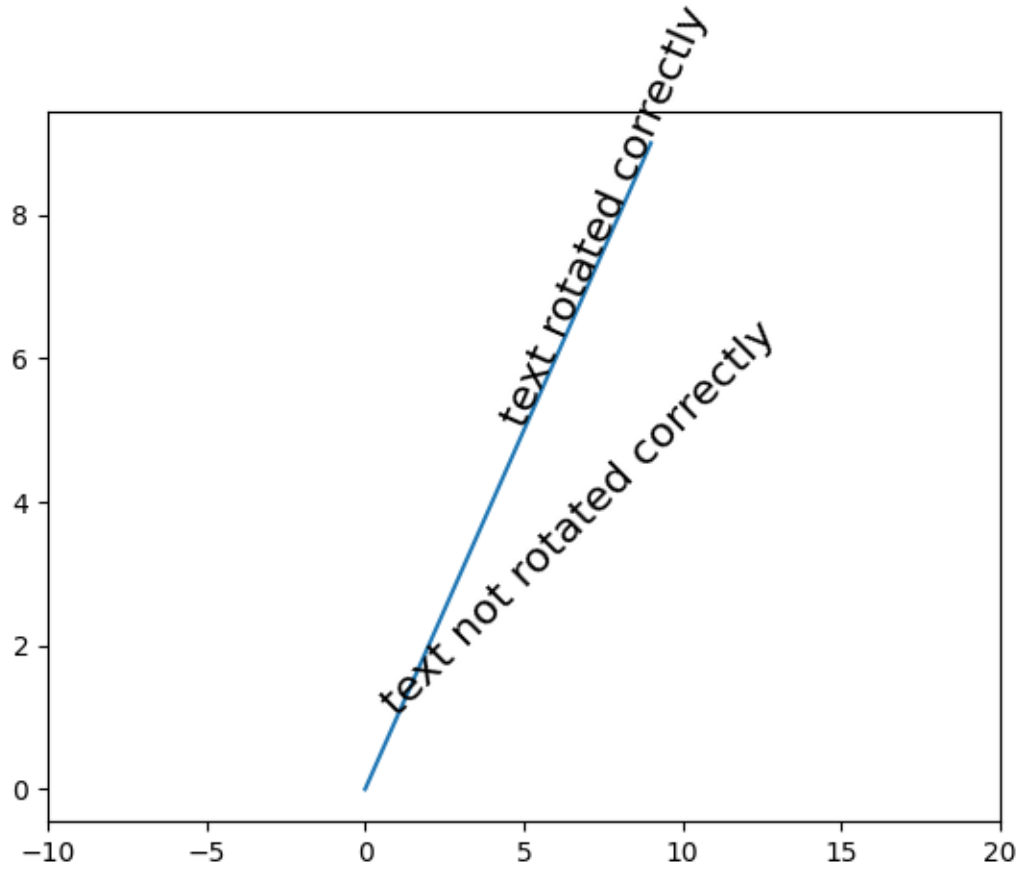
This example shows how to share parameters across many text objects and labels by creating a dictionary of options passed across several functions.




```
import matplotlib.pyplot as plt
import numpy as np

font = {'family': 'serif',
        'color':  'darkred',
        'weight': 'normal',
        'size': 16,
        }

x = np.linspace(0.0, 5.0, 100)
y = np.cos(2*np.pi*x) * np.exp(-x)

plt.plot(x, y, 'k')
plt.title('Damped exponential decay', fontdict=font)
plt.text(2, 0.65, r'$\cos(2 \pi t) \exp(-t)$', fontdict=font)
plt.xlabel('time (s)', fontdict=font)
plt.ylabel('voltage (mV)', fontdict=font)

# Tweak spacing to prevent clipping of ylabel
plt.subplots_adjust(left=0.15)
plt.show()
```

### Text Rotation Relative To Line

Text objects in matplotlib are normally rotated with respect to the screen coordinate system (i.e., 45 degrees rotation plots text along a line that is in between horizontal and vertical no matter how the axes are changed). However, at times one wants to rotate text with respect to something on the plot. In this case, the correct angle won't be the angle of that object in the plot coordinate system, but the angle that that object APPEARS in the screen coordinate system. This angle can be determined automatically by setting the parameter *transform_rotates_text*, as shown in the example below.




```
import matplotlib.pyplot as plt
import numpy as np

fig, ax = plt.subplots()

# Plot diagonal line (45 degrees)
h = ax.plot(range(0, 10), range(0, 10))

# set limits so that it no longer looks on screen to be 45 degrees
ax.set_xlim([-10, 20])

# Locations to plot text
l1 = np.array((1, 1))
```

(continues on next page)

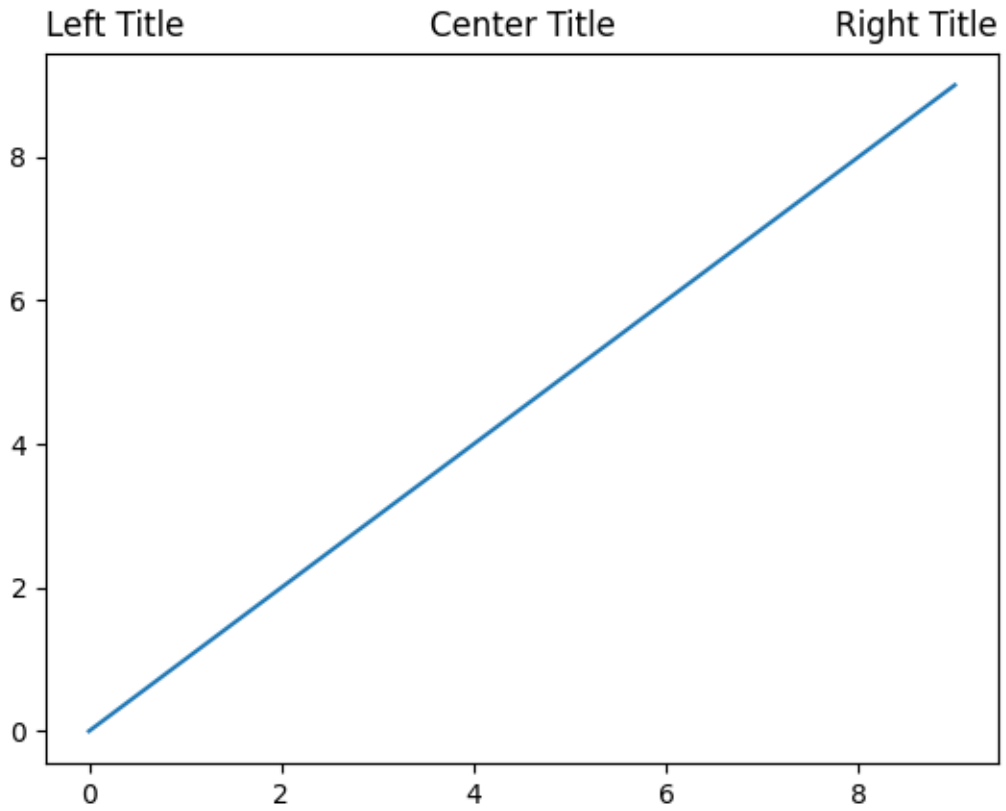
(continued from previous page)

```
l2 = np.array((5, 5))

# Rotate angle
angle = 45

# Plot text
th1 = ax.text(*l1, 'text not rotated correctly', fontsize=16,
              rotation=angle, rotation_mode='anchor')
th2 = ax.text(*l2, 'text rotated correctly', fontsize=16,
              rotation=angle, rotation_mode='anchor',
              transform_rotates_text=True)

plt.show()
```

### Title positioning

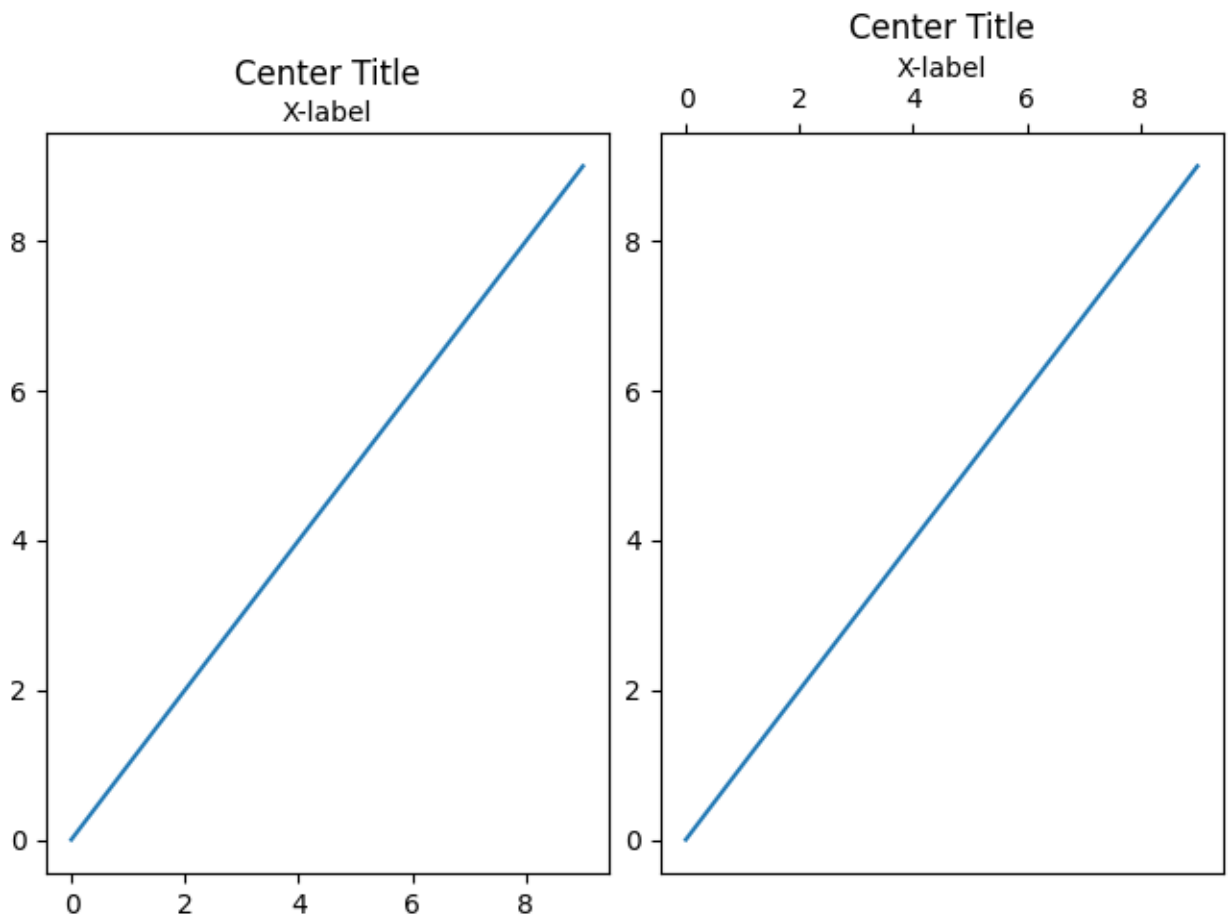
Matplotlib can display plot titles centered, flush with the left side of a set of Axes, and flush with the right side of a set of Axes.

```
import matplotlib.pyplot as plt

plt.plot(range(10))

plt.title('Center Title')
plt.title('Left Title', loc='left')
plt.title('Right Title', loc='right')

plt.show()
```



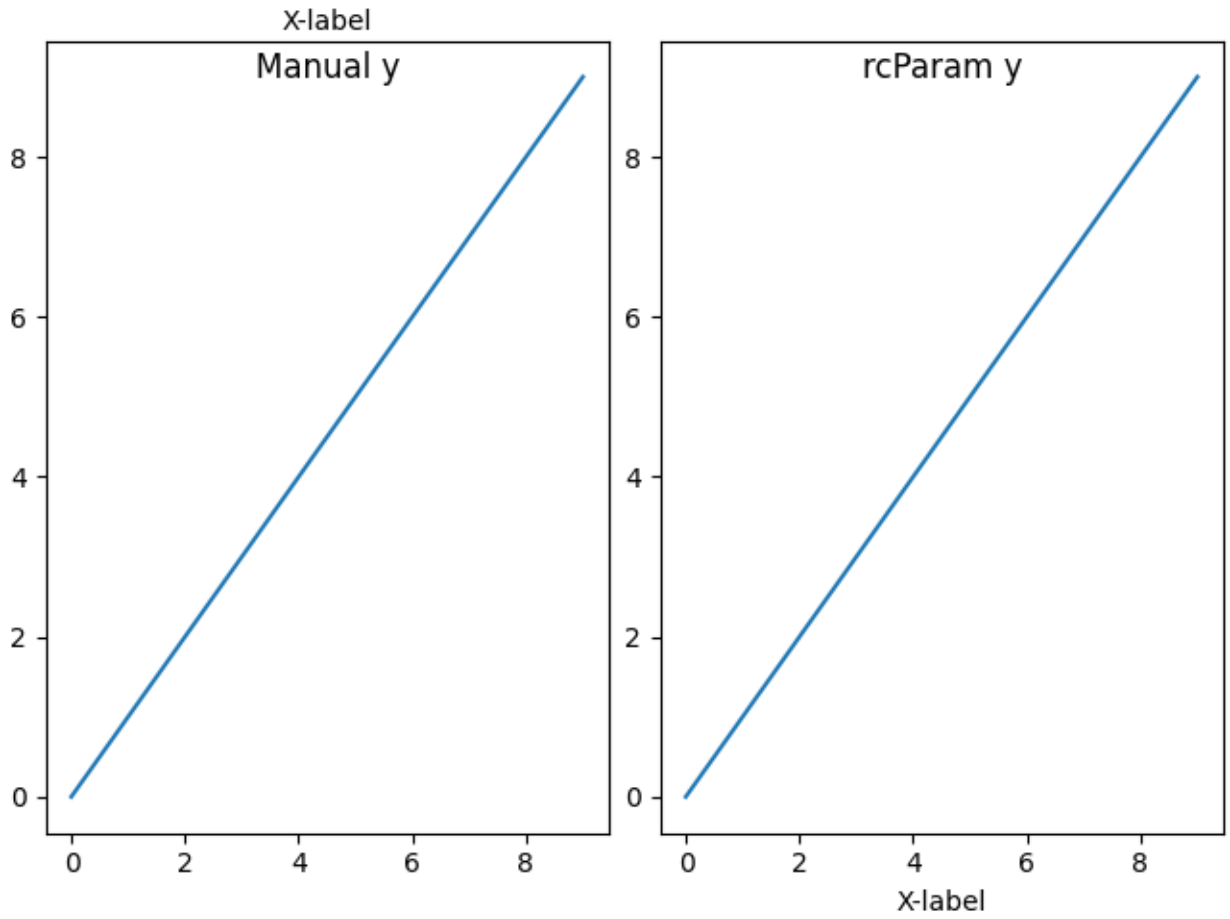

The vertical position is automatically chosen to avoid decorations (i.e. labels and ticks) on the topmost x-axis:

```
fig, axs = plt.subplots(1, 2, layout='constrained')

ax = axs[0]
ax.plot(range(10))
ax.xaxis.set_label_position('top')
ax.set_xlabel('X-label')
ax.set_title('Center Title')

ax = axs[1]
ax.plot(range(10))
ax.xaxis.set_label_position('top')
ax.xaxis.tick_top()
ax.set_xlabel('X-label')
ax.set_title('Center Title')
plt.show()
```




Automatic positioning can be turned off by manually specifying the *y* keyword argument for the title or setting *rcParams["axes.titley"]* (default: None) in the rcParams.

```
fig, axs = plt.subplots(1, 2, layout='constrained')

ax = axs[0]
ax.plot(range(10))
ax.xaxis.set_label_position('top')
ax.set_xlabel('X-label')
ax.set_title('Manual y', y=1.0, pad=-14)

plt.rcParams['axes.titley'] = 1.0    # y is in axes-relative coordinates.
plt.rcParams['axes.titlepad'] = -14  # pad is in points...
ax = axs[1]
ax.plot(range(10))
ax.set_xlabel('X-label')
ax.set_title('rcParam y')

plt.show()
```



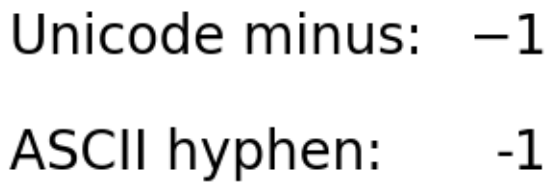

**Total running time of the script:** (0 minutes 1.198 seconds)

### Unicode minus

By default, tick labels at negative values are rendered using a Unicode minus (U+2212) rather than an ASCII hyphen (U+002D). This can be controlled by setting *rcParams["axes.unicode_minus"]* (default: `True`).

The replacement is performed at draw time of the tick labels (usually during a *pyplot.show()* or *pyplot.savefig()* call). Therefore, all tick labels of the figure follow the same setting and we cannot demonstrate both glyphs on real tick labels of the same figure simultaneously.

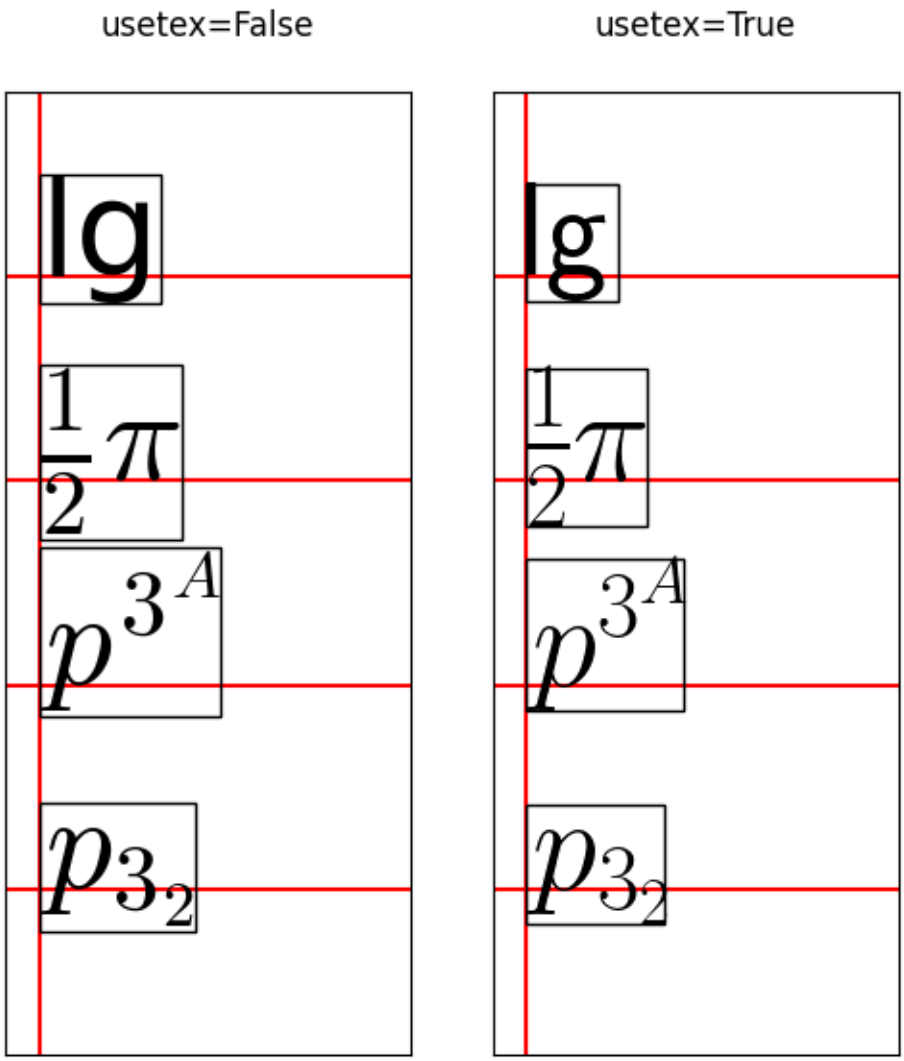
Instead, this example simply showcases the difference between the two glyphs in a magnified font.




```
import matplotlib.pyplot as plt

fig = plt.figure(figsize=(4, 2))
fig.text(.15, .6, "Unicode minus:", fontsize=20)
fig.text(.85, .6, "\N{MINUS SIGN}1", ha='right', fontsize=20)
fig.text(.15, .3, "ASCII hyphen:", fontsize=20)
fig.text(.85, .3, "-1", ha='right', fontsize=20)
plt.show()
```

### Usetex Baseline Test

Comparison of text baselines computed for mathtext and usetex.




```
import matplotlib.pyplot as plt

plt.rcParams.update({"mathtext.fontset": "cm", "mathtext.rm": "serif"})
axs = plt.figure(figsize=(2 * 3, 6.5)).subplots(1, 2)
for ax, usetex in zip(axs, [False, True]):
    ax.axvline(0, color="r")
    test_strings = ["lg", r"$\frac{1}{2}\pi$", r"$p^{3^A}$", r"$p_{3_2}$"]
    for i, s in enumerate(test_strings):
        ax.axhline(i, color="r")
        ax.text(0., 3 - i, s,
                usetex=usetex,
                verticalalignment="baseline",
                size=50,
```

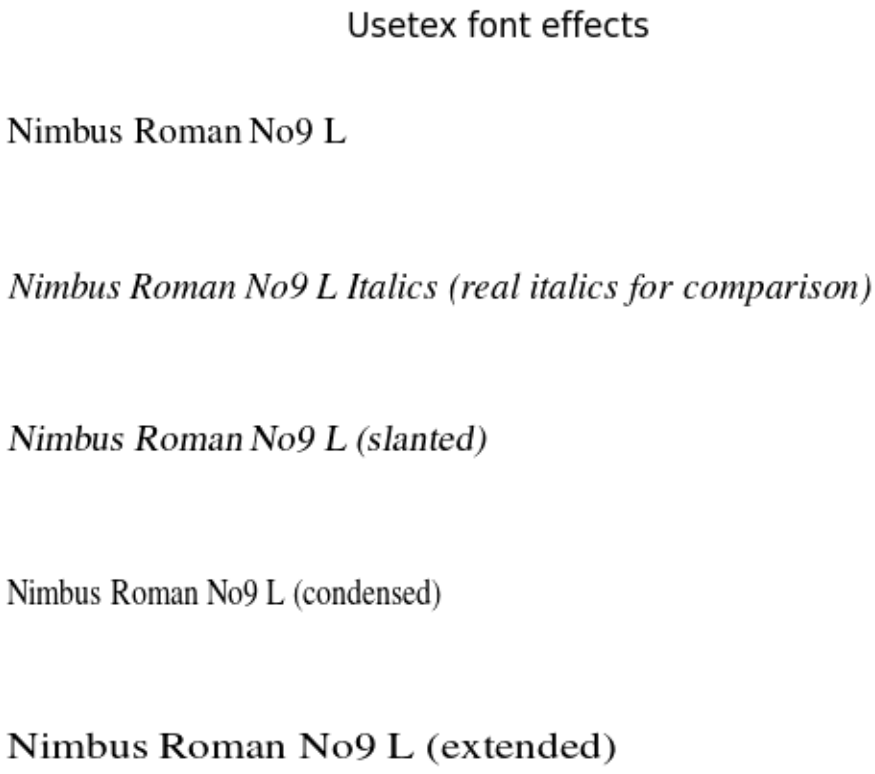
(continues on next page)

(continued from previous page)

```
                bbox=dict(pad=0, ec="k", fc="none"))
    ax.set(xlim=(-0.1, 1.1), ylim=(-.8, 3.9), xticks=[], yticks=[],
           title=f"usetex={usetex}\n")
plt.show()
```

### Usetex Fonteffects

This script demonstrates that font effects specified in your pdftex.map are now supported in usetex mode.




```
import matplotlib.pyplot as plt


def setfont(font):
    return rf'\font\a {font} at 14pt\a '


fig = plt.figure()
for y, font, text in zip(
    range(5),
    ['ptmr8r', 'ptmri8r', 'ptmro8r', 'ptmr8rn', 'ptmrr8re'],
```

(continues on next page)

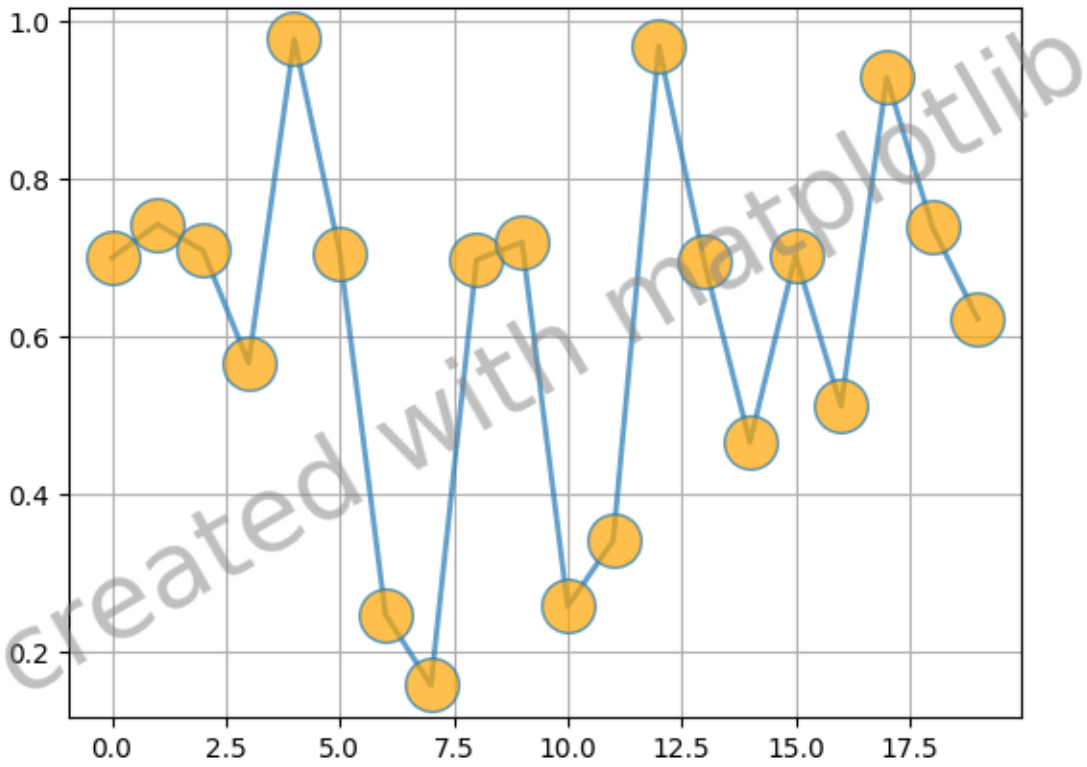
(continued from previous page)

```
    [f'Nimbus Roman No9 L {x}'
     for x in ['', 'Italics (real italics for comparison)',
               '(slanted)', '(condensed)', '(extended)']],
):
    fig.text(.1, 1 - (y + 1) / 6, setfont(font) + text, usetex=True)

fig.suptitle('Usetex font effects')
# Would also work if saving to pdf.
plt.show()
```

### Text watermark

A watermark effect can be achieved by drawing a semi-transparent text.

```
import matplotlib.pyplot as plt
import numpy as np

# Fixing random state for reproducibility
np.random.seed(19680801)


fig, ax = plt.subplots()
ax.plot(np.random.rand(20), '-o', ms=20, lw=2, alpha=0.7, mfc='orange')
ax.grid()

ax.text(0.5, 0.5, 'created with matplotlib', transform=ax.transAxes,
        fontsize=40, color='gray', alpha=0.5,
        ha='center', va='center', rotation=30)

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.figure.Figure.text*

### 6.25.7 Color

For a description of the colormaps available in Matplotlib, see the *colormaps tutorial*.

#### Color Demo

Matplotlib recognizes the following formats to specify a color:

1) an RGB or RGBA tuple of float values in `[0, 1]` (e.g. `(0.1, 0.2, 0.5)` or `(0.1, 0.2, 0.5, 0.3)`). RGBA is short for Red, Green, Blue, Alpha;
2) a hex RGB or RGBA string (e.g., `'#0F0F0F'` or `'#0F0F0F0F'`);

3) a shorthand hex RGB or RGBA string, equivalent to the hex RGB or RGBA string obtained by duplicating each character, (e.g., `'#abc'`, equivalent to `'#aabbcc'`, or `'#abcd'`, equivalent to `'#aabbccdd'`);

4) a string representation of a float value in `[0, 1]` inclusive for gray level (e.g., `'0.5'`);

5) a single letter string, i.e. one of `{'b', 'g', 'r', 'c', 'm', 'y', 'k', 'w'}`, which are short-hand notations for shades of blue, green, red, cyan, magenta, yellow, black, and white;

6) a X11/CSS4 ("html") color name, e.g. `"blue"`;

7) a name from the xkcd color survey, prefixed with `'xkcd:'` (e.g., `'xkcd:sky blue'`);

8) a "Cn" color spec, i.e. `'C'` followed by a number, which is an index into the default property cycle (*`rcParams["axes.prop_cycle"]`* (default: `cycler('color', ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728', '#9467bd', '#8c564b', '#e377c2', '#7f7f7f', '#bcbd22', '#17becf'])`)); the indexing is intended to occur at rendering time, and defaults to black if the cycle does not include color.

9) one of `{'tab:blue', 'tab:orange', 'tab:green', 'tab:red', 'tab:purple', 'tab:brown', 'tab:pink', 'tab:gray', 'tab:olive', 'tab:cyan'}` which are the Tableau Colors from the 'tab10' categorical palette (which is the default color cycle);

For more information on colors in matplotlib see

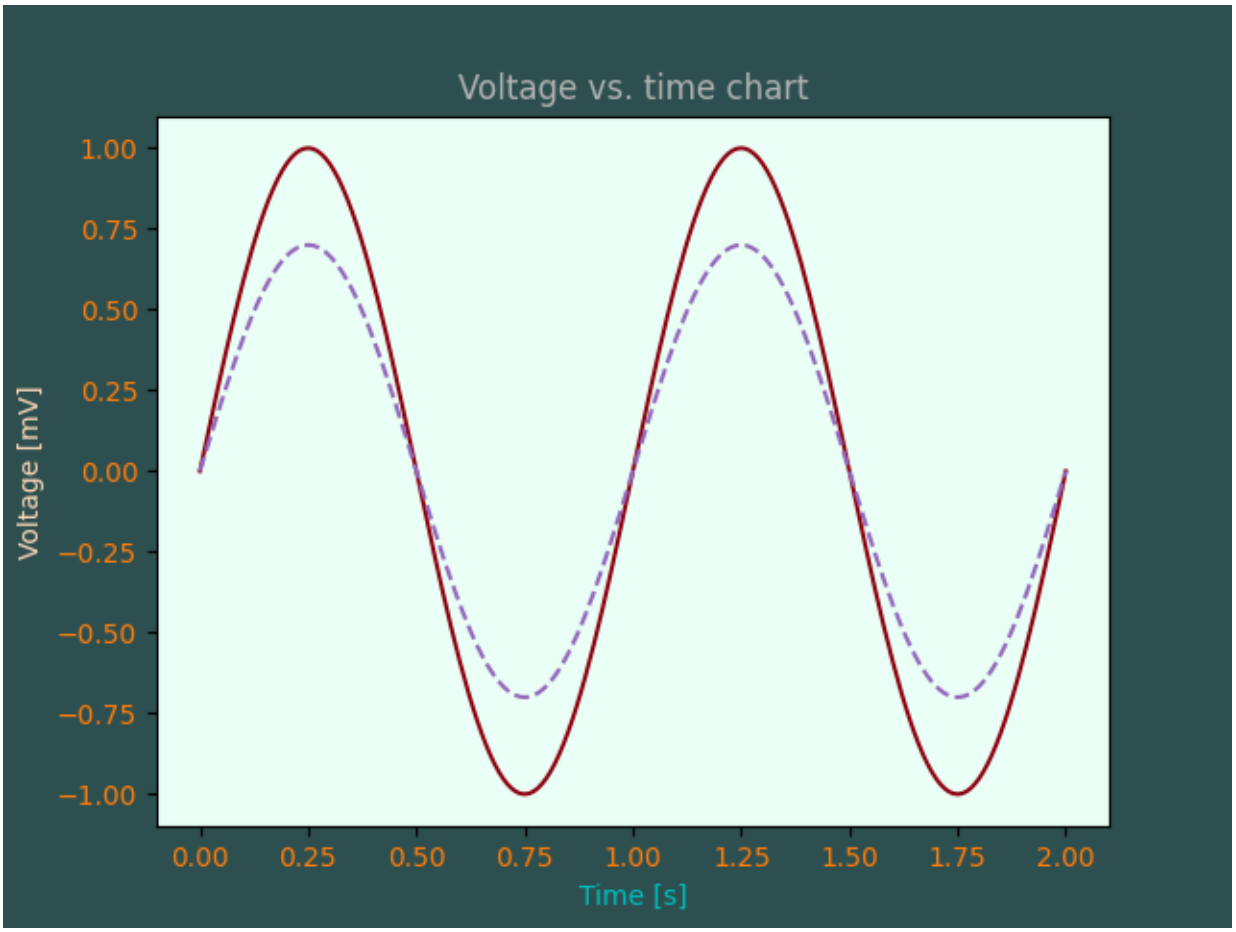
- the *Specifying colors* tutorial;
- the *`matplotlib.colors`* API;
- the *List of named colors* example.

```
import matplotlib.pyplot as plt
import numpy as np

t = np.linspace(0.0, 2.0, 201)
s = np.sin(2 * np.pi * t)

# 1) RGB tuple:
fig, ax = plt.subplots(facecolor=(.18, .31, .31))
# 2) hex string:
ax.set_facecolor('#eafff5')
# 3) gray level string:
ax.set_title('Voltage vs. time chart', color='0.7')
# 4) single letter color string
ax.set_xlabel('Time [s]', color='c')
# 5) a named color:
ax.set_ylabel('Voltage [mV]', color='peachpuff')
# 6) a named xkcd color:
ax.plot(t, s, 'xkcd:crimson')
# 7) Cn notation:
ax.plot(t, .7*s, color='C4', linestyle='--')
# 8) tab notation:
ax.tick_params(labelcolor='tab:orange')
```

(continues on next page)

(continued from previous page)

```
plt.show()
```




**References**

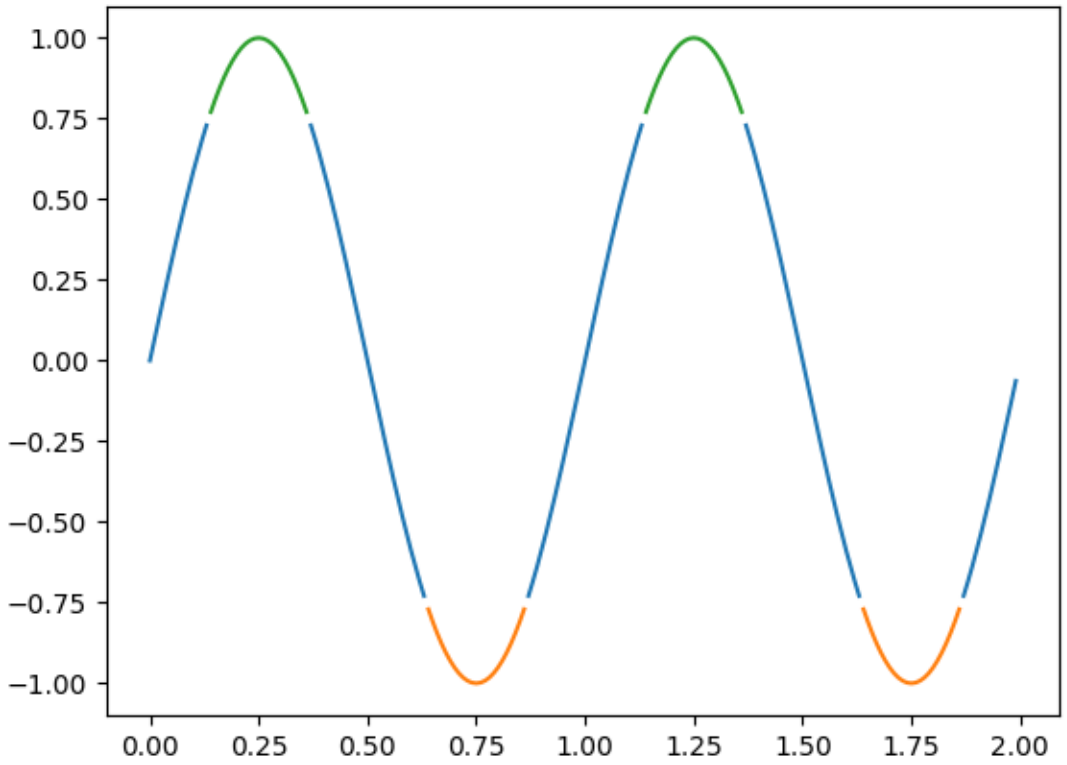
The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.colors*
- *matplotlib.axes.Axes.plot*
- *matplotlib.axes.Axes.set_facecolor*
- *matplotlib.axes.Axes.set_title*
- *matplotlib.axes.Axes.set_xlabel*
- *matplotlib.axes.Axes.set_ylabel*
- *matplotlib.axes.Axes.tick_params*

### Color by y-value

Use masked arrays to plot a line with different colors by y-value.

```
import matplotlib.pyplot as plt
import numpy as np

t = np.arange(0.0, 2.0, 0.01)
s = np.sin(2 * np.pi * t)

upper = 0.77
lower = -0.77

supper = np.ma.masked_where(s < upper, s)
slower = np.ma.masked_where(s > lower, s)
smiddle = np.ma.masked_where((s < lower) | (s > upper), s)

fig, ax = plt.subplots()
ax.plot(t, smiddle, t, slower, t, supper)
plt.show()
```




**References**

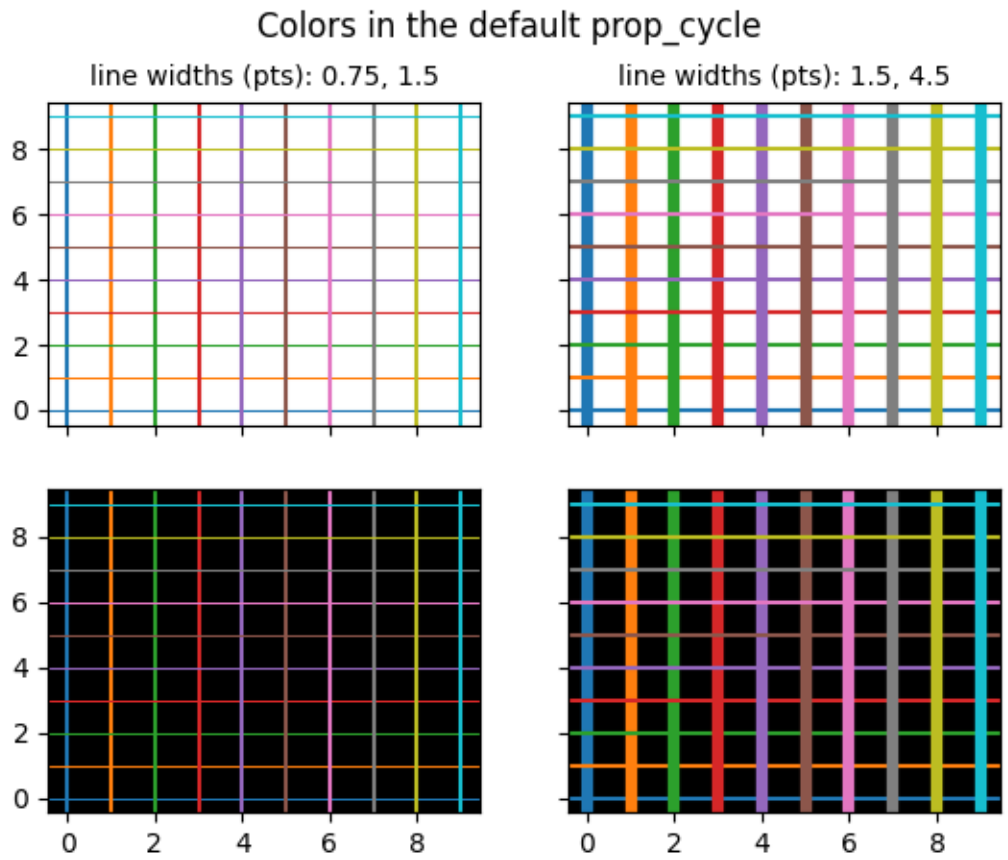
The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.plot* / *matplotlib.pyplot.plot*

### Colors in the default property cycle

Display the colors from the default prop_cycle, which is obtained from the *rc parameters*.

```
import matplotlib.pyplot as plt
import numpy as np

prop_cycle = plt.rcParams['axes.prop_cycle']
colors = prop_cycle.by_key()['color']

lwbase = plt.rcParams['lines.linewidth']
thin = lwbase / 2
thick = lwbase * 3

fig, axs = plt.subplots(nrows=2, ncols=2, sharex=True, sharey=True)
for icol in range(2):
    if icol == 0:
        lwx, lwy = thin, lwbase
    else:
        lwx, lwy = lwbase, thick
    for irow in range(2):
        for i, color in enumerate(colors):
            axs[irow, icol].axhline(i, color=color, lw=lwx)
            axs[irow, icol].axvline(i, color=color, lw=lwy)

    axs[1, icol].set_facecolor('k')
    axs[1, icol].xaxis.set_ticks(np.arange(0, 10, 2))
    axs[0, icol].set_title(f'line widths (pts): {lwx:g}, {lwy:g}',
                           fontsize='medium')

for irow in range(2):
    axs[irow, 0].yaxis.set_ticks(np.arange(0, 10, 2))

fig.suptitle('Colors in the default prop_cycle', fontsize='large')

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.axhline* / *matplotlib.pyplot.axhline*
- *matplotlib.axes.Axes.axvline* / *matplotlib.pyplot.axvline*
- *matplotlib.axes.Axes.set_facecolor*
- *matplotlib.figure.Figure.suptitle*

## Colorbar

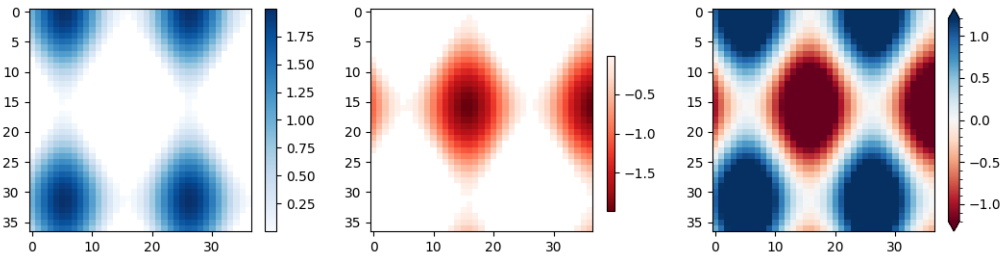
Use *colorbar* by specifying the mappable object (here the *AxesImage* returned by *imshow*) and the Axes to attach the colorbar to.

```
import matplotlib.pyplot as plt
import numpy as np

# setup some generic data
```

(continues on next page)

(continued from previous page)

```
N = 37
x, y = np.mgrid[:N, :N]
Z = (np.cos(x*0.2) + np.sin(y*0.3))

# mask out the negative and positive values, respectively
Zpos = np.ma.masked_less(Z, 0)
Zneg = np.ma.masked_greater(Z, 0)

fig, (ax1, ax2, ax3) = plt.subplots(figsize=(13, 3), ncols=3)

# plot just the positive data and save the
# color "mappable" object returned by ax1.imshow
pos = ax1.imshow(Zpos, cmap='Blues', interpolation='none')

# add the colorbar using the figure's method,
# telling which mappable we're talking about and
# which Axes object it should be near
fig.colorbar(pos, ax=ax1)

# repeat everything above for the negative data
# you can specify location, anchor and shrink the colorbar
neg = ax2.imshow(Zneg, cmap='Reds_r', interpolation='none')
fig.colorbar(neg, ax=ax2, location='right', anchor=(0, 0.3), shrink=0.7)

# Plot both positive and negative values between +/- 1.2
pos_neg_clipped = ax3.imshow(Z, cmap='RdBu', vmin=-1.2, vmax=1.2,
                             interpolation='none')
# Add minorticks on the colorbar to make it easy to read the
# values off the colorbar.
cbar = fig.colorbar(pos_neg_clipped, ax=ax3, extend='both')
cbar.minorticks_on()
plt.show()
```




## References

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.imshow* / *matplotlib.pyplot.imshow*
- *matplotlib.figure.Figure.colorbar* / *matplotlib.pyplot.colorbar*
- *matplotlib.colorbar.Colorbar.minorticks_on*

- *matplotlib.colorbar.Colorbar.minorticks_off*

### Colormap reference

Reference for colormaps included with Matplotlib.

A reversed version of each of these colormaps is available by appending `_r` to the name, as shown in *Reversed colormaps*.

See *Choosing Colormaps in Matplotlib* for an in-depth discussion about colormaps, including colorblind-friendliness, and *Creating Colormaps in Matplotlib* for a guide to creating colormaps.

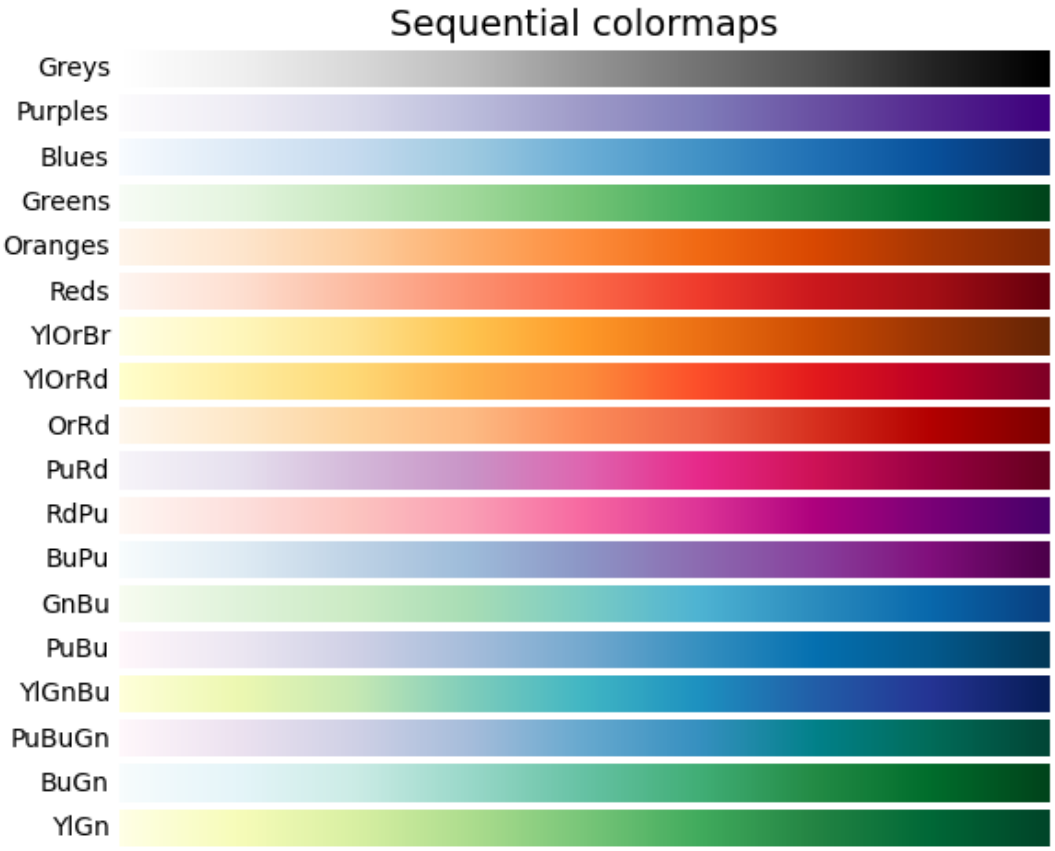
```
import matplotlib.pyplot as plt
import numpy as np

cmaps = [('Perceptually Uniform Sequential', [
            'viridis', 'plasma', 'inferno', 'magma', 'cividis']),
         ('Sequential', [
            'Greys', 'Purples', 'Blues', 'Greens', 'Oranges', 'Reds',
            'YlOrBr', 'YlOrRd', 'OrRd', 'PuRd', 'RdPu', 'BuPu',
            'GnBu', 'PuBu', 'YlGnBu', 'PuBuGn', 'BuGn', 'YlGn']),
         ('Sequential (2)', [
            'binary', 'gist_yarg', 'gist_gray', 'gray', 'bone', 'pink',
            'spring', 'summer', 'autumn', 'winter', 'cool', 'Wistia',
            'hot', 'afmhot', 'gist_heat', 'copper']),
         ('Diverging', [
            'PiYG', 'PRGn', 'BrBG', 'PuOr', 'RdGy', 'RdBu',
            'RdYlBu', 'RdYlGn', 'Spectral', 'coolwarm', 'bwr', 'seismic']),
         ('Cyclic', ['twilight', 'twilight_shifted', 'hsv']),
         ('Qualitative', [
            'Pastel1', 'Pastel2', 'Paired', 'Accent',
            'Dark2', 'Set1', 'Set2', 'Set3',
            'tab10', 'tab20', 'tab20b', 'tab20c']),
         ('Miscellaneous', [
            'flag', 'prism', 'ocean', 'gist_earth', 'terrain', 'gist_stern',
            'gnuplot', 'gnuplot2', 'CMRmap', 'cubehelix', 'brg',
            'gist_rainbow', 'rainbow', 'jet', 'turbo', 'nipy_spectral',
            'gist_ncar'])]

gradient = np.linspace(0, 1, 256)
gradient = np.vstack((gradient, gradient))


def plot_color_gradients(cmap_category, cmap_list):
    # Create figure and adjust figure height to number of colormaps
    nrows = len(cmap_list)
    figh = 0.35 + 0.15 + (nrows + (nrows-1)*0.1)*0.22
    fig, axs = plt.subplots(nrows=nrows, figsize=(6.4, figh))
    fig.subplots_adjust(top=1-.35/figh, bottom=.15/figh, left=0.2, right=0.99)

    axs[0].set_title(f"{cmap_category} colormaps", fontsize=14)
```

(continues on next page)

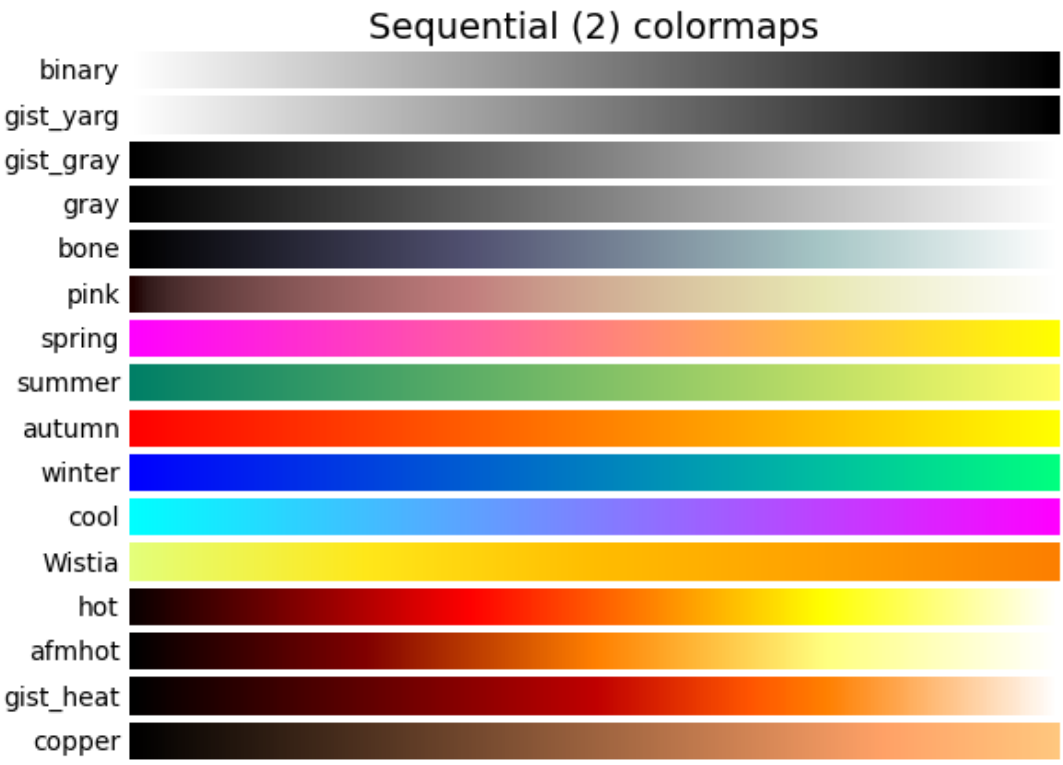
(continued from previous page)

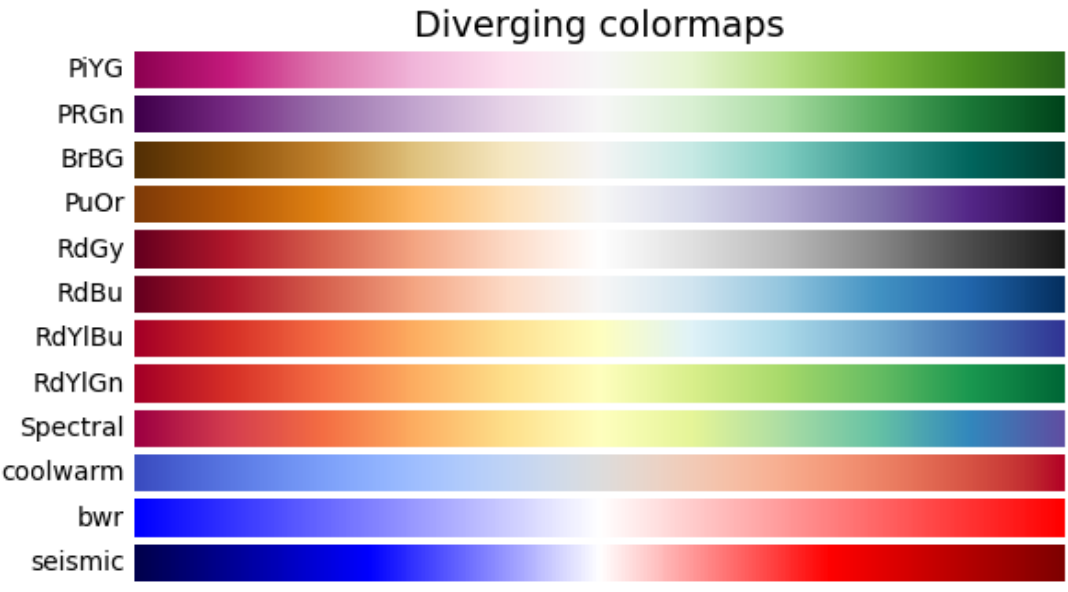
```
    for ax, cmap_name in zip(axs, cmap_list):
        ax.imshow(gradient, aspect='auto', cmap=cmap_name)
        ax.text(-.01, .5, cmap_name, va='center', ha='right', fontsize=10,
                transform=ax.transAxes)

    # Turn off *all* ticks & spines, not just the ones with colormaps.
    for ax in axs:
        ax.set_axis_off()


for cmap_category, cmap_list in cmaps:
    plot_color_gradients(cmap_category, cmap_list)
```

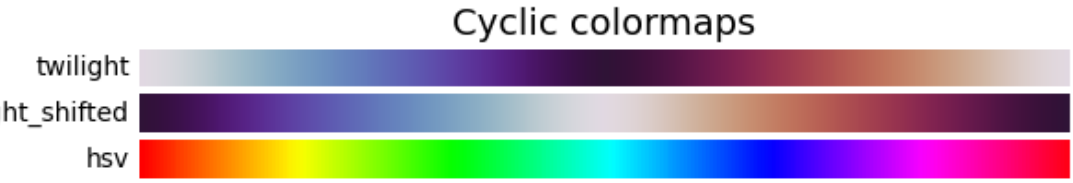
- 



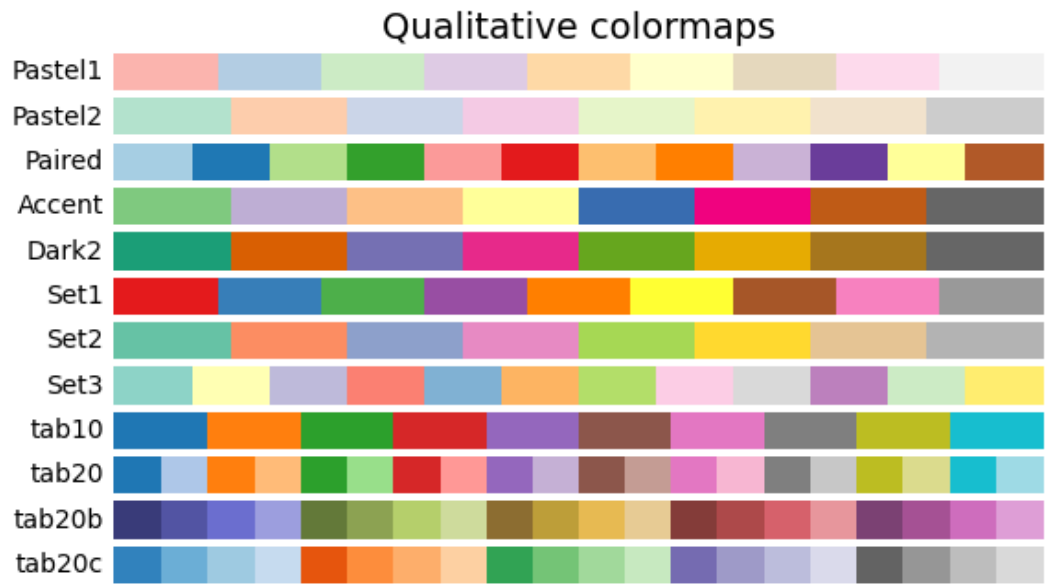

- 

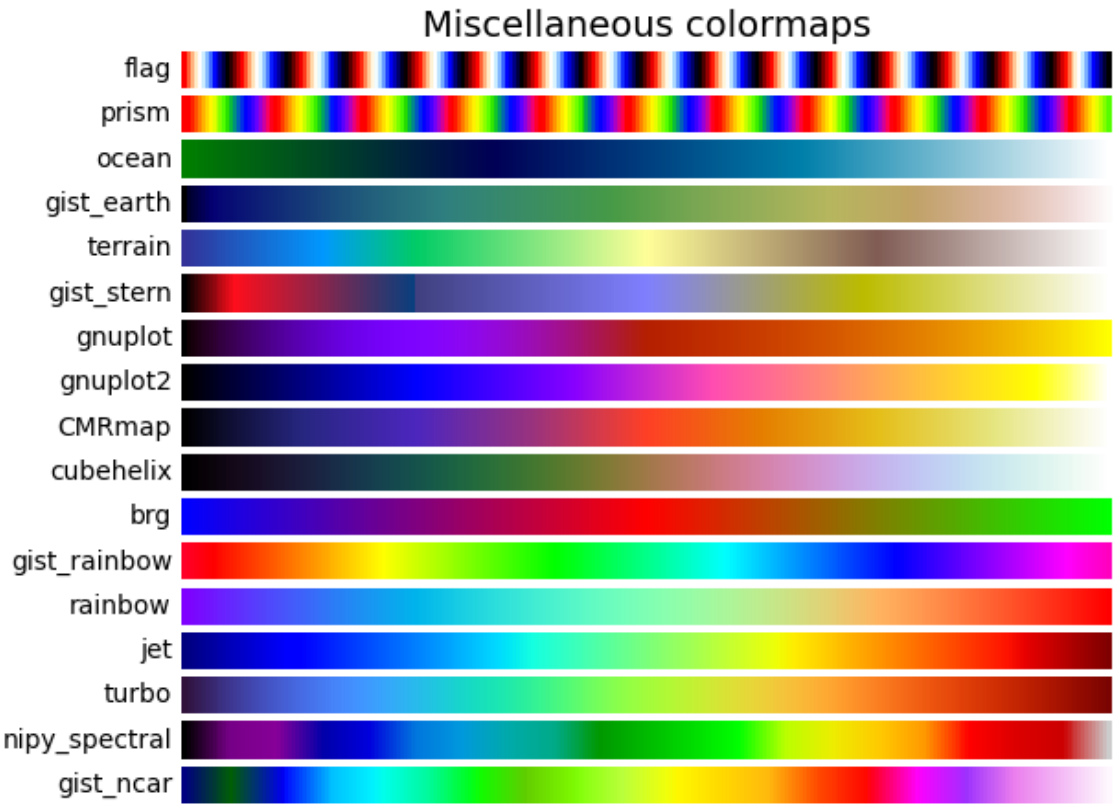

Sequential (2) colormaps
binary
gist_yarg
gist_gray
gray
bone
pink
spring
summer
autumn
winter
cool
Wistia
hot
afmhot
gist_heat
copper

- 
Diverging colormaps
PiYG
PRGn
BrBG
PuOr
RdGy
RdBu
RdYlBu
RdYlGn
Spectral
coolwarm
bwr
seismic

- 
Cyclic colormaps
twilight
twilight_shifted
hsv

-

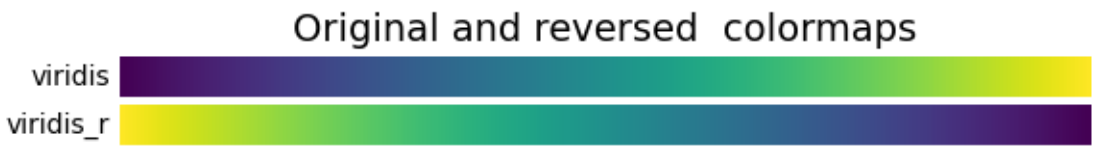
- 
Qualitative colormaps
Pastel1
Pastel2
Paired
Accent
Dark2
Set1
Set2
Set3
tab10
tab20
tab20b
tab20c

- 
Miscellaneous colormaps
flag
prism
ocean
gist_earth
terrain
gist_stern
gnuplot
gnuplot2
CMRmap
cubehelix
brg
gist_rainbow
rainbow
jet
turbo
nipy_spectral
gist_ncar

### Reversed colormaps

Append `_r` to the name of any built-in colormap to get the reversed version:

```
plot_color_gradients("Original and reversed ", ['viridis', 'viridis_r'])
```




The built-in reversed colormaps are generated using *`Colormap.reversed`*. For an example, see *Reversing a colormap*

---

**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *`matplotlib.colors`*
- *`matplotlib.axes.Axes.imshow`*
- *`matplotlib.figure.Figure.text`*
- *`matplotlib.axes.Axes.set_axis_off`*

---

**Total running time of the script:** (0 minutes 2.314 seconds)

### Creating a colormap from a list of colors

For more detail on creating and manipulating colormaps see *Creating Colormaps in Matplotlib*.

Creating a *colormap* from a list of colors can be done with the *`LinearSegmentedColormap.from_list`* method. You must pass a list of RGB tuples that define the mixture of colors from 0 to 1.

### Creating custom colormaps

It is also possible to create a custom mapping for a colormap. This is accomplished by creating dictionary that specifies how the RGB channels change from one end of the cmap to the other.

Example: suppose you want red to increase from 0 to 1 over the bottom half, green to do the same over the middle half, and blue over the top half. Then you would use:

```
cdict = {
    'red': (
        (0.0,  0.0, 0.0),
        (0.5,  1.0, 1.0),
        (1.0,  1.0, 1.0),
```

(continues on next page)

(continued from previous page)

```
    ),
    'green': (
        (0.0,  0.0, 0.0),
        (0.25, 0.0, 0.0),
        (0.75, 1.0, 1.0),
        (1.0,  1.0, 1.0),
    ),
    'blue': (
        (0.0,  0.0, 0.0),
        (0.5,  0.0, 0.0),
        (1.0,  1.0, 1.0),
    )
}
```

If, as in this example, there are no discontinuities in the r, g, and b components, then it is quite simple: the second and third element of each tuple, above, is the same -- call it "`y`". The first element ("`x`") defines interpolation intervals over the full range of 0 to 1, and it must span that whole range. In other words, the values of `x` divide the 0-to-1 range into a set of segments, and `y` gives the end-point color values for each segment.

Now consider the green, `cdict['green']` is saying that for:

- 0 <= `x` <= 0.25, `y` is zero; no green.
- 0.25 < `x` <= 0.75, `y` varies linearly from 0 to 1.
- 0.75 < `x` <= 1, `y` remains at 1, full green.

If there are discontinuities, then it is a little more complicated. Label the 3 elements in each row in the `cdict` entry for a given color as `(x, y0, y1)`. Then for values of `x` between `x[i]` and `x[i+1]` the color value is interpolated between `y1[i]` and `y0[i+1]`.

Going back to a cookbook example:

```
cdict = {
    'red': (
        (0.0,  0.0, 0.0),
        (0.5,  1.0, 0.7),
        (1.0,  1.0, 1.0),
    ),
    'green': (
        (0.0,  0.0, 0.0),
        (0.5,  1.0, 0.0),
        (1.0,  1.0, 1.0),
    ),
    'blue': (
        (0.0,  0.0, 0.0),
        (0.5,  0.0, 0.0),
        (1.0,  1.0, 1.0),
    )
}
```

and look at `cdict['red'][1]`; because `y0 != y1`, it is saying that for `x` from 0 to 0.5, red increases

from 0 to 1, but then it jumps down, so that for `x` from 0.5 to 1, red increases from 0.7 to 1. Green ramps from 0 to 1 as `x` goes from 0 to 0.5, then jumps back to 0, and ramps back to 1 as `x` goes from 0.5 to 1.

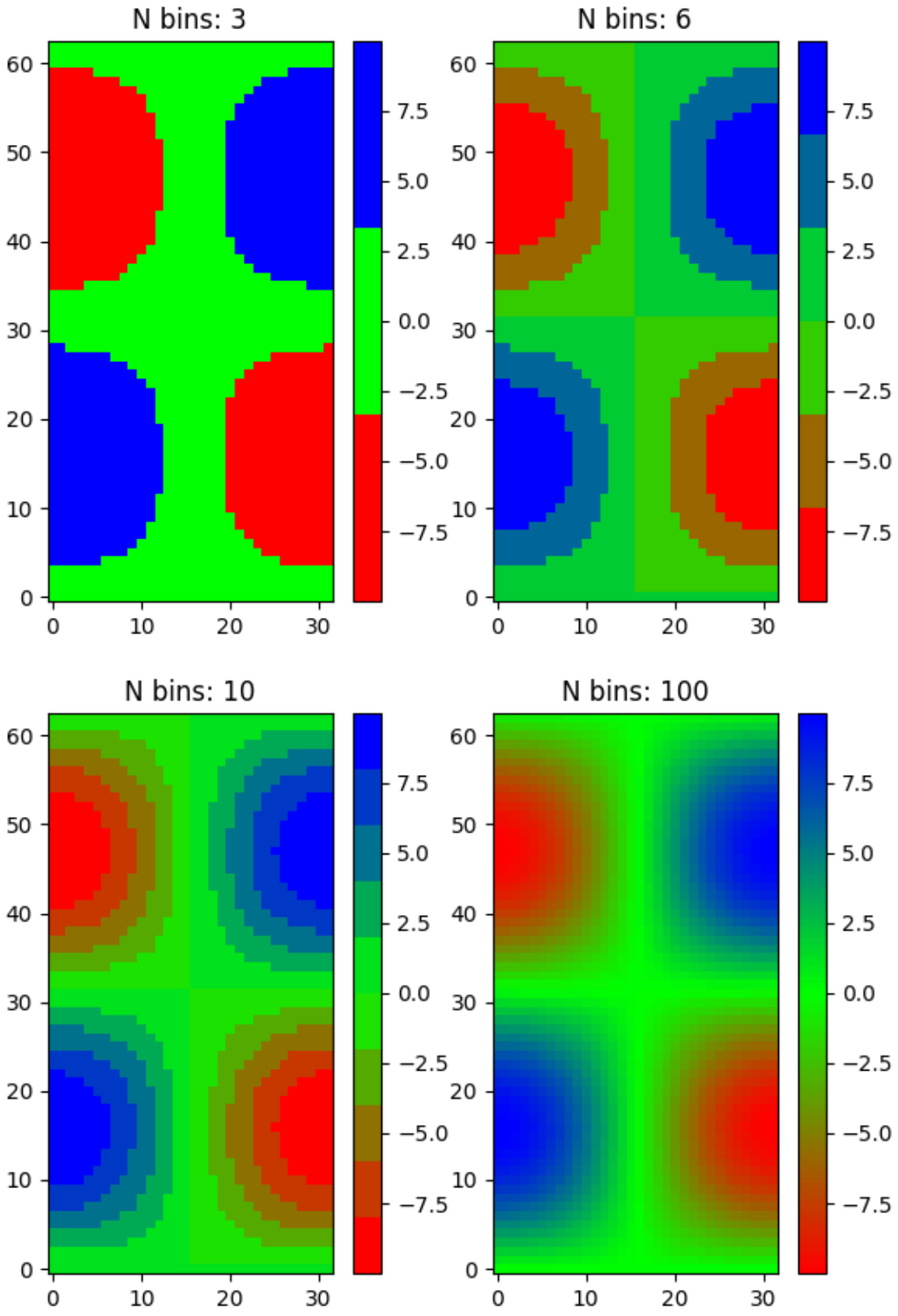
```
row i:   x  y0  y1
               /
              /
row i+1: x  y0  y1
```

Above is an attempt to show that for `x` in the range `x[i]` to `x[i+1]`, the interpolation is between `y1[i]` and `y0[i+1]`. So, `y0[0]` and `y1[-1]` are never used.

```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib as mpl
from matplotlib.colors import LinearSegmentedColormap

# Make some illustrative fake data:

x = np.arange(0, np.pi, 0.1)
y = np.arange(0, 2 * np.pi, 0.1)
X, Y = np.meshgrid(x, y)
Z = np.cos(X) * np.sin(Y) * 10
```

### Colormaps from a list

```
colors = [(1, 0, 0), (0, 1, 0), (0, 0, 1)]  # R -> G -> B
n_bins = [3, 6, 10, 100]  # Discretizes the interpolation into bins
cmap_name = 'my_list'
fig, axs = plt.subplots(2, 2, figsize=(6, 9))
fig.subplots_adjust(left=0.02, bottom=0.06, right=0.95, top=0.94, wspace=0.05)
for n_bin, ax in zip(n_bins, axs.flat):
    # Create the colormap
    cmap = LinearSegmentedColormap.from_list(cmap_name, colors, N=n_bin)
    # Fewer bins will result in "coarser" colomap interpolation
    im = ax.imshow(Z, origin='lower', cmap=cmap)
    ax.set_title("N bins: %s" % n_bin)
    fig.colorbar(im, ax=ax)
```


N bins: 3
N bins: 6
N bins: 10
N bins: 100
0
10
20
30
40
50
60
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5

### Custom colormaps

```
cdict1 = {
    'red': (
        (0.0, 0.0, 0.0),
        (0.5, 0.0, 0.1),
        (1.0, 1.0, 1.0),
    ),
    'green': (
        (0.0, 0.0, 0.0),
        (1.0, 0.0, 0.0),
    ),
    'blue': (
        (0.0, 0.0, 1.0),
        (0.5, 0.1, 0.0),
        (1.0, 0.0, 0.0),
    )
}

cdict2 = {
    'red': (
        (0.0, 0.0, 0.0),
        (0.5, 0.0, 1.0),
        (1.0, 0.1, 1.0),
    ),
    'green': (
        (0.0, 0.0, 0.0),
        (1.0, 0.0, 0.0),
    ),
    'blue': (
        (0.0, 0.0, 0.1),
        (0.5, 1.0, 0.0),
        (1.0, 0.0, 0.0),
    )
}

cdict3 = {
    'red': (
        (0.0, 0.0, 0.0),
        (0.25, 0.0, 0.0),
        (0.5, 0.8, 1.0),
        (0.75, 1.0, 1.0),
        (1.0, 0.4, 1.0),
    ),
    'green': (
        (0.0, 0.0, 0.0),
        (0.25, 0.0, 0.0),
        (0.5, 0.9, 0.9),
        (0.75, 0.0, 0.0),
        (1.0, 0.0, 0.0),
    ),
    'blue': (
```

(continues on next page)

(continued from previous page)

```
        (0.0, 0.0, 0.4),
        (0.25, 1.0, 1.0),
        (0.5, 1.0, 0.8),
        (0.75, 0.0, 0.0),
        (1.0, 0.0, 0.0),
    )
}

# Make a modified version of cdict3 with some transparency
# in the middle of the range.
cdict4 = {
    **cdict3,
    'alpha': (
        (0.0, 1.0, 1.0),
        # (0.25, 1.0, 1.0),
        (0.5, 0.3, 0.3),
        # (0.75, 1.0, 1.0),
        (1.0, 1.0, 1.0),
    ),
}
```

Now we will use this example to illustrate 2 ways of handling custom colormaps. First, the most direct and explicit:

```
blue_red1 = LinearSegmentedColormap('BlueRed1', cdict1)
```

Second, create the map explicitly and register it. Like the first method, this method works with any kind of Colormap, not just a LinearSegmentedColormap:

```
mpl.colormaps.register(LinearSegmentedColormap('BlueRed2', cdict2))
mpl.colormaps.register(LinearSegmentedColormap('BlueRed3', cdict3))
mpl.colormaps.register(LinearSegmentedColormap('BlueRedAlpha', cdict4))
```

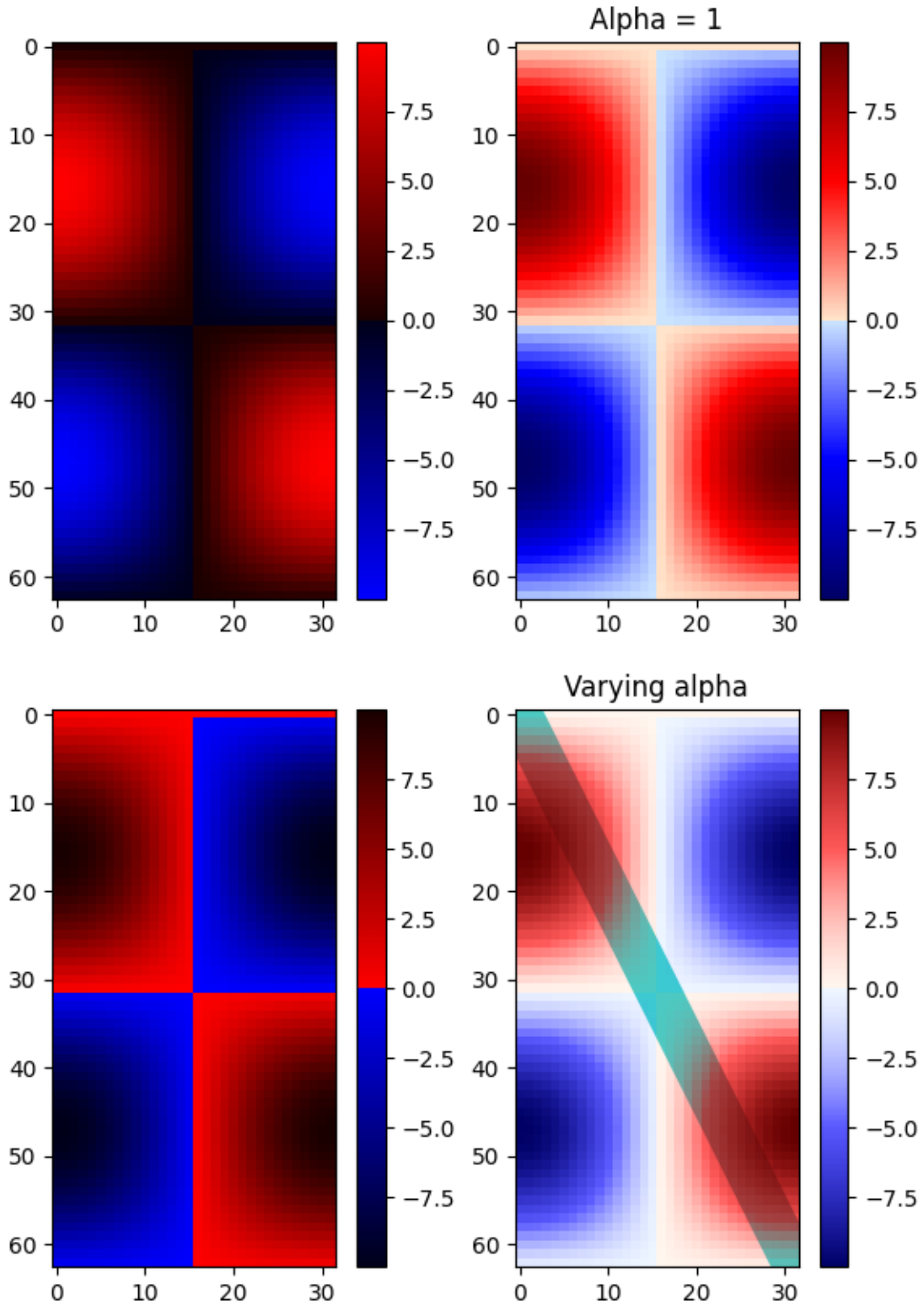
Make the figure, with 4 subplots:

```
fig, axs = plt.subplots(2, 2, figsize=(6, 9))
fig.subplots_adjust(left=0.02, bottom=0.06, right=0.95, top=0.94, wspace=0.05)

im1 = axs[0, 0].imshow(Z, cmap=blue_red1)
fig.colorbar(im1, ax=axs[0, 0])

im2 = axs[1, 0].imshow(Z, cmap='BlueRed2')
fig.colorbar(im2, ax=axs[1, 0])

# Now we will set the third cmap as the default.  One would
# not normally do this in the middle of a script like this;
# it is done here just to illustrate the method.

plt.rcParams['image.cmap'] = 'BlueRed3'

im3 = axs[0, 1].imshow(Z)
```

(continues on next page)

(continued from previous page)

```
fig.colorbar(im3, ax=axs[0, 1])
axs[0, 1].set_title("Alpha = 1")

# Or as yet another variation, we can replace the rcParams
# specification *before* the imshow with the following *after*
# imshow.
# This sets the new default *and* sets the colormap of the last
# image-like item plotted via pyplot, if any.
#

# Draw a line with low zorder so it will be behind the image.
axs[1, 1].plot([0, 10 * np.pi], [0, 20 * np.pi], color='c', lw=20, zorder=-1)

im4 = axs[1, 1].imshow(Z)
fig.colorbar(im4, ax=axs[1, 1])

# Here it is: changing the colormap for the current image and its
# colorbar after they have been plotted.
im4.set_cmap('BlueRedAlpha')
axs[1, 1].set_title("Varying alpha")

fig.suptitle('Custom Blue-Red colormaps', fontsize=16)
fig.subplots_adjust(top=0.9)

plt.show()
```


Custom Blue-Red colormaps
Alpha = 1
Varying alpha
0
10
20
30
40
50
60
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5

**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.imshow* / *matplotlib.pyplot.imshow*
- *matplotlib.figure.Figure.colorbar* / *matplotlib.pyplot.colorbar*
- *matplotlib.colors*
- *matplotlib.colors.LinearSegmentedColormap*
- *matplotlib.colors.LinearSegmentedColormap.from_list*
- *matplotlib.cm*
- *matplotlib.cm.ScalarMappable.set_cmap*
- *matplotlib.cm.ColormapRegistry.register*

**Total running time of the script:** (0 minutes 1.464 seconds)

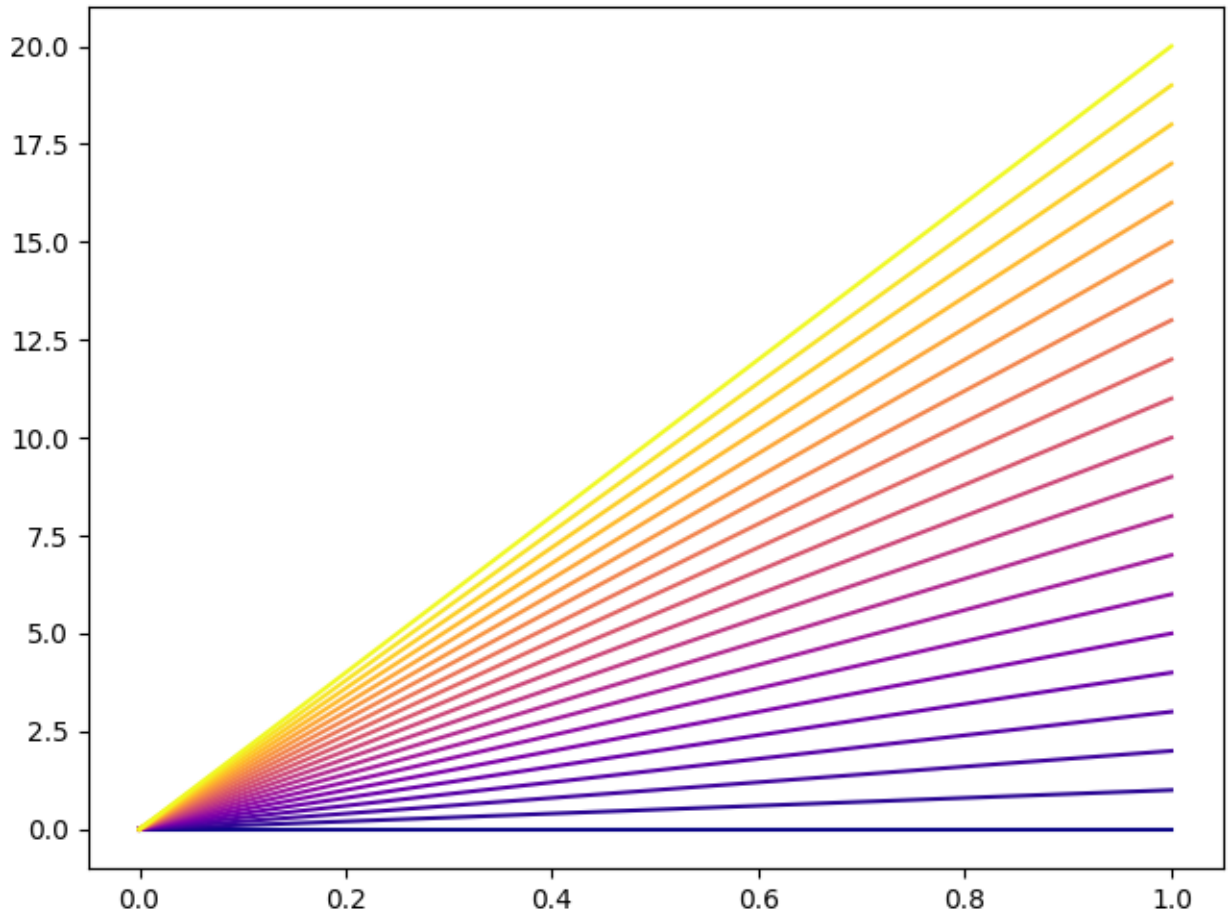
### Selecting individual colors from a colormap

Sometimes we want to use more colors or a different set of colors than the default color cycle provides. Selecting individual colors from one of the provided colormaps can be a convenient way to do this.

We can retrieve colors from any *Colormap* by calling it with a float or a list of floats in the range [0, 1]; e.g. `cmap(0.5)` will give the middle color. See also *Colormap.__call__*.

### Extracting colors from a continuous colormap

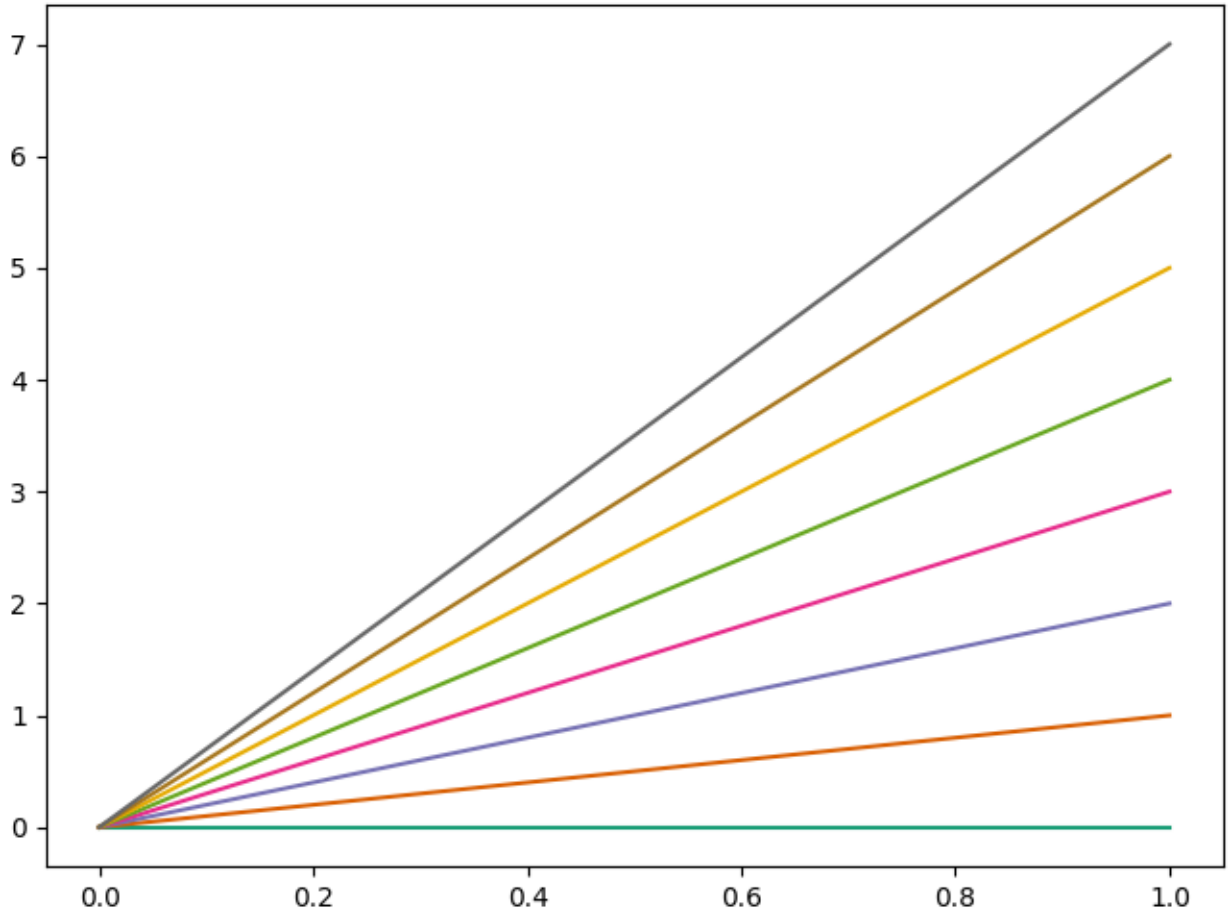
```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib as mpl

n_lines = 21
cmap = mpl.colormaps['plasma']

# Take colors at regular intervals spanning the colormap.
colors = cmap(np.linspace(0, 1, n_lines))

fig, ax = plt.subplots(layout='constrained')

for i, color in enumerate(colors):
    ax.plot([0, i], color=color)

plt.show()
```




### Extracting colors from a discrete colormap

The list of all colors in a *ListedColormap* is available as the `colors` attribute.

```
colors = mpl.colormaps['Dark2'].colors

fig, ax = plt.subplots(layout='constrained')

for i, color in enumerate(colors):
    ax.plot([0, i], color=color)

plt.show()
```




## See Also

For more details about manipulating colormaps, see *Creating Colormaps in Matplotlib*. To change the default color cycle, see *Styling with cycler*.

**References**

The use of the following functions, methods, classes and modules is shown in this example:

- `matplotlib.colors.Colormap`
- `matplotlib.colors.Colormap.resampled`

### List of named colors

This plots a list of the named colors supported by Matplotlib. For more information on colors in matplotlib see

- the *Specifying colors* tutorial;
- the `matplotlib.colors` API;
- the *Color Demo*.

### Helper Function for Plotting

First we define a helper function for making a table of colors, then we use it on some common color categories.

```
import math

import matplotlib.pyplot as plt

import matplotlib.colors as mcolors
from matplotlib.patches import Rectangle


def plot_colortable(colors, *, ncols=4, sort_colors=True):

    cell_width = 212
    cell_height = 22
    swatch_width = 48
    margin = 12

    # Sort colors by hue, saturation, value and name.
    if sort_colors is True:
        names = sorted(
            colors, key=lambda c: tuple(mcolors.rgb_to_hsv(mcolors.to_
 ↪rgb(c))))
    else:
        names = list(colors)

    n = len(names)
    nrows = math.ceil(n / ncols)

    width = cell_width * ncols + 2 * margin
    height = cell_height * nrows + 2 * margin
    dpi = 72

    fig, ax = plt.subplots(figsize=(width / dpi, height / dpi), dpi=dpi)
    fig.subplots_adjust(margin/width, margin/height,
                        (width-margin)/width, (height-margin)/height)
    ax.set_xlim(0, cell_width * ncols)
    ax.set_ylim(cell_height * (nrows-0.5), -cell_height/2.)
    ax.yaxis.set_visible(False)
    ax.xaxis.set_visible(False)
```

(continues on next page)

(continued from previous page)

```
    ax.set_axis_off()

    for i, name in enumerate(names):
        row = i % nrows
        col = i // nrows
        y = row * cell_height

        swatch_start_x = cell_width * col
        text_pos_x = cell_width * col + swatch_width + 7

        ax.text(text_pos_x, y, name, fontsize=14,
                horizontalalignment='left',
                verticalalignment='center')

        ax.add_patch(
            Rectangle(xy=(swatch_start_x, y-9), width=swatch_width,
                      height=18, facecolor=colors[name], edgecolor='0.7')
        )

    return fig
```

### Base colors

```
plot_colortable(mcolors.BASE_COLORS, ncols=3, sort_colors=False)
```

### Tableau Palette

```
plot_colortable(mcolors.TABLEAU_COLORS, ncols=2, sort_colors=False)
```

## CSS Colors

```
plot_colortable(mcolors.CSS4_COLORS)
plt.show()
```

### XKCD Colors

Matplotlib supports colors from the xkcd color survey, e.g. `"xkcd:sky blue"`. Since this contains almost 1000 colors, a figure of this would be very large and is thus omitted here. You can use the following code to generate the overview yourself

```
xkcd_fig = plot_colortable(mcolors.XKCD_COLORS)
xkcd_fig.savefig("XKCD_Colors.png")
```

**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.colors*
- *matplotlib.colors.rgb_to_hsv*
- *matplotlib.colors.to_rgba*
- *matplotlib.figure.Figure.get_size_inches*
- *matplotlib.figure.Figure.subplots_adjust*
- *matplotlib.axes.Axes.text*
- *matplotlib.patches.Rectangle*

**Total running time of the script:** (0 minutes 1.216 seconds)

### Ways to set a color's alpha value

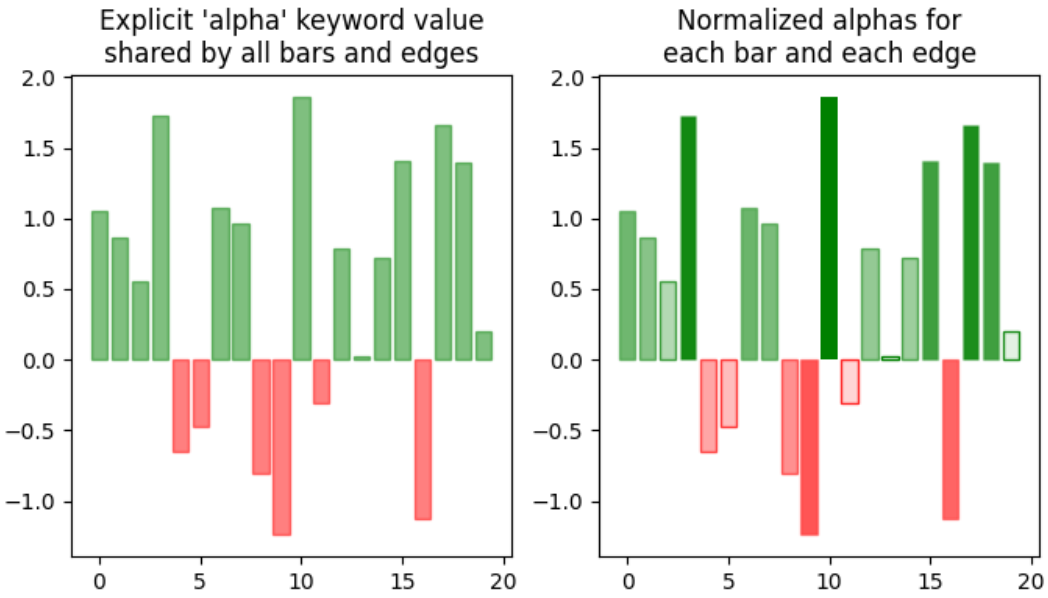
Compare setting alpha by the *alpha* keyword argument and by one of the Matplotlib color formats. Often, the *alpha* keyword is the only tool needed to add transparency to a color. In some cases, the *(matplotlib_color, alpha)* color format provides an easy way to fine-tune the appearance of a Figure.

```
import matplotlib.pyplot as plt
import numpy as np

# Fixing random state for reproducibility.
np.random.seed(19680801)

fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(8, 4))

x_values = [n for n in range(20)]
y_values = np.random.randn(20)

facecolors = ['green' if y > 0 else 'red' for y in y_values]
edgecolors = facecolors

ax1.bar(x_values, y_values, color=facecolors, edgecolor=edgecolors, alpha=0.5)
ax1.set_title("Explicit 'alpha' keyword value\nshared by all bars and edges")
```

(continues on next page)

(continued from previous page)

```
# Normalize y values to get distinct face alpha values.
abs_y = [abs(y) for y in y_values]
face_alphas = [n / max(abs_y) for n in abs_y]
edge_alphas = [1 - alpha for alpha in face_alphas]

colors_with_alphas = list(zip(facecolors, face_alphas))
edgecolors_with_alphas = list(zip(edgecolors, edge_alphas))

ax2.bar(x_values, y_values, color=colors_with_alphas,
        edgecolor=edgecolors_with_alphas)
ax2.set_title('Normalized alphas for\neach bar and each edge')

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.bar*
- *matplotlib.pyplot.subplots*

### 6.25.8 Shapes and collections

#### Arrow guide

Adding arrow patches to plots.

Arrows are often used to annotate plots. This tutorial shows how to plot arrows that behave differently when the data limits on a plot are changed. In general, points on a plot can either be fixed in "data space" or "display space". Something plotted in data space moves when the data limits are altered - an example would be the points in a scatter plot. Something plotted in display space stays static when data limits are altered - an example would be a figure title or the axis labels.

Arrows consist of a head (and possibly a tail) and a stem drawn between a start point and end point, called 'anchor points' from now on. Here we show three use cases for plotting arrows, depending on whether the head or anchor points need to be fixed in data or display space:

1. Head shape fixed in display space, anchor points fixed in data space
2. Head shape and anchor points fixed in display space
3. Entire patch fixed in data space

Below each use case is presented in turn.

```
import matplotlib.pyplot as plt

import matplotlib.patches as mpatches

x_tail = 0.1
y_tail = 0.5
x_head = 0.9
y_head = 0.8
dx = x_head - x_tail
dy = y_head - y_tail
```

#### Head shape fixed in display space and anchor points fixed in data space

This is useful if you are annotating a plot, and don't want the arrow to change shape or position if you pan or scale the plot.

In this case we use *patches.FancyArrowPatch*.

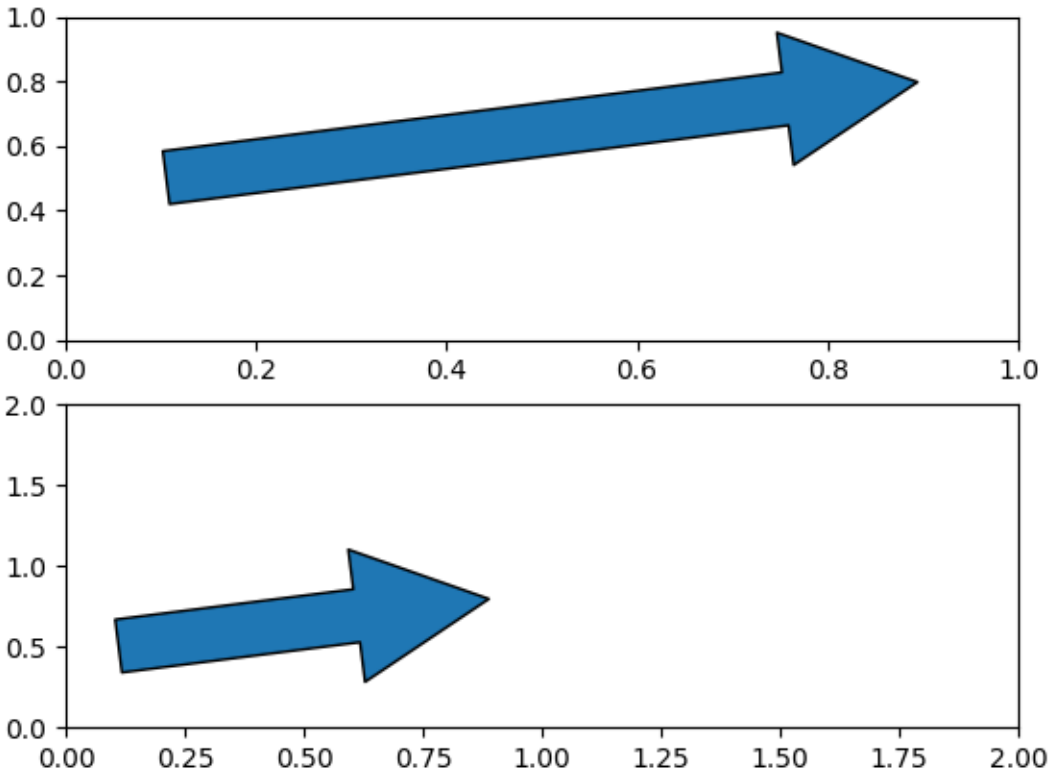
Note that when the axis limits are changed, the arrow shape stays the same, but the anchor points move.

```
fig, axs = plt.subplots(nrows=2)
arrow = mpatches.FancyArrowPatch((x_tail, y_tail), (x_head, y_head),
                                 mutation_scale=100)
axs[0].add_patch(arrow)

arrow = mpatches.FancyArrowPatch((x_tail, y_tail), (x_head, y_head),
                                 mutation_scale=100)
```

(continues on next page)

(continued from previous page)

```
axs[1].add_patch(arrow)
axs[1].set(xlim=(0, 2), ylim=(0, 2))
```




### Head shape and anchor points fixed in display space

This is useful if you are annotating a plot, and don't want the arrow to change shape or position if you pan or scale the plot.

In this case we use *patches.FancyArrowPatch*, and pass the keyword argument `transform=ax.transAxes` where `ax` is the Axes we are adding the patch to.

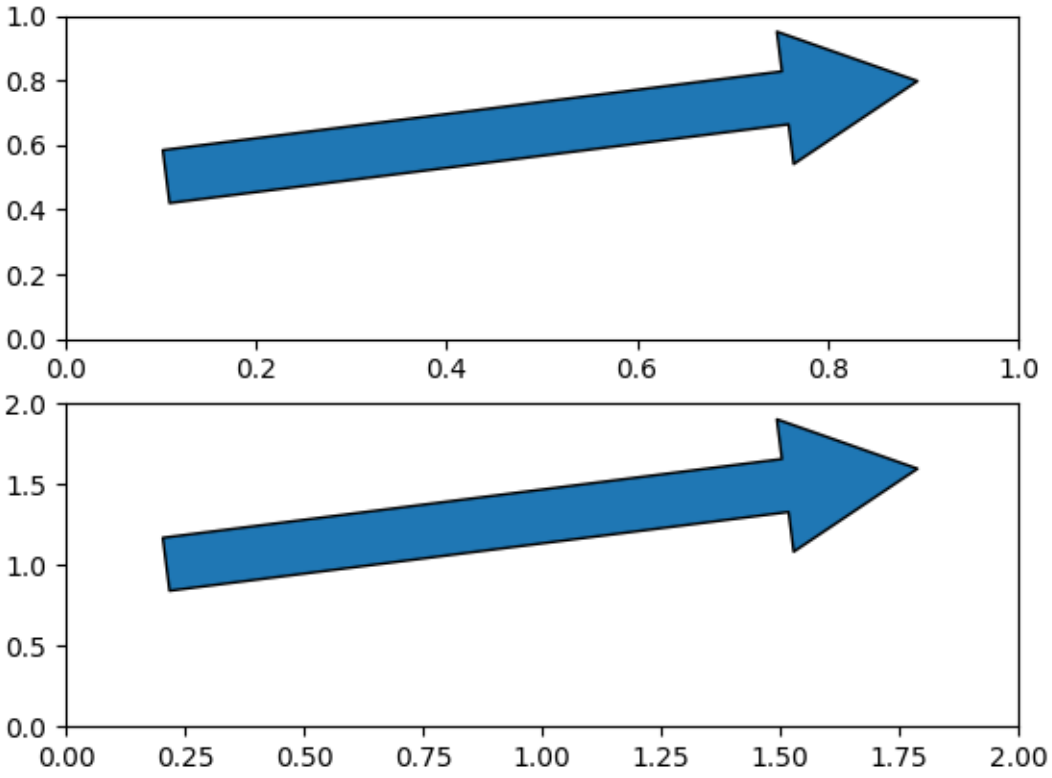
Note that when the axis limits are changed, the arrow shape and location stay the same.

```
fig, axs = plt.subplots(nrows=2)
arrow = mpatches.FancyArrowPatch((x_tail, y_tail), (x_head, y_head),
                                 mutation_scale=100,
                                 transform=axs[0].transAxes)
axs[0].add_patch(arrow)

arrow = mpatches.FancyArrowPatch((x_tail, y_tail), (x_head, y_head),
```

(continues on next page)

(continued from previous page)

```
                                    mutation_scale=100,
                                    transform=axs[1].transAxes)
axs[1].add_patch(arrow)
axs[1].set(xlim=(0, 2), ylim=(0, 2))
```




### Head shape and anchor points fixed in data space

In this case we use *patches.Arrow*, or *patches.FancyArrow* (the latter is in orange).

Note that when the axis limits are changed, the arrow shape and location change.

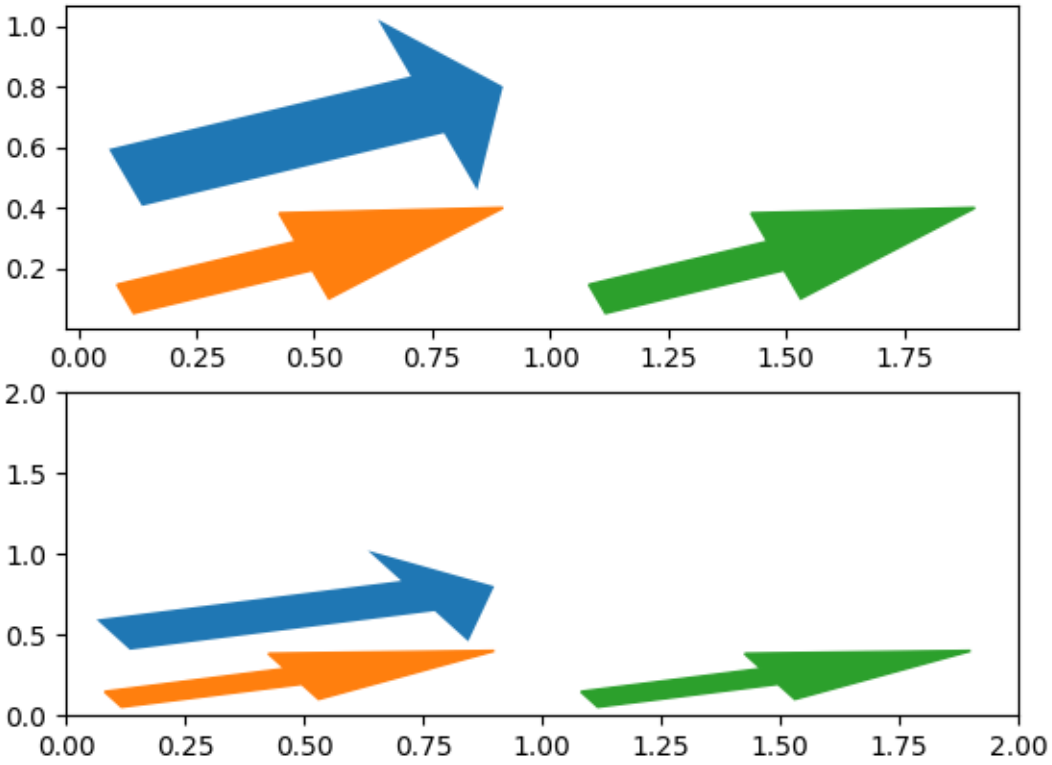
*FancyArrow*'s API is relatively awkward, and requires in particular passing `length_includes_head=True` so that the arrow *tip* is `(dx, dy)` away from the arrow start. It is only included in this reference because it is the arrow class returned by *Axes.arrow* (in green).

```
fig, axs = plt.subplots(nrows=2)

arrow = mpatches.Arrow(x_tail, y_tail, dx, dy)
axs[0].add_patch(arrow)
arrow = mpatches.FancyArrow(x_tail, y_tail - .4, dx, dy,
```

(continues on next page)

(continued from previous page)

```
                              width=.1, length_includes_head=True, color="C1")
axs[0].add_patch(arrow)
axs[0].arrow(x_tail + 1, y_tail - .4, dx, dy,
             width=.1, length_includes_head=True, color="C2")

arrow = mpatches.Arrow(x_tail, y_tail, dx, dy)
axs[1].add_patch(arrow)
arrow = mpatches.FancyArrow(x_tail, y_tail - .4, dx, dy,
                              width=.1, length_includes_head=True, color="C1")
axs[1].add_patch(arrow)
axs[1].arrow(x_tail + 1, y_tail - .4, dx, dy,
             width=.1, length_includes_head=True, color="C2")
axs[1].set(xlim=(0, 2), ylim=(0, 2))
```




```
plt.show()
```

### Reference for Matplotlib artists

This example displays several of Matplotlib's graphics primitives (artists). A full list of artists is documented at *the artist API*.

See also *Circles, Wedges and Polygons*, which groups all artists into a single `PatchCollection` instead.

Copyright (c) 2010, Bartosz Telenczuk BSD License

```
import matplotlib.pyplot as plt

import matplotlib as mpl
import matplotlib.lines as mlines
import matplotlib.patches as mpatches
import matplotlib.path as mpath

# Prepare the data for the PathPatch below.
Path = mpath.Path
codes, verts = zip(*[
    (Path.MOVETO, [0.018, -0.11]),
    (Path.CURVE4, [-0.031, -0.051]),
    (Path.CURVE4, [-0.115, 0.073]),
    (Path.CURVE4, [-0.03, 0.073]),
    (Path.LINETO, [-0.011, 0.039]),
    (Path.CURVE4, [0.043, 0.121]),
    (Path.CURVE4, [0.075, -0.005]),
    (Path.CURVE4, [0.035, -0.027]),
    (Path.CLOSEPOLY, [0.018, -0.11])])

artists = [
    mpatches.Circle((0, 0), 0.1, ec="none"),
    mpatches.Rectangle((-0.025, -0.05), 0.05, 0.1, ec="none"),
    mpatches.Wedge((0, 0), 0.1, 30, 270, ec="none"),
    mpatches.RegularPolygon((0, 0), 5, radius=0.1),
    mpatches.Ellipse((0, 0), 0.2, 0.1),
    mpatches.Arrow(-0.05, -0.05, 0.1, 0.1, width=0.1),
    mpatches.PathPatch(mpath.Path(verts, codes), ec="none"),
    mpatches.FancyBboxPatch((-0.025, -0.05), 0.05, 0.1, ec="none",
                            boxstyle=mpatches.BoxStyle("Round", pad=0.02)),
    mlines.Line2D([-0.06, 0.0, 0.1], [0.05, -0.05, 0.05], lw=5),
]

axs = plt.figure(figsize=(6, 6), layout="constrained").subplots(3, 3)
for i, (ax, artist) in enumerate(zip(axs.flat, artists)):
    artist.set(color=mpl.colormaps["hsv"](i / len(artists)))
    ax.add_artist(artist)
    ax.set(title=type(artist).__name__,
           aspect=1, xlim=(-.2, .2), ylim=(-.2, .2))
    ax.set_axis_off()
plt.show()
```

**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.path*
- *matplotlib.path.Path*
- *matplotlib.lines*
- *matplotlib.lines.Line2D*
- *matplotlib.patches*
- *matplotlib.patches.Circle*
- *matplotlib.patches.Ellipse*

- *matplotlib.patches.Wedge*
- *matplotlib.patches.Rectangle*
- *matplotlib.patches.Arrow*
- *matplotlib.patches.PathPatch*
- *matplotlib.patches.FancyBboxPatch*
- *matplotlib.patches.RegularPolygon*
- *matplotlib.axes.Axes.add_artist*

### Line, Poly and RegularPoly Collection with autoscaling

For the first two subplots, we will use spirals. Their size will be set in plot units, not data units. Their positions will be set in data units by using the *offsets* and *offset_transform* keyword arguments of the *LineCollection* and *PolyCollection*.

The third subplot will make regular polygons, with the same type of scaling and positioning as in the first two.

The last subplot illustrates the use of `offsets=(xo, yo)`, that is, a single tuple instead of a list of tuples, to generate successively offset curves, with the offset given in data units. This behavior is available only for the LineCollection.

```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib import collections, transforms

nverts = 50
npts = 100

# Make some spirals
r = np.arange(nverts)
theta = np.linspace(0, 2*np.pi, nverts)
xx = r * np.sin(theta)
yy = r * np.cos(theta)
spiral = np.column_stack([xx, yy])

# Fixing random state for reproducibility
rs = np.random.RandomState(19680801)

# Make some offsets
xyo = rs.randn(npts, 2)

# Make a list of colors cycling through the default series.
colors = plt.rcParams['axes.prop_cycle'].by_key()['color']

fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2)
```

(continues on next page)

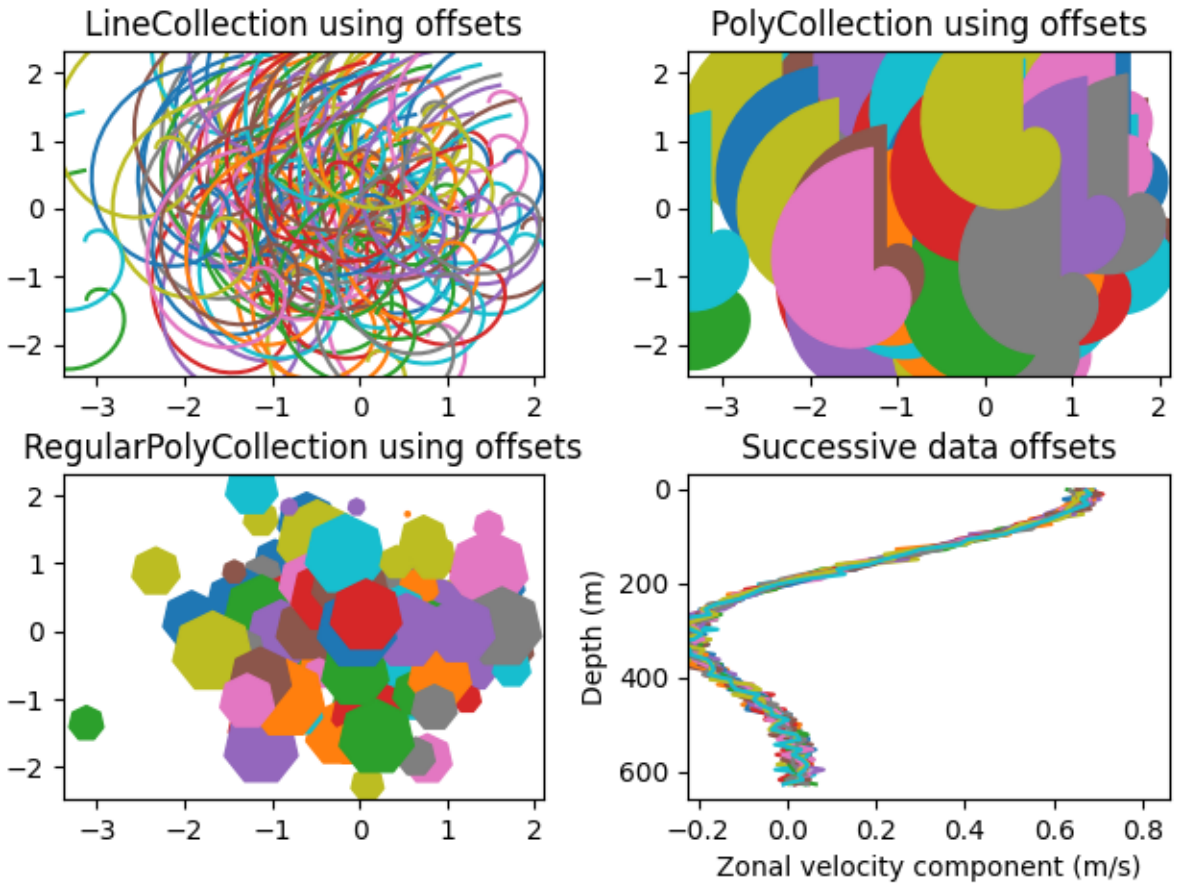
(continued from previous page)

```
fig.subplots_adjust(top=0.92, left=0.07, right=0.97,
                    hspace=0.3, wspace=0.3)


col = collections.LineCollection(
    [spiral], offsets=xyo, offset_transform=ax1.transData)
trans = fig.dpi_scale_trans + transforms.Affine2D().scale(1.0/72.0)
col.set_transform(trans)  # the points to pixels transform
# Note: the first argument to the collection initializer
# must be a list of sequences of (x, y) tuples; we have only
# one sequence, but we still have to put it in a list.
ax1.add_collection(col, autolim=True)
# autolim=True enables autoscaling.  For collections with
# offsets like this, it is neither efficient nor accurate,
# but it is good enough to generate a plot that you can use
# as a starting point.  If you know beforehand the range of
# x and y that you want to show, it is better to set them
# explicitly, leave out the *autolim* keyword argument (or set it to False),
# and omit the 'ax1.autoscale_view()' call below.

# Make a transform for the line segments such that their size is
# given in points:
col.set_color(colors)

ax1.autoscale_view()  # See comment above, after ax1.add_collection.
ax1.set_title('LineCollection using offsets')


# The same data as above, but fill the curves.
col = collections.PolyCollection(
    [spiral], offsets=xyo, offset_transform=ax2.transData)
trans = transforms.Affine2D().scale(fig.dpi/72.0)
col.set_transform(trans)  # the points to pixels transform
ax2.add_collection(col, autolim=True)
col.set_color(colors)


ax2.autoscale_view()
ax2.set_title('PolyCollection using offsets')

# 7-sided regular polygons

col = collections.RegularPolyCollection(
    7, sizes=np.abs(xx) * 10.0, offsets=xyo, offset_transform=ax3.transData)
trans = transforms.Affine2D().scale(fig.dpi / 72.0)
col.set_transform(trans)  # the points to pixels transform
ax3.add_collection(col, autolim=True)
col.set_color(colors)
ax3.autoscale_view()
ax3.set_title('RegularPolyCollection using offsets')
```

(continues on next page)

(continued from previous page)

```
# Simulate a series of ocean current profiles, successively
# offset by 0.1 m/s so that they form what is sometimes called
# a "waterfall" plot or a "stagger" plot.

nverts = 60
ncurves = 20
offs = (0.1, 0.0)

yy = np.linspace(0, 2*np.pi, nverts)
ym = np.max(yy)
xx = (0.2 + (ym - yy) / ym) ** 2 * np.cos(yy - 0.4) * 0.5
segs = []
for i in range(ncurves):
    xxx = xx + 0.02*rs.randn(nverts)
    curve = np.column_stack([xxx, yy * 100])
    segs.append(curve)

col = collections.LineCollection(segs, offsets=offs)
ax4.add_collection(col, autolim=True)
col.set_color(colors)
ax4.autoscale_view()
ax4.set_title('Successive data offsets')
ax4.set_xlabel('Zonal velocity component (m/s)')
ax4.set_ylabel('Depth (m)')
# Reverse the y-axis so depth increases downward
ax4.set_ylim(ax4.get_ylim()[::-1])


plt.show()
```




**References**

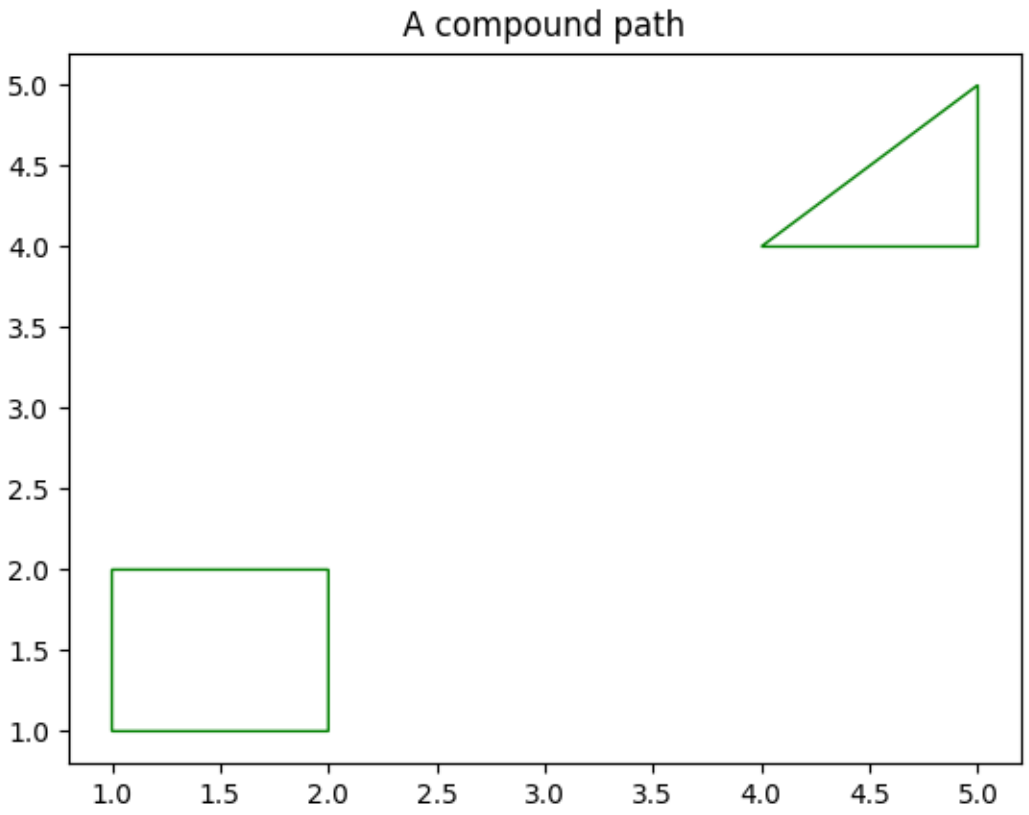
The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.figure.Figure*
- *matplotlib.collections*
- *matplotlib.collections.LineCollection*
- *matplotlib.collections.RegularPolyCollection*
- *matplotlib.axes.Axes.add_collection*
- *matplotlib.axes.Axes.autoscale_view*
- *matplotlib.transforms.Affine2D*
- *matplotlib.transforms.Affine2D.scale*

### Compound path

Make a compound path -- in this case two simple polygons, a rectangle and a triangle. Use `CLOSEPOLY` and `MOVETO` for the different parts of the compound path

```
import matplotlib.pyplot as plt

from matplotlib.patches import PathPatch
from matplotlib.path import Path

vertices = []
codes = []

codes = [Path.MOVETO] + [Path.LINETO]*3 + [Path.CLOSEPOLY]
vertices = [(1, 1), (1, 2), (2, 2), (2, 1), (0, 0)]

codes += [Path.MOVETO] + [Path.LINETO]*2 + [Path.CLOSEPOLY]
vertices += [(4, 4), (5, 5), (5, 4), (0, 0)]

path = Path(vertices, codes)

pathpatch = PathPatch(path, facecolor='none', edgecolor='green')

fig, ax = plt.subplots()
ax.add_patch(pathpatch)
ax.set_title('A compound path')

ax.autoscale_view()

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.path*
- *matplotlib.path.Path*
- *matplotlib.patches*
- *matplotlib.patches.PathPatch*
- *matplotlib.axes.Axes.add_patch*
- *matplotlib.axes.Axes.autoscale_view*

### Dolphins

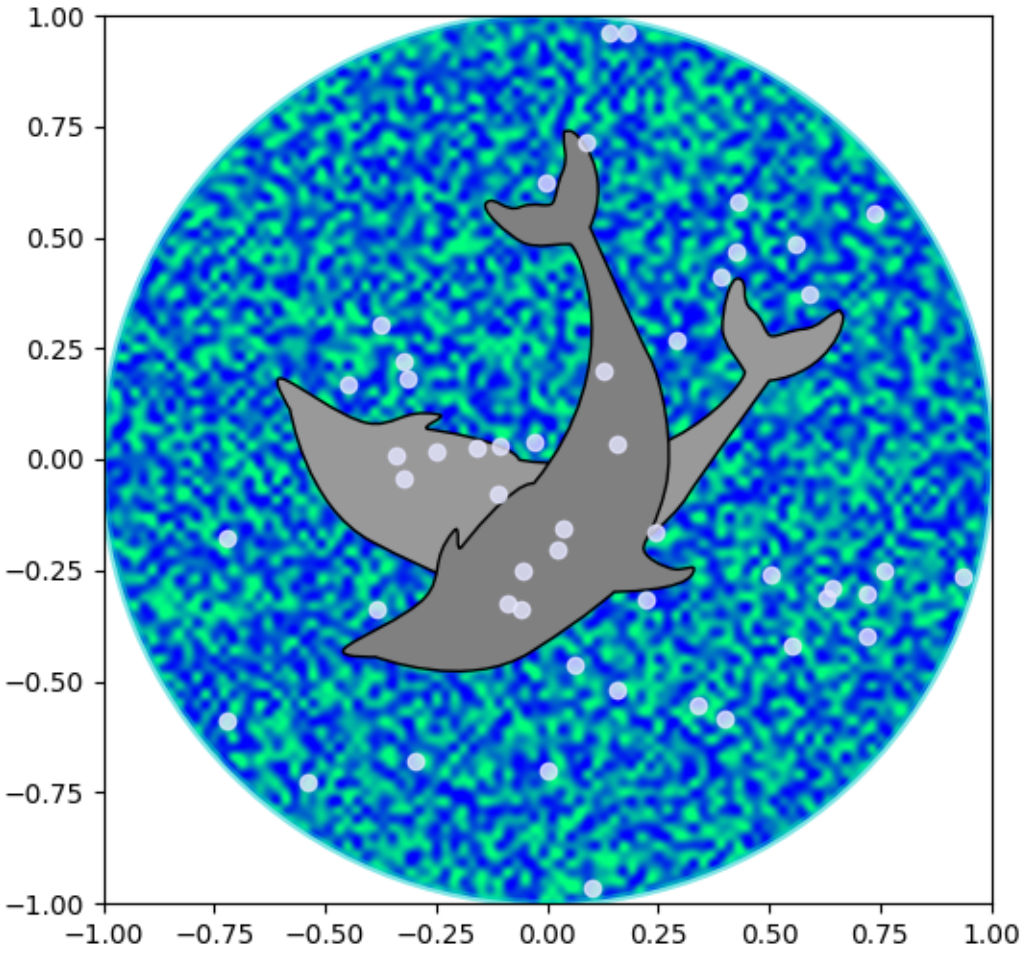
This example shows how to draw, and manipulate shapes given vertices and nodes using the *Path*, *PathPatch* and *transforms* classes.

```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib.cm as cm
from matplotlib.patches import Circle, PathPatch
from matplotlib.path import Path
from matplotlib.transforms import Affine2D

# Fixing random state for reproducibility
np.random.seed(19680801)


r = np.random.rand(50)
t = np.random.rand(50) * np.pi * 2.0
x = r * np.cos(t)
y = r * np.sin(t)

fig, ax = plt.subplots(figsize=(6, 6))
circle = Circle((0, 0), 1, facecolor='none',
                edgecolor=(0, 0.8, 0.8), linewidth=3, alpha=0.5)
ax.add_patch(circle)

im = plt.imshow(np.random.random((100, 100)),
                origin='lower', cmap=cm.winter,
                interpolation='spline36',
                extent=(-1, 1, -1, 1))
im.set_clip_path(circle)

plt.plot(x, y, 'o', color=(0.9, 0.9, 1.0), alpha=0.8)

# Dolphin from OpenClipart library by Andy Fitzsimon
#   <cc:License rdf:about="http://web.resource.org/cc/PublicDomain">
#     <cc:permits rdf:resource="http://web.resource.org/cc/Reproduction"/>
#     <cc:permits rdf:resource="http://web.resource.org/cc/Distribution"/>
#     <cc:permits rdf:resource="http://web.resource.org/cc/DerivativeWorks"/>
#   </cc:License>

dolphin = """
M -0.59739425,160.18173 C -0.62740401,160.18885 -0.57867129,160.11183
-0.57867129,160.11183 C -0.57867129,160.11183 -0.5438361,159.89315
-0.39514638,159.81496 C -0.24645668,159.73678 -0.18316813,159.71981
-0.18316813,159.71981 C -0.18316813,159.71981 -0.10322971,159.58124
-0.057804323,159.58725 C -0.029723983,159.58913 -0.061841603,159.60356
-0.071265813,159.62815 C -0.080250183,159.65325 -0.082918513,159.70554
-0.061841203,159.71248 C -0.040763903,159.7194 -0.0066711426,159.71091
0.077336307,159.73612 C 0.16879567,159.76377 0.28380306,159.86448
0.31516668,159.91533 C 0.3465303,159.96618 0.5011127,160.1771
0.5011127,160.1771 C 0.63668998,160.19238 0.67763022,160.31259
```

(continues on next page)

(continued from previous page)

```
0.66556395,160.32668 C 0.65339985,160.34212 0.66350443,160.33642
0.64907098,160.33088 C 0.63463742,160.32533 0.61309688,160.297
0.5789627,160.29339 C 0.54348657,160.28968 0.52329693,160.27674
0.50728856,160.27737 C 0.49060916,160.27795 0.48965803,160.31565
0.46114204,160.33673 C 0.43329696,160.35786 0.4570711,160.39871
0.43309565,160.40685 C 0.4105108,160.41442 0.39416631,160.33027
0.3954995,160.2935 C 0.39683269,160.25672 0.43807996,160.21522
0.44567915,160.19734 C 0.45327833,160.17946 0.27946869,159.9424
-0.061852613,159.99845 C -0.083965233,160.0427 -0.26176109,160.06683
-0.26176109,160.06683 C -0.30127962,160.07028 -0.21167141,160.09731
-0.24649368,160.1011 C -0.32642366,160.11569 -0.34521187,160.06895
-0.40622293,160.0819 C -0.467234,160.09485 -0.56738444,160.17461
-0.59739425,160.18173
"""

vertices = []
codes = []
parts = dolphin.split()
i = 0
code_map = {
    'M': Path.MOVETO,
    'C': Path.CURVE4,
    'L': Path.LINETO,
}

while i < len(parts):
    path_code = code_map[parts[i]]
    npoints = Path.NUM_VERTICES_FOR_CODE[path_code]
    codes.extend([path_code] * npoints)
    vertices.extend([[*map(float, y.split(','))]
                     for y in parts[i + 1:][:npoints]])
    i += npoints + 1
vertices = np.array(vertices)
vertices[:, 1] -= 160

dolphin_path = Path(vertices, codes)
dolphin_patch = PathPatch(dolphin_path, facecolor=(0.6, 0.6, 0.6),
                          edgecolor=(0.0, 0.0, 0.0))
ax.add_patch(dolphin_patch)

vertices = Affine2D().rotate_deg(60).transform(vertices)
dolphin_path2 = Path(vertices, codes)
dolphin_patch2 = PathPatch(dolphin_path2, facecolor=(0.5, 0.5, 0.5),
                           edgecolor=(0.0, 0.0, 0.0))
ax.add_patch(dolphin_patch2)

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.path*
- *matplotlib.path.Path*
- *matplotlib.patches*
- *matplotlib.patches.PathPatch*
- *matplotlib.patches.Circle*
- *matplotlib.axes.Axes.add_patch*
- *matplotlib.transforms*

- *matplotlib.transforms.Affine2D*
- *matplotlib.transforms.Affine2D.rotate_deg*

### Mmh Donuts!!!

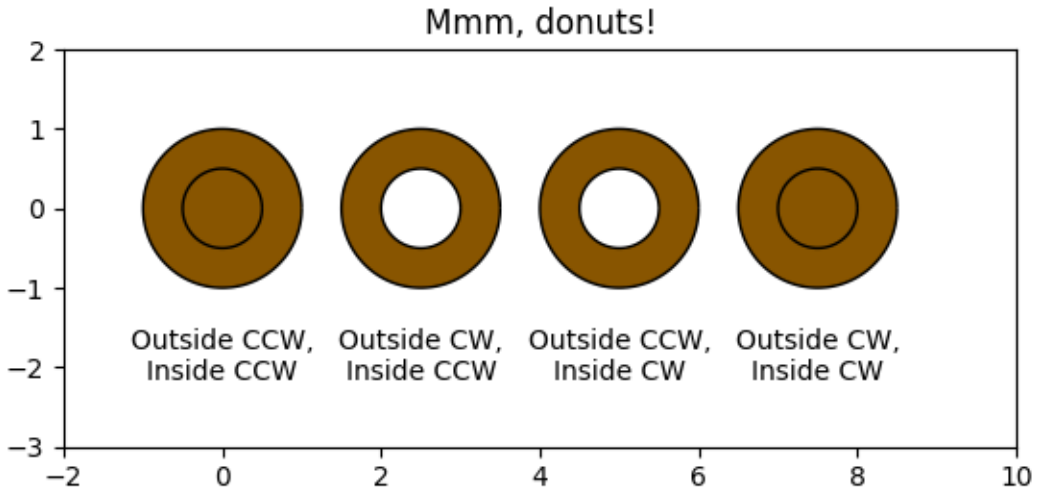
Draw donuts (miam!) using *Path*s and *PathPatch*es. This example shows the effect of the path's orientations in a compound path.

```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib.patches as mpatches
import matplotlib.path as mpath


def wise(v):
    if v == 1:
        return "CCW"
    else:
        return "CW"


def make_circle(r):
    t = np.arange(0, np.pi * 2.0, 0.01)
    t = t.reshape((len(t), 1))
    x = r * np.cos(t)
    y = r * np.sin(t)
    return np.hstack((x, y))

Path = mpath.Path

fig, ax = plt.subplots()

inside_vertices = make_circle(0.5)
outside_vertices = make_circle(1.0)
codes = np.ones(
    len(inside_vertices), dtype=mpath.Path.code_type) * mpath.Path.LINETO
codes[0] = mpath.Path.MOVETO

for i, (inside, outside) in enumerate(((1, 1), (1, -1), (-1, 1), (-1, -1))):
    # Concatenate the inside and outside subpaths together, changing their
    # order as needed
    vertices = np.concatenate((outside_vertices[::outside],
                               inside_vertices[::inside]))
    # Shift the path
    vertices[:, 0] += i * 2.5
    # The codes will be all "LINETO" commands, except for "MOVETO"s at the
    # beginning of each subpath
    all_codes = np.concatenate((codes, codes))
    # Create the Path object
```

(continues on next page)

(continued from previous page)

```
    path = mpath.Path(vertices, all_codes)
    # Add plot it
    patch = mpatches.PathPatch(path, facecolor='#885500', edgecolor='black')
    ax.add_patch(patch)

    ax.annotate(f"Outside {wise(outside)},\nInside {wise(inside)}",
                (i * 2.5, -1.5), va="top", ha="center")

ax.set_xlim(-2, 10)
ax.set_ylim(-3, 2)
ax.set_title('Mmm, donuts!')
ax.set_aspect(1.0)
plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

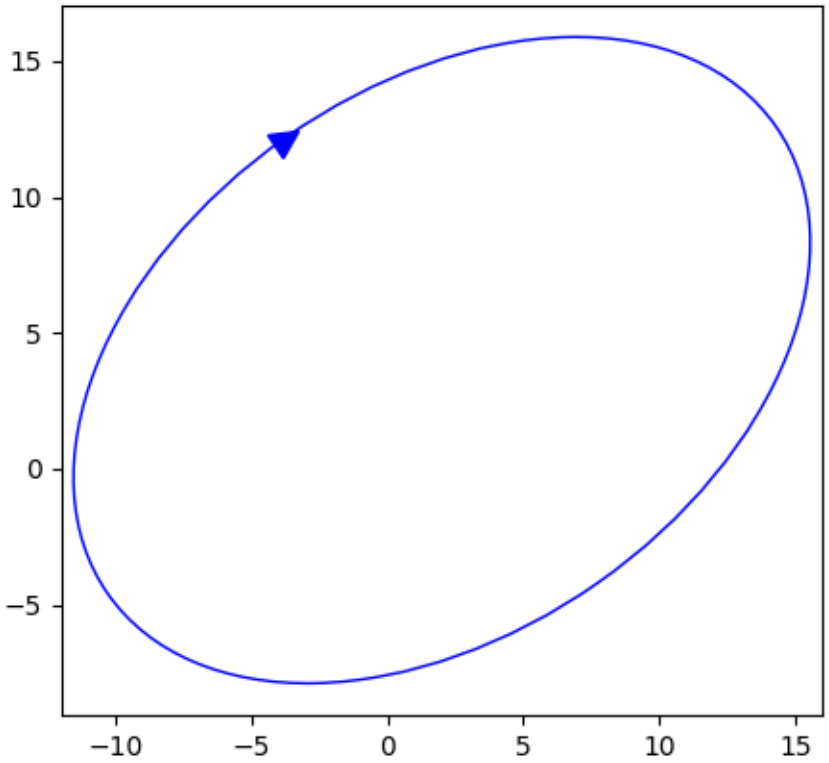
- *matplotlib.path*
- *matplotlib.path.Path*
- *matplotlib.patches*

- *matplotlib.patches.PathPatch*
- *matplotlib.patches.Circle*
- *matplotlib.axes.Axes.add_patch*
- *matplotlib.axes.Axes.annotate*
- *matplotlib.axes.Axes.set_aspect*
- *matplotlib.axes.Axes.set_xlim*
- *matplotlib.axes.Axes.set_ylim*
- *matplotlib.axes.Axes.set_title*

### Ellipse with orientation arrow demo

This demo shows how to draw an ellipse with an orientation arrow (clockwise or counterclockwise). Compare this to the *Ellipse collection example*.

```
import matplotlib.pyplot as plt

from matplotlib.markers import MarkerStyle
from matplotlib.patches import Ellipse
from matplotlib.transforms import Affine2D

# Create a figure and axis
fig, ax = plt.subplots(subplot_kw={"aspect": "equal"})

ellipse = Ellipse(
    xy=(2, 4),
    width=30,
    height=20,
    angle=35,
    facecolor="none",
    edgecolor="b"
)
ax.add_patch(ellipse)

# Plot an arrow marker at the end point of minor axis
vertices = ellipse.get_co_vertices()
t = Affine2D().rotate_deg(ellipse.angle)
ax.plot(
    vertices[0][0],
    vertices[0][1],
    color="b",
    marker=MarkerStyle(">", "full", t),
    markersize=10
)
# Note: To reverse the orientation arrow, switch the marker type from > to <.

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.patches*
- *matplotlib.patches.Ellipse*

### Ellipse Collection

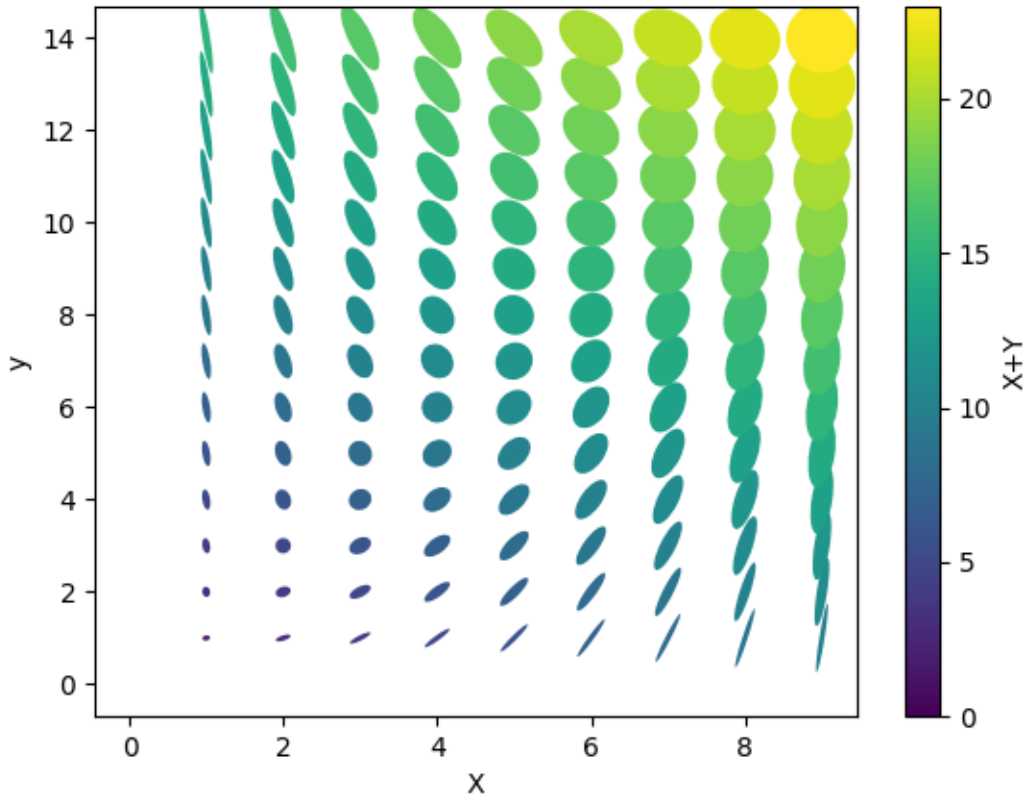
Drawing a collection of ellipses. While this would equally be possible using a *EllipseCollection* or *PathCollection*, the use of an *EllipseCollection* allows for much shorter code.

```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.collections import EllipseCollection

x = np.arange(10)
y = np.arange(15)
X, Y = np.meshgrid(x, y)
```

(continues on next page)

(continued from previous page)

```
XY = np.column_stack((X.ravel(), Y.ravel()))

ww = X / 10.0
hh = Y / 15.0
aa = X * 9


fig, ax = plt.subplots()

ec = EllipseCollection(ww, hh, aa, units='x', offsets=XY,
                       offset_transform=ax.transData)
ec.set_array((X + Y).ravel())
ax.add_collection(ec)
ax.autoscale_view()
ax.set_xlabel('X')
ax.set_ylabel('y')
cbar = plt.colorbar(ec)
cbar.set_label('X+Y')
plt.show()
```




**References**

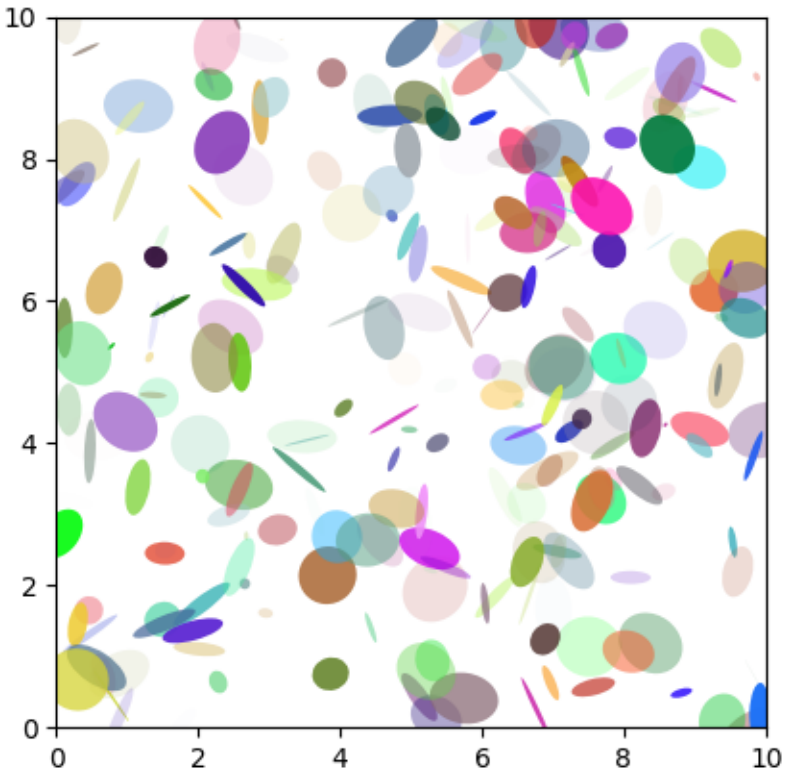
The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.collections*
- *matplotlib.collections.EllipseCollection*
- *matplotlib.axes.Axes.add_collection*
- *matplotlib.axes.Axes.autoscale_view*
- *matplotlib.cm.ScalarMappable.set_array*

### Ellipse Demo

Draw many ellipses. Here individual ellipses are drawn. Compare this to the *Ellipse collection example*.

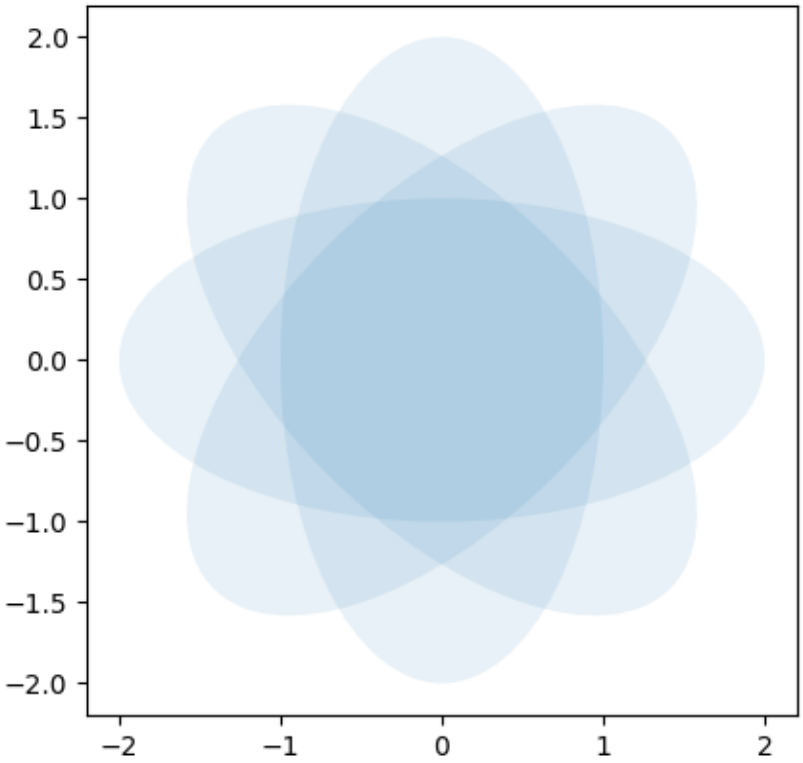
```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.patches import Ellipse

# Fixing random state for reproducibility
np.random.seed(19680801)

NUM = 250

ells = [Ellipse(xy=np.random.rand(2) * 10,
                width=np.random.rand(), height=np.random.rand(),
                angle=np.random.rand() * 360)
        for i in range(NUM)]

fig, ax = plt.subplots()
ax.set(xlim=(0, 10), ylim=(0, 10), aspect="equal")

for e in ells:
    ax.add_artist(e)
    e.set_clip_box(ax.bbox)
    e.set_alpha(np.random.rand())
    e.set_facecolor(np.random.rand(3))

plt.show()
```




### Ellipse Rotated

Draw many ellipses with different angles.

```
angle_step = 45  # degrees
angles = np.arange(0, 180, angle_step)

fig, ax = plt.subplots()
ax.set(xlim=(-2.2, 2.2), ylim=(-2.2, 2.2), aspect="equal")

for angle in angles:
    ellipse = Ellipse((0, 0), 4, 2, angle=angle, alpha=0.1)
    ax.add_artist(ellipse)

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.patches*
- *matplotlib.patches.Ellipse*
- *matplotlib.axes.Axes.add_artist*
- *matplotlib.artist.Artist.set_clip_box*
- *matplotlib.artist.Artist.set_alpha*
- *matplotlib.patches.Patch.set_facecolor*

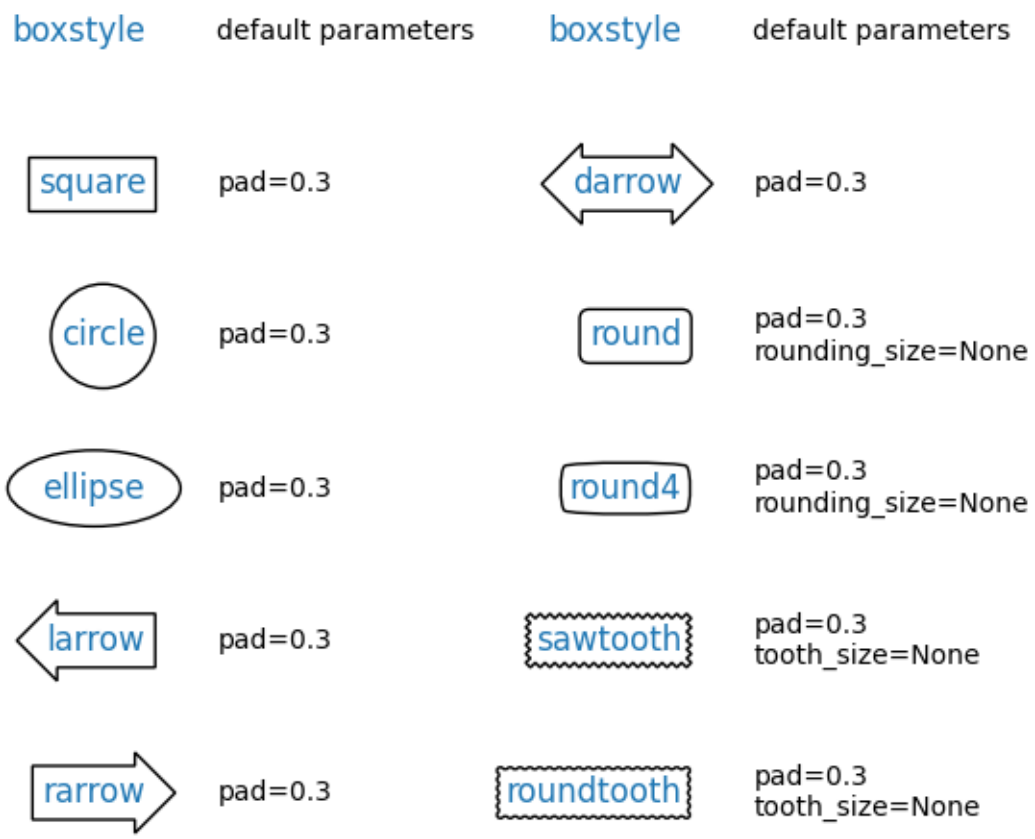
### Drawing fancy boxes

The following examples show how to plot boxes with different visual properties.

```
import inspect

import matplotlib.pyplot as plt

import matplotlib.patches as mpatch
from matplotlib.patches import FancyBboxPatch
import matplotlib.transforms as mtransforms
```

First we'll show some sample boxes with fancybox.

```
styles = mpatch.BoxStyle.get_styles()
ncol = 2
nrow = (len(styles) + 1) // ncol
axs = (plt.figure(figsize=(3 * ncol, 1 + nrow))
       .add_gridspec(1 + nrow, ncol, wspace=.5).subplots())
for ax in axs.flat:
    ax.set_axis_off()
for ax in axs[0, :]:
    ax.text(.2, .5, "boxstyle",
            transform=ax.transAxes, size="large", color="tab:blue",
            horizontalalignment="right", verticalalignment="center")
    ax.text(.4, .5, "default parameters",
            transform=ax.transAxes,
            horizontalalignment="left", verticalalignment="center")
for ax, (stylename, stylecls) in zip(axs[1:, :].T.flat, styles.items()):
    ax.text(.2, .5, stylename, bbox=dict(boxstyle=stylename, fc="w", ec="k"),
            transform=ax.transAxes, size="large", color="tab:blue",
            horizontalalignment="right", verticalalignment="center")
    ax.text(.4, .5, str(inspect.signature(stylecls))[1:-1].replace(", ", "\n
 ↪"),
            transform=ax.transAxes,
            horizontalalignment="left", verticalalignment="center")
```




Next we'll show off multiple fancy boxes at once.

```
def add_fancy_patch_around(ax, bb, **kwargs):
    fancy = FancyBboxPatch(bb.p0, bb.width, bb.height,
                           fc=(1, 0.8, 1, 0.5), ec=(1, 0.5, 1, 0.5),
                           **kwargs)
    ax.add_patch(fancy)
    return fancy


def draw_control_points_for_patches(ax):
    for patch in ax.patches:
        patch.axes.plot(*patch.get_path().vertices.T, ".",
                        c=patch.get_edgecolor())
```

(continues on next page)

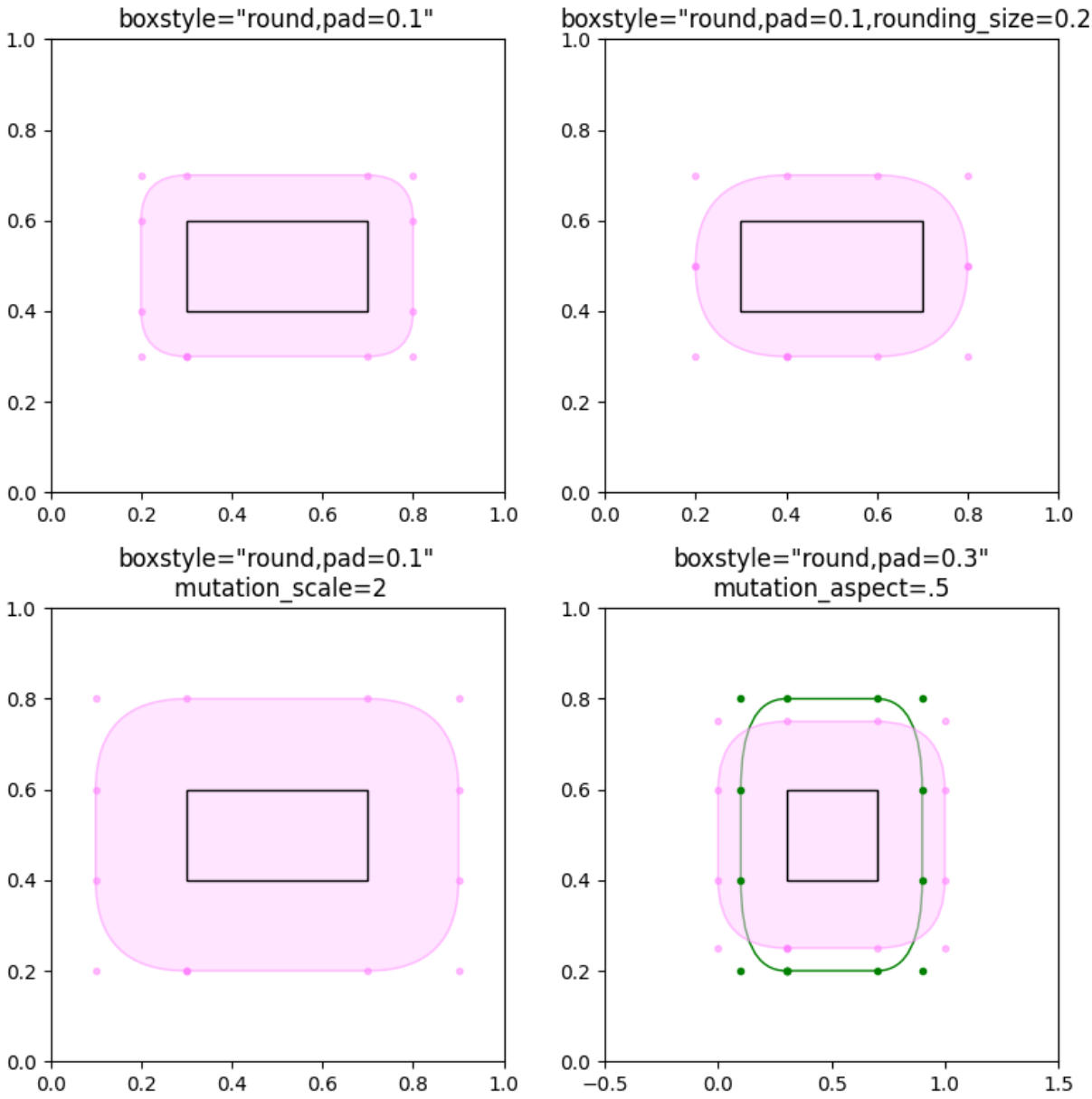
(continued from previous page)

```
fig, axs = plt.subplots(2, 2, figsize=(8, 8))

# Bbox object around which the fancy box will be drawn.
bb = mtransforms.Bbox([[0.3, 0.4], [0.7, 0.6]])

ax = axs[0, 0]
# a fancy box with round corners. pad=0.1
fancy = add_fancy_patch_around(ax, bb, boxstyle="round,pad=0.1")
ax.set(xlim=(0, 1), ylim=(0, 1), aspect=1,
       title='boxstyle="round,pad=0.1"')

ax = axs[0, 1]
# bbox=round has two optional arguments: pad and rounding_size.
# They can be set during the initialization.
fancy = add_fancy_patch_around(ax, bb, boxstyle="round,pad=0.1")
# The boxstyle and its argument can be later modified with set_boxstyle().
# Note that the old attributes are simply forgotten even if the boxstyle name
# is same.
fancy.set_boxstyle("round,pad=0.1,rounding_size=0.2")
# or: fancy.set_boxstyle("round", pad=0.1, rounding_size=0.2)
ax.set(xlim=(0, 1), ylim=(0, 1), aspect=1,
       title='boxstyle="round,pad=0.1,rounding_size=0.2"')

ax = axs[1, 0]
# mutation_scale determines the overall scale of the mutation, i.e. both pad
# and rounding_size is scaled according to this value.
fancy = add_fancy_patch_around(
    ax, bb, boxstyle="round,pad=0.1", mutation_scale=2)
ax.set(xlim=(0, 1), ylim=(0, 1), aspect=1,
       title='boxstyle="round,pad=0.1"\n mutation_scale=2')

ax = axs[1, 1]
# When the aspect ratio of the Axes is not 1, the fancy box may not be what␣
 ↪you
# expected (green).
fancy = add_fancy_patch_around(ax, bb, boxstyle="round,pad=0.2")
fancy.set(facecolor="none", edgecolor="green")
# You can compensate this by setting the mutation_aspect (pink).
fancy = add_fancy_patch_around(
    ax, bb, boxstyle="round,pad=0.3", mutation_aspect=0.5)
ax.set(xlim=(-.5, 1.5), ylim=(0, 1), aspect=2,
       title='boxstyle="round,pad=0.3"\nmutation_aspect=.5')

for ax in axs.flat:
    draw_control_points_for_patches(ax)
    # Draw the original bbox (using boxstyle=square with pad=0).
    fancy = add_fancy_patch_around(ax, bb, boxstyle="square,pad=0")
    fancy.set(edgecolor="black", facecolor="none", zorder=10)

fig.tight_layout()
```

(continues on next page)

(continued from previous page)

```
plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.patches*
- *matplotlib.patches.FancyBboxPatch*
- *matplotlib.patches.BoxStyle*
- matplotlib.patches.BoxStyle.get_styles

- *matplotlib.transforms.Bbox*
- *matplotlib.figure.Figure.text*
- *matplotlib.axes.Axes.text*

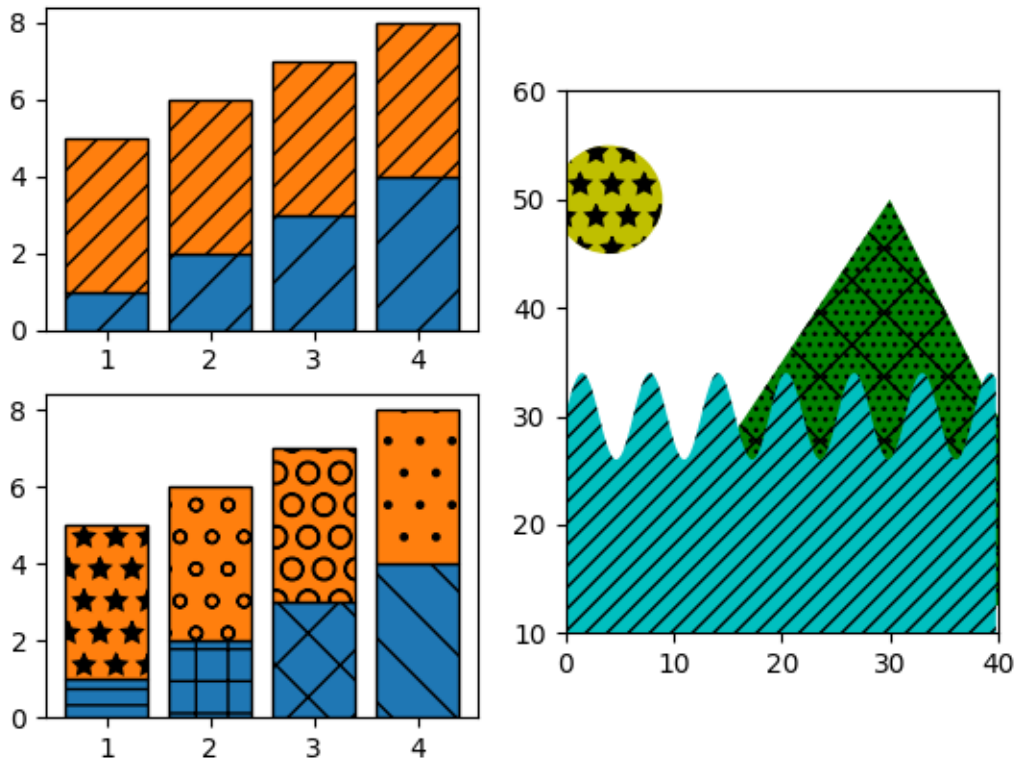
**Total running time of the script:** (0 minutes 1.185 seconds)

## Hatch demo

Hatches can be added to most polygons in Matplotlib, including *bar*, *fill_between*, *contourf*, and children of *Polygon*. They are currently supported in the PS, PDF, SVG, macosx, and Agg backends. The WX and Cairo backends do not currently support hatching.

See also *Contourf Hatching* for an example using *contourf*, and *Hatch style reference* for swatches of the existing hatches.

```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.patches import Ellipse, Polygon

x = np.arange(1, 5)
y1 = np.arange(1, 5)
y2 = np.ones(y1.shape) * 4

fig = plt.figure()
axs = fig.subplot_mosaic([['bar1', 'patches'], ['bar2', 'patches']])

axs['bar1'].bar(x, y1, edgecolor='black', hatch="/")
axs['bar1'].bar(x, y2, bottom=y1, edgecolor='black', hatch='//')

axs['bar2'].bar(x, y1, edgecolor='black', hatch=['--', '+', 'x', '\\'])
axs['bar2'].bar(x, y2, bottom=y1, edgecolor='black',
                hatch=['*', 'o', 'O', '.'])

x = np.arange(0, 40, 0.2)
axs['patches'].fill_between(x, np.sin(x) * 4 + 30, y2=0,
                            hatch='///', zorder=2, fc='c')
axs['patches'].add_patch(Ellipse((4, 50), 10, 10, fill=True,
                                 hatch='*', facecolor='y'))
axs['patches'].add_patch(Polygon([(10, 20), (30, 50), (50, 10)],
                                 hatch='\\/...', facecolor='g'))
axs['patches'].set_xlim([0, 40])
axs['patches'].set_ylim([10, 60])
axs['patches'].set_aspect(1)
plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

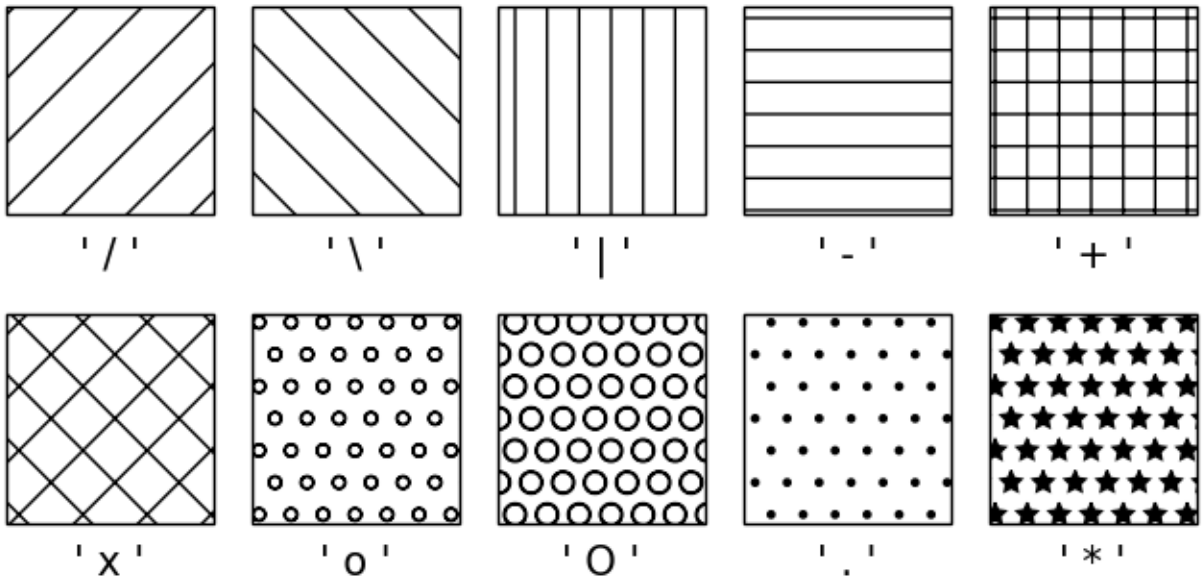
- *matplotlib.patches*
- *matplotlib.patches.Ellipse*
- *matplotlib.patches.Polygon*
- *matplotlib.axes.Axes.add_patch*
- *matplotlib.patches.Patch.set_hatch*
- *matplotlib.axes.Axes.bar* / *matplotlib.pyplot.bar*

### Hatch style reference

Hatches can be added to most polygons in Matplotlib, including *bar*, *fill_between*, *contourf*, and children of *Polygon*. They are currently supported in the PS, PDF, SVG, macosx, and Agg backends. The WX and Cairo backends do not currently support hatching.

See also *Contourf Hatching* for an example using *contourf*, and *Hatch demo* for more usage examples.

```
import matplotlib.pyplot as plt

from matplotlib.patches import Rectangle

fig, axs = plt.subplots(2, 5, layout='constrained', figsize=(6.4, 3.2))

hatches = ['/', '\\', '|', '-', '+', 'x', 'o', 'O', '.', '*']


def hatches_plot(ax, h):
    ax.add_patch(Rectangle((0, 0), 2, 2, fill=False, hatch=h))
    ax.text(1, -0.5, f"' {h} '", size=15, ha="center")
    ax.axis('equal')
    ax.axis('off')

for ax, h in zip(axs.flat, hatches):
    hatches_plot(ax, h)
```



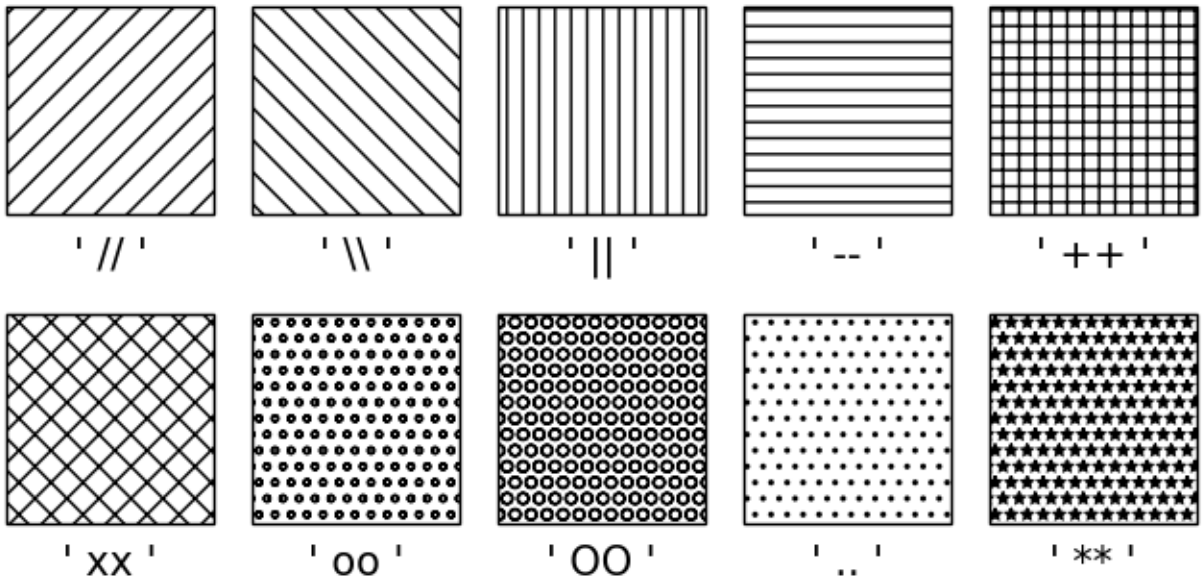

Hatching patterns can be repeated to increase the density.

```
fig, axs = plt.subplots(2, 5, layout='constrained', figsize=(6.4, 3.2))

hatches = ['//', '\\\\', '||', '--', '++', 'xx', 'oo', 'OO', '..', '**']

for ax, h in zip(axs.flat, hatches):
```

(continues on next page)

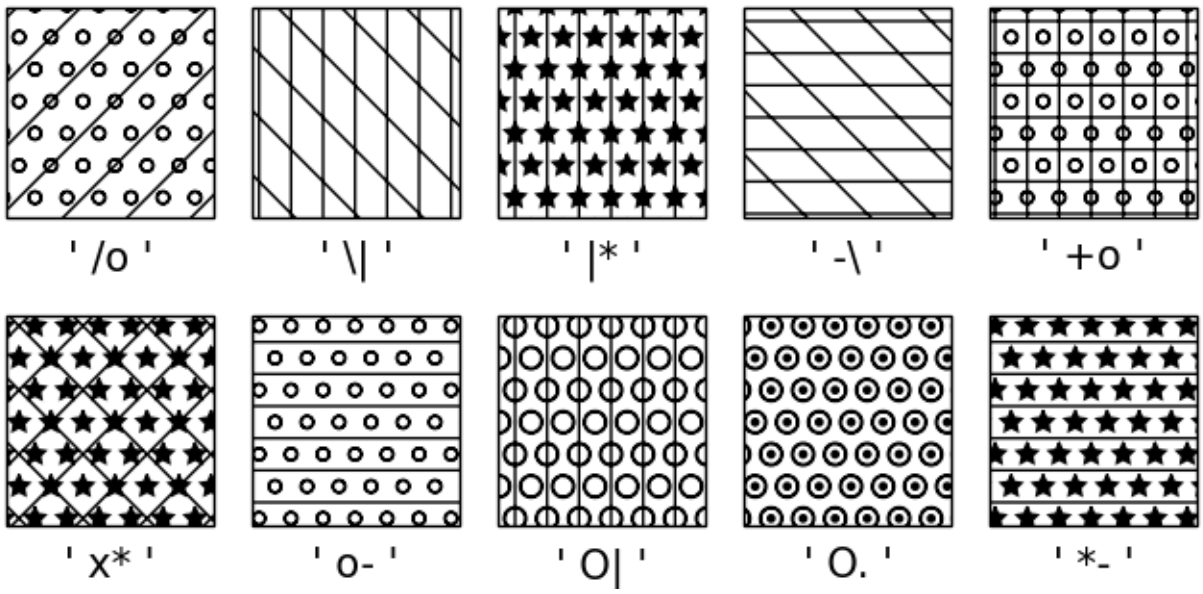
(continued from previous page)

```
    hatches_plot(ax, h)
```




Hatching patterns can be combined to create additional patterns.

```
fig, axs = plt.subplots(2, 5, layout='constrained', figsize=(6.4, 3.2))

hatches = ['/o', '\\|', '|*', '-\\', '+o', 'x*', 'o-', 'O|', 'O.', '*-']

for ax, h in zip(axs.flat, hatches):
    hatches_plot(ax, h)
```



**References**

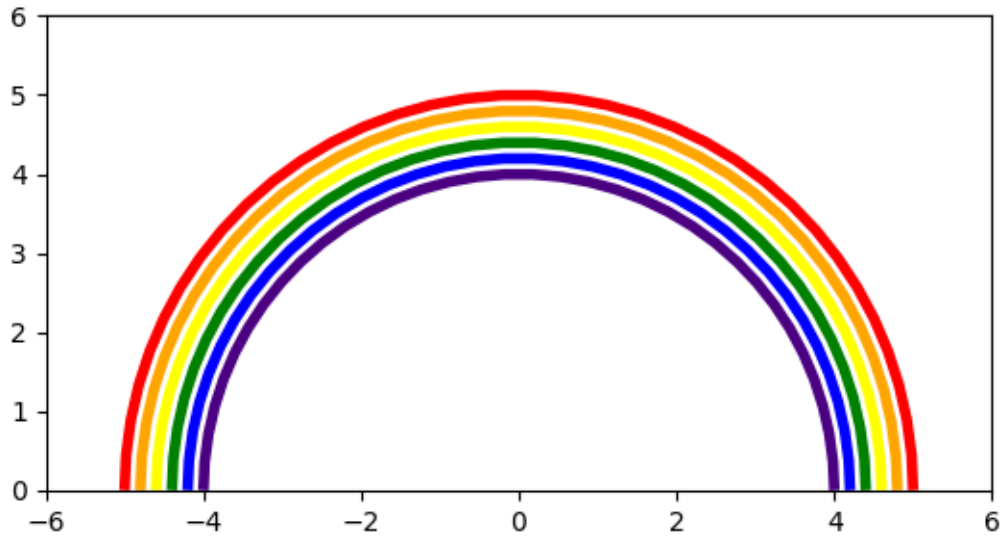
The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.patches*
- *matplotlib.patches.Rectangle*
- *matplotlib.axes.Axes.add_patch*
- *matplotlib.axes.Axes.text*

## Plotting multiple lines with a LineCollection

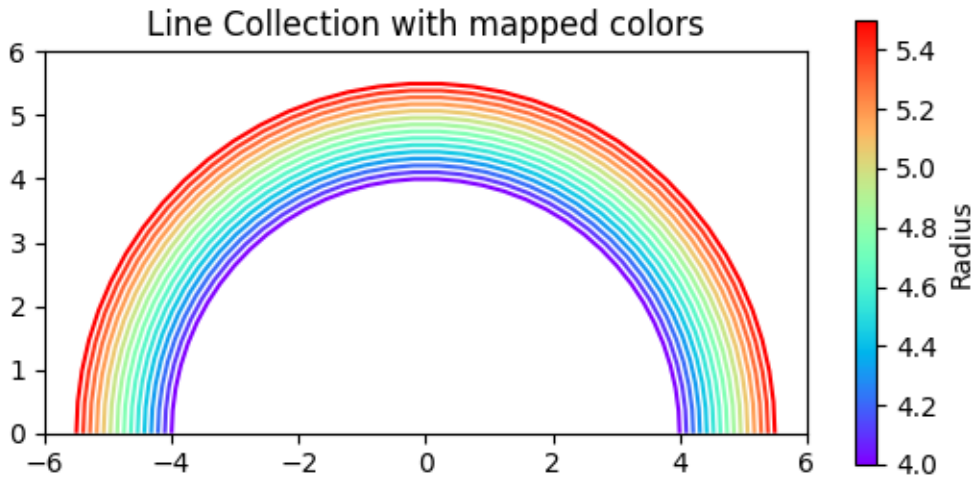
Matplotlib can efficiently draw multiple lines at once using a *LineCollection*.

```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.collections import LineCollection

colors = ["indigo", "blue", "green", "yellow", "orange", "red"]

# create a list of half-circles with varying radii
theta = np.linspace(0, np.pi, 36)
radii = np.linspace(4, 5, num=len(colors))
arcs = [np.column_stack([r * np.cos(theta), r * np.sin(theta)]) for r in␣
 ↪radii]

fig, ax = plt.subplots(figsize=(6.4, 3.2))
# set axes limits manually because Collections do not take part in autoscaling
ax.set_xlim(-6, 6)
ax.set_ylim(0, 6)
ax.set_aspect("equal")  # to make the arcs look circular

# create a LineCollection with the half-circles
# its properties can be set per line by passing a sequence (here used for␣
 ↪*colors*)
# or they can be set for all lines by passing a scalar (here used for␣
 ↪*linewidths*)
line_collection = LineCollection(arcs, colors=colors, linewidths=4)
ax.add_collection(line_collection)

plt.show()
```




Instead of passing a list of colors (`colors=colors`), we can alternatively use colormapping. The lines are then color-coded based on an additional array of values passed to the *array* parameter. In the below example, we color the lines based on their radius by passing `array=radii`.

```
num_arcs = 15
theta = np.linspace(0, np.pi, 36)
radii = np.linspace(4, 5.5, num=num_arcs)
arcs = [np.column_stack([r * np.cos(theta), r * np.sin(theta)]) for r in␣
 ↪radii]

fig, ax = plt.subplots(figsize=(6.4, 3))
# set axes limits manually because Collections do not take part in autoscaling
ax.set_xlim(-6, 6)
ax.set_ylim(0, 6)
ax.set_aspect("equal")  # to make the arcs look circular

# create a LineCollection with the half-circles and color mapping
line_collection = LineCollection(arcs, array=radii, cmap="rainbow")
ax.add_collection(line_collection)

fig.colorbar(line_collection, label="Radius")
ax.set_title("Line Collection with mapped colors")

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.collections.LineCollection*
- *matplotlib.collections.Collection.set_array*
- *matplotlib.axes.Axes.add_collection*
- *matplotlib.figure.Figure.colorbar* / *matplotlib.pyplot.colorbar*

## Circles, Wedges and Polygons

This example demonstrates how to use *collections.PatchCollection*.

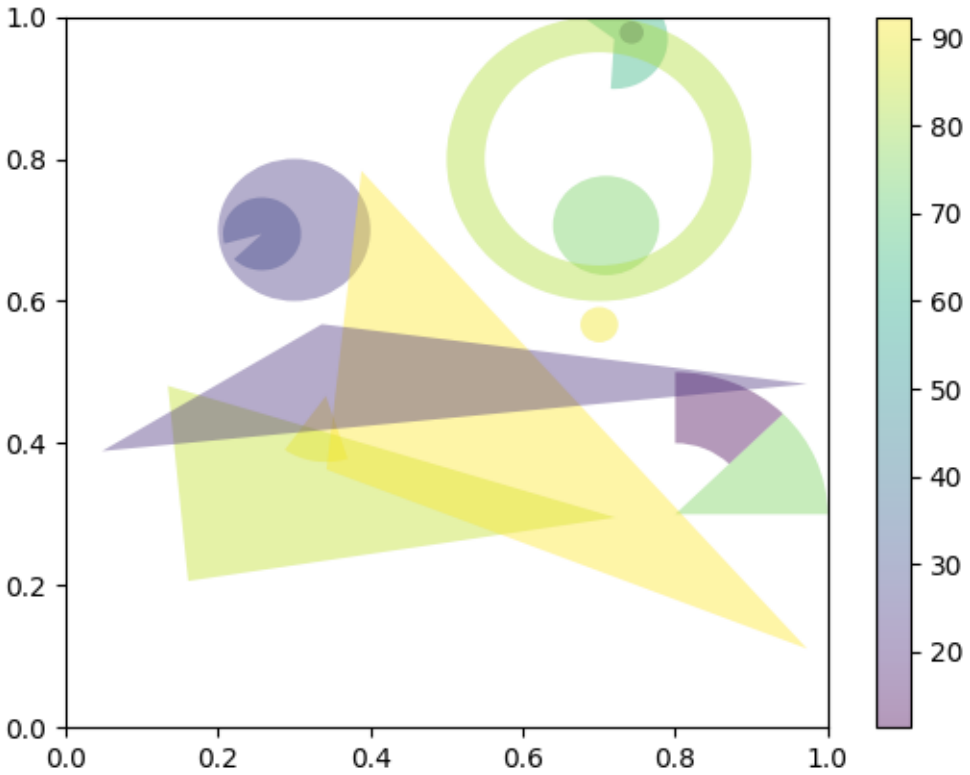
See also *Reference for Matplotlib artists*, which instead adds each artist separately to its own Axes.

```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.collections import PatchCollection
from matplotlib.patches import Circle, Polygon, Wedge

# Fixing random state for reproducibility
np.random.seed(19680801)


fig, ax = plt.subplots()

resolution = 50  # the number of vertices
N = 3
```

(continues on next page)

(continued from previous page)

```
x = np.random.rand(N)
y = np.random.rand(N)
radii = 0.1*np.random.rand(N)
patches = []
for x1, y1, r in zip(x, y, radii):
    circle = Circle((x1, y1), r)
    patches.append(circle)

x = np.random.rand(N)
y = np.random.rand(N)
radii = 0.1*np.random.rand(N)
theta1 = 360.0*np.random.rand(N)
theta2 = 360.0*np.random.rand(N)
for x1, y1, r, t1, t2 in zip(x, y, radii, theta1, theta2):
    wedge = Wedge((x1, y1), r, t1, t2)
    patches.append(wedge)

# Some limiting conditions on Wedge
patches += [
    Wedge((.3, .7), .1, 0, 360),             # Full circle
    Wedge((.7, .8), .2, 0, 360, width=0.05),  # Full ring
    Wedge((.8, .3), .2, 0, 45),              # Full sector
    Wedge((.8, .3), .2, 45, 90, width=0.10),  # Ring sector
]

for i in range(N):
    polygon = Polygon(np.random.rand(N, 2), closed=True)
    patches.append(polygon)

colors = 100 * np.random.rand(len(patches))
p = PatchCollection(patches, alpha=0.4)
p.set_array(colors)
ax.add_collection(p)
fig.colorbar(p, ax=ax)

plt.show()
```




**References**

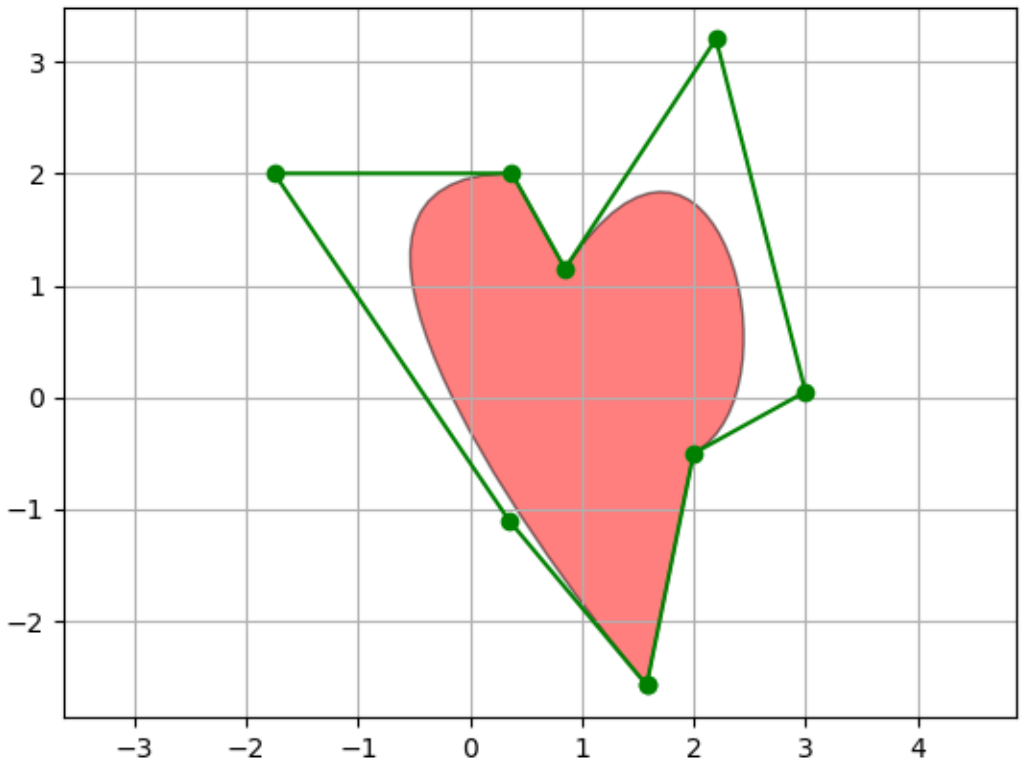
The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.patches*
- *matplotlib.patches.Circle*
- *matplotlib.patches.Wedge*
- *matplotlib.patches.Polygon*
- *matplotlib.collections.PatchCollection*
- *matplotlib.collections.Collection.set_array*
- *matplotlib.axes.Axes.add_collection*
- *matplotlib.figure.Figure.colorbar*

### PathPatch object

This example shows how to create *Path* and *PathPatch* objects through Matplotlib's API.

```
import matplotlib.pyplot as plt

import matplotlib.patches as mpatches
import matplotlib.path as mpath

fig, ax = plt.subplots()

Path = mpath.Path
path_data = [
    (Path.MOVETO, (1.58, -2.57)),
    (Path.CURVE4, (0.35, -1.1)),
    (Path.CURVE4, (-1.75, 2.0)),
    (Path.CURVE4, (0.375, 2.0)),
    (Path.LINETO, (0.85, 1.15)),
    (Path.CURVE4, (2.2, 3.2)),
    (Path.CURVE4, (3, 0.05)),
    (Path.CURVE4, (2.0, -0.5)),
    (Path.CLOSEPOLY, (1.58, -2.57)),
    ]
codes, verts = zip(*path_data)
path = mpath.Path(verts, codes)
patch = mpatches.PathPatch(path, facecolor='r', alpha=0.5)
ax.add_patch(patch)

# plot control points and connecting lines
x, y = zip(*path.vertices)
line, = ax.plot(x, y, 'go-')

ax.grid()
ax.axis('equal')
plt.show()
```




**References**

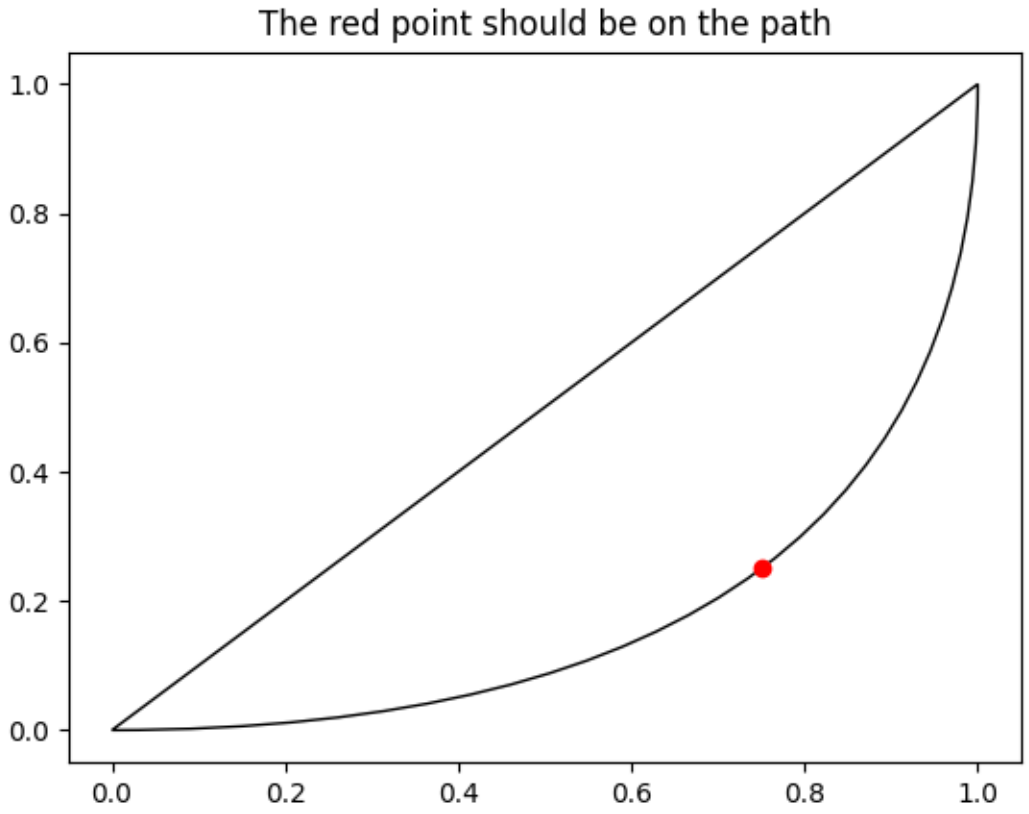
The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.path*
- *matplotlib.path.Path*
- *matplotlib.patches*
- *matplotlib.patches.PathPatch*
- *matplotlib.axes.Axes.add_patch*

### Bezier Curve

This example showcases the *PathPatch* object to create a Bezier polycurve path patch.

```
import matplotlib.pyplot as plt

import matplotlib.patches as mpatches
import matplotlib.path as mpath

Path = mpath.Path

fig, ax = plt.subplots()
pp1 = mpatches.PathPatch(
    Path([(0, 0), (1, 0), (1, 1), (0, 0)],
         [Path.MOVETO, Path.CURVE3, Path.CURVE3, Path.CLOSEPOLY]),
    fc="none", transform=ax.transData)

ax.add_patch(pp1)
ax.plot([0.75], [0.25], "ro")
ax.set_title('The red point should be on the path')

plt.show()
```



**References**

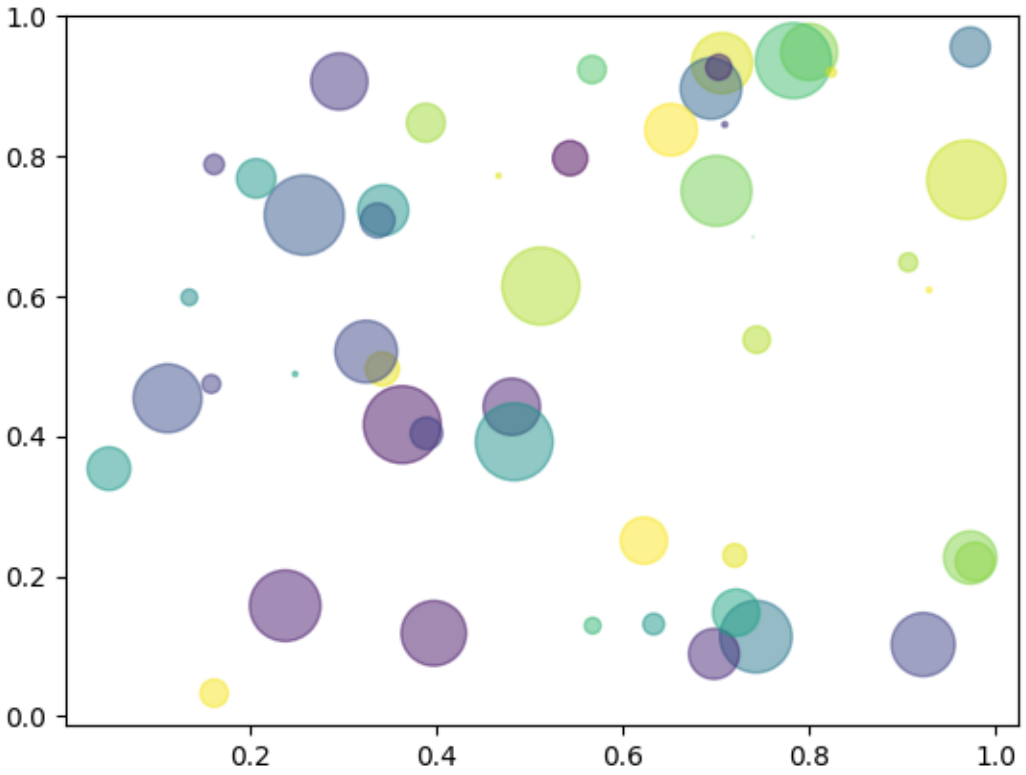
The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.path*
- *matplotlib.path.Path*
- *matplotlib.patches*
- *matplotlib.patches.PathPatch*
- *matplotlib.axes.Axes.add_patch*

## Scatter plot

This example showcases a simple scatter plot.

```
import matplotlib.pyplot as plt
import numpy as np

# Fixing random state for reproducibility
np.random.seed(19680801)


N = 50
x = np.random.rand(N)
y = np.random.rand(N)
colors = np.random.rand(N)
area = (30 * np.random.rand(N))**2  # 0 to 15 point radii

plt.scatter(x, y, s=area, c=colors, alpha=0.5)
plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

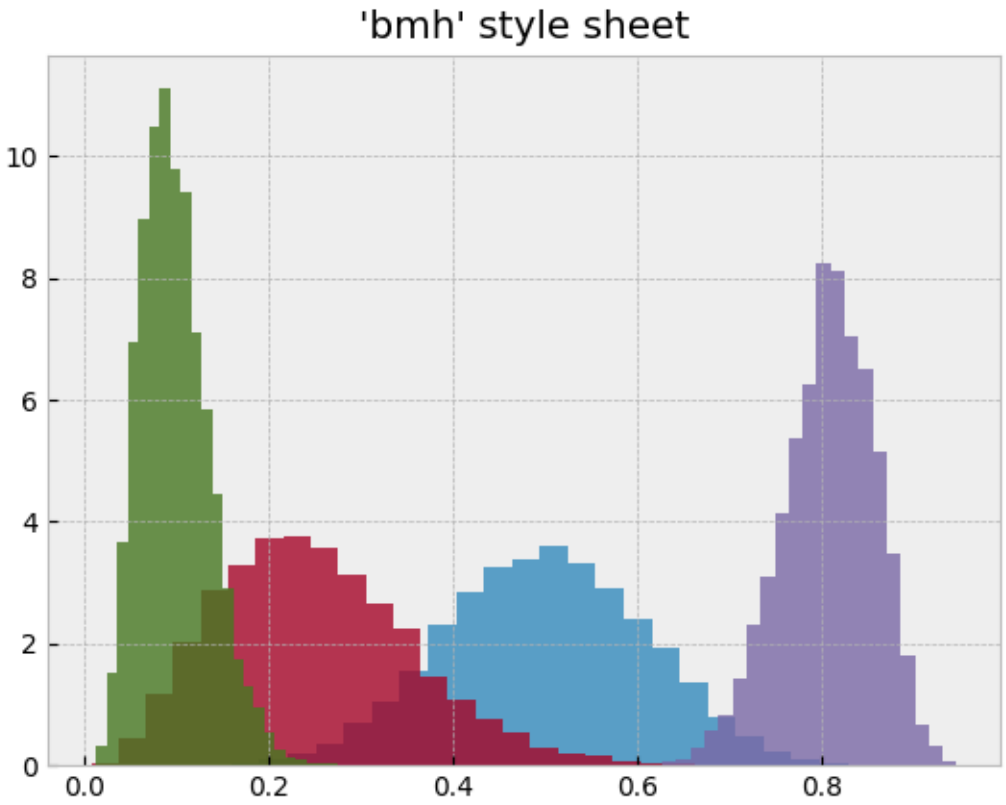
- *matplotlib.axes.Axes.scatter* / *matplotlib.pyplot.scatter*

### 6.25.9 Style sheets

#### Bayesian Methods for Hackers style sheet

This example demonstrates the style used in the Bayesian Methods for Hackers[1] online book.

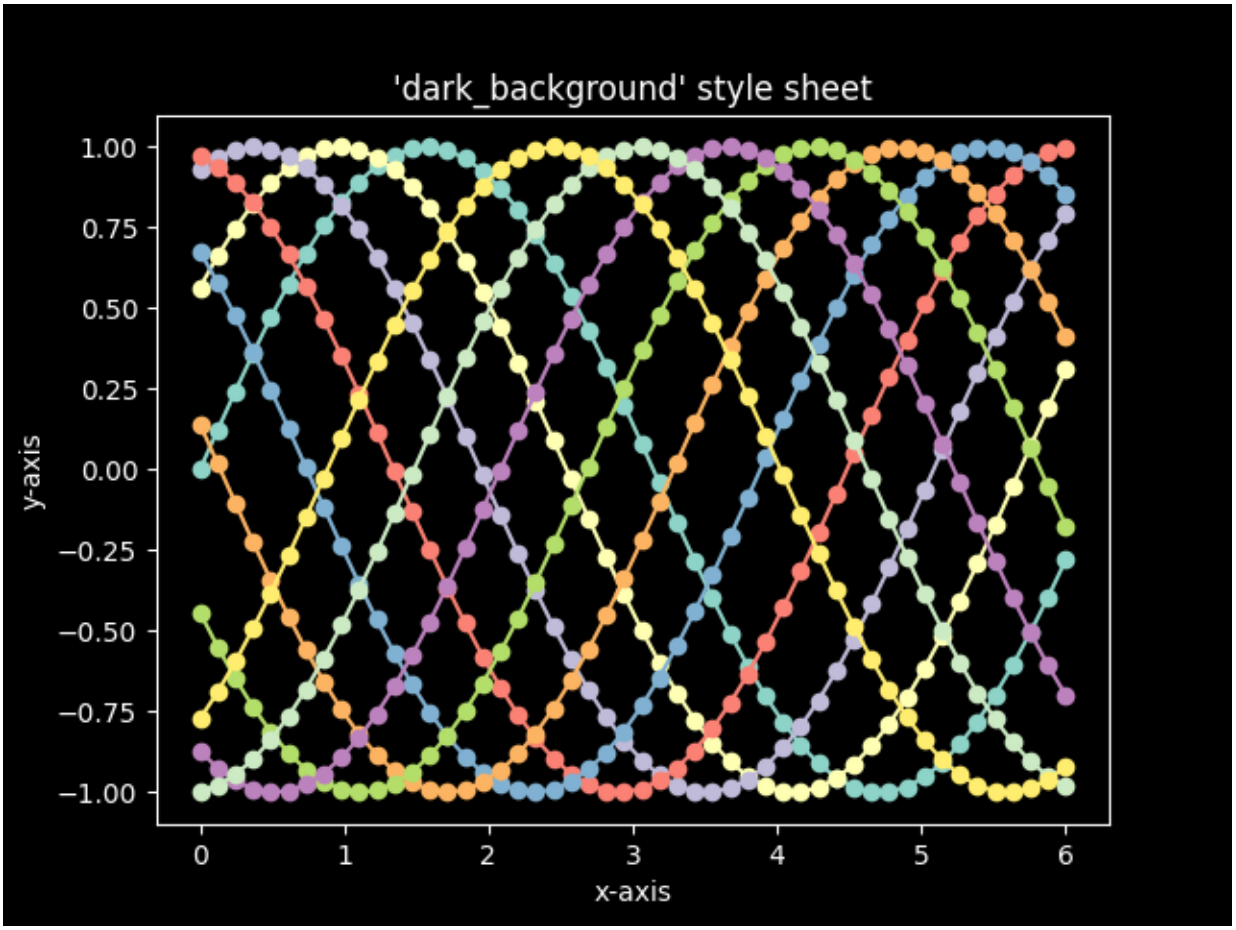
[1] http://camdavidsonpilon.github.io/Probabilistic-Programming-and-Bayesian-Methods-for-Hackers/




```
import matplotlib.pyplot as plt
import numpy as np

# Fixing random state for reproducibility
np.random.seed(19680801)

plt.style.use('bmh')


def plot_beta_hist(ax, a, b):
    ax.hist(np.random.beta(a, b, size=10000),
            histtype="stepfilled", bins=25, alpha=0.8, density=True)


fig, ax = plt.subplots()
plot_beta_hist(ax, 10, 10)
plot_beta_hist(ax, 4, 12)
plot_beta_hist(ax, 50, 12)
plot_beta_hist(ax, 6, 55)
ax.set_title("'bmh' style sheet")

plt.show()
```

### Dark background style sheet

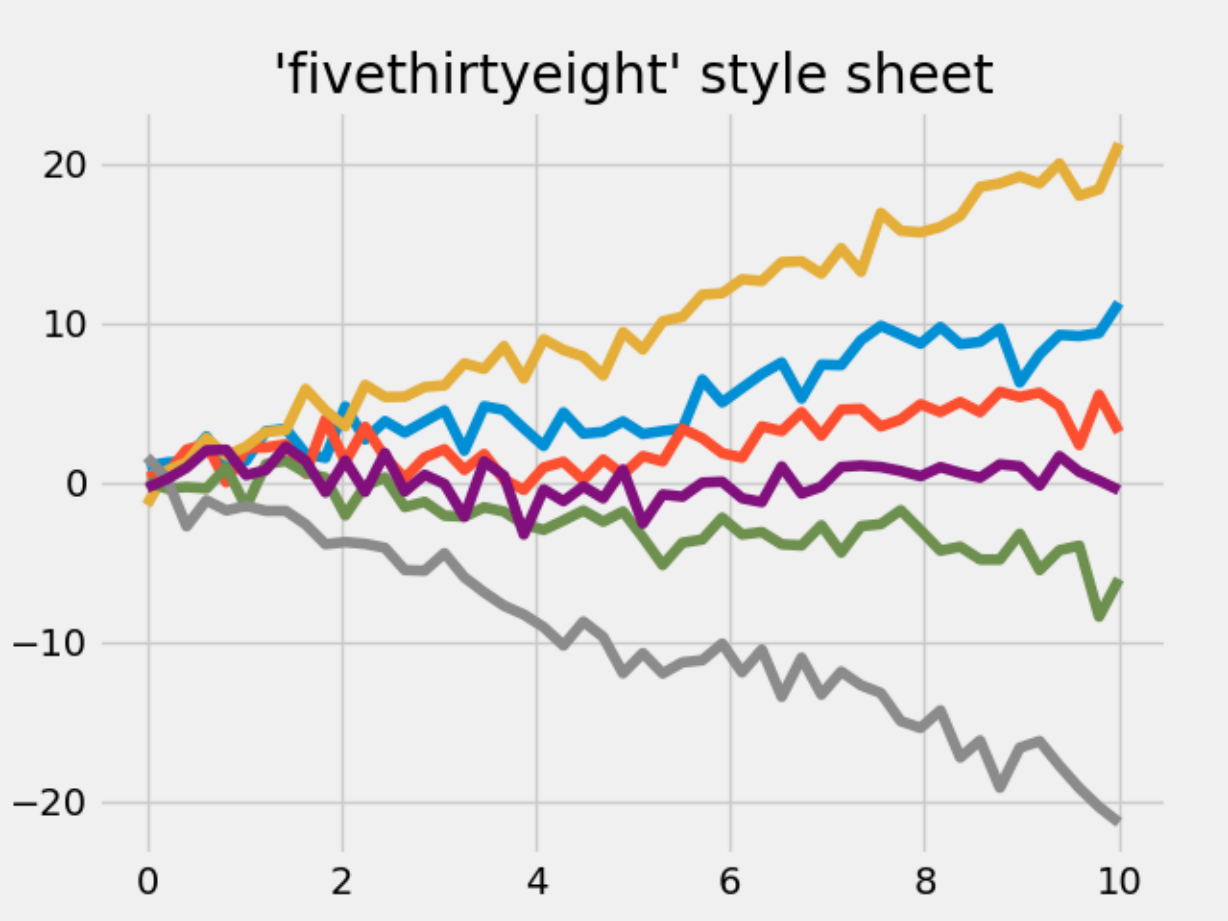
This example demonstrates the "dark_background" style, which uses white for elements that are typically black (text, borders, etc). Note that not all plot elements default to colors defined by an rc parameter.




```
import matplotlib.pyplot as plt
import numpy as np

plt.style.use('dark_background')

fig, ax = plt.subplots()

L = 6
x = np.linspace(0, L)
ncolors = len(plt.rcParams['axes.prop_cycle'])
shift = np.linspace(0, L, ncolors, endpoint=False)
for s in shift:
    ax.plot(x, np.sin(x + s), 'o-')
ax.set_xlabel('x-axis')
ax.set_ylabel('y-axis')
ax.set_title("'dark_background' style sheet")

plt.show()
```

### FiveThirtyEight style sheet

This shows an example of the "fivethirtyeight" styling, which tries to replicate the styles from FiveThirtyEight.com.




```
import matplotlib.pyplot as plt
import numpy as np

plt.style.use('fivethirtyeight')

x = np.linspace(0, 10)

# Fixing random state for reproducibility
np.random.seed(19680801)

fig, ax = plt.subplots()

ax.plot(x, np.sin(x) + x + np.random.randn(50))
ax.plot(x, np.sin(x) + 0.5 * x + np.random.randn(50))
ax.plot(x, np.sin(x) + 2 * x + np.random.randn(50))
ax.plot(x, np.sin(x) - 0.5 * x + np.random.randn(50))
ax.plot(x, np.sin(x) - 2 * x + np.random.randn(50))
ax.plot(x, np.sin(x) + np.random.randn(50))
```

(continues on next page)

(continued from previous page)

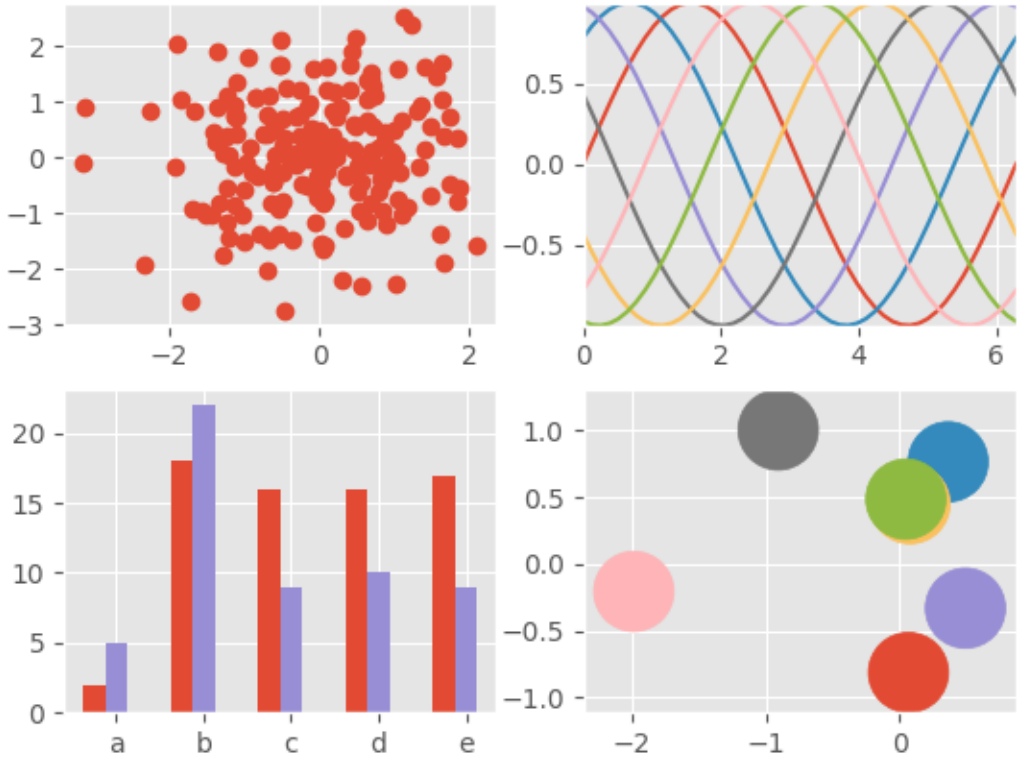
```
ax.set_title("'fivethirtyeight' style sheet")

plt.show()
```

**Total running time of the script:** (0 minutes 1.082 seconds)

### ggplot style sheet

This example demonstrates the "ggplot" style, which adjusts the style to emulate ggplot (a popular plotting package for R).

These settings were shamelessly stolen from[1] (with permission).




```
import matplotlib.pyplot as plt
import numpy as np

plt.style.use('ggplot')

# Fixing random state for reproducibility
```

(continues on next page)

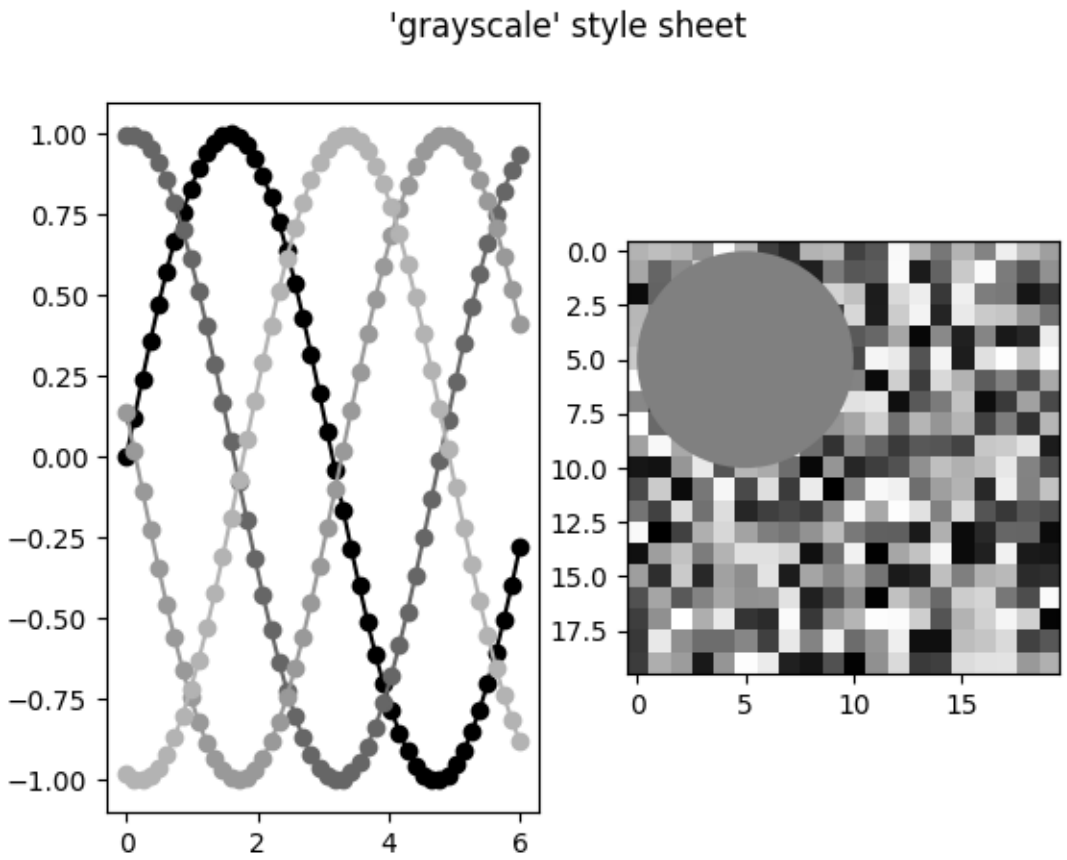
[1] https://everyhue.me/posts/sane-color-scheme-for-matplotlib/

(continued from previous page)

```
np.random.seed(19680801)

fig, axs = plt.subplots(ncols=2, nrows=2)
ax1, ax2, ax3, ax4 = axs.flat

# scatter plot (Note: `plt.scatter` doesn't use default colors)
x, y = np.random.normal(size=(2, 200))
ax1.plot(x, y, 'o')

# sinusoidal lines with colors from default color cycle
L = 2*np.pi
x = np.linspace(0, L)
ncolors = len(plt.rcParams['axes.prop_cycle'])
shift = np.linspace(0, L, ncolors, endpoint=False)
for s in shift:
    ax2.plot(x, np.sin(x + s), '-')
ax2.margins(0)

# bar graphs
x = np.arange(5)
y1, y2 = np.random.randint(1, 25, size=(2, 5))
width = 0.25
ax3.bar(x, y1, width)
ax3.bar(x + width, y2, width,
        color=list(plt.rcParams['axes.prop_cycle'])[2]['color'])
ax3.set_xticks(x + width, labels=['a', 'b', 'c', 'd', 'e'])

# circles with colors from default color cycle
for i, color in enumerate(plt.rcParams['axes.prop_cycle']):
    xy = np.random.normal(size=2)
    ax4.add_patch(plt.Circle(xy, radius=0.3, color=color['color']))
ax4.axis('equal')
ax4.margins(0)

plt.show()
```

### Grayscale style sheet

This example demonstrates the "grayscale" style sheet, which changes all colors that are defined as *rcParams* to grayscale. Note, however, that not all plot elements respect *rcParams*.




```
import matplotlib.pyplot as plt
import numpy as np

# Fixing random state for reproducibility
np.random.seed(19680801)


def color_cycle_example(ax):
    L = 6
    x = np.linspace(0, L)
    ncolors = len(plt.rcParams['axes.prop_cycle'])
    shift = np.linspace(0, L, ncolors, endpoint=False)
    for s in shift:
        ax.plot(x, np.sin(x + s), 'o-')


def image_and_patch_example(ax):
    ax.imshow(np.random.random(size=(20, 20)), interpolation='none')
    c = plt.Circle((5, 5), radius=5, label='patch')
    ax.add_patch(c)


plt.style.use('grayscale')
```

(continues on next page)

(continued from previous page)

```
fig, (ax1, ax2) = plt.subplots(ncols=2)
fig.suptitle("'grayscale' style sheet")

color_cycle_example(ax1)
image_and_patch_example(ax2)

plt.show()
```

### Solarized Light stylesheet

This shows an example of "Solarized_Light" styling, which tries to replicate the styles of:

- https://ethanschoonover.com/solarized/
- https://github.com/jrnold/ggthemes
- http://www.pygal.org/en/stable/documentation/builtin_styles.html#light-solarized

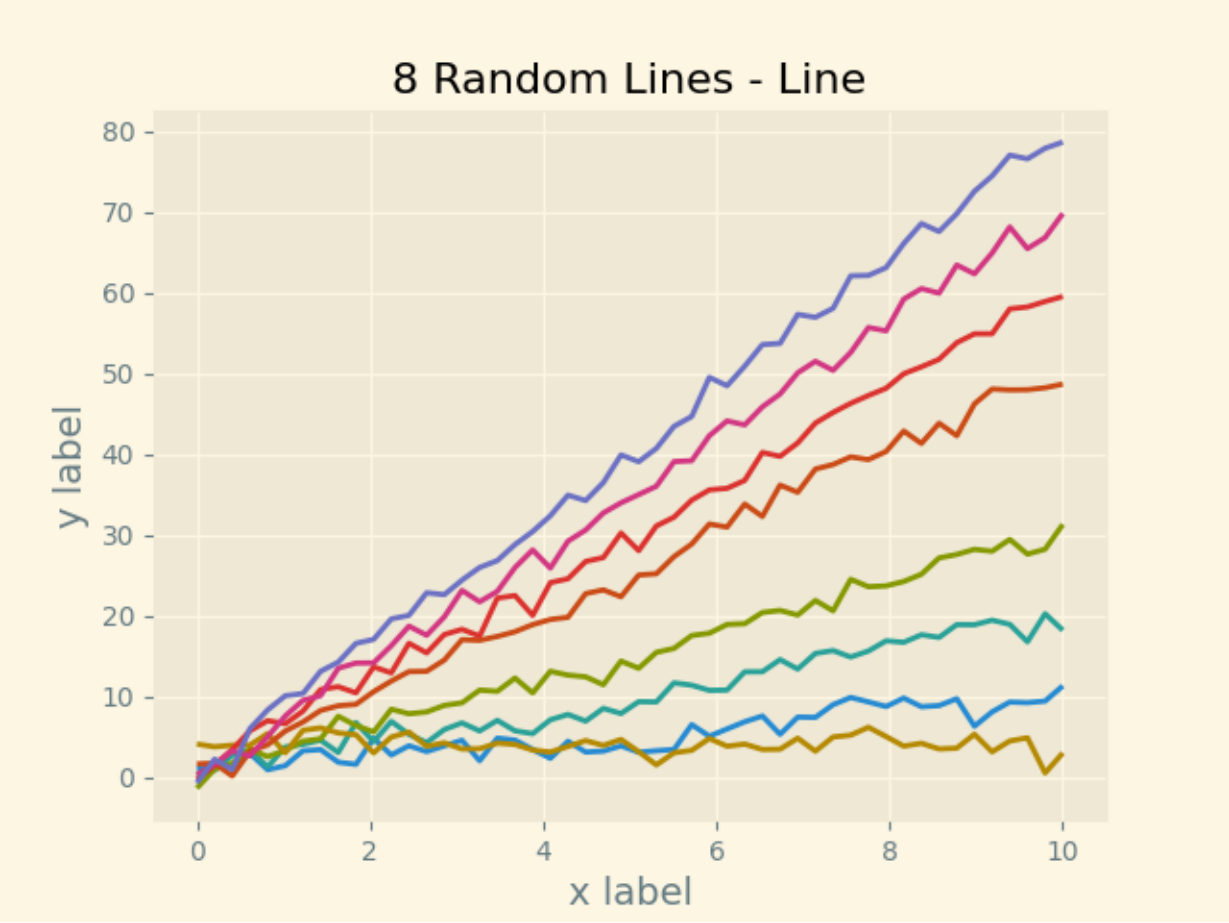
and work of:

- https://github.com/tonysyu/mpltools

using all 8 accents of the color palette - starting with blue

Still TODO:

- Create alpha values for bar and stacked charts. .33 or .5
- Apply Layout Rules




```
import matplotlib.pyplot as plt
import numpy as np

# Fixing random state for reproducibility
np.random.seed(19680801)

x = np.linspace(0, 10)
with plt.style.context('Solarize_Light2'):
    plt.plot(x, np.sin(x) + x + np.random.randn(50))
    plt.plot(x, np.sin(x) + 2 * x + np.random.randn(50))
    plt.plot(x, np.sin(x) + 3 * x + np.random.randn(50))
    plt.plot(x, np.sin(x) + 4 + np.random.randn(50))
    plt.plot(x, np.sin(x) + 5 * x + np.random.randn(50))
    plt.plot(x, np.sin(x) + 6 * x + np.random.randn(50))
    plt.plot(x, np.sin(x) + 7 * x + np.random.randn(50))
    plt.plot(x, np.sin(x) + 8 * x + np.random.randn(50))
    # Number of accent colors in the color scheme
    plt.title('8 Random Lines - Line')
    plt.xlabel('x label', fontsize=14)
    plt.ylabel('y label', fontsize=14)

plt.show()
```

## Style sheets reference

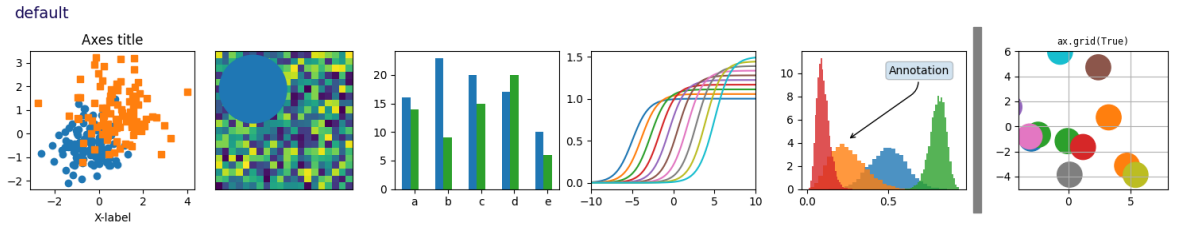
This script demonstrates the different available style sheets on a common set of example plots: scatter plot, image, bar graph, patches, line plot and histogram.

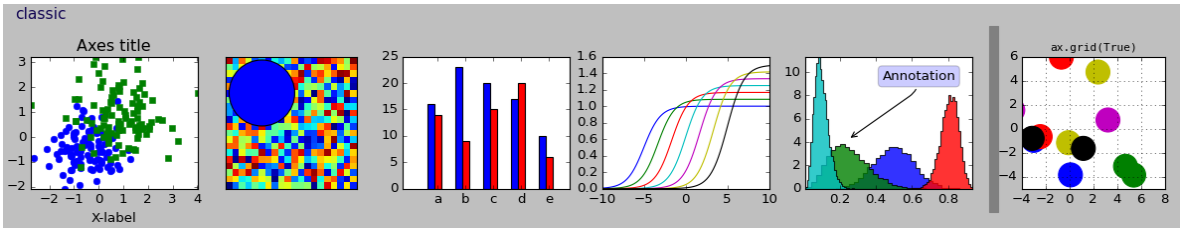
Any of these style sheets can be imported (i.e. activated) by its name. For example for the ggplot style:

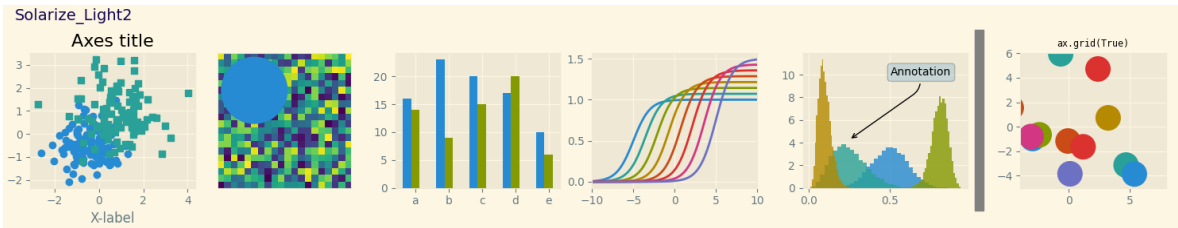
```
>>> plt.style.use('ggplot')
```

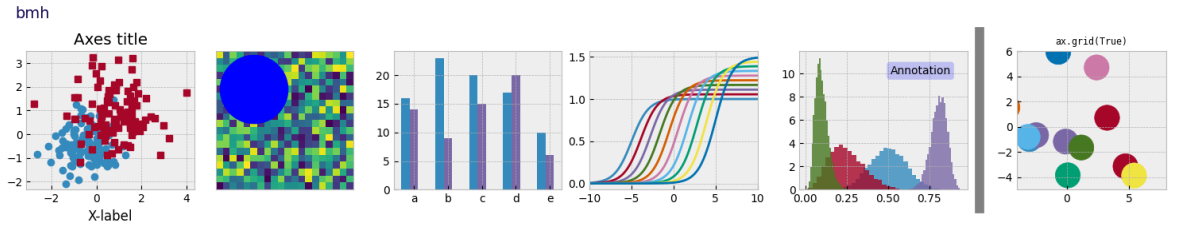
The names of the available style sheets can be found in the list `matplotlib.style.available` (they are also printed in the corner of each plot below).

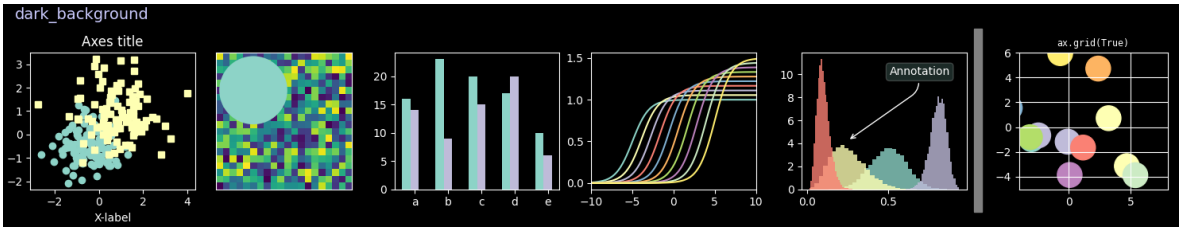
See more details in *Customizing Matplotlib using style sheets*.

- 

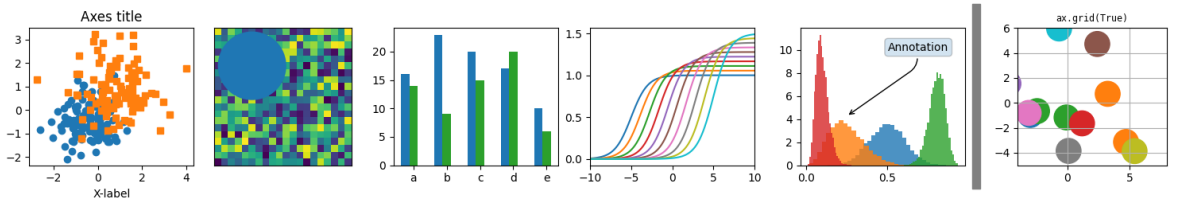
- 

- 

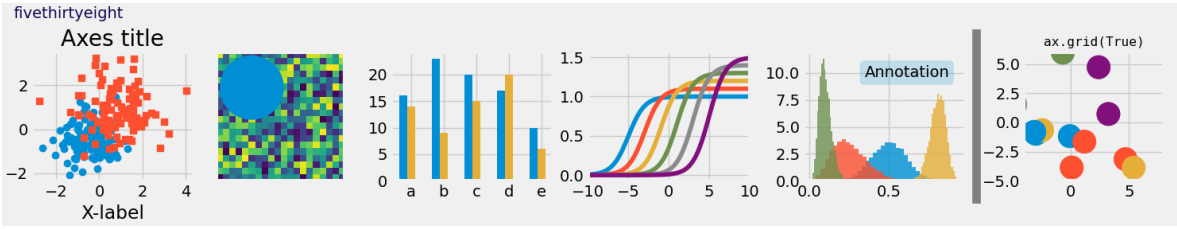
- 

- 

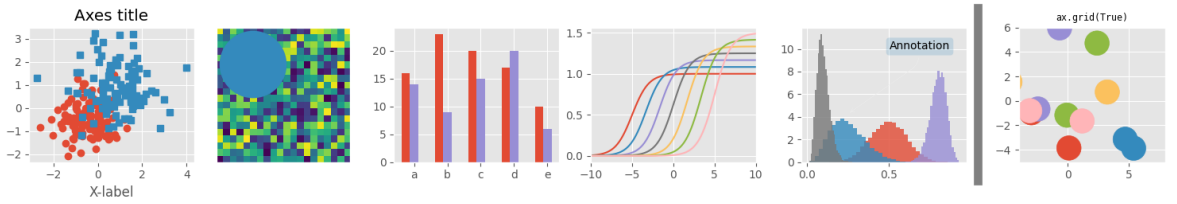
fast



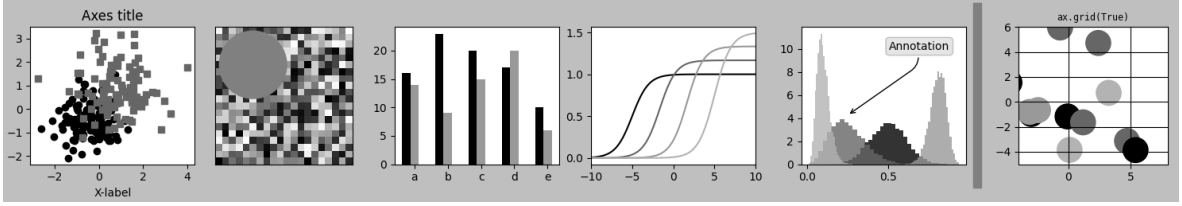

fivethirtyeight



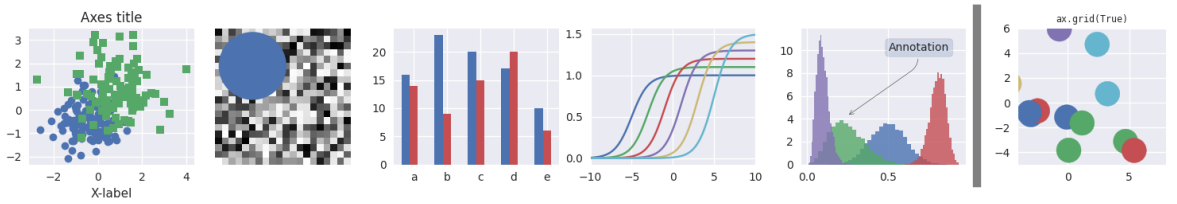

ggplot



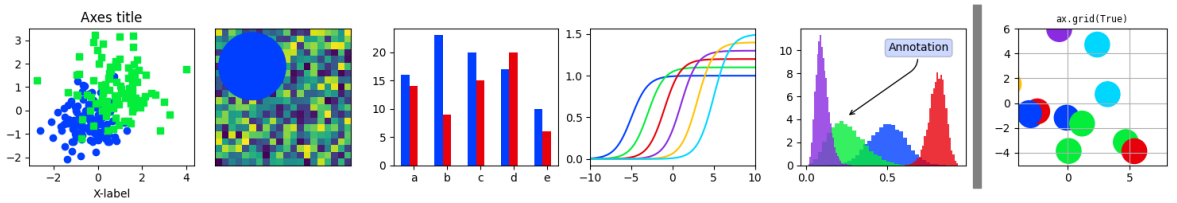

grayscale



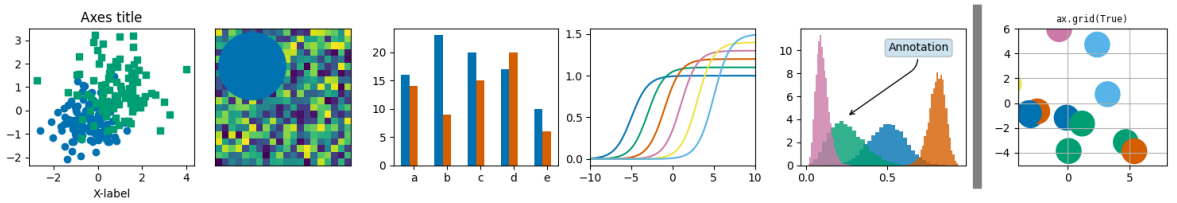

seaborn-v0_8



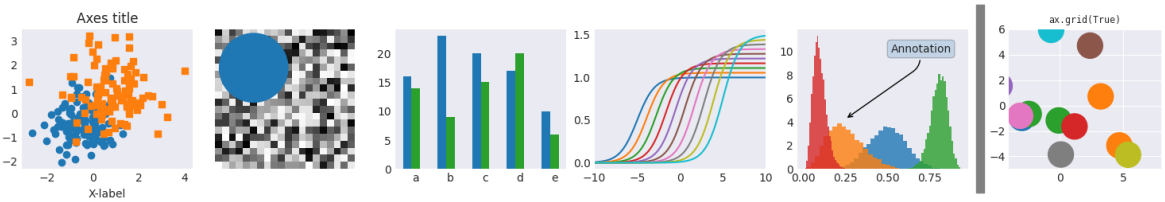

seaborn-v0_8-bright



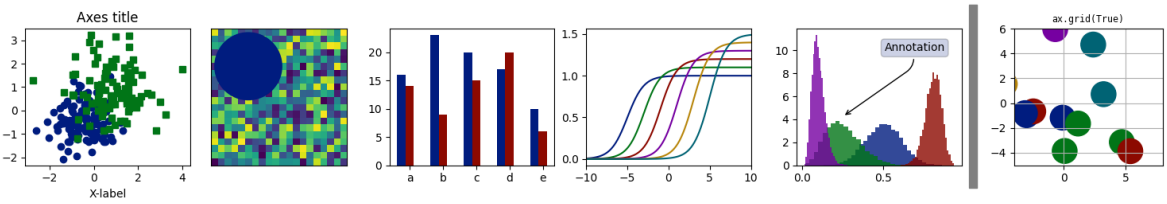

seaborn-v0_8-colorblind



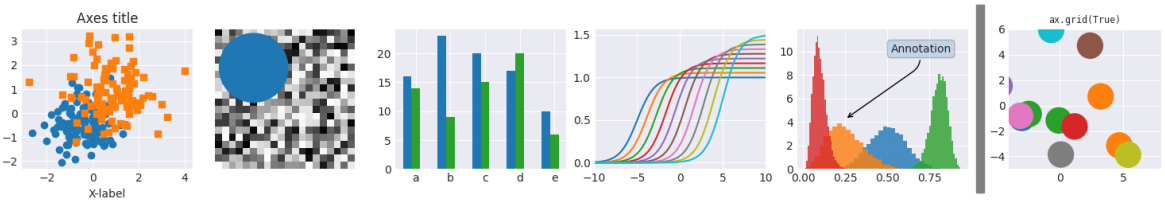
- seaborn-v0_8-dark



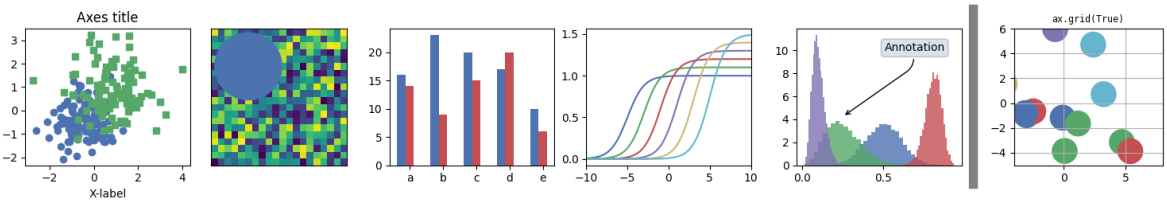

- seaborn-v0_8-dark-palette



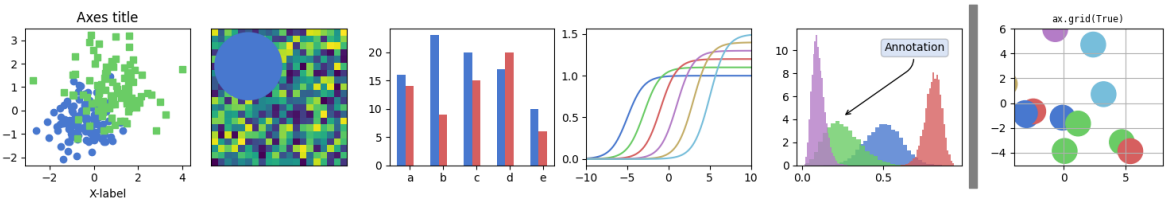

- seaborn-v0_8-darkgrid



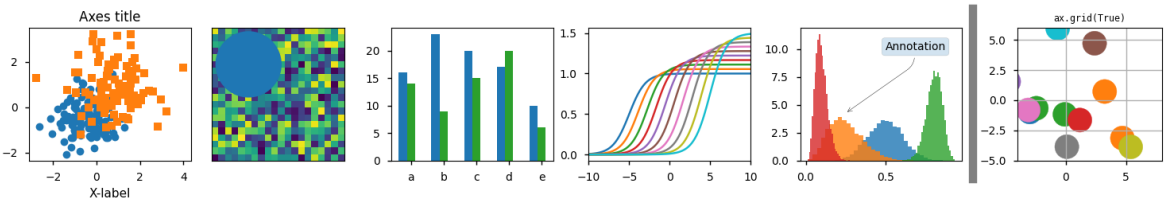

- seaborn-v0_8-deep



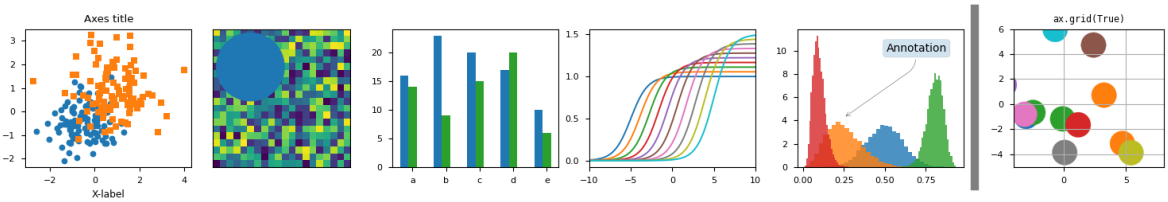

- seaborn-v0_8-muted



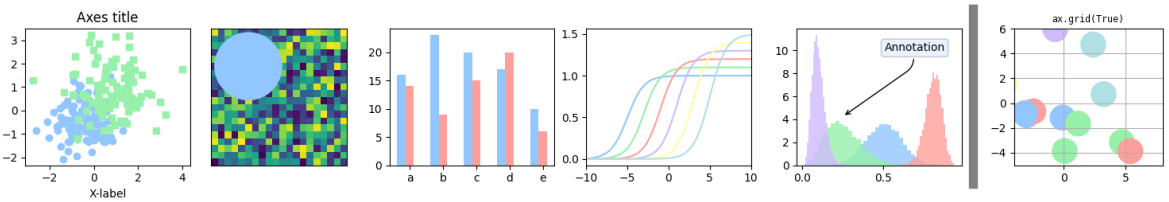

- seaborn-v0_8-notebook



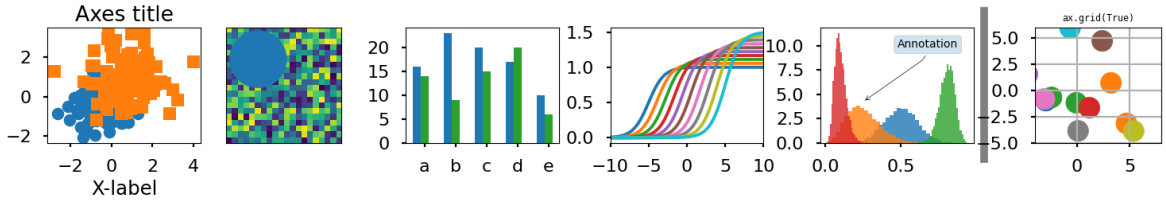

- seaborn-v0_8-paper



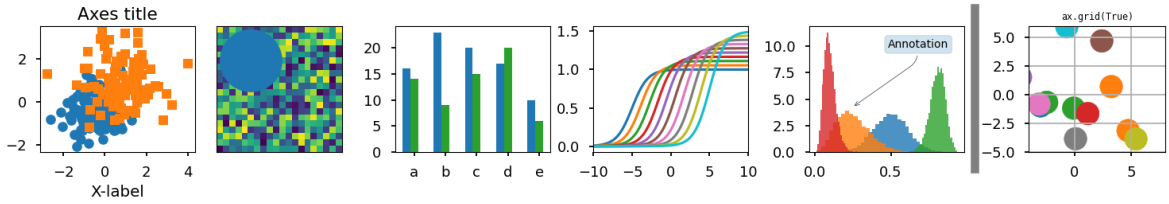
- seaborn-v0_8-pastel



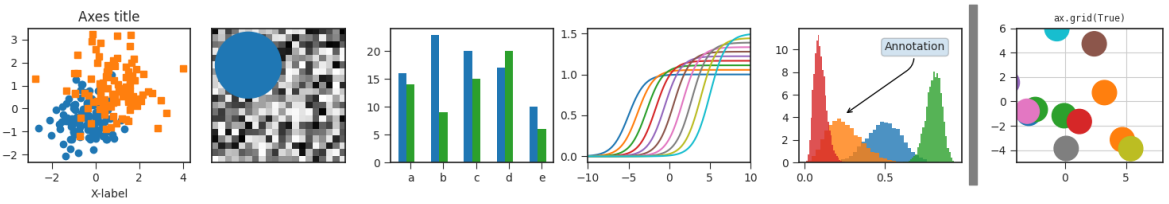

- seaborn-v0_8-poster



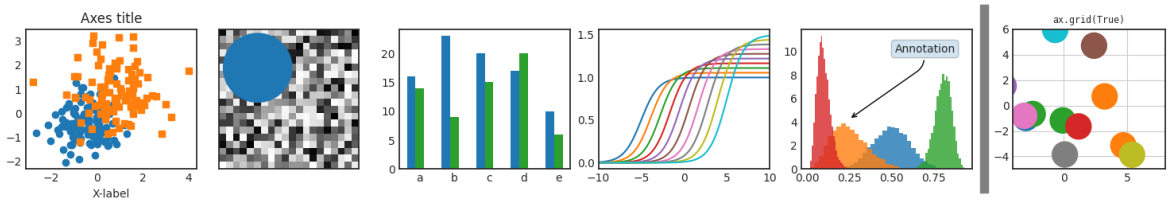

- seaborn-v0_8-talk



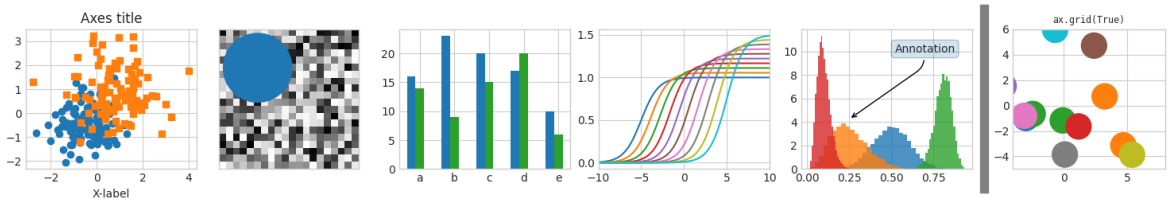

- seaborn-v0_8-ticks



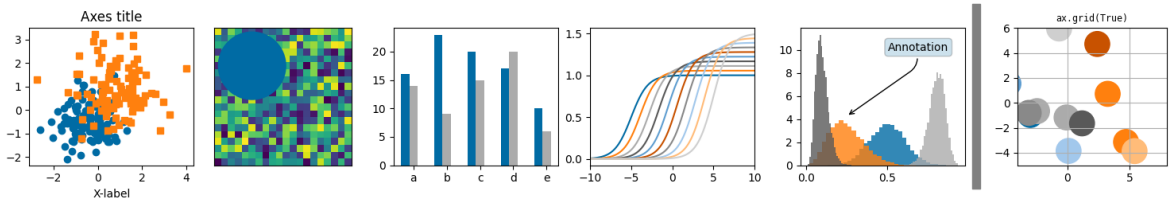

- seaborn-v0_8-white




- seaborn-v0_8-whitegrid




- tableau-colorblind10



```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib.colors as mcolors
from matplotlib.patches import Rectangle

# Fixing random state for reproducibility
np.random.seed(19680801)


def plot_scatter(ax, prng, nb_samples=100):
    """Scatter plot."""
    for mu, sigma, marker in [(-.5, 0.75, 'o'), (0.75, 1., 's')]:
        x, y = prng.normal(loc=mu, scale=sigma, size=(2, nb_samples))
        ax.plot(x, y, ls='none', marker=marker)
    ax.set_xlabel('X-label')
    ax.set_title('Axes title')
    return ax


def plot_colored_lines(ax):
    """Plot lines with colors following the style color cycle."""
    t = np.linspace(-10, 10, 100)

    def sigmoid(t, t0):
        return 1 / (1 + np.exp(-(t - t0)))

    nb_colors = len(plt.rcParams['axes.prop_cycle'])
    shifts = np.linspace(-5, 5, nb_colors)
    amplitudes = np.linspace(1, 1.5, nb_colors)
    for t0, a in zip(shifts, amplitudes):
        ax.plot(t, a * sigmoid(t, t0), '-')
    ax.set_xlim(-10, 10)
    return ax


def plot_bar_graphs(ax, prng, min_value=5, max_value=25, nb_samples=5):
    """Plot two bar graphs side by side, with letters as x-tick labels."""
    x = np.arange(nb_samples)
    ya, yb = prng.randint(min_value, max_value, size=(2, nb_samples))
    width = 0.25
    ax.bar(x, ya, width)
    ax.bar(x + width, yb, width, color='C2')
    ax.set_xticks(x + width, labels=['a', 'b', 'c', 'd', 'e'])
    return ax


def plot_colored_circles(ax, prng, nb_samples=15):
    """
    Plot circle patches.

    NB: draws a fixed amount of samples, rather than using the length of
```

(continues on next page)

(continued from previous page)

```
    the color cycle, because different styles may have different numbers
    of colors.
    """
    for sty_dict, j in zip(plt.rcParams['axes.prop_cycle'](),
                           range(nb_samples)):
        ax.add_patch(plt.Circle(prng.normal(scale=3, size=2),
                                radius=1.0, color=sty_dict['color']))
    ax.grid(visible=True)

    # Add title for enabling grid
    plt.title('ax.grid(True)', family='monospace', fontsize='small')

    ax.set_xlim([-4, 8])
    ax.set_ylim([-5, 6])
    ax.set_aspect('equal', adjustable='box')  # to plot circles as circles
    return ax


def plot_image_and_patch(ax, prng, size=(20, 20)):
    """Plot an image with random values and superimpose a circular patch."""
    values = prng.random_sample(size=size)
    ax.imshow(values, interpolation='none')
    c = plt.Circle((5, 5), radius=5, label='patch')
    ax.add_patch(c)
    # Remove ticks
    ax.set_xticks([])
    ax.set_yticks([])


def plot_histograms(ax, prng, nb_samples=10000):
    """Plot 4 histograms and a text annotation."""
    params = ((10, 10), (4, 12), (50, 12), (6, 55))
    for a, b in params:
        values = prng.beta(a, b, size=nb_samples)
        ax.hist(values, histtype="stepfilled", bins=30,
                alpha=0.8, density=True)

    # Add a small annotation.
    ax.annotate('Annotation', xy=(0.25, 4.25),
                xytext=(0.9, 0.9), textcoords=ax.transAxes,
                va="top", ha="right",
                bbox=dict(boxstyle="round", alpha=0.2),
                arrowprops=dict(
                          arrowstyle="->",
                          connectionstyle="angle,angleA=-95,angleB=35,rad=10
 ↪"),
                )
    return ax


def plot_figure(style_label=""):
    """Setup and plot the demonstration figure with a given style."""
```

(continues on next page)

(continued from previous page)

```
    # Use a dedicated RandomState instance to draw the same "random" values
    # across the different figures.
    prng = np.random.RandomState(96917002)

    fig, axs = plt.subplots(ncols=6, nrows=1, num=style_label,
                            figsize=(14.8, 2.8), layout='constrained')

    # make a suptitle, in the same style for all subfigures,
    # except those with dark backgrounds, which get a lighter color:
    background_color = mcolors.rgb_to_hsv(
        mcolors.to_rgb(plt.rcParams['figure.facecolor']))[2]
    if background_color < 0.5:
        title_color = [0.8, 0.8, 1]
    else:
        title_color = np.array([19, 6, 84]) / 256
    fig.suptitle(style_label, x=0.01, ha='left', color=title_color,
                 fontsize=14, fontfamily='DejaVu Sans', fontweight='normal')

    plot_scatter(axs[0], prng)
    plot_image_and_patch(axs[1], prng)
    plot_bar_graphs(axs[2], prng)
    plot_colored_lines(axs[3])
    plot_histograms(axs[4], prng)
    plot_colored_circles(axs[5], prng)

    # add divider
    rec = Rectangle((1 + 0.025, -2), 0.05, 16,
                    clip_on=False, color='gray')

    axs[4].add_artist(rec)

if __name__ == "__main__":

    # Set up a list of all available styles, in alphabetical order but
    # the `default` and `classic` ones, which will be forced resp. in
    # first and second position.
    # styles with leading underscores are for internal use such as testing
    # and plot types gallery. These are excluded here.
    style_list = ['default', 'classic'] + sorted(
        style for style in plt.style.available
        if style != 'classic' and not style.startswith('_'))

    # Plot a demonstration figure for every available style sheet.
    for style_label in style_list:
        with plt.rc_context({"figure.max_open_warning": len(style_list)}):
            with plt.style.context(style_label):
                plot_figure(style_label=style_label)

    plt.show()
```

**Total running time of the script:** (0 minutes 24.265 seconds)

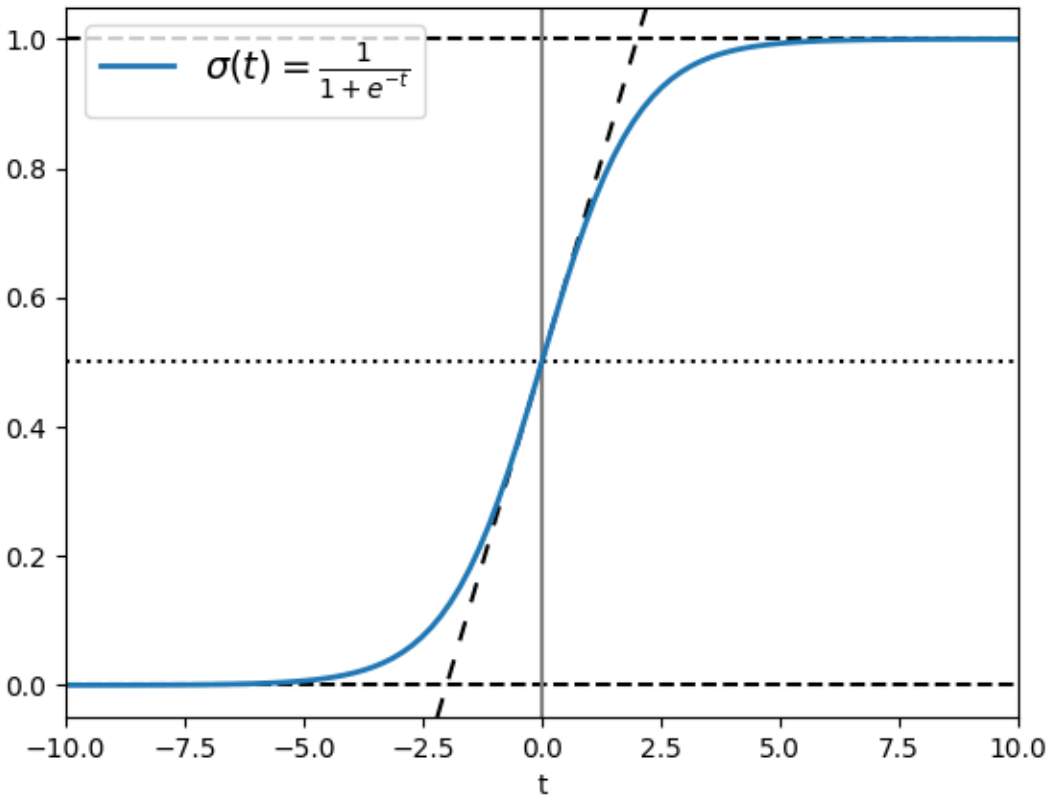
### 6.25.10 Module - pyplot

#### Infinite lines

*axvline* and *axhline* draw infinite vertical / horizontal lines, at given *x* / *y* positions. They are usually used to mark special data values, e.g. in this example the center and limit values of the sigmoid function.

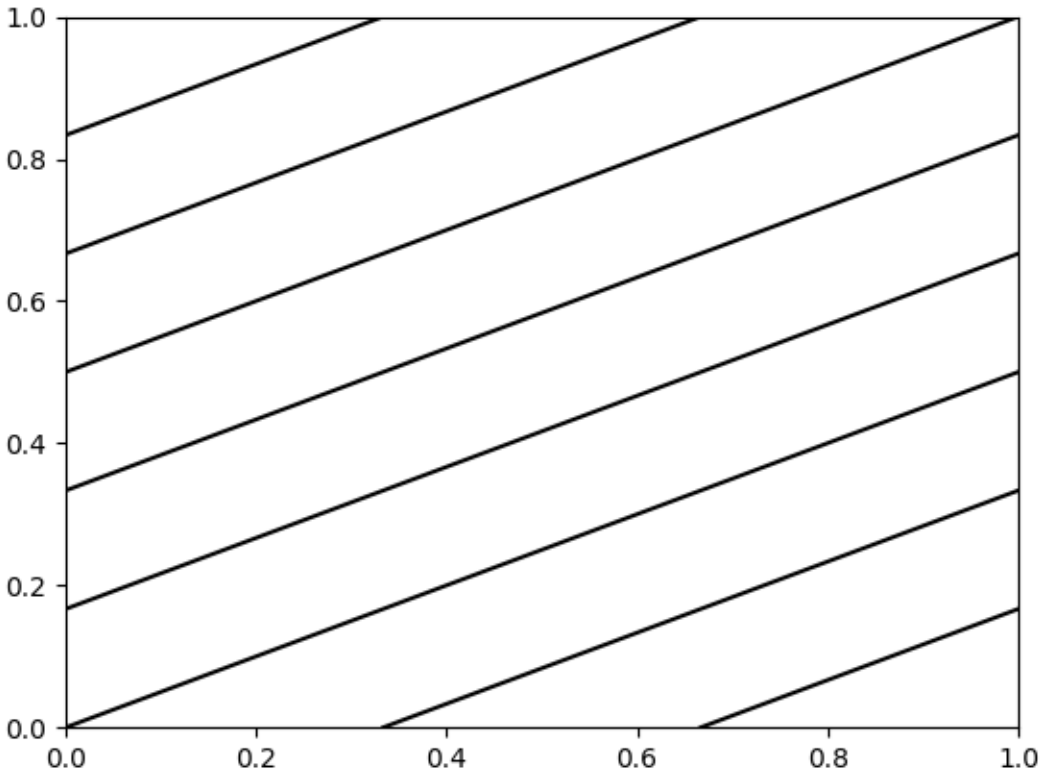
*axline* draws infinite straight lines in arbitrary directions.

```
import matplotlib.pyplot as plt
import numpy as np

t = np.linspace(-10, 10, 100)
sig = 1 / (1 + np.exp(-t))

plt.axhline(y=0, color="black", linestyle="--")
plt.axhline(y=0.5, color="black", linestyle=":")
plt.axhline(y=1.0, color="black", linestyle="--")
plt.axvline(color="grey")
plt.axline((0, 0.5), slope=0.25, color="black", linestyle=(0, (5, 5)))
plt.plot(t, sig, linewidth=2, label=r"$\sigma(t) = \frac{1}{1 + e^{-t}}$")
plt.xlim(-10, 10)
plt.xlabel("t")
plt.legend(fontsize=14)
plt.show()
```




*axline* can also be used with a `transform` parameter, which applies to the point, but not to the slope. This can be useful for drawing diagonal grid lines with a fixed slope, which stay in place when the plot limits are moved.

```
for pos in np.linspace(-2, 1, 10):
    plt.axline((pos, 0), slope=0.5, color='k', transform=plt.gca().transAxes)

plt.ylim([0, 1])
plt.xlim([0, 1])
plt.show()
```




**References**

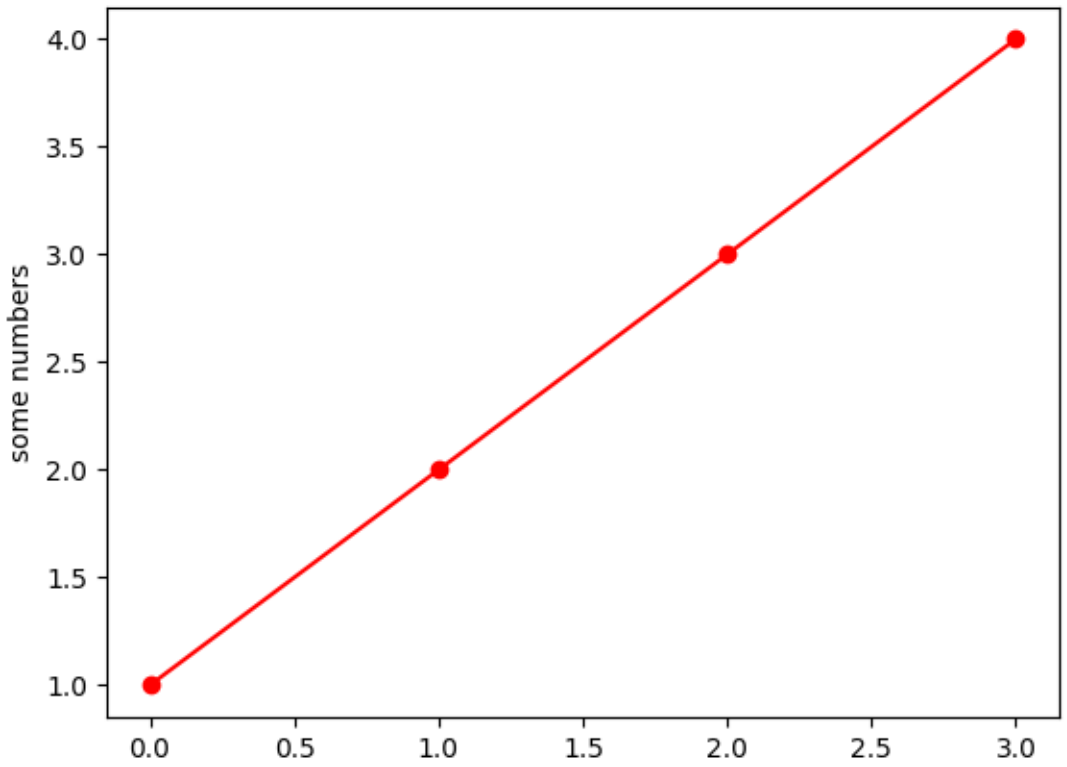
The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.axhline* / *matplotlib.pyplot.axhline*
- *matplotlib.axes.Axes.axvline* / *matplotlib.pyplot.axvline*
- *matplotlib.axes.Axes.axline* / *matplotlib.pyplot.axline*

## Simple plot

A simple plot where a list of numbers are plotted against their index, resulting in a straight line. Use a format string (here, 'o-r') to set the markers (circles), linestyle (solid line) and color (red).

```
import matplotlib.pyplot as plt

plt.plot([1, 2, 3, 4], 'o-r')
plt.ylabel('some numbers')
plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.pyplot.plot*
- *matplotlib.pyplot.ylabel*
- *matplotlib.pyplot.show*

## Text and mathtext using pyplot

Set the special text objects *title*, *xlabel*, and *ylabel* through the dedicated pyplot functions. Additional text objects can be placed in the Axes using *text*.

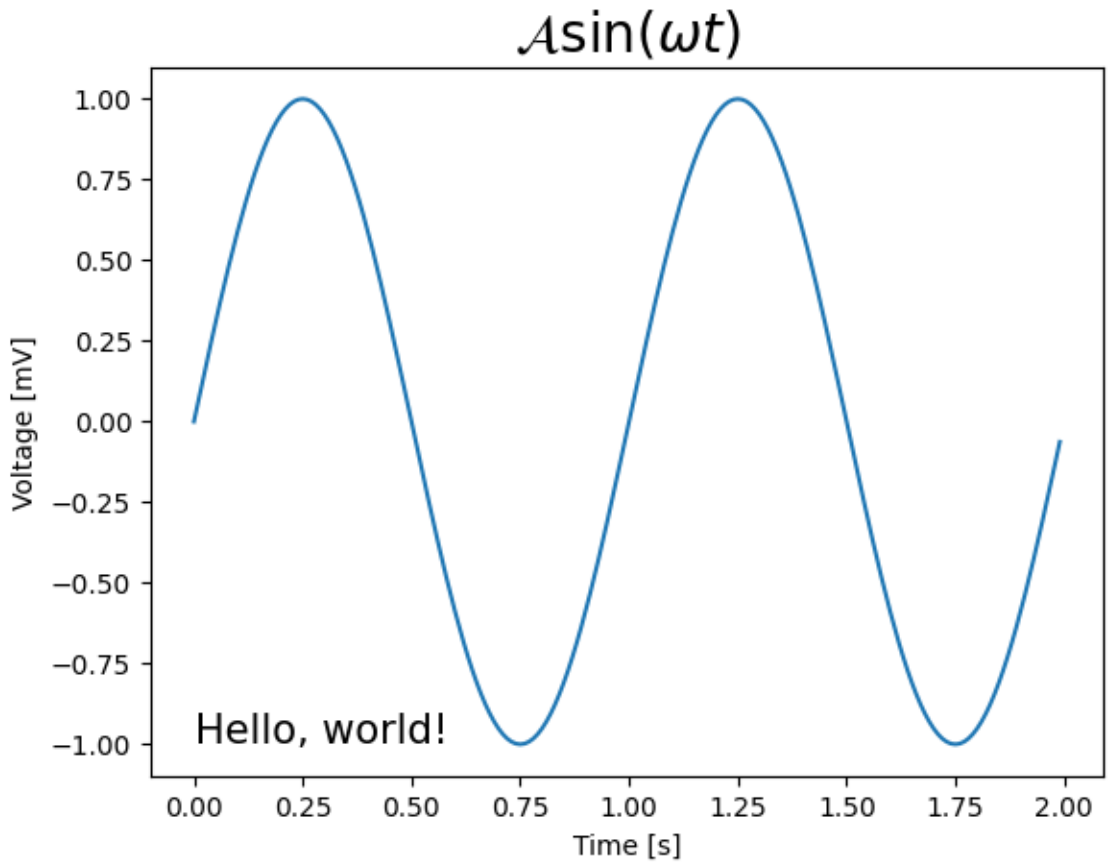
You can use TeX-like mathematical typesetting in all texts; see also *Writing mathematical expressions*.

```
import matplotlib.pyplot as plt
import numpy as np

t = np.arange(0.0, 2.0, 0.01)
```

(continues on next page)

(continued from previous page)

```
s = np.sin(2*np.pi*t)

plt.plot(t, s)
plt.text(0, -1, r'Hello, world!', fontsize=15)
plt.title(r'$\mathcal{A}\sin(\omega t)$', fontsize=20)
plt.xlabel('Time [s]')
plt.ylabel('Voltage [mV]')
plt.show()
```




**References**

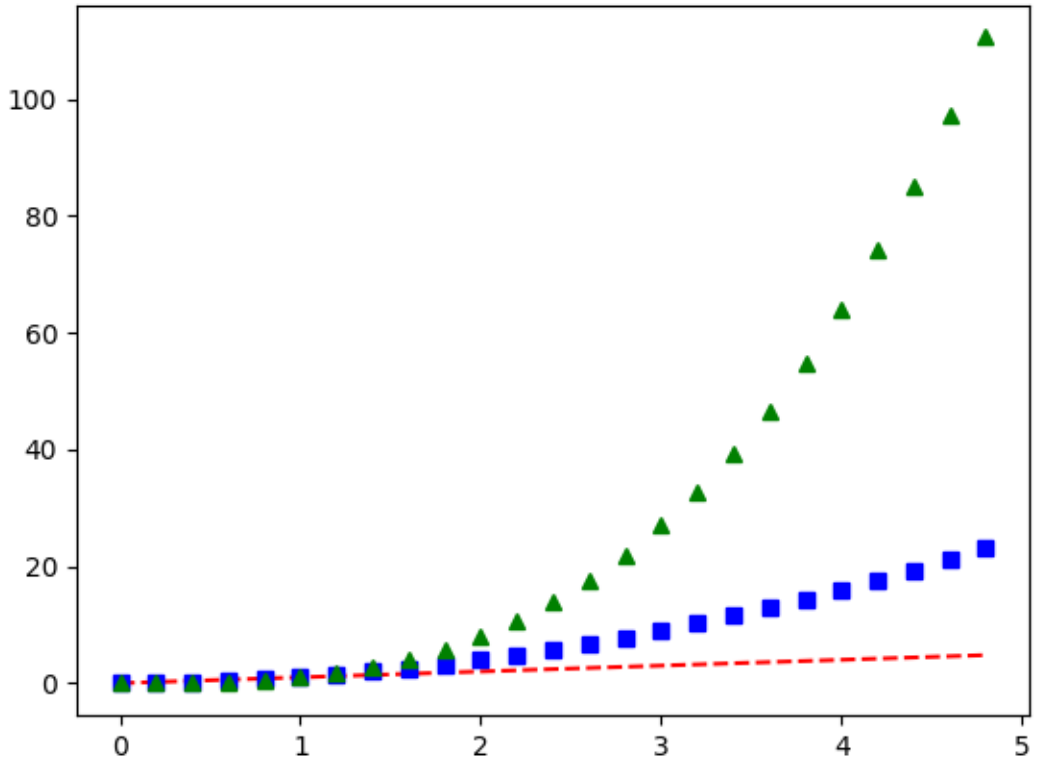
The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.pyplot.hist*
- *matplotlib.pyplot.xlabel*
- *matplotlib.pyplot.ylabel*
- *matplotlib.pyplot.text*
- *matplotlib.pyplot.grid*
- *matplotlib.pyplot.show*

### Multiple lines using pyplot

Plot three datasets with a single call to *plot*.

```
import matplotlib.pyplot as plt
import numpy as np

# evenly sampled time at 200ms intervals
t = np.arange(0., 5., 0.2)

# red dashes, blue squares and green triangles
plt.plot(t, t, 'r--', t, t**2, 'bs', t, t**3, 'g^')
plt.show()
```




**References**

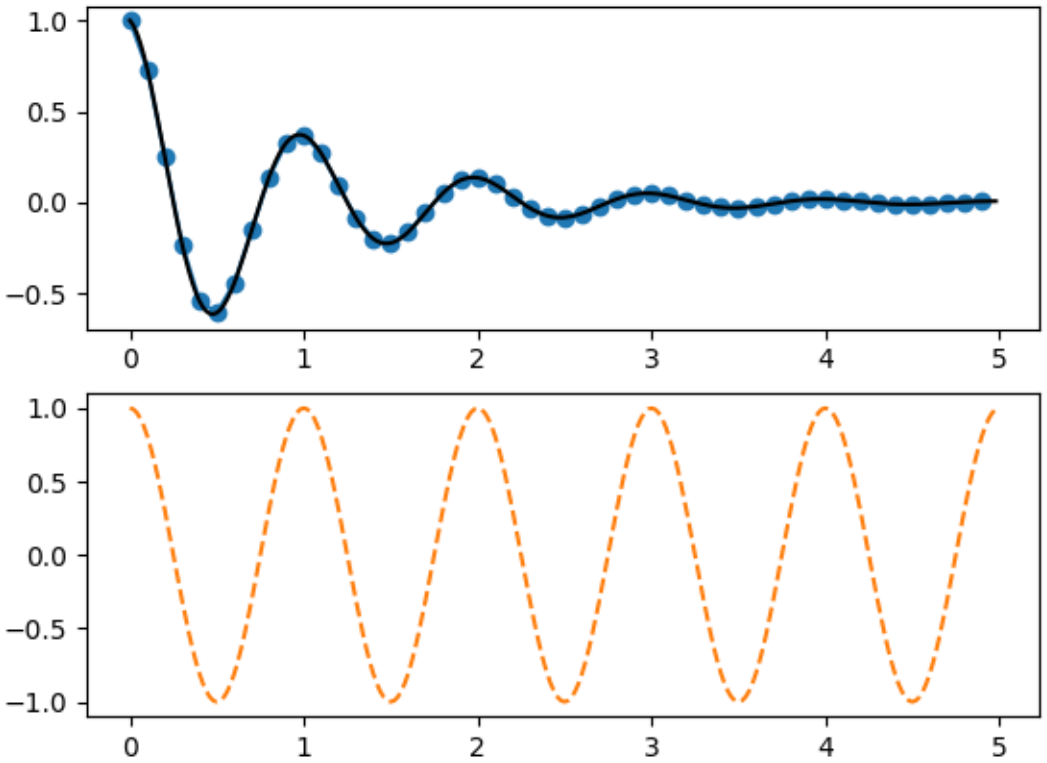
The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.plot* / *matplotlib.pyplot.plot*

### Two subplots using pyplot

Create a figure with two subplots using *pyplot.subplot*.

```
import matplotlib.pyplot as plt
import numpy as np


def f(t):
    return np.exp(-t) * np.cos(2*np.pi*t)


t1 = np.arange(0.0, 5.0, 0.1)
t2 = np.arange(0.0, 5.0, 0.02)

plt.figure()
plt.subplot(211)
plt.plot(t1, f(t1), color='tab:blue', marker='o')
plt.plot(t2, f(t2), color='black')

plt.subplot(212)
plt.plot(t2, np.cos(2*np.pi*t2), color='tab:orange', linestyle='--')
plt.show()
```



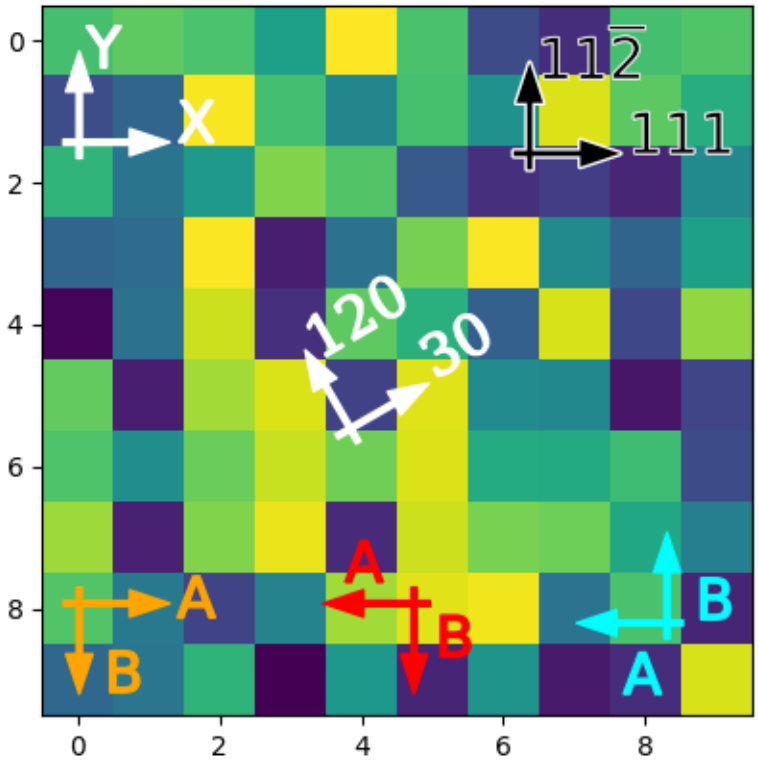
**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.pyplot.figure*
- *matplotlib.pyplot.subplot*

### 6.25.11 Module - axes_grid1

#### Anchored Direction Arrow




```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib.font_manager as fm
from mpl_toolkits.axes_grid1.anchored_artists import AnchoredDirectionArrows

# Fixing random state for reproducibility
np.random.seed(19680801)
```

(continues on next page)

(continued from previous page)

```
fig, ax = plt.subplots()
ax.imshow(np.random.random((10, 10)))

# Simple example
simple_arrow = AnchoredDirectionArrows(ax.transAxes, 'X', 'Y')
ax.add_artist(simple_arrow)

# High contrast arrow
high_contrast_part_1 = AnchoredDirectionArrows(
                           ax.transAxes,
                           '111', r'11$\overline{2}$',
                           loc='upper right',
                           arrow_props={'ec': 'w', 'fc': 'none', 'alpha': 1,
                                        'lw': 2}
                           )
ax.add_artist(high_contrast_part_1)

high_contrast_part_2 = AnchoredDirectionArrows(
                           ax.transAxes,
                           '111', r'11$\overline{2}$',
                           loc='upper right',
                           arrow_props={'ec': 'none', 'fc': 'k'},
                           text_props={'ec': 'w', 'fc': 'k', 'lw': 0.4}
                           )
ax.add_artist(high_contrast_part_2)

# Rotated arrow
fontprops = fm.FontProperties(family='serif')

rotated_arrow = AnchoredDirectionArrows(
                    ax.transAxes,
                    '30', '120',
                    loc='center',
                    color='w',
                    angle=30,
                    fontproperties=fontprops
                    )
ax.add_artist(rotated_arrow)

# Altering arrow directions
a1 = AnchoredDirectionArrows(
        ax.transAxes, 'A', 'B', loc='lower center',
        length=-0.15,
        sep_x=0.03, sep_y=0.03,
        color='r'
    )
ax.add_artist(a1)

a2 = AnchoredDirectionArrows(
        ax.transAxes, 'A', ' B', loc='lower left',
        aspect_ratio=-1,
```

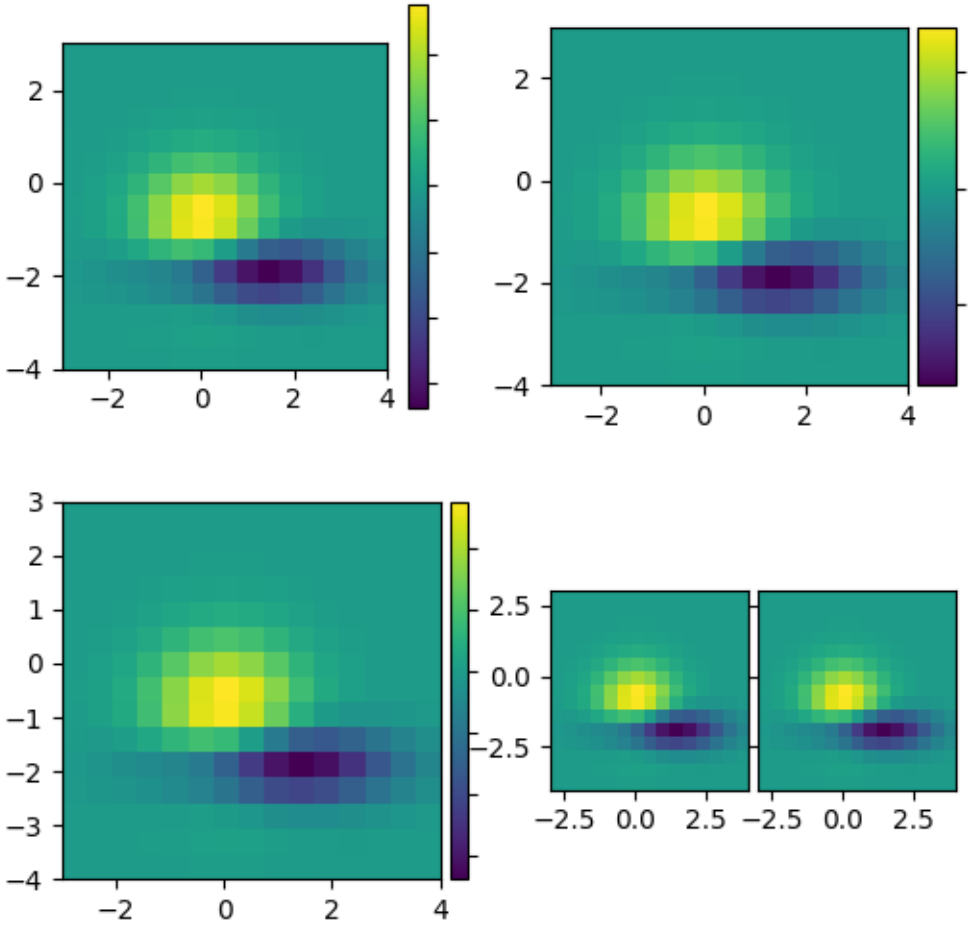
(continues on next page)

(continued from previous page)

```
        sep_x=0.01, sep_y=-0.02,
        color='orange'
        )
ax.add_artist(a2)


a3 = AnchoredDirectionArrows(
        ax.transAxes, ' A', 'B', loc='lower right',
        length=-0.15,
        aspect_ratio=-1,
        sep_y=-0.1, sep_x=0.04,
        color='cyan'
        )
ax.add_artist(a3)

plt.show()
```

### Axes divider

Axes divider to calculate location of Axes and create a divider for them using existing Axes instances.




```
import matplotlib.pyplot as plt

from matplotlib import cbook


def get_demo_image():
    z = cbook.get_sample_data("axes_grid/bivariate_normal.npy")   # 15x15 array
    return z, (-3, 4, -4, 3)


def demo_simple_image(ax):
    Z, extent = get_demo_image()

    im = ax.imshow(Z, extent=extent)
    cb = plt.colorbar(im)
    cb.ax.yaxis.set_tick_params(labelright=False)
```

(continues on next page)

(continued from previous page)

```
def demo_locatable_axes_hard(fig):
    from mpl_toolkits.axes_grid1 import Size, SubplotDivider

    divider = SubplotDivider(fig, 2, 2, 2, aspect=True)

    # Axes for image
    ax = fig.add_subplot(axes_locator=divider.new_locator(nx=0, ny=0))
    # Axes for colorbar
    ax_cb = fig.add_subplot(axes_locator=divider.new_locator(nx=2, ny=0))

    divider.set_horizontal([
        Size.AxesX(ax),  # main Axes
        Size.Fixed(0.05),  # padding, 0.1 inch
        Size.Fixed(0.2),  # colorbar, 0.3 inch
    ])
    divider.set_vertical([Size.AxesY(ax)])

    Z, extent = get_demo_image()

    im = ax.imshow(Z, extent=extent)
    plt.colorbar(im, cax=ax_cb)
    ax_cb.yaxis.set_tick_params(labelright=False)


def demo_locatable_axes_easy(ax):
    from mpl_toolkits.axes_grid1 import make_axes_locatable

    divider = make_axes_locatable(ax)

    ax_cb = divider.append_axes("right", size="5%", pad=0.05)
    fig = ax.get_figure()
    fig.add_axes(ax_cb)

    Z, extent = get_demo_image()
    im = ax.imshow(Z, extent=extent)

    plt.colorbar(im, cax=ax_cb)
    ax_cb.yaxis.tick_right()
    ax_cb.yaxis.set_tick_params(labelright=False)


def demo_images_side_by_side(ax):
    from mpl_toolkits.axes_grid1 import make_axes_locatable

    divider = make_axes_locatable(ax)

    Z, extent = get_demo_image()
    ax2 = divider.append_axes("right", size="100%", pad=0.05)
    fig1 = ax.get_figure()
    fig1.add_axes(ax2)
```

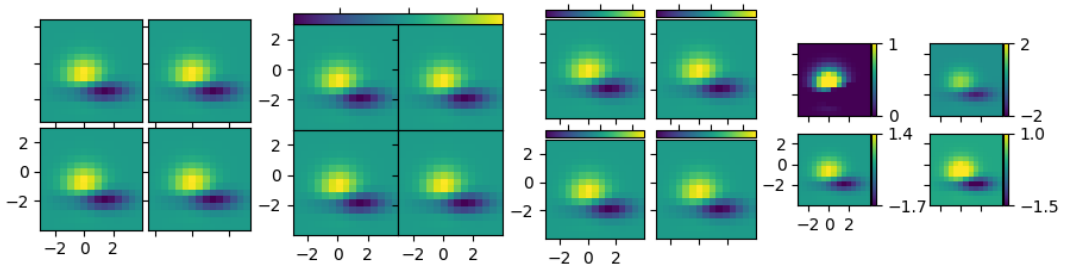
(continues on next page)

(continued from previous page)

```
    ax.imshow(Z, extent=extent)
    ax2.imshow(Z, extent=extent)
    ax2.yaxis.set_tick_params(labelleft=False)


def demo():
    fig = plt.figure(figsize=(6, 6))

    # PLOT 1
    # simple image & colorbar
    ax = fig.add_subplot(2, 2, 1)
    demo_simple_image(ax)

    # PLOT 2
    # image and colorbar with draw-time positioning -- a hard way
    demo_locatable_axes_hard(fig)

    # PLOT 3
    # image and colorbar with draw-time positioning -- an easy way
    ax = fig.add_subplot(2, 2, 3)
    demo_locatable_axes_easy(ax)

    # PLOT 4
    # two images side by side with fixed padding.
    ax = fig.add_subplot(2, 2, 4)
    demo_images_side_by_side(ax)

    plt.show()


demo()
```

## Demo Axes Grid

Grid of 2x2 images with a single colorbar or with one colorbar per Axes.



```
import matplotlib.pyplot as plt

from matplotlib import cbook
```

(continues on next page)

(continued from previous page)

```
from mpl_toolkits.axes_grid1 import ImageGrid

fig = plt.figure(figsize=(10.5, 2.5))
Z = cbook.get_sample_data("axes_grid/bivariate_normal.npy")  # 15x15 array
extent = (-3, 4, -4, 3)


# A grid of 2x2 images with 0.05 inch pad between images and only the
# lower-left Axes is labeled.
grid = ImageGrid(
    fig, 141,  # similar to fig.add_subplot(141).
     nrows_ncols=(2, 2), axes_pad=0.05, label_mode="1")
for ax in grid:
    ax.imshow(Z, extent=extent)
# This only affects Axes in first column and second row as share_all=False.
grid.axes_llc.set(xticks=[-2, 0, 2], yticks=[-2, 0, 2])


# A grid of 2x2 images with a single colorbar.
grid = ImageGrid(
    fig, 142,  # similar to fig.add_subplot(142).
    nrows_ncols=(2, 2), axes_pad=0.0, label_mode="L", share_all=True,
    cbar_location="top", cbar_mode="single")
for ax in grid:
    im = ax.imshow(Z, extent=extent)
grid.cbar_axes[0].colorbar(im)
for cax in grid.cbar_axes:
    cax.tick_params(labeltop=False)
# This affects all Axes as share_all = True.
grid.axes_llc.set(xticks=[-2, 0, 2], yticks=[-2, 0, 2])


# A grid of 2x2 images. Each image has its own colorbar.
grid = ImageGrid(
    fig, 143,  # similar to fig.add_subplot(143).
    nrows_ncols=(2, 2), axes_pad=0.1, label_mode="1", share_all=True,
    cbar_location="top", cbar_mode="each", cbar_size="7%", cbar_pad="2%")
for ax, cax in zip(grid, grid.cbar_axes):
    im = ax.imshow(Z, extent=extent)
    cax.colorbar(im)
    cax.tick_params(labeltop=False)
# This affects all Axes as share_all = True.
grid.axes_llc.set(xticks=[-2, 0, 2], yticks=[-2, 0, 2])


# A grid of 2x2 images. Each image has its own colorbar.
grid = ImageGrid(
    fig, 144,  # similar to fig.add_subplot(144).
    nrows_ncols=(2, 2), axes_pad=(0.45, 0.15), label_mode="1", share_all=True,
    cbar_location="right", cbar_mode="each", cbar_size="7%", cbar_pad="2%")
# Use a different colorbar range every time
limits = ((0, 1), (-2, 2), (-1.7, 1.4), (-1.5, 1))
```

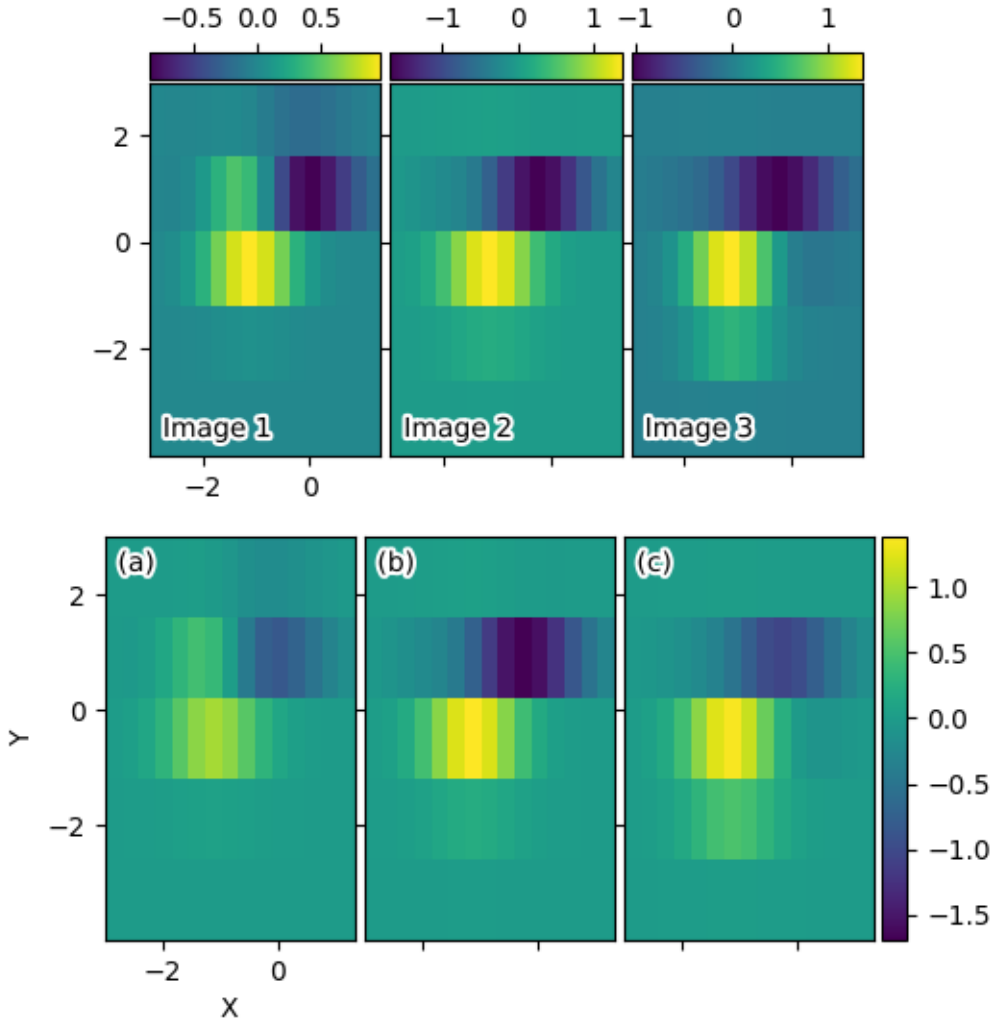
(continues on next page)

(continued from previous page)

```
for ax, cax, vlim in zip(grid, grid.cbar_axes, limits):
    im = ax.imshow(Z, extent=extent, vmin=vlim[0], vmax=vlim[1])
    cb = cax.colorbar(im)
    cb.set_ticks((vlim[0], vlim[1]))
# This affects all Axes as share_all = True.
grid.axes_llc.set(xticks=[-2, 0, 2], yticks=[-2, 0, 2])


plt.show()
```

### Axes Grid2

Grid of images with shared xaxis and yaxis.



```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib import cbook
from mpl_toolkits.axes_grid1 import ImageGrid


def add_inner_title(ax, title, loc, **kwargs):
    from matplotlib.offsetbox import AnchoredText
    from matplotlib.patheffects import withStroke
    prop = dict(path_effects=[withStroke(foreground='w', linewidth=3)],
                size=plt.rcParams['legend.fontsize'])
    at = AnchoredText(title, loc=loc, prop=prop,
                      pad=0., borderpad=0.5,
                      frameon=False, **kwargs)
    ax.add_artist(at)
    return at


fig = plt.figure(figsize=(6, 6))

# Prepare images
Z = cbook.get_sample_data("axes_grid/bivariate_normal.npy")
extent = (-3, 4, -4, 3)
ZS = [Z[i::3, :] for i in range(3)]
extent = extent[0], extent[1]/3., extent[2], extent[3]

# *** Demo 1: colorbar at each Axes ***
grid = ImageGrid(
    # 211 = at the position of fig.add_subplot(211)
    fig, 211, nrows_ncols=(1, 3), axes_pad=0.05, label_mode="1", share_
 ↪all=True,
    cbar_location="top", cbar_mode="each", cbar_size="7%", cbar_pad="1%")
grid[0].set(xticks=[-2, 0], yticks=[-2, 0, 2])

for i, (ax, z) in enumerate(zip(grid, ZS)):
    im = ax.imshow(z, origin="lower", extent=extent)
    cb = ax.cax.colorbar(im)
    # Changing the colorbar ticks
    if i in [1, 2]:
        cb.set_ticks([-1, 0, 1])

for ax, im_title in zip(grid, ["Image 1", "Image 2", "Image 3"]):
    add_inner_title(ax, im_title, loc='lower left')

# *** Demo 2: shared colorbar ***
grid2 = ImageGrid(
    fig, 212, nrows_ncols=(1, 3), axes_pad=0.05, label_mode="1", share_
 ↪all=True,
    cbar_location="right", cbar_mode="single", cbar_size="10%", cbar_pad=0.05)
grid2[0].set(xlabel="X", ylabel="Y", xticks=[-2, 0], yticks=[-2, 0, 2])
```

(continues on next page)

(continued from previous page)

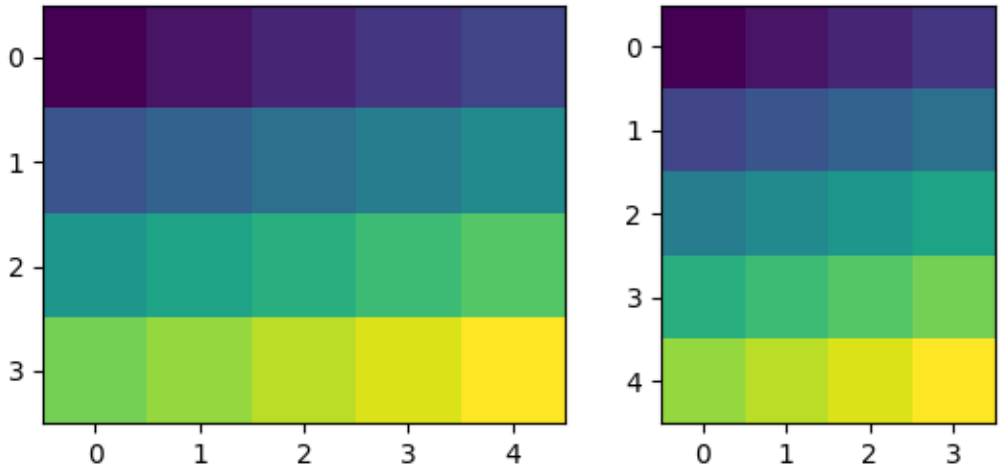
```
clim = (np.min(ZS), np.max(ZS))
for ax, z in zip(grid2, ZS):
    im = ax.imshow(z, clim=clim, origin="lower", extent=extent)

# With cbar_mode="single", cax attribute of all Axes are identical.
ax.cax.colorbar(im)

for ax, im_title in zip(grid2, ["(a)", "(b)", "(c)"]):
    add_inner_title(ax, im_title, loc='upper left')

plt.show()
```

### HBoxDivider and VBoxDivider demo

Using an *HBoxDivider* to arrange subplots.

Note that both Axes' location are adjusted so that they have equal heights while maintaining their aspect ratios.

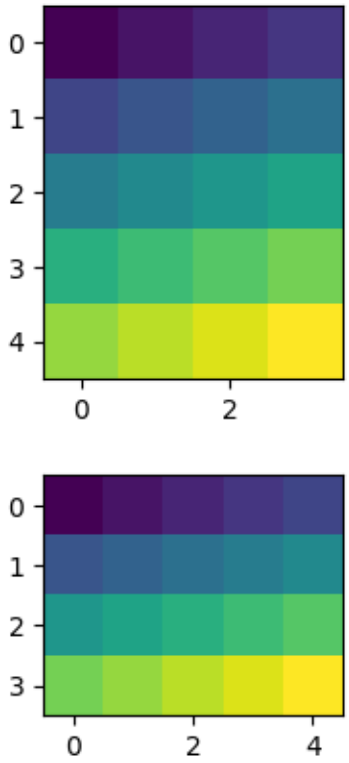
```
import matplotlib.pyplot as plt
import numpy as np

from mpl_toolkits.axes_grid1.axes_divider import HBoxDivider, VBoxDivider
import mpl_toolkits.axes_grid1.axes_size as Size

arr1 = np.arange(20).reshape((4, 5))
arr2 = np.arange(20).reshape((5, 4))

fig, (ax1, ax2) = plt.subplots(1, 2)
ax1.imshow(arr1)
ax2.imshow(arr2)

pad = 0.5  # pad in inches
divider = HBoxDivider(
    fig, 111,
    horizontal=[Size.AxesX(ax1), Size.Fixed(pad), Size.AxesX(ax2)],
    vertical=[Size.AxesY(ax1), Size.Scaled(1), Size.AxesY(ax2)])
ax1.set_axes_locator(divider.new_locator(0))
ax2.set_axes_locator(divider.new_locator(2))

plt.show()
```




Using a *VBoxDivider* to arrange subplots.

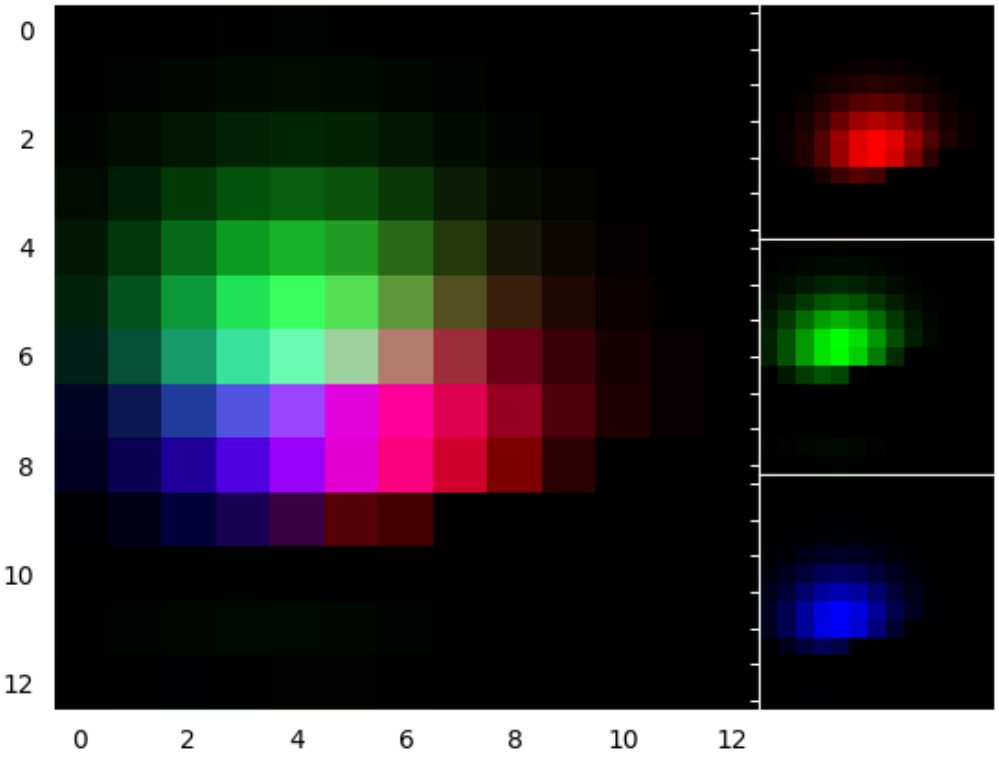
Note that both Axes' location are adjusted so that they have equal widths while maintaining their aspect ratios.

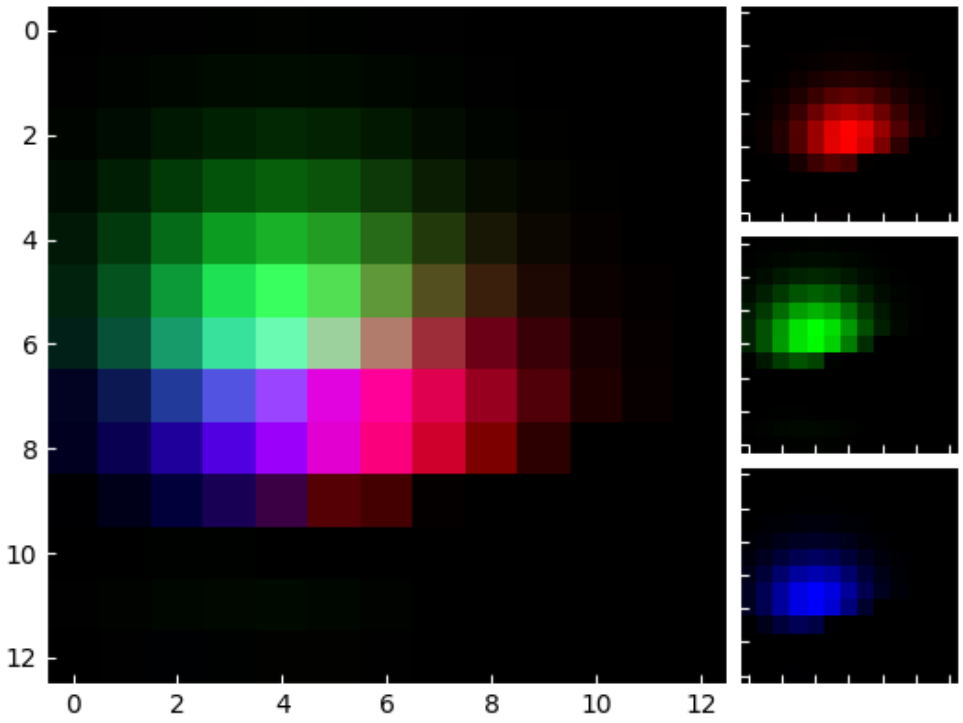
```
fig, (ax1, ax2) = plt.subplots(2, 1)
ax1.imshow(arr1)
ax2.imshow(arr2)

divider = VBoxDivider(
    fig, 111,
    horizontal=[Size.AxesX(ax1), Size.Scaled(1), Size.AxesX(ax2)],
    vertical=[Size.AxesY(ax1), Size.Fixed(pad), Size.AxesY(ax2)])

ax1.set_axes_locator(divider.new_locator(0))
ax2.set_axes_locator(divider.new_locator(2))

plt.show()
```




### Showing RGB channels using RGBAxes

`RGBAxes` creates a layout of 4 Axes for displaying RGB channels: one large Axes for the RGB image and 3 smaller Axes for the R, G, B channels.


0
2
4
6
8
10
12
0
2
4
6
8
10
12




- 

```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib import cbook
from mpl_toolkits.axes_grid1.axes_rgb import RGBAxes, make_rgb_axes


def get_rgb():
    Z = cbook.get_sample_data("axes_grid/bivariate_normal.npy")
    Z[Z < 0] = 0.
    Z = Z / Z.max()

    R = Z[:13, :13]
    G = Z[2:, 2:]
    B = Z[:13, 2:]

    return R, G, B


def make_cube(r, g, b):
    ny, nx = r.shape
    R = np.zeros((ny, nx, 3))
    R[:, :, 0] = r
    G = np.zeros_like(R)
```

(continues on next page)

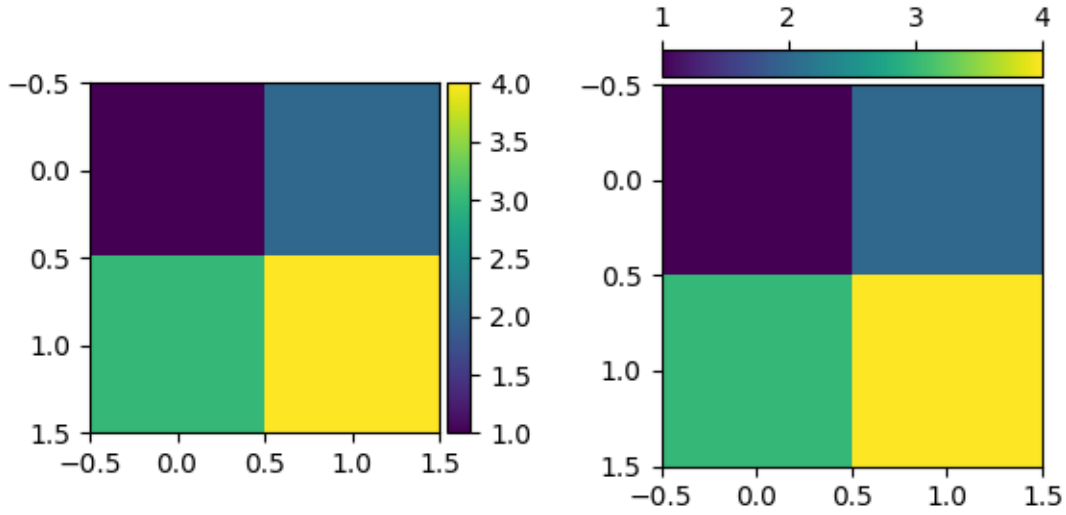
(continued from previous page)

```
    G[:, :, 1] = g
    B = np.zeros_like(R)
    B[:, :, 2] = b

    RGB = R + G + B

    return R, G, B, RGB


def demo_rgb1():
    fig = plt.figure()
    ax = RGBAxes(fig, [0.1, 0.1, 0.8, 0.8], pad=0.0)
    r, g, b = get_rgb()
    ax.imshow_rgb(r, g, b)


def demo_rgb2():
    fig, ax = plt.subplots()
    ax_r, ax_g, ax_b = make_rgb_axes(ax, pad=0.02)

    r, g, b = get_rgb()
    im_r, im_g, im_b, im_rgb = make_cube(r, g, b)
    ax.imshow(im_rgb)
    ax_r.imshow(im_r)
    ax_g.imshow(im_g)
    ax_b.imshow(im_b)

    for ax in fig.axes:
        ax.tick_params(direction='in', color='w')
        ax.spines[:].set_color("w")


demo_rgb1()
demo_rgb2()

plt.show()
```

### Colorbar with AxesDivider

The *axes_divider.make_axes_locatable* function takes an existing Axes, adds it to a new *AxesDivider* and returns the *AxesDivider*. The *append_axes* method of the *AxesDivider* can then be used to create a new Axes on a given side ("top", "right", "bottom", or "left") of the original Axes. This example uses *append_axes* to add colorbars next to Axes.

Users should consider simply passing the main Axes to the *ax* keyword argument of *colorbar* instead of creating a locatable Axes manually like this. See *Placing colorbars*.



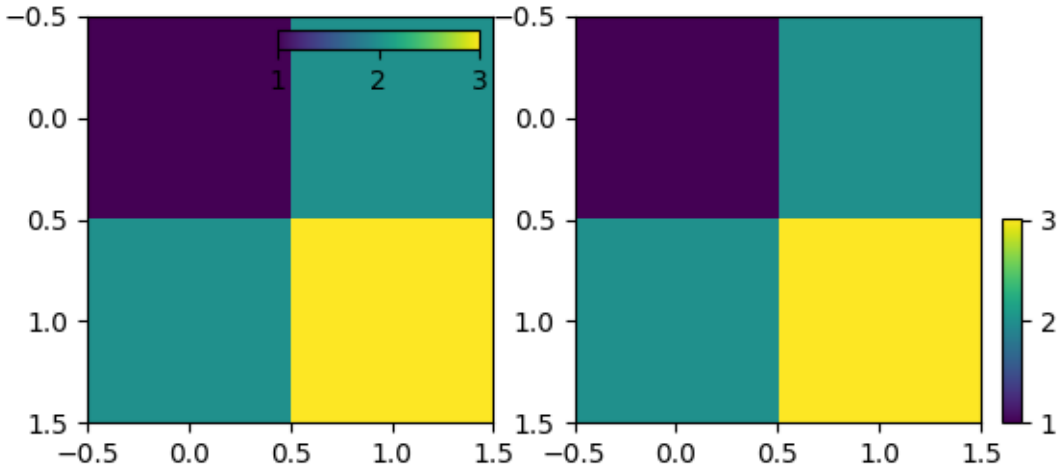

```
import matplotlib.pyplot as plt

from mpl_toolkits.axes_grid1.axes_divider import make_axes_locatable

fig, (ax1, ax2) = plt.subplots(1, 2)
fig.subplots_adjust(wspace=0.5)

im1 = ax1.imshow([[1, 2], [3, 4]])
ax1_divider = make_axes_locatable(ax1)
# Add an Axes to the right of the main Axes.
cax1 = ax1_divider.append_axes("right", size="7%", pad="2%")
cb1 = fig.colorbar(im1, cax=cax1)

im2 = ax2.imshow([[1, 2], [3, 4]])
ax2_divider = make_axes_locatable(ax2)
# Add an Axes above the main Axes.
cax2 = ax2_divider.append_axes("top", size="7%", pad="2%")
cb2 = fig.colorbar(im2, cax=cax2, orientation="horizontal")
# Change tick position to top (with the default tick position "bottom", ticks
# overlap the image).
cax2.xaxis.set_ticks_position("top")

plt.show()
```

### Controlling the position and size of colorbars with Inset Axes

This example shows how to control the position, height, and width of colorbars using `inset_axes`.

Inset Axes placement is controlled as for legends: either by providing a *loc* option ("upper right", "best", ...), or by providing a locator with respect to the parent bbox. Parameters such as *bbox_to_anchor* and *borderpad* likewise work in the same way, and are also demonstrated here.

Users should consider using `Axes.inset_axes` instead (see *Placing colorbars*).




```
import matplotlib.pyplot as plt

from mpl_toolkits.axes_grid1.inset_locator import inset_axes

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=[6, 3])

im1 = ax1.imshow([[1, 2], [2, 3]])
axins1 = inset_axes(
    ax1,
    width="50%",  # width: 50% of parent_bbox width
    height="5%",  # height: 5%
    loc="upper right",
)
axins1.xaxis.set_ticks_position("bottom")
fig.colorbar(im1, cax=axins1, orientation="horizontal", ticks=[1, 2, 3])

im = ax2.imshow([[1, 2], [2, 3]])
axins = inset_axes(
    ax2,
    width="5%",  # width: 5% of parent_bbox width
    height="50%",  # height: 50%
    loc="lower left",
    bbox_to_anchor=(1.05, 0., 1, 1),
    bbox_transform=ax2.transAxes,
```

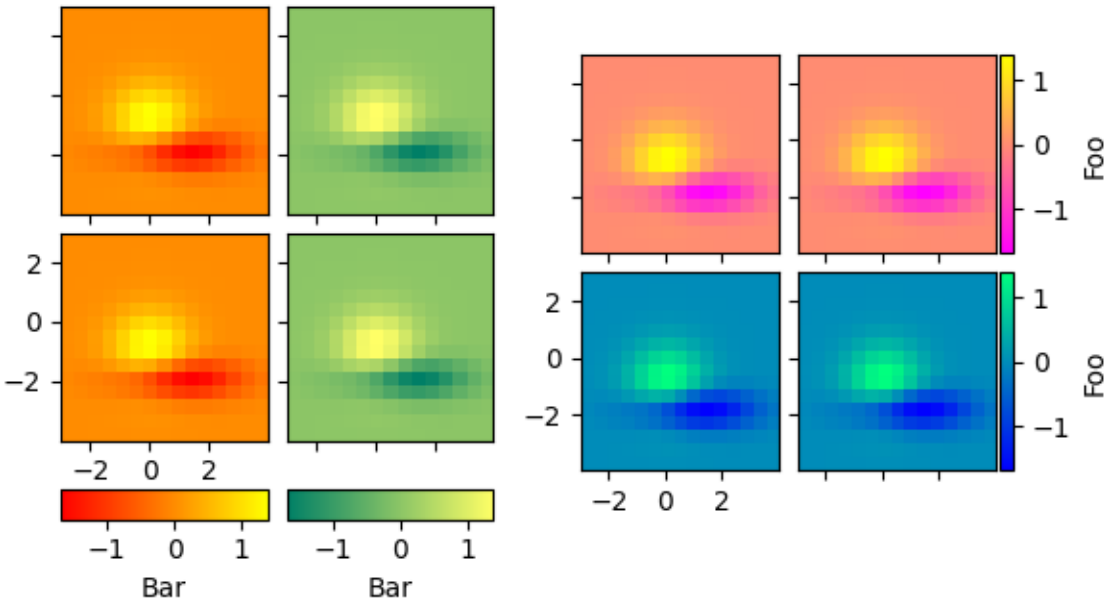
(continues on next page)

(continued from previous page)

```
    borderpad=0,
)
fig.colorbar(im, cax=axins, ticks=[1, 2, 3])

plt.show()
```

### Per-row or per-column colorbars

This example shows how to use one common colorbar for each row or column of an image grid.




```
import matplotlib.pyplot as plt

from matplotlib import cbook
from mpl_toolkits.axes_grid1 import AxesGrid


def get_demo_image():
    z = cbook.get_sample_data("axes_grid/bivariate_normal.npy")  # 15x15 array
    return z, (-3, 4, -4, 3)
```

(continues on next page)

(continued from previous page)

```
def demo_bottom_cbar(fig):
    """
    A grid of 2x2 images with a colorbar for each column.
    """
    grid = AxesGrid(fig, 121,  # similar to subplot(121)
                    nrows_ncols=(2, 2),
                    axes_pad=0.10,
                    share_all=True,
                    label_mode="1",
                    cbar_location="bottom",
                    cbar_mode="edge",
                    cbar_pad=0.25,
                    cbar_size="15%",
                    direction="column"
                    )

    Z, extent = get_demo_image()
    cmaps = ["autumn", "summer"]
    for i in range(4):
        im = grid[i].imshow(Z, extent=extent, cmap=cmaps[i//2])
        if i % 2:
            grid.cbar_axes[i//2].colorbar(im)

    for cax in grid.cbar_axes:
        cax.axis[cax.orientation].set_label("Bar")

    # This affects all Axes as share_all = True.
    grid.axes_llc.set_xticks([-2, 0, 2])
    grid.axes_llc.set_yticks([-2, 0, 2])


def demo_right_cbar(fig):
    """
    A grid of 2x2 images. Each row has its own colorbar.
    """
    grid = AxesGrid(fig, 122,  # similar to subplot(122)
                    nrows_ncols=(2, 2),
                    axes_pad=0.10,
                    label_mode="1",
                    share_all=True,
                    cbar_location="right",
                    cbar_mode="edge",
                    cbar_size="7%",
                    cbar_pad="2%",
                    )
    Z, extent = get_demo_image()
    cmaps = ["spring", "winter"]
    for i in range(4):
        im = grid[i].imshow(Z, extent=extent, cmap=cmaps[i//2])
        if i % 2:
            grid.cbar_axes[i//2].colorbar(im)
```

(continues on next page)

(continued from previous page)

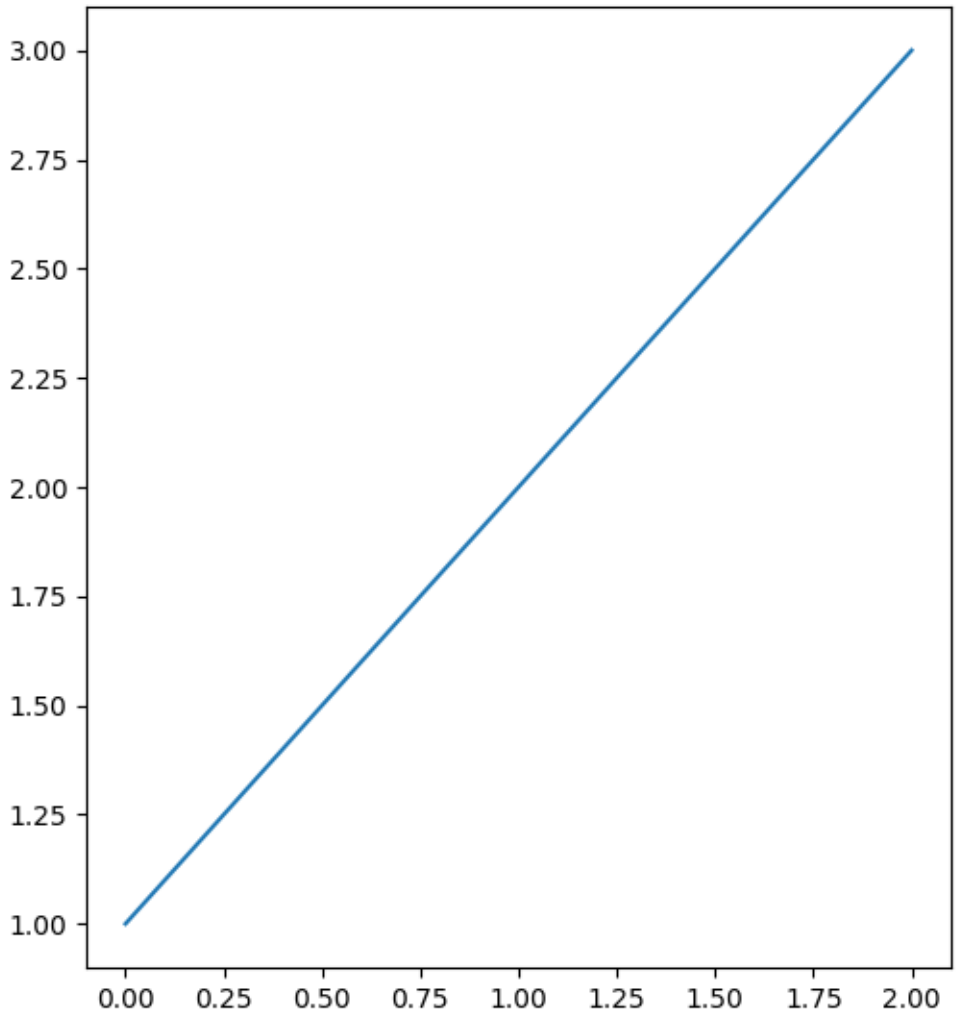
```
    for cax in grid.cbar_axes:
        cax.axis[cax.orientation].set_label('Foo')

    # This affects all Axes because we set share_all = True.
    grid.axes_llc.set_xticks([-2, 0, 2])
    grid.axes_llc.set_yticks([-2, 0, 2])


fig = plt.figure()

demo_bottom_cbar(fig)
demo_right_cbar(fig)

plt.show()
```

### Axes with a fixed physical size

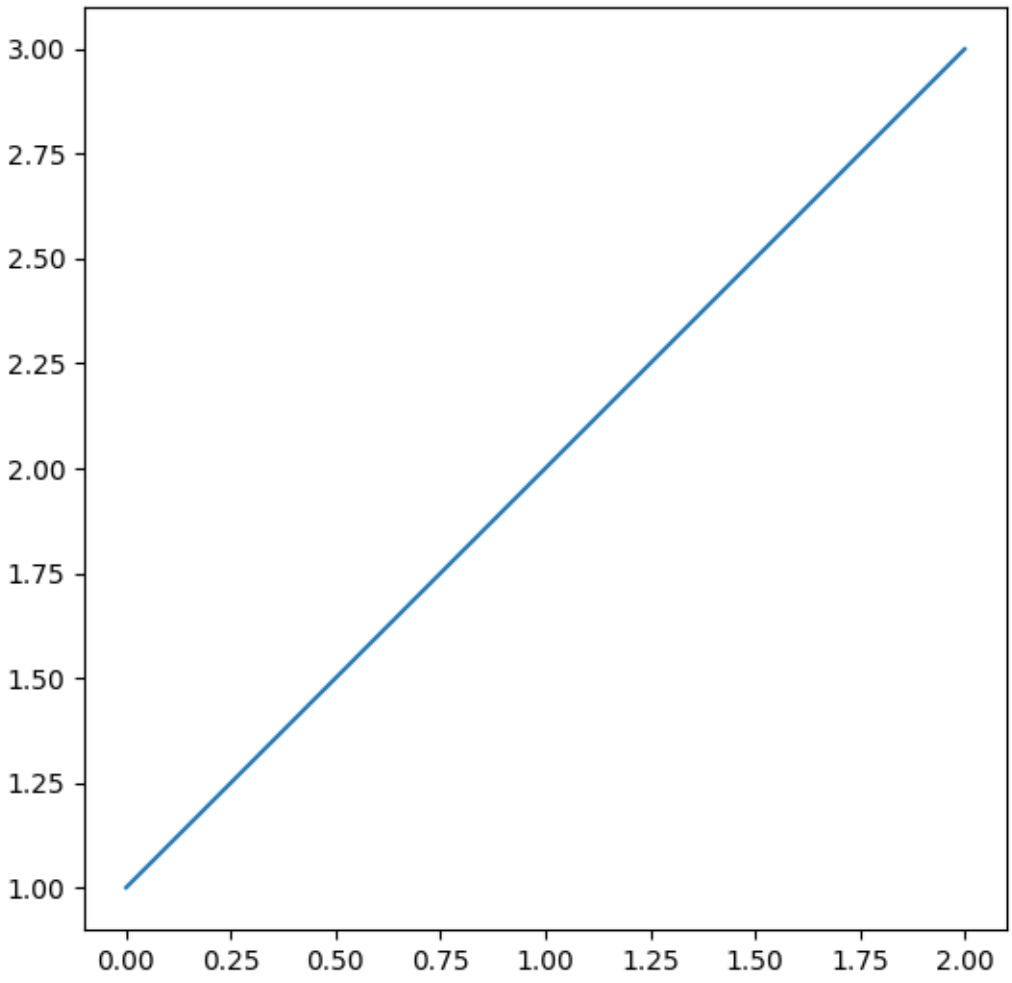
Note that this can be accomplished with the main library for Axes on Figures that do not change size: *Adding single Axes at a time*

```
import matplotlib.pyplot as plt

from mpl_toolkits.axes_grid1 import Divider, Size
```

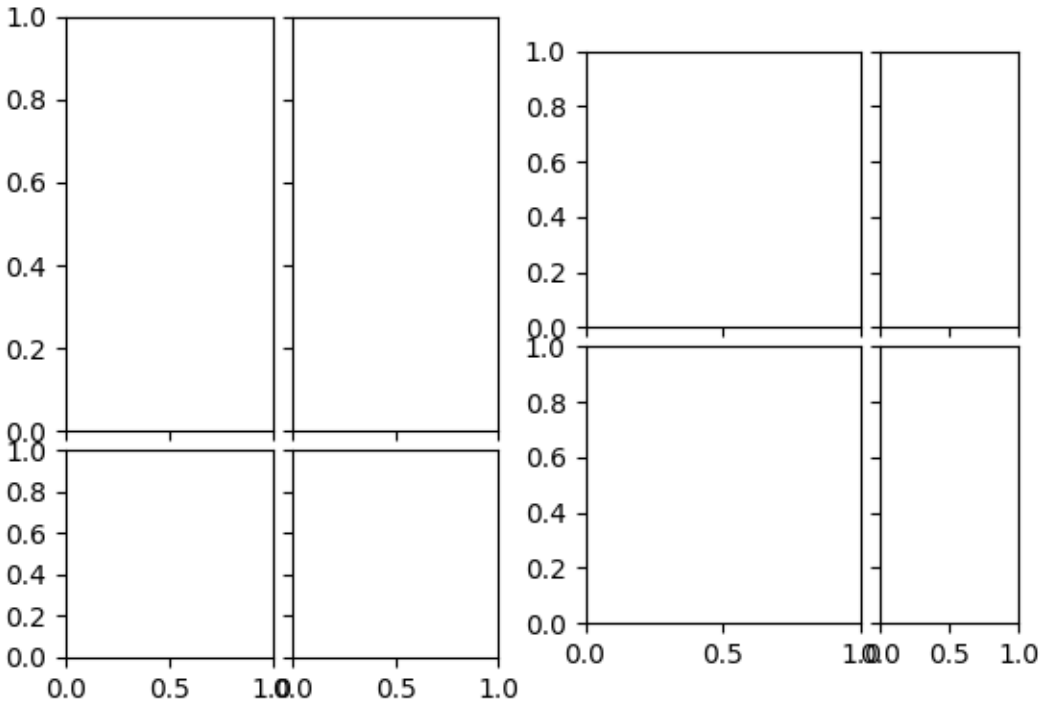
```
fig = plt.figure(figsize=(6, 6))

# The first items are for padding and the second items are for the Axes.
# sizes are in inch.
h = [Size.Fixed(1.0), Size.Fixed(4.5)]
v = [Size.Fixed(0.7), Size.Fixed(5.)]

divider = Divider(fig, (0, 0, 1, 1), h, v, aspect=False)
# The width and height of the rectangle are ignored.

ax = fig.add_axes(divider.get_position(),
                  axes_locator=divider.new_locator(nx=1, ny=1))

ax.plot([1, 2, 3])
```




```
fig = plt.figure(figsize=(6, 6))

# The first & third items are for padding and the second items are for the
# Axes. Sizes are in inches.
h = [Size.Fixed(1.0), Size.Scaled(1.), Size.Fixed(.2)]
v = [Size.Fixed(0.7), Size.Scaled(1.), Size.Fixed(.5)]

divider = Divider(fig, (0, 0, 1, 1), h, v, aspect=False)
# The width and height of the rectangle are ignored.

ax = fig.add_axes(divider.get_position(),
                  axes_locator=divider.new_locator(nx=1, ny=1))

ax.plot([1, 2, 3])

plt.show()
```


3.00
2.75
2.50
2.25
2.00
1.75
1.50
1.25
1.00
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00

### Setting a fixed aspect on ImageGrid cells

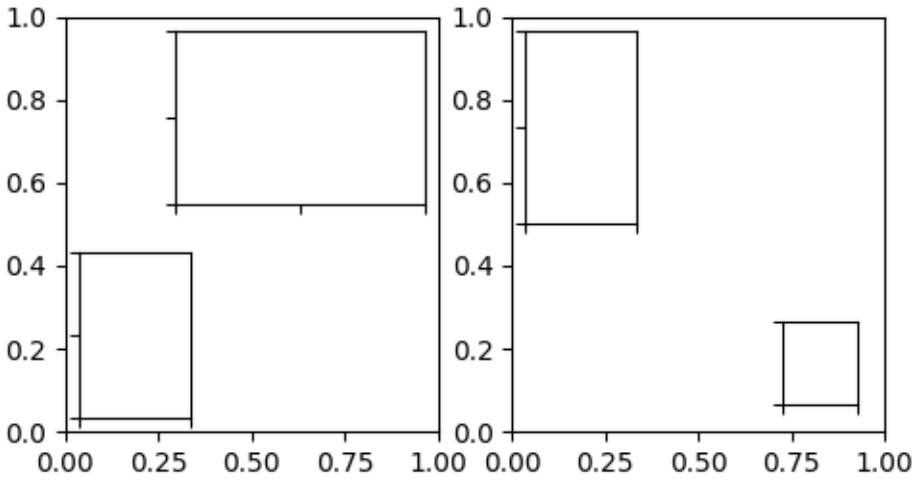



```
import matplotlib.pyplot as plt

from mpl_toolkits.axes_grid1 import ImageGrid

fig = plt.figure()

grid1 = ImageGrid(fig, 121, (2, 2), axes_pad=0.1,
                  aspect=True, share_all=True)
for i in [0, 1]:
    grid1[i].set_aspect(2)

grid2 = ImageGrid(fig, 122, (2, 2), axes_pad=0.1,
                  aspect=True, share_all=True)
for i in [1, 3]:
    grid2[i].set_aspect(2)

plt.show()
```

### Inset locator demo

The *inset_locator*'s *inset_axes* allows easily placing insets in the corners of the Axes by specifying a width and height and optionally a location (loc) that accepts locations as codes, similar to *legend*. By default, the inset is offset by some points from the axes, controlled via the *borderpad* parameter.

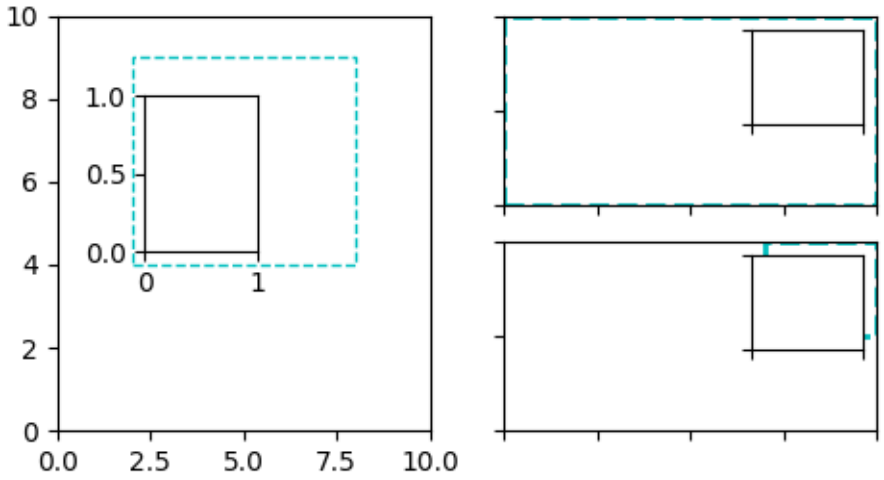
```
import matplotlib.pyplot as plt

from mpl_toolkits.axes_grid1.inset_locator import inset_axes

fig, (ax, ax2) = plt.subplots(1, 2, figsize=[5.5, 2.8])

# Create inset of width 1.3 inches and height 0.9 inches
# at the default upper right location
axins = inset_axes(ax, width=1.3, height=0.9)

# Create inset of width 30% and height 40% of the parent Axes' bounding box
# at the lower left corner (loc=3)
axins2 = inset_axes(ax, width="30%", height="40%", loc=3)

# Create inset of mixed specifications in the second subplot;
# width is 30% of parent Axes' bounding box and
# height is 1 inch at the upper left corner (loc=2)
axins3 = inset_axes(ax2, width="30%", height=1., loc=2)

# Create an inset in the lower right corner (loc=4) with borderpad=1, i.e.
# 10 points padding (as 10pt is the default fontsize) to the parent Axes
axins4 = inset_axes(ax2, width="20%", height="20%", loc=4, borderpad=1)

# Turn ticklabels of insets off
for axi in [axins, axins2, axins3, axins4]:
    axi.tick_params(labelleft=False, labelbottom=False)

plt.show()
```




The parameters *bbox_to_anchor* and *bbox_transform* can be used for a more fine-grained control over the

inset position and size or even to position the inset at completely arbitrary positions. The *bbox_to_anchor* sets the bounding box in coordinates according to the *bbox_transform*.

```
fig = plt.figure(figsize=[5.5, 2.8])
ax = fig.add_subplot(121)

# We use the Axes transform as bbox_transform. Therefore, the bounding box
# needs to be specified in axes coordinates ((0, 0) is the lower left corner
# of the Axes, (1, 1) is the upper right corner).
# The bounding box (.2, .4, .6, .5) starts at (.2, .4) and ranges to (.8, .9)
# in those coordinates.
# Inside this bounding box an inset of half the bounding box' width and
# three quarters of the bounding box' height is created. The lower left corner
# of the inset is aligned to the lower left corner of the bounding box␣
 ↪(loc=3).
# The inset is then offset by the default 0.5 in units of the font size.

axins = inset_axes(ax, width="50%", height="75%",
                   bbox_to_anchor=(.2, .4, .6, .5),
                   bbox_transform=ax.transAxes, loc=3)

# For visualization purposes we mark the bounding box by a rectangle
ax.add_patch(plt.Rectangle((.2, .4), .6, .5, ls="--", ec="c", fc="none",
                           transform=ax.transAxes))

# We set the axis limits to something other than the default, in order to not
# distract from the fact that axes coordinates are used here.
ax.set(xlim=(0, 10), ylim=(0, 10))


# Note how the two following insets are created at the same positions, one by
# use of the default parent Axes' bbox and the other via a bbox in Axes
# coordinates and the respective transform.
ax2 = fig.add_subplot(222)
axins2 = inset_axes(ax2, width="30%", height="50%")

ax3 = fig.add_subplot(224)
axins3 = inset_axes(ax3, width="100%", height="100%",
                    bbox_to_anchor=(.7, .5, .3, .5),
                    bbox_transform=ax3.transAxes)

# For visualization purposes we mark the bounding box by a rectangle
ax2.add_patch(plt.Rectangle((0, 0), 1, 1, ls="--", lw=2, ec="c", fc="none"))
ax3.add_patch(plt.Rectangle((.7, .5), .3, .5, ls="--", lw=2,
                            ec="c", fc="none"))

# Turn ticklabels off
for axi in [axins2, axins3, ax2, ax3]:
    axi.tick_params(labelleft=False, labelbottom=False)

plt.show()
```



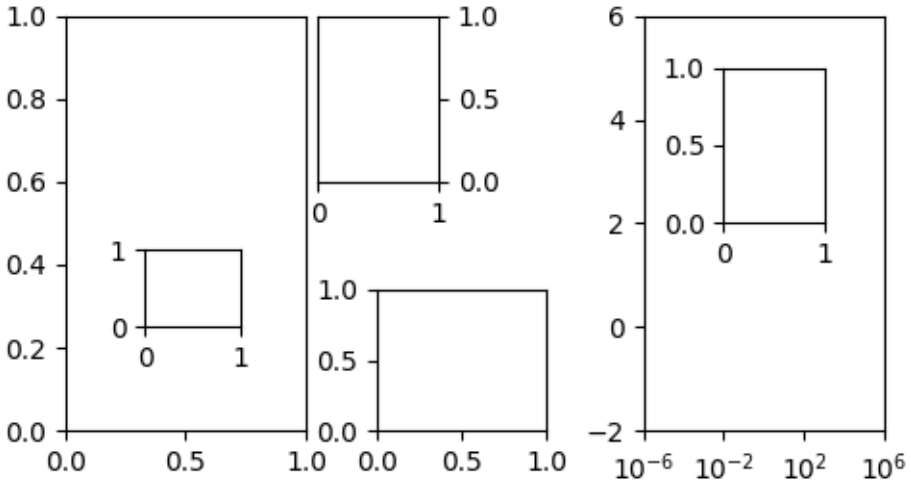

In the above the Axes transform together with 4-tuple bounding boxes has been used as it mostly is useful to specify an inset relative to the Axes it is an inset to. However, other use cases are equally possible. The following example examines some of those.

```
fig = plt.figure(figsize=[5.5, 2.8])
ax = fig.add_subplot(131)

# Create an inset outside the Axes
axins = inset_axes(ax, width="100%", height="100%",
                   bbox_to_anchor=(1.05, .6, .5, .4),
                   bbox_transform=ax.transAxes, loc=2, borderpad=0)
axins.tick_params(left=False, right=True, labelleft=False, labelright=True)

# Create an inset with a 2-tuple bounding box. Note that this creates a
# bbox without extent. This hence only makes sense when specifying
# width and height in absolute units (inches).
axins2 = inset_axes(ax, width=0.5, height=0.4,
                    bbox_to_anchor=(0.33, 0.25),
                    bbox_transform=ax.transAxes, loc=3, borderpad=0)


ax2 = fig.add_subplot(133)
ax2.set_xscale("log")
ax2.set(xlim=(1e-6, 1e6), ylim=(-2, 6))

# Create inset in data coordinates using ax.transData as transform
axins3 = inset_axes(ax2, width="100%", height="100%",
                    bbox_to_anchor=(1e-2, 2, 1e3, 3),
                    bbox_transform=ax2.transData, loc=2, borderpad=0)

# Create an inset horizontally centered in figure coordinates and vertically
# bound to line up with the Axes.
from matplotlib.transforms import blended_transform_factory  # noqa

transform = blended_transform_factory(fig.transFigure, ax2.transAxes)
```

(continues on next page)

(continued from previous page)

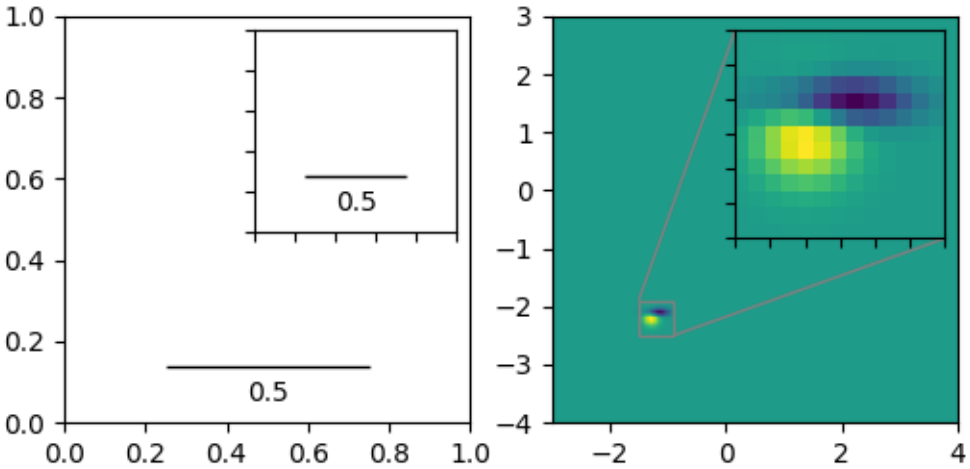
```
axins4 = inset_axes(ax2, width="16%", height="34%",
                    bbox_to_anchor=(0, 0, 1, 1),
                    bbox_transform=transform, loc=8, borderpad=0)

plt.show()
```




## Inset locator demo 2

This demo shows how to create a zoomed inset via *zoomed_inset_axes*. In the first subplot an *AnchoredSizeBar* shows the zoom effect. In the second subplot a connection to the region of interest is created via *mark_inset*.

A version of the second subplot, not using the toolkit, is available in *Zoom region inset Axes*.



```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib import cbook
from mpl_toolkits.axes_grid1.anchored_artists import AnchoredSizeBar
from mpl_toolkits.axes_grid1.inset_locator import mark_inset, zoomed_inset_
 ↪axes

fig, (ax, ax2) = plt.subplots(ncols=2, figsize=[6, 3])


# First subplot, showing an inset with a size bar.
ax.set_aspect(1)

axins = zoomed_inset_axes(ax, zoom=0.5, loc='upper right')
# fix the number of ticks on the inset Axes
axins.yaxis.get_major_locator().set_params(nbins=7)
axins.xaxis.get_major_locator().set_params(nbins=7)
axins.tick_params(labelleft=False, labelbottom=False)


def add_sizebar(ax, size):
    asb = AnchoredSizeBar(ax.transData,
                          size,
                          str(size),
                          loc=8,
                          pad=0.1, borderpad=0.5, sep=5,
                          frameon=False)
    ax.add_artist(asb)

add_sizebar(ax, 0.5)
add_sizebar(axins, 0.5)


# Second subplot, showing an image with an inset zoom and a marked inset
Z = cbook.get_sample_data("axes_grid/bivariate_normal.npy")  # 15x15 array
extent = (-3, 4, -4, 3)
Z2 = np.zeros((150, 150))
ny, nx = Z.shape
Z2[30:30+ny, 30:30+nx] = Z

ax2.imshow(Z2, extent=extent, origin="lower")

axins2 = zoomed_inset_axes(ax2, zoom=6, loc=1)
axins2.imshow(Z2, extent=extent, origin="lower")

# subregion of the original image
x1, x2, y1, y2 = -1.5, -0.9, -2.5, -1.9
axins2.set_xlim(x1, x2)
axins2.set_ylim(y1, y2)
# fix the number of ticks on the inset Axes
axins2.yaxis.get_major_locator().set_params(nbins=7)
```

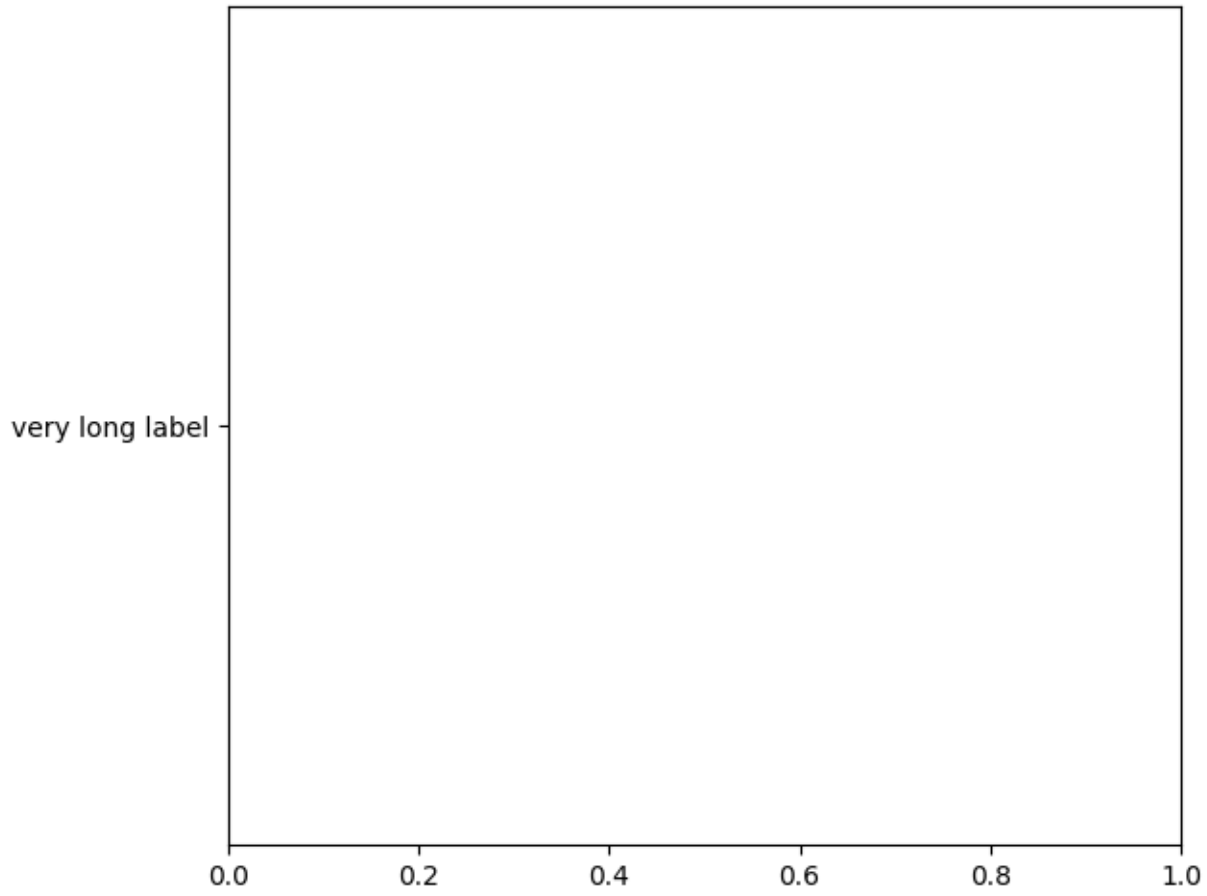
(continues on next page)

(continued from previous page)

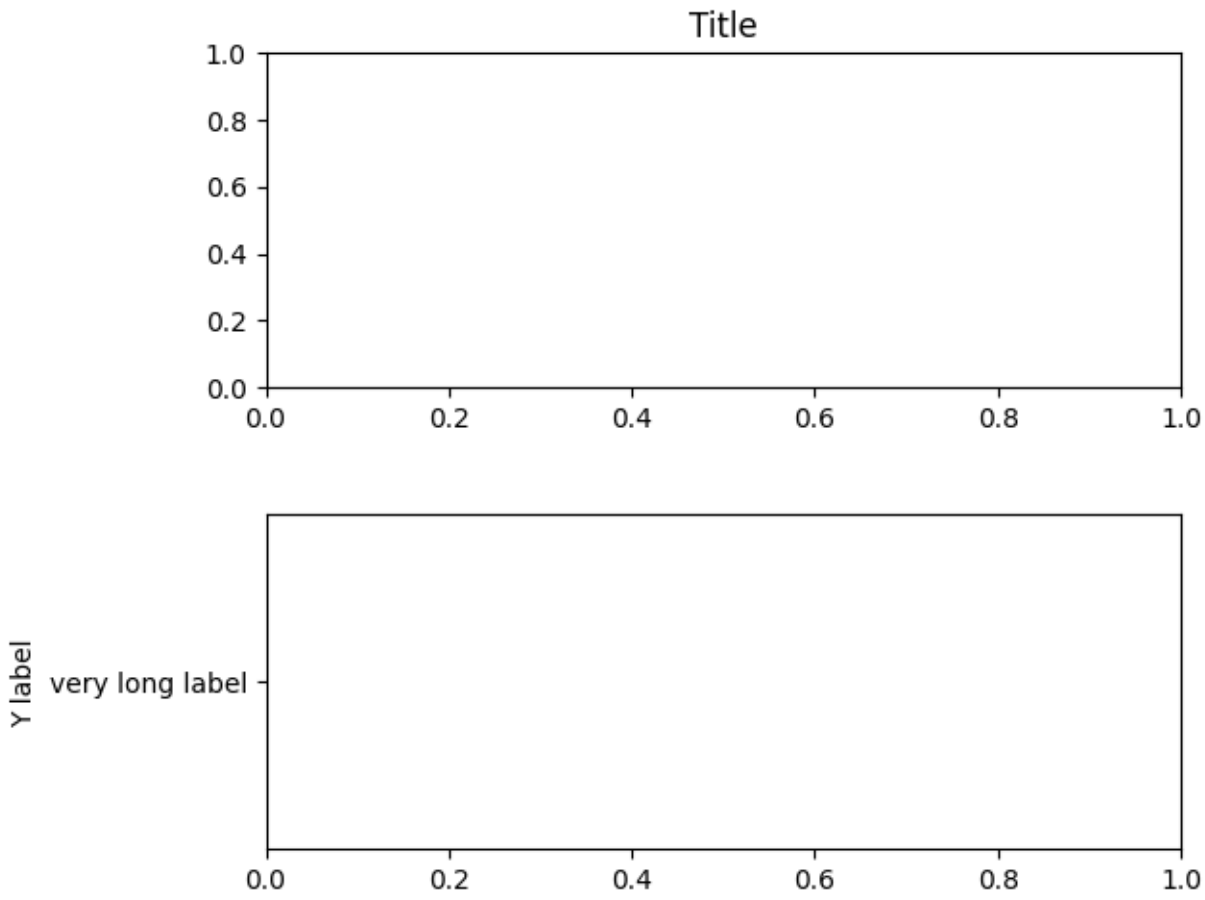
```
axins2.xaxis.get_major_locator().set_params(nbins=7)
axins2.tick_params(labelleft=False, labelbottom=False)

# draw a bbox of the region of the inset Axes in the parent Axes and
# connecting lines between the bbox and the inset Axes area
mark_inset(ax2, axins2, loc1=2, loc2=4, fc="none", ec="0.5")

plt.show()
```

### Make room for ylabel using axes_grid

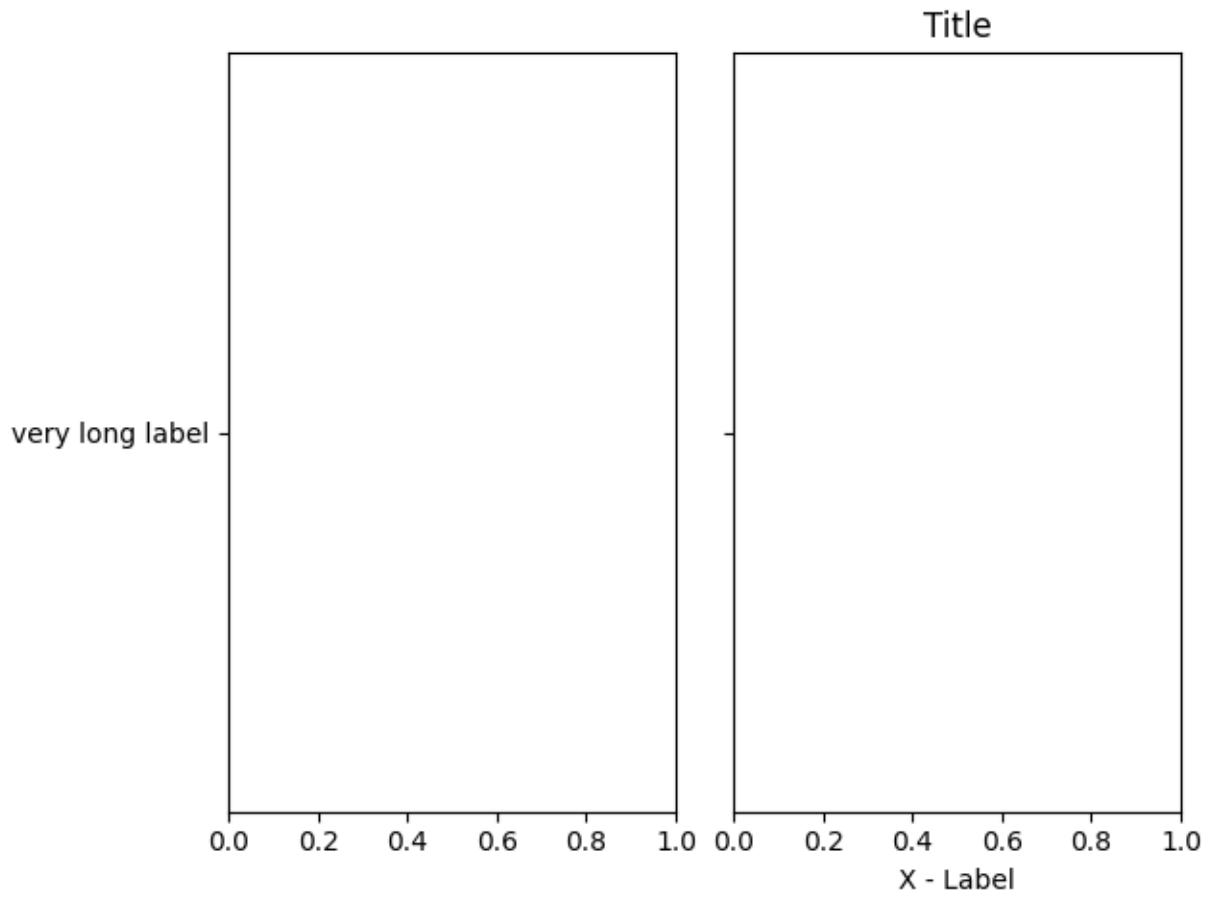
```
import matplotlib.pyplot as plt

from mpl_toolkits.axes_grid1 import make_axes_locatable
from mpl_toolkits.axes_grid1.axes_divider import make_axes_area_auto_
 ↪adjustable

fig = plt.figure()
ax = fig.add_axes([0, 0, 1, 1])

ax.set_yticks([0.5], labels=["very long label"])

make_axes_area_auto_adjustable(ax)
```




```
fig = plt.figure()
ax1 = fig.add_axes([0, 0, 1, 0.5])
ax2 = fig.add_axes([0, 0.5, 1, 0.5])

ax1.set_yticks([0.5], labels=["very long label"])
ax1.set_ylabel("Y label")

ax2.set_title("Title")

make_axes_area_auto_adjustable(ax1, pad=0.1, use_axes=[ax1, ax2])
make_axes_area_auto_adjustable(ax2, pad=0.1, use_axes=[ax1, ax2])
```

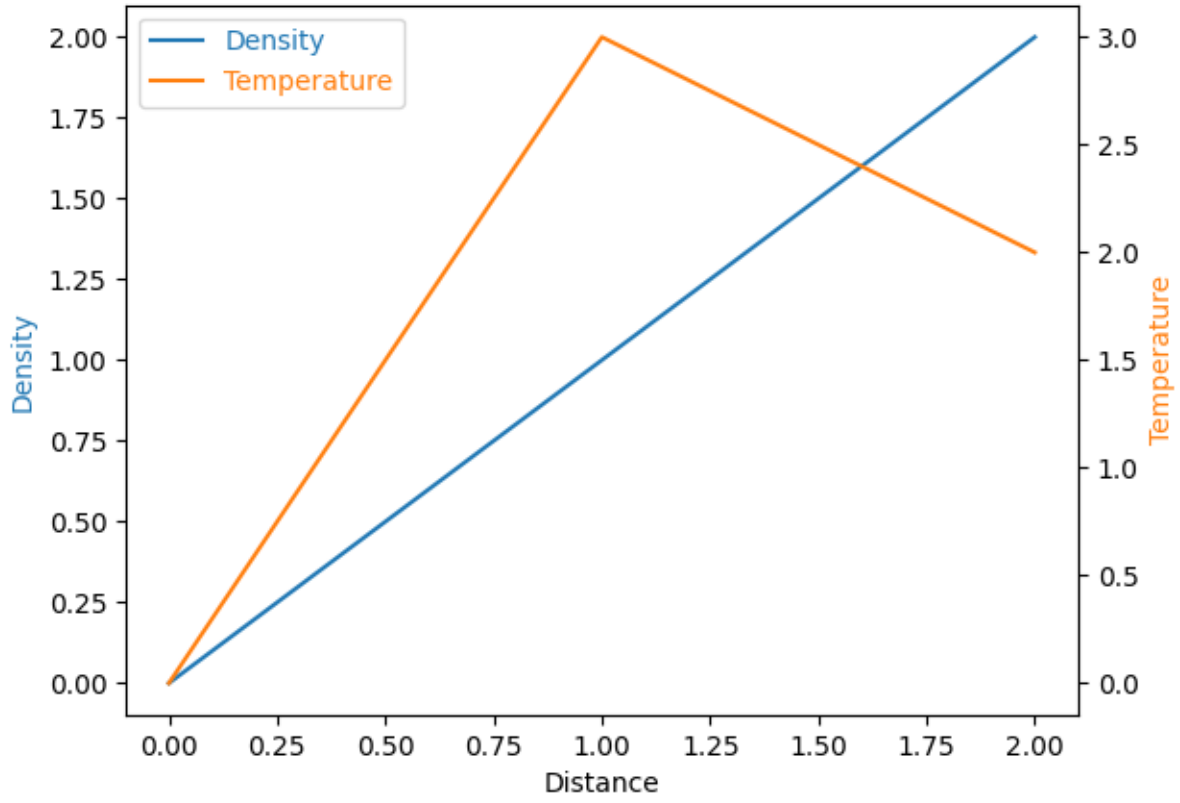



```
fig = plt.figure()
ax1 = fig.add_axes([0, 0, 1, 1])
divider = make_axes_locatable(ax1)

ax2 = divider.append_axes("right", "100%", pad=0.3, sharey=ax1)
ax2.tick_params(labelleft=False)
fig.add_axes(ax2)

divider.add_auto_adjustable_area(use_axes=[ax1], pad=0.1,
                                 adjust_dirs=["left"])
divider.add_auto_adjustable_area(use_axes=[ax2], pad=0.1,
                                 adjust_dirs=["right"])
divider.add_auto_adjustable_area(use_axes=[ax1, ax2], pad=0.1,
                                 adjust_dirs=["top", "bottom"])

ax1.set_yticks([0.5], labels=["very long label"])

ax2.set_title("Title")
ax2.set_xlabel("X - Label")

plt.show()
```



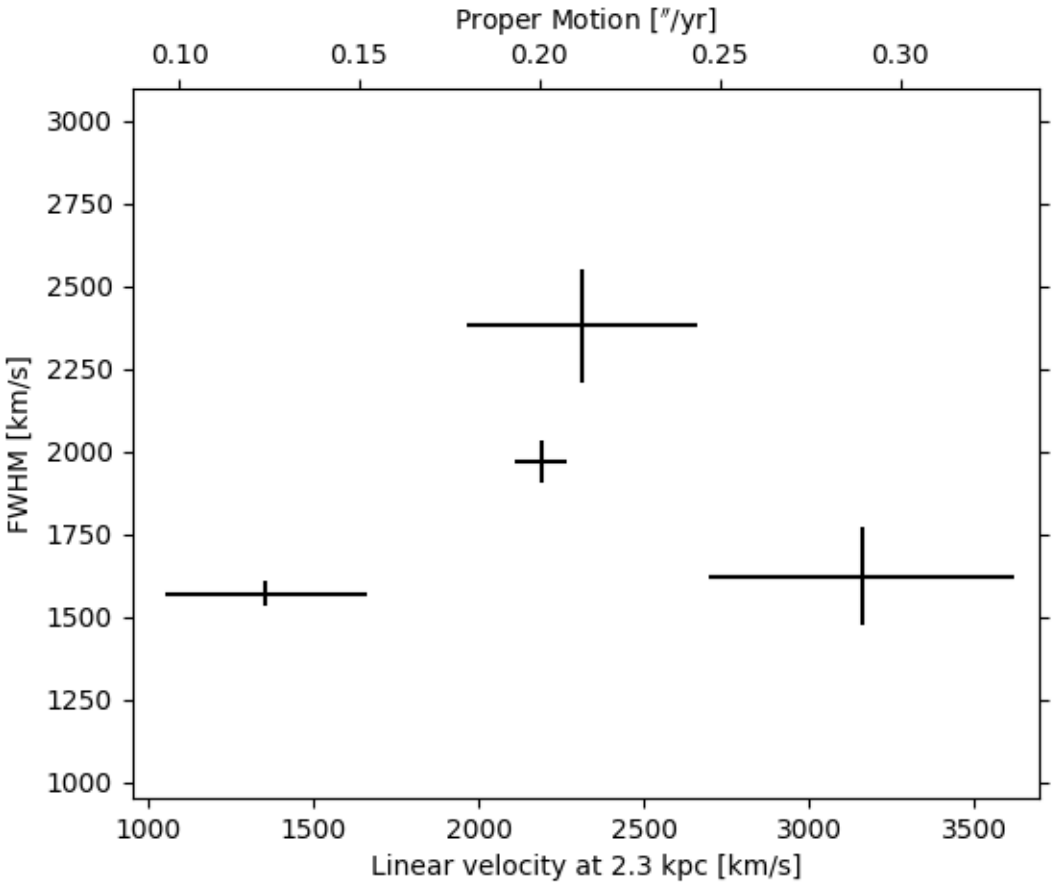

**Total running time of the script:** (0 minutes 1.092 seconds)

### Parasite Simple




```
import matplotlib.pyplot as plt

from mpl_toolkits.axes_grid1 import host_subplot

host = host_subplot(111)
par = host.twinx()

host.set_xlabel("Distance")
host.set_ylabel("Density")
par.set_ylabel("Temperature")

p1, = host.plot([0, 1, 2], [0, 1, 2], label="Density")
p2, = par.plot([0, 1, 2], [0, 3, 2], label="Temperature")

host.legend(labelcolor="linecolor")

host.yaxis.get_label().set_color(p1.get_color())
par.yaxis.get_label().set_color(p2.get_color())

plt.show()
```

### Parasite Simple2




```
import matplotlib.pyplot as plt

import matplotlib.transforms as mtransforms
from mpl_toolkits.axes_grid1.parasite_axes import HostAxes

obs = [["01_S1", 3.88, 0.14, 1970, 63],
       ["01_S4", 5.6, 0.82, 1622, 150],
       ["02_S1", 2.4, 0.54, 1570, 40],
       ["03_S1", 4.1, 0.62, 2380, 170]]


fig = plt.figure()

ax_kms = fig.add_subplot(axes_class=HostAxes, aspect=1)

# angular proper motion("/yr) to linear velocity(km/s) at distance=2.3kpc
pm_to_kms = 1./206265.*2300*3.085e18/3.15e7/1.e5

aux_trans = mtransforms.Affine2D().scale(pm_to_kms, 1.)
ax_pm = ax_kms.twin(aux_trans)
```

(continues on next page)

(continued from previous page)

```
for n, ds, dse, w, we in obs:
    time = ((2007 + (10. + 4/30.)/12) - 1988.5)
    v = ds / time * pm_to_kms
    ve = dse / time * pm_to_kms
    ax_kms.errorbar([v], [w], xerr=[ve], yerr=[we], color="k")


ax_kms.axis["bottom"].set_label("Linear velocity at 2.3 kpc [km/s]")
ax_kms.axis["left"].set_label("FWHM [km/s]")
ax_pm.axis["top"].set_label(r"Proper Motion [$''$/yr]")
ax_pm.axis["top"].label.set_visible(True)
ax_pm.axis["right"].major_ticklabels.set_visible(False)

ax_kms.set_xlim(950, 3700)
ax_kms.set_ylim(950, 3100)
# xlim and ylim of ax_pms will be automatically adjusted.

plt.show()
```

### Scatter Histogram (Locatable Axes)

Show the marginal distributions of a scatter plot as histograms at the sides of the plot.

For a nice alignment of the main Axes with the marginals, the Axes positions are defined by a `Divider`, produced via *`make_axes_locatable`*. Note that the `Divider` API allows setting Axes sizes and pads in inches, which is its main feature.

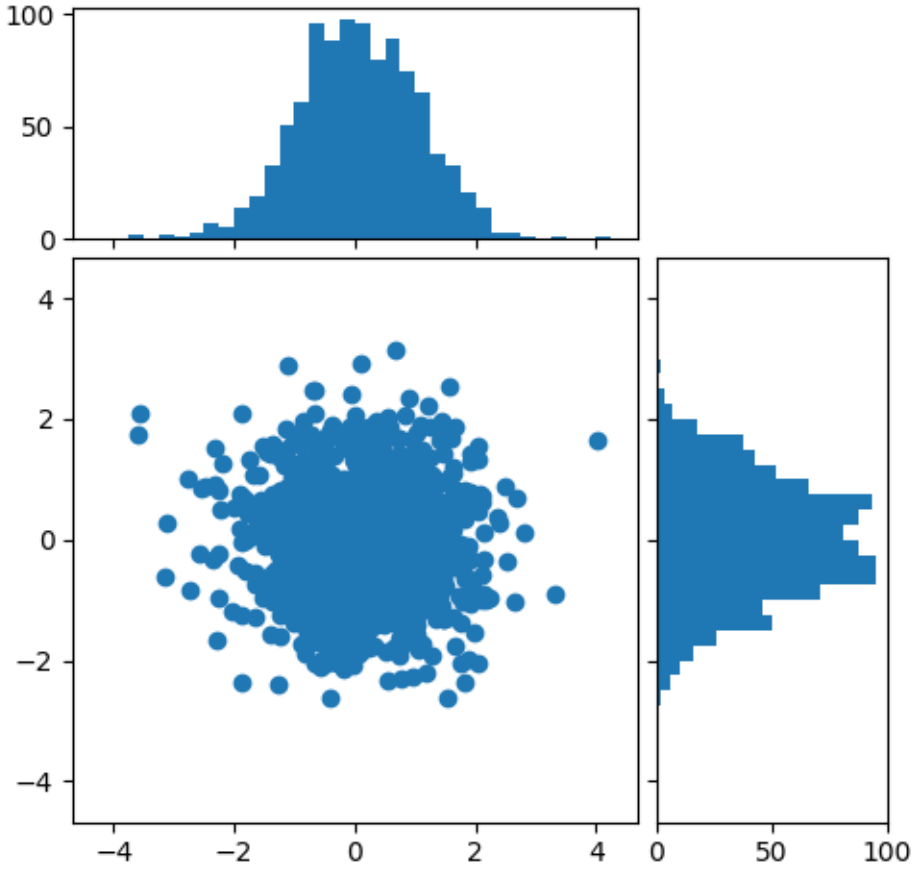
If one wants to set Axes sizes and pads relative to the main Figure, see the *Scatter plot with histograms* example.

```
import matplotlib.pyplot as plt
import numpy as np

from mpl_toolkits.axes_grid1 import make_axes_locatable

# Fixing random state for reproducibility
np.random.seed(19680801)

# the random data
x = np.random.randn(1000)
y = np.random.randn(1000)


fig, ax = plt.subplots(figsize=(5.5, 5.5))

# the scatter plot:
ax.scatter(x, y)

# Set aspect of the main Axes.
ax.set_aspect(1.)
```

(continues on next page)

(continued from previous page)

```
# create new Axes on the right and on the top of the current Axes
divider = make_axes_locatable(ax)
# below height and pad are in inches
ax_histx = divider.append_axes("top", 1.2, pad=0.1, sharex=ax)
ax_histy = divider.append_axes("right", 1.2, pad=0.1, sharey=ax)

# make some labels invisible
ax_histx.xaxis.set_tick_params(labelbottom=False)
ax_histy.yaxis.set_tick_params(labelleft=False)

# now determine nice limits by hand:
binwidth = 0.25
xymax = max(np.max(np.abs(x)), np.max(np.abs(y)))
lim = (int(xymax/binwidth) + 1)*binwidth

bins = np.arange(-lim, lim + binwidth, binwidth)
ax_histx.hist(x, bins=bins)
ax_histy.hist(y, bins=bins, orientation='horizontal')

# the xaxis of ax_histx and yaxis of ax_histy are shared with ax,
# thus there is no need to manually adjust the xlim and ylim of these
# axis.

ax_histx.set_yticks([0, 50, 100])
ax_histy.set_xticks([0, 50, 100])

plt.show()
```



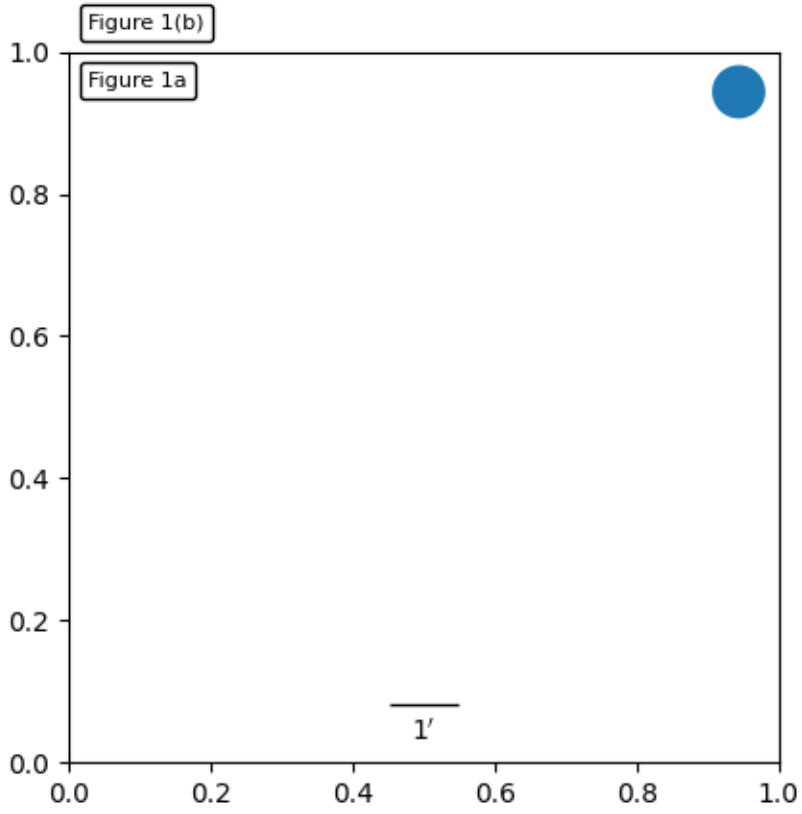

**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *mpl_toolkits.axes_grid1.axes_divider.make_axes_locatable*
- *matplotlib.axes.Axes.set_aspect*
- *matplotlib.axes.Axes.scatter*
- *matplotlib.axes.Axes.hist*

### Simple Anchored Artists

This example illustrates the use of the anchored helper classes found in *matplotlib.offsetbox* and in *mpl_toolkits.axes_grid1*. An implementation of a similar figure, but without use of the toolkit, can be found in *Anchored Artists*.




```
import matplotlib.pyplot as plt


def draw_text(ax):
    """
    Draw two text-boxes, anchored by different corners to the upper-left
    corner of the figure.
    """
    from matplotlib.offsetbox import AnchoredText
    at = AnchoredText("Figure 1a",
                      loc='upper left', prop=dict(size=8), frameon=True,
                      )
    at.patch.set_boxstyle("round,pad=0.,rounding_size=0.2")
    ax.add_artist(at)

    at2 = AnchoredText("Figure 1(b)",
                       loc='lower left', prop=dict(size=8), frameon=True,
```

(continues on next page)

(continued from previous page)

```
                        bbox_to_anchor=(0., 1.),
                        bbox_transform=ax.transAxes
                        )
    at2.patch.set_boxstyle("round,pad=0.,rounding_size=0.2")
    ax.add_artist(at2)


def draw_circle(ax):
    """
    Draw a circle in axis coordinates
    """
    from matplotlib.patches import Circle
    from mpl_toolkits.axes_grid1.anchored_artists import AnchoredDrawingArea
    ada = AnchoredDrawingArea(20, 20, 0, 0,
                              loc='upper right', pad=0., frameon=False)
    p = Circle((10, 10), 10)
    ada.da.add_artist(p)
    ax.add_artist(ada)


def draw_sizebar(ax):
    """
    Draw a horizontal bar with length of 0.1 in data coordinates,
    with a fixed label underneath.
    """
    from mpl_toolkits.axes_grid1.anchored_artists import AnchoredSizeBar
    asb = AnchoredSizeBar(ax.transData,
                          0.1,
                          r"1$^{\prime}$",
                          loc='lower center',
                          pad=0.1, borderpad=0.5, sep=5,
                          frameon=False)
    ax.add_artist(asb)


fig, ax = plt.subplots()
ax.set_aspect(1.)

draw_text(ax)
draw_circle(ax)
draw_sizebar(ax)

plt.show()
```

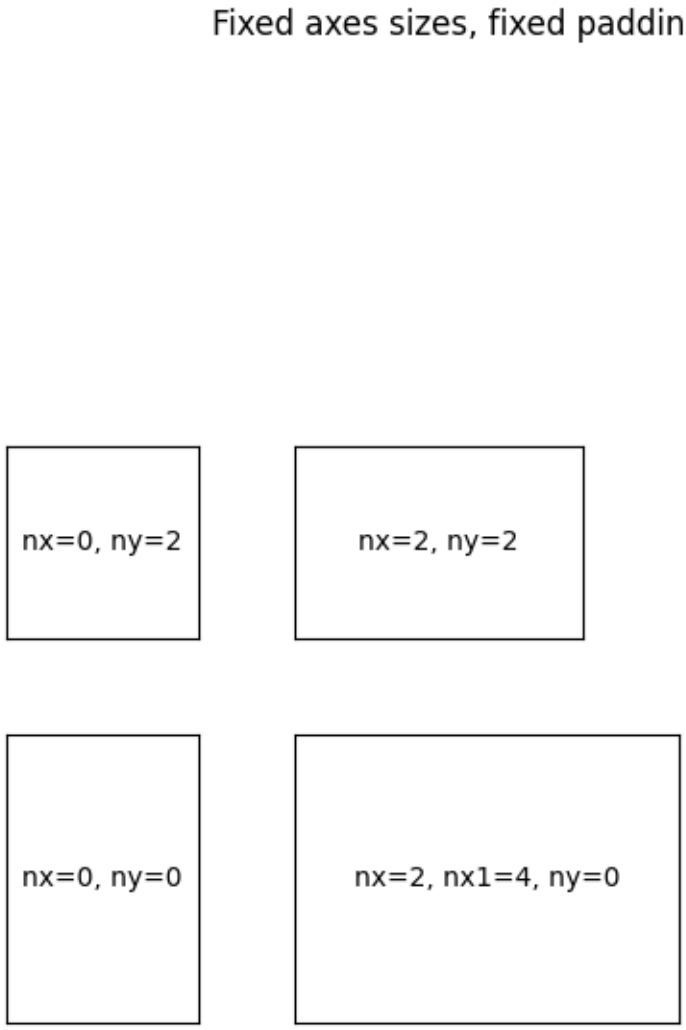
### Simple Axes Divider 1

See also *The axes_grid1 toolkit*.

```
import matplotlib.pyplot as plt

from mpl_toolkits.axes_grid1 import Divider, Size


def label_axes(ax, text):
    """Place a label at the center of an Axes, and remove the axis ticks."""
    ax.text(.5, .5, text, transform=ax.transAxes,
            horizontalalignment="center", verticalalignment="center")
    ax.tick_params(bottom=False, labelbottom=False,
                   left=False, labelleft=False)
```

Fixed Axes sizes; fixed paddings.

```
fig = plt.figure(figsize=(6, 6))
fig.suptitle("Fixed axes sizes, fixed paddings")

# Sizes are in inches.
horiz = [Size.Fixed(1.), Size.Fixed(.5), Size.Fixed(1.5), Size.Fixed(.5)]
vert = [Size.Fixed(1.5), Size.Fixed(.5), Size.Fixed(1.)]

rect = (0.1, 0.1, 0.8, 0.8)
# Divide the Axes rectangle into a grid with sizes specified by horiz * vert.
div = Divider(fig, rect, horiz, vert, aspect=False)

# The rect parameter will actually be ignored and overridden by axes_locator.
ax1 = fig.add_axes(rect, axes_locator=div.new_locator(nx=0, ny=0))
label_axes(ax1, "nx=0, ny=0")
ax2 = fig.add_axes(rect, axes_locator=div.new_locator(nx=0, ny=2))
label_axes(ax2, "nx=0, ny=2")
ax3 = fig.add_axes(rect, axes_locator=div.new_locator(nx=2, ny=2))
label_axes(ax3, "nx=2, ny=2")
ax4 = fig.add_axes(rect, axes_locator=div.new_locator(nx=2, nx1=4, ny=0))
label_axes(ax4, "nx=2, nx1=4, ny=0")
```



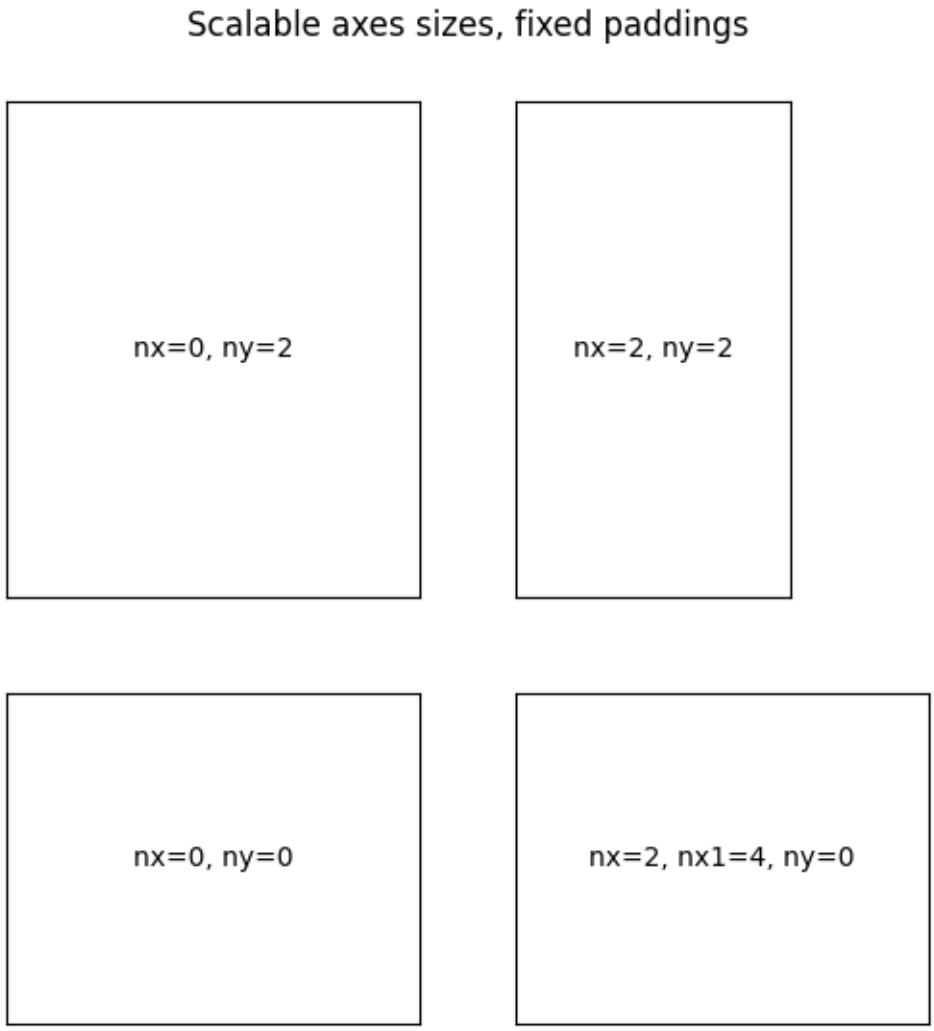

Axes sizes that scale with the figure size; fixed paddings.

```
fig = plt.figure(figsize=(6, 6))
fig.suptitle("Scalable axes sizes, fixed paddings")

horiz = [Size.Scaled(1.5), Size.Fixed(.5), Size.Scaled(1.), Size.Scaled(.5)]
vert = [Size.Scaled(1.), Size.Fixed(.5), Size.Scaled(1.5)]

rect = (0.1, 0.1, 0.8, 0.8)
# Divide the Axes rectangle into a grid with sizes specified by horiz * vert.
div = Divider(fig, rect, horiz, vert, aspect=False)

# The rect parameter will actually be ignored and overridden by axes_locator.
ax1 = fig.add_axes(rect, axes_locator=div.new_locator(nx=0, ny=0))
label_axes(ax1, "nx=0, ny=0")
ax2 = fig.add_axes(rect, axes_locator=div.new_locator(nx=0, ny=2))
```

(continues on next page)

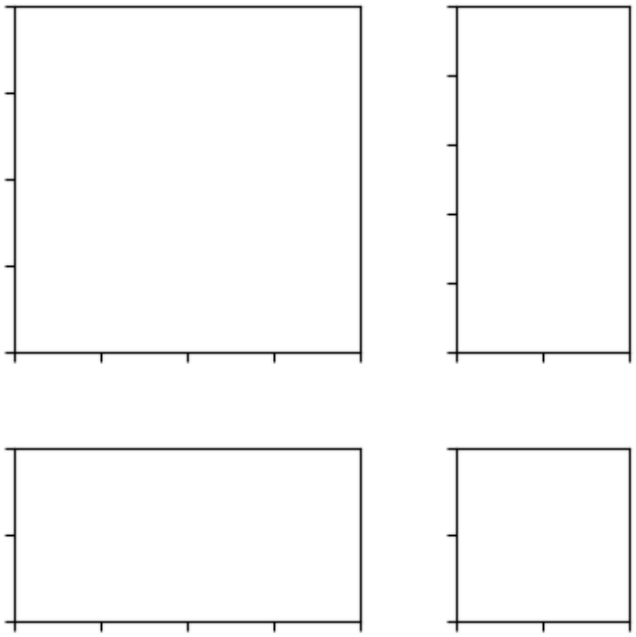
(continued from previous page)

```
label_axes(ax2, "nx=0, ny=2")
ax3 = fig.add_axes(rect, axes_locator=div.new_locator(nx=2, ny=2))
label_axes(ax3, "nx=2, ny=2")
ax4 = fig.add_axes(rect, axes_locator=div.new_locator(nx=2, nx1=4, ny=0))
label_axes(ax4, "nx=2, nx1=4, ny=0")

plt.show()
```



### Simple axes divider 3

See also *The axes_grid1 toolkit*.



```
import matplotlib.pyplot as plt

from mpl_toolkits.axes_grid1 import Divider
import mpl_toolkits.axes_grid1.axes_size as Size

fig = plt.figure(figsize=(5.5, 4))

# the rect parameter will be ignored as we will set axes_locator
rect = (0.1, 0.1, 0.8, 0.8)
ax = [fig.add_axes(rect, label="%d" % i) for i in range(4)]


horiz = [Size.AxesX(ax[0]), Size.Fixed(.5), Size.AxesX(ax[1])]
vert = [Size.AxesY(ax[0]), Size.Fixed(.5), Size.AxesY(ax[2])]

# divide the Axes rectangle into grid whose size is specified by horiz * vert
divider = Divider(fig, rect, horiz, vert, aspect=False)


ax[0].set_axes_locator(divider.new_locator(nx=0, ny=0))
ax[1].set_axes_locator(divider.new_locator(nx=2, ny=0))
ax[2].set_axes_locator(divider.new_locator(nx=0, ny=2))
ax[3].set_axes_locator(divider.new_locator(nx=2, ny=2))

```

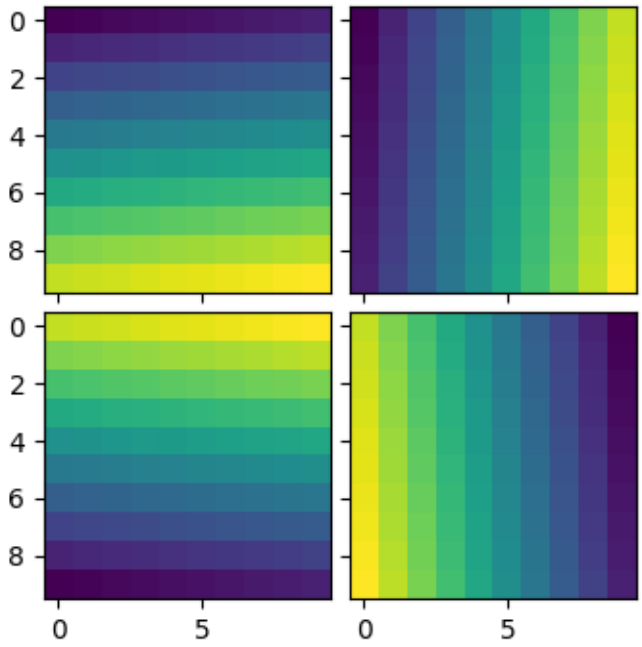
(continues on next page)

(continued from previous page)

```
ax[0].set_xlim(0, 2)
ax[1].set_xlim(0, 1)

ax[0].set_ylim(0, 1)
ax[2].set_ylim(0, 2)

divider.set_aspect(1.)

for ax1 in ax:
    ax1.tick_params(labelbottom=False, labelleft=False)

plt.show()
```

### Simple ImageGrid

Align multiple images using *ImageGrid*.




```
import matplotlib.pyplot as plt
import numpy as np

from mpl_toolkits.axes_grid1 import ImageGrid

im1 = np.arange(100).reshape((10, 10))
im2 = im1.T
im3 = np.flipud(im1)
```

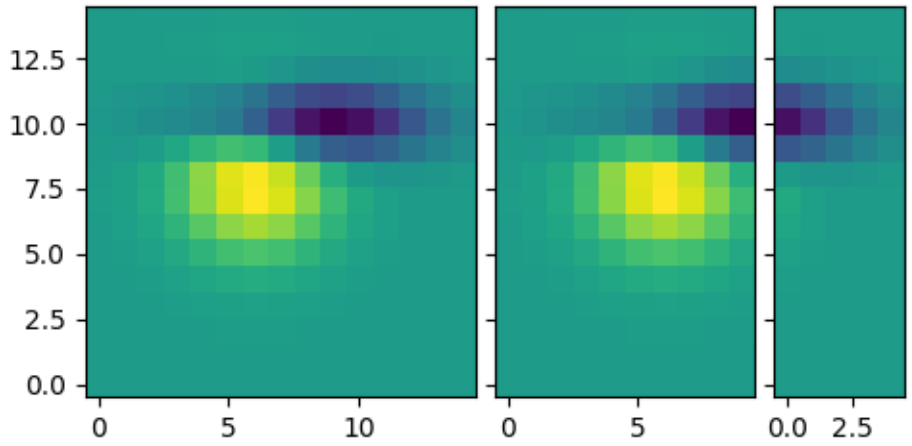
(continues on next page)

(continued from previous page)

```
im4 = np.fliplr(im2)

fig = plt.figure(figsize=(4., 4.))
grid = ImageGrid(fig, 111,  # similar to subplot(111)
                 nrows_ncols=(2, 2),  # creates 2x2 grid of Axes
                 axes_pad=0.1,  # pad between Axes in inch.
                 )

for ax, im in zip(grid, [im1, im2, im3, im4]):
    # Iterating over the grid returns the Axes.
    ax.imshow(im)

plt.show()
```

### Simple ImageGrid 2

Align multiple images of different sizes using *ImageGrid*.



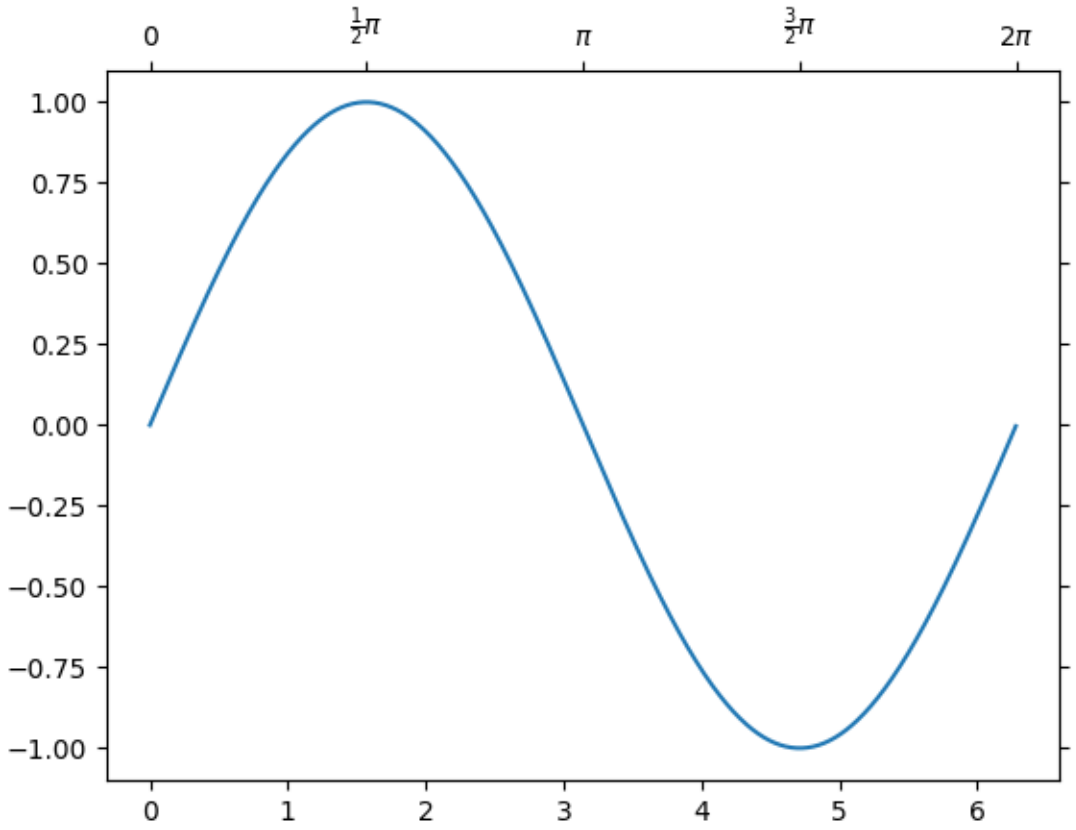

```
import matplotlib.pyplot as plt

from matplotlib import cbook
from mpl_toolkits.axes_grid1 import ImageGrid

fig = plt.figure(figsize=(5.5, 3.5))
grid = ImageGrid(fig, 111,  # similar to subplot(111)
                 nrows_ncols=(1, 3),
                 axes_pad=0.1,
                 label_mode="L",
```

(continues on next page)

(continued from previous page)

```
                  )

# demo image
Z = cbook.get_sample_data("axes_grid/bivariate_normal.npy")
im1 = Z
im2 = Z[:, :10]
im3 = Z[:, 10:]
vmin, vmax = Z.min(), Z.max()
for ax, im in zip(grid, [im1, im2, im3]):
    ax.imshow(im, origin="lower", vmin=vmin, vmax=vmax)

plt.show()
```

### Simple Axisline4




```
import matplotlib.pyplot as plt
import numpy as np

from mpl_toolkits.axes_grid1 import host_subplot
```

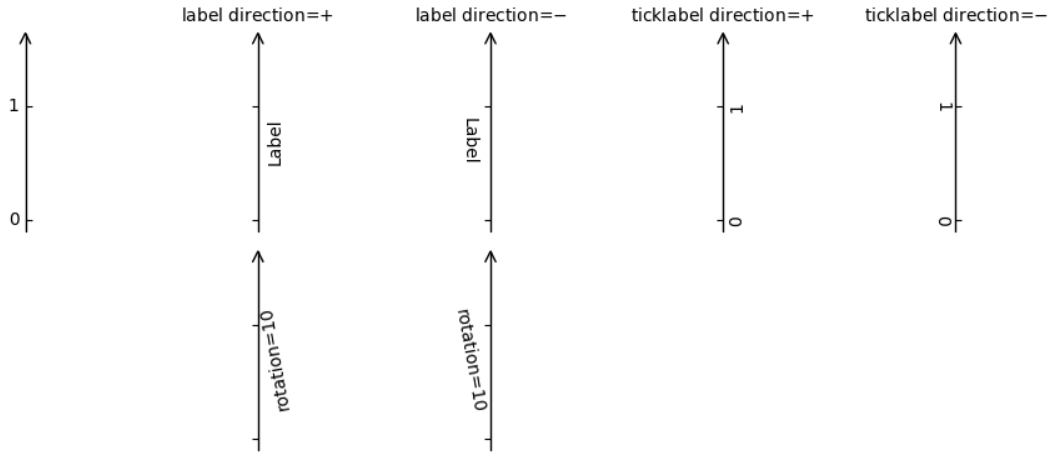
(continues on next page)

(continued from previous page)

```
ax = host_subplot(111)
xx = np.arange(0, 2*np.pi, 0.01)
ax.plot(xx, np.sin(xx))

ax2 = ax.twin()  # ax2 is responsible for "top" axis and "right" axis
ax2.set_xticks([0., .5*np.pi, np.pi, 1.5*np.pi, 2*np.pi],
               labels=["$0$", r"$\frac{1}{2}\pi$",
                       r"$\pi$", r"$\frac{3}{2}\pi$", r"$2\pi$"])

ax2.axis["right"].major_ticklabels.set_visible(False)
ax2.axis["top"].major_ticklabels.set_visible(True)

plt.show()
```

### 6.25.12 Module - axisartist

#### Axis Direction



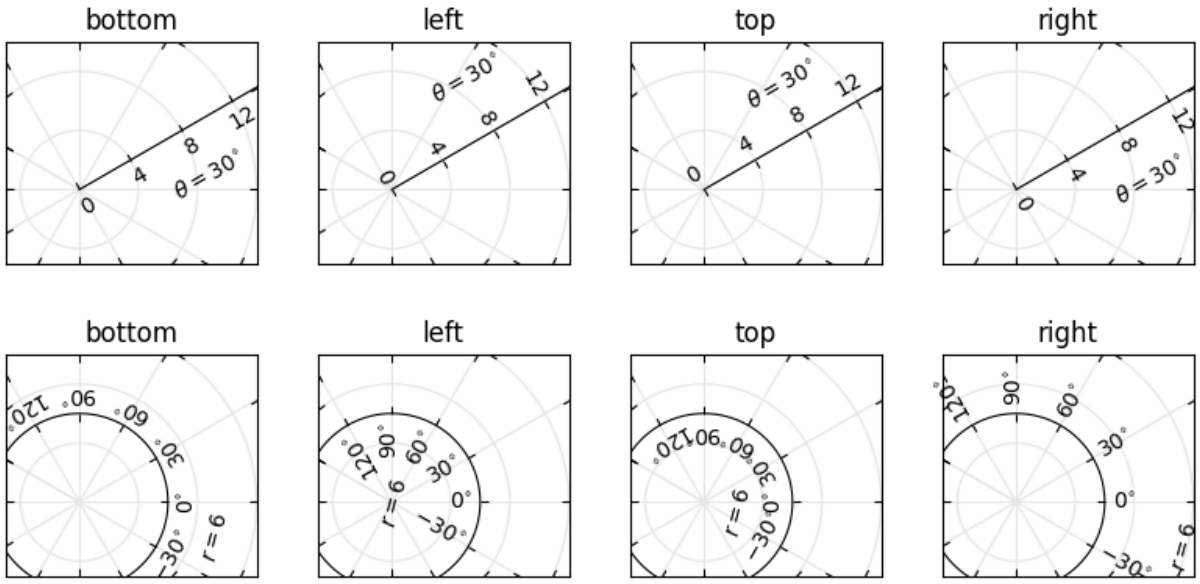

```
import matplotlib.pyplot as plt

import mpl_toolkits.axisartist as axisartist


def setup_axes(fig, pos):
    ax = fig.add_subplot(pos, axes_class=axisartist.Axes)

    ax.set_ylim(-0.1, 1.5)
    ax.set_yticks([0, 1])

    ax.axis[:].set_visible(False)

    ax.axis["x"] = ax.new_floating_axis(1, 0.5)
    ax.axis["x"].set_axisline_style("->", size=1.5)
```

(continues on next page)

(continued from previous page)

```
    return ax


plt.rcParams.update({
    "axes.titlesize": "medium",
    "axes.titley": 1.1,
})

fig = plt.figure(figsize=(10, 4))
fig.subplots_adjust(bottom=0.1, top=0.9, left=0.05, right=0.95)

ax1 = setup_axes(fig, 251)
ax1.axis["x"].set_axis_direction("left")

ax2 = setup_axes(fig, 252)
ax2.axis["x"].label.set_text("Label")
ax2.axis["x"].toggle(ticklabels=False)
ax2.axis["x"].set_axislabel_direction("+")
ax2.set_title("label direction=$+$")

ax3 = setup_axes(fig, 253)
ax3.axis["x"].label.set_text("Label")
ax3.axis["x"].toggle(ticklabels=False)
ax3.axis["x"].set_axislabel_direction("-")
ax3.set_title("label direction=$-$")

ax4 = setup_axes(fig, 254)
ax4.axis["x"].set_ticklabel_direction("+")
ax4.set_title("ticklabel direction=$+$")

ax5 = setup_axes(fig, 255)
ax5.axis["x"].set_ticklabel_direction("-")
ax5.set_title("ticklabel direction=$-$")

ax7 = setup_axes(fig, 257)
ax7.axis["x"].label.set_text("rotation=10")
ax7.axis["x"].label.set_rotation(10)
ax7.axis["x"].toggle(ticklabels=False)

ax8 = setup_axes(fig, 258)
ax8.axis["x"].set_axislabel_direction("-")
ax8.axis["x"].label.set_text("rotation=10")
ax8.axis["x"].label.set_rotation(10)
ax8.axis["x"].toggle(ticklabels=False)

plt.show()
```

### axis_direction demo




```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.projections import PolarAxes
from matplotlib.transforms import Affine2D
import mpl_toolkits.axisartist as axisartist
import mpl_toolkits.axisartist.angle_helper as angle_helper
import mpl_toolkits.axisartist.grid_finder as grid_finder
from mpl_toolkits.axisartist.grid_helper_curvelinear import \
    GridHelperCurveLinear


def setup_axes(fig, rect):
    """Polar projection, but in a rectangular box."""
    # see demo_curvelinear_grid.py for details
    grid_helper = GridHelperCurveLinear(
        (
            Affine2D().scale(np.pi/180., 1.) +
            PolarAxes.PolarTransform(apply_theta_transforms=False)
        ),
        extreme_finder=angle_helper.ExtremeFinderCycle(
            20, 20,
            lon_cycle=360, lat_cycle=None,
            lon_minmax=None, lat_minmax=(0, np.inf),
        ),
        grid_locator1=angle_helper.LocatorDMS(12),
        grid_locator2=grid_finder.MaxNLocator(5),
        tick_formatter1=angle_helper.FormatterDMS(),
    )
    ax = fig.add_subplot(
```

(continues on next page)

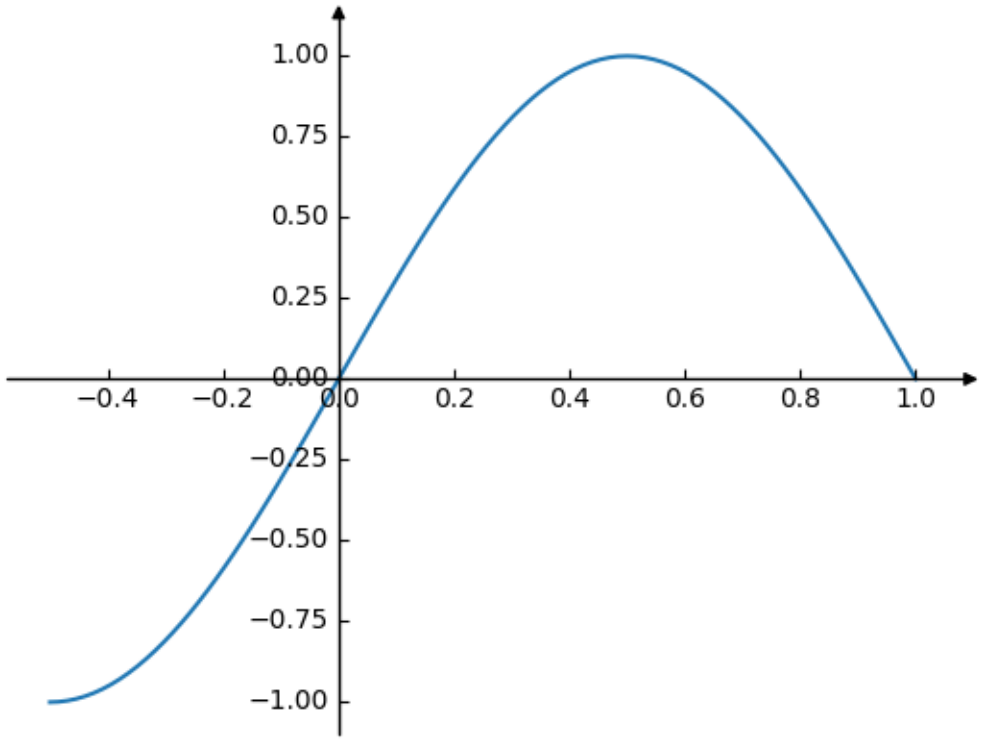
(continued from previous page)

```
        rect, axes_class=axisartist.Axes, grid_helper=grid_helper,
        aspect=1, xlim=(-5, 12), ylim=(-5, 10))
    ax.axis[:].toggle(ticklabels=False)
    ax.grid(color=".9")
    return ax


def add_floating_axis1(ax):
    ax.axis["lat"] = axis = ax.new_floating_axis(0, 30)
    axis.label.set_text(r"$\theta = 30^{\circ}$")
    axis.label.set_visible(True)
    return axis


def add_floating_axis2(ax):
    ax.axis["lon"] = axis = ax.new_floating_axis(1, 6)
    axis.label.set_text(r"$r = 6$")
    axis.label.set_visible(True)
    return axis


fig = plt.figure(figsize=(8, 4), layout="constrained")

for i, d in enumerate(["bottom", "left", "top", "right"]):
    ax = setup_axes(fig, rect=241+i)
    axis = add_floating_axis1(ax)
    axis.set_axis_direction(d)
    ax.set(title=d)

for i, d in enumerate(["bottom", "left", "top", "right"]):
    ax = setup_axes(fig, rect=245+i)
    axis = add_floating_axis2(ax)
    axis.set_axis_direction(d)
    ax.set(title=d)

plt.show()
```

### Axis line styles

This example shows some configurations for axis style.

Note: The *mpl_toolkits.axisartist* Axes classes may be confusing for new users. If the only aim is to obtain arrow heads at the ends of the axes, rather check out the *Centered spines with arrows* example.

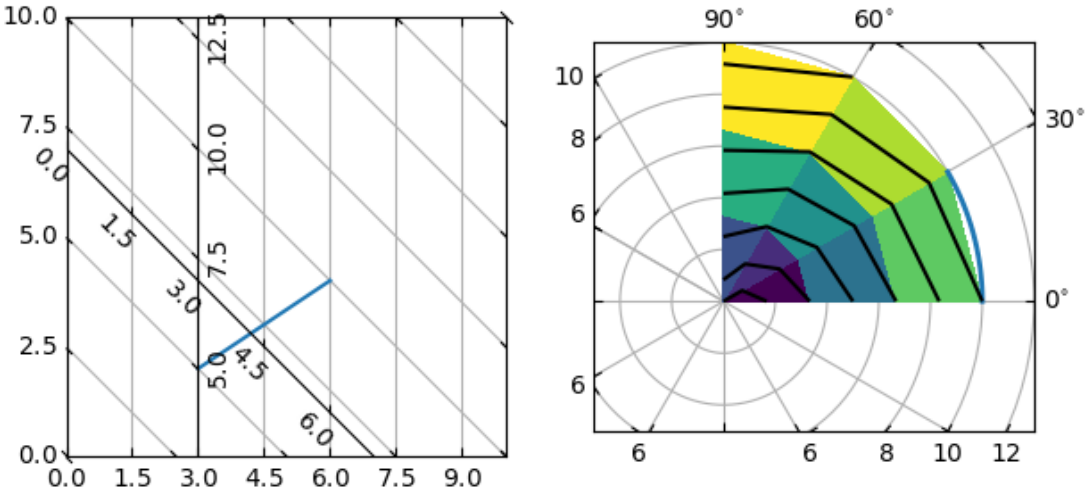



```
import matplotlib.pyplot as plt
import numpy as np

from mpl_toolkits.axisartist.axislines import AxesZero

fig = plt.figure()
ax = fig.add_subplot(axes_class=AxesZero)

for direction in ["xzero", "yzero"]:
    # adds arrows at the ends of each axis
    ax.axis[direction].set_axisline_style("-|>")

    # adds X and Y-axis from the origin
    ax.axis[direction].set_visible(True)

for direction in ["left", "right", "bottom", "top"]:
    # hides borders
    ax.axis[direction].set_visible(False)

x = np.linspace(-0.5, 1., 100)
ax.plot(x, np.sin(x*np.pi))

plt.show()
```

### Curvilinear grid demo

Custom grid and ticklines.

This example demonstrates how to use *GridHelperCurveLinear* to define custom grids and ticklines by applying a transformation on the grid. This can be used, as shown on the second plot, to create polar projections in a rectangular box.




```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.projections import PolarAxes
from matplotlib.transforms import Affine2D
from mpl_toolkits.axisartist import Axes, HostAxes, angle_helper
from mpl_toolkits.axisartist.grid_helper_curvelinear import \
    GridHelperCurveLinear


def curvelinear_test1(fig):
    """
    Grid for custom transform.
    """

    def tr(x, y): return x, y - x
    def inv_tr(x, y): return x, y + x

    grid_helper = GridHelperCurveLinear((tr, inv_tr))

    ax1 = fig.add_subplot(1, 2, 1, axes_class=Axes, grid_helper=grid_helper)
    # ax1 will have ticks and gridlines defined by the given transform (+
```

(continues on next page)

(continued from previous page)

```
    # transData of the Axes).  Note that the transform of the Axes itself
    # (i.e., transData) is not affected by the given transform.
    xx, yy = tr(np.array([3, 6]), np.array([5, 10]))
    ax1.plot(xx, yy)

    ax1.set_aspect(1)
    ax1.set_xlim(0, 10)
    ax1.set_ylim(0, 10)

    ax1.axis["t"] = ax1.new_floating_axis(0, 3)
    ax1.axis["t2"] = ax1.new_floating_axis(1, 7)
    ax1.grid(True, zorder=0)


def curvelinear_test2(fig):
    """
    Polar projection, but in a rectangular box.
    """

    # PolarAxes.PolarTransform takes radian. However, we want our coordinate
    # system in degree
    tr = Affine2D().scale(np.pi/180, 1) + PolarAxes.PolarTransform(
        apply_theta_transforms=False)
    # Polar projection, which involves cycle, and also has limits in
    # its coordinates, needs a special method to find the extremes
    # (min, max of the coordinate within the view).
    extreme_finder = angle_helper.ExtremeFinderCycle(
        nx=20, ny=20,  # Number of sampling points in each direction.
        lon_cycle=360, lat_cycle=None,
        lon_minmax=None, lat_minmax=(0, np.inf),
    )
    # Find grid values appropriate for the coordinate (degree, minute,␣
 ↪second).
    grid_locator1 = angle_helper.LocatorDMS(12)
    # Use an appropriate formatter.  Note that the acceptable Locator and
    # Formatter classes are a bit different than that of Matplotlib, which
    # cannot directly be used here (this may be possible in the future).
    tick_formatter1 = angle_helper.FormatterDMS()

    grid_helper = GridHelperCurveLinear(
        tr, extreme_finder=extreme_finder,
        grid_locator1=grid_locator1, tick_formatter1=tick_formatter1)
    ax1 = fig.add_subplot(
        1, 2, 2, axes_class=HostAxes, grid_helper=grid_helper)

    # make ticklabels of right and top axis visible.
    ax1.axis["right"].major_ticklabels.set_visible(True)
    ax1.axis["top"].major_ticklabels.set_visible(True)
    # let right axis shows ticklabels for 1st coordinate (angle)
    ax1.axis["right"].get_helper().nth_coord_ticks = 0
    # let bottom axis shows ticklabels for 2nd coordinate (radius)
    ax1.axis["bottom"].get_helper().nth_coord_ticks = 1
```

(continues on next page)

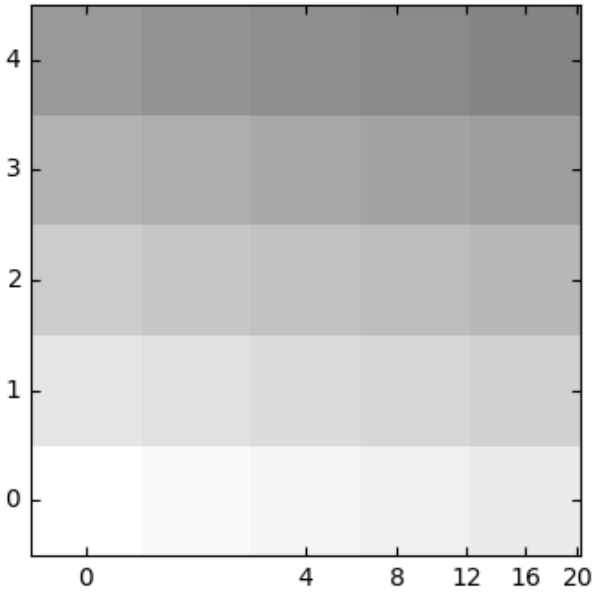
(continued from previous page)

```
    ax1.set_aspect(1)
    ax1.set_xlim(-5, 12)
    ax1.set_ylim(-5, 10)

    ax1.grid(True, zorder=0)

    # A parasite Axes with given transform
    ax2 = ax1.get_aux_axes(tr)
    # note that ax2.transData == tr + ax1.transData
    # Anything you draw in ax2 will match the ticks and grids of ax1.
    ax2.plot(np.linspace(0, 30, 51), np.linspace(10, 10, 51), linewidth=2)

    ax2.pcolor(np.linspace(0, 90, 4), np.linspace(0, 10, 4),
               np.arange(9).reshape((3, 3)))
    ax2.contour(np.linspace(0, 90, 4), np.linspace(0, 10, 4),
                np.arange(16).reshape((4, 4)), colors="k")


if __name__ == "__main__":
    fig = plt.figure(figsize=(7, 4))

    curvelinear_test1(fig)
    curvelinear_test2(fig)

    plt.show()
```

### Demo CurveLinear Grid2

Custom grid and ticklines.

This example demonstrates how to use GridHelperCurveLinear to define custom grids and ticklines by applying a transformation on the grid. As showcase on the plot, a 5x5 matrix is displayed on the Axes.




```
import matplotlib.pyplot as plt
import numpy as np

from mpl_toolkits.axisartist.axislines import Axes
from mpl_toolkits.axisartist.grid_finder import (ExtremeFinderSimple,
                                                 MaxNLocator)
from mpl_toolkits.axisartist.grid_helper_curvelinear import \
    GridHelperCurveLinear


def curvelinear_test1(fig):
    """Grid for custom transform."""

    def tr(x, y):
        return np.sign(x)*abs(x)**.5, y

    def inv_tr(x, y):
        return np.sign(x)*x**2, y

    grid_helper = GridHelperCurveLinear(
        (tr, inv_tr),
        extreme_finder=ExtremeFinderSimple(20, 20),
        # better tick density
        grid_locator1=MaxNLocator(nbins=6), grid_
 ↪locator2=MaxNLocator(nbins=6))

    ax1 = fig.add_subplot(axes_class=Axes, grid_helper=grid_helper)
    # ax1 will have a ticks and gridlines defined by the given
    # transform (+ transData of the Axes). Note that the transform of the Axes
    # itself (i.e., transData) is not affected by the given transform.
```

(continues on next page)

(continued from previous page)

```
    ax1.imshow(np.arange(25).reshape(5, 5),
               vmax=50, cmap=plt.cm.gray_r, origin="lower")


if __name__ == "__main__":
    fig = plt.figure(figsize=(7, 4))
    curvelinear_test1(fig)
    plt.show()
```

### `floating_axes` features

Demonstration of features of the *floating_axes* module:

- Using *scatter* and *bar* with changing the shape of the plot.
- Using *GridHelperCurveLinear* to rotate the plot and set the plot boundary.
- Using *add_subplot* to create a subplot using the return value from *GridHelperCurveLinear*.
- Making a sector plot by adding more features to *GridHelperCurveLinear*.

```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.projections import PolarAxes
from matplotlib.transforms import Affine2D
import mpl_toolkits.axisartist.angle_helper as angle_helper
import mpl_toolkits.axisartist.floating_axes as floating_axes
from mpl_toolkits.axisartist.grid_finder import (DictFormatter, FixedLocator,
                                                 MaxNLocator)

# Fixing random state for reproducibility
np.random.seed(19680801)


def setup_axes1(fig, rect):
    """
    A simple one.
    """
    tr = Affine2D().scale(2, 1).rotate_deg(30)

    grid_helper = floating_axes.GridHelperCurveLinear(
        tr, extremes=(-0.5, 3.5, 0, 4),
        grid_locator1=MaxNLocator(nbins=4),
        grid_locator2=MaxNLocator(nbins=4))

    ax1 = fig.add_subplot(
        rect, axes_class=floating_axes.FloatingAxes, grid_helper=grid_helper)
    ax1.grid()
```

(continues on next page)

(continued from previous page)

```
    aux_ax = ax1.get_aux_axes(tr)

    return ax1, aux_ax


def setup_axes2(fig, rect):
    """
    With custom locator and formatter.
    Note that the extreme values are swapped.
    """
    tr = PolarAxes.PolarTransform(apply_theta_transforms=False)

    pi = np.pi
    angle_ticks = [(0, r"$0$"),
                   (.25*pi, r"$\frac{1}{4}\pi$"),
                   (.5*pi, r"$\frac{1}{2}\pi$")]
    grid_locator1 = FixedLocator([v for v, s in angle_ticks])
    tick_formatter1 = DictFormatter(dict(angle_ticks))

    grid_locator2 = MaxNLocator(2)

    grid_helper = floating_axes.GridHelperCurveLinear(
        tr, extremes=(.5*pi, 0, 2, 1),
        grid_locator1=grid_locator1,
        grid_locator2=grid_locator2,
        tick_formatter1=tick_formatter1,
        tick_formatter2=None)

    ax1 = fig.add_subplot(
        rect, axes_class=floating_axes.FloatingAxes, grid_helper=grid_helper)
    ax1.grid()

    # create a parasite Axes whose transData in RA, cz
    aux_ax = ax1.get_aux_axes(tr)

    aux_ax.patch = ax1.patch  # for aux_ax to have a clip path as in ax
    ax1.patch.zorder = 0.9  # but this has a side effect that the patch is
    # drawn twice, and possibly over some other
    # artists. So, we decrease the zorder a bit to
    # prevent this.

    return ax1, aux_ax


def setup_axes3(fig, rect):
    """
    Sometimes, things like axis_direction need to be adjusted.
    """

    # rotate a bit for better orientation
    tr_rotate = Affine2D().translate(-95, 0)
```

(continues on next page)

(continued from previous page)

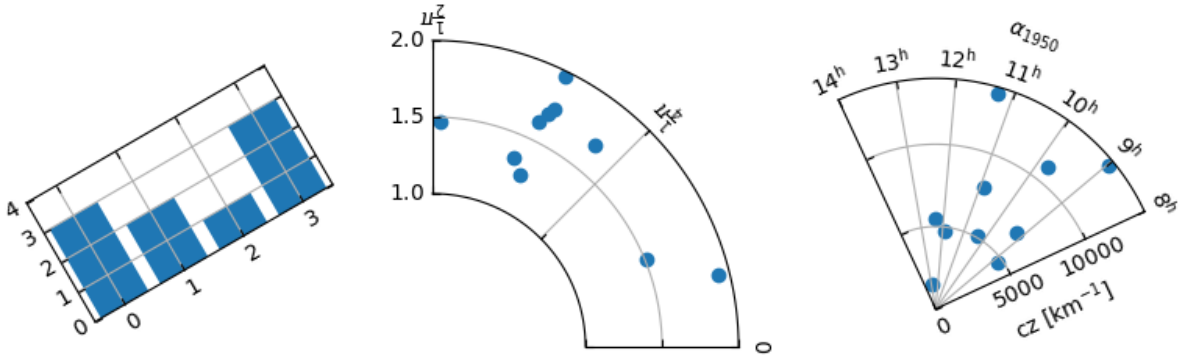
```
    # scale degree to radians
    tr_scale = Affine2D().scale(np.pi/180., 1.)

    tr = tr_rotate + tr_scale + PolarAxes.PolarTransform(
        apply_theta_transforms=False)

    grid_locator1 = angle_helper.LocatorHMS(4)
    tick_formatter1 = angle_helper.FormatterHMS()

    grid_locator2 = MaxNLocator(3)

    # Specify theta limits in degrees
    ra0, ra1 = 8.*15, 14.*15
    # Specify radial limits
    cz0, cz1 = 0, 14000
    grid_helper = floating_axes.GridHelperCurveLinear(
        tr, extremes=(ra0, ra1, cz0, cz1),
        grid_locator1=grid_locator1,
        grid_locator2=grid_locator2,
        tick_formatter1=tick_formatter1,
        tick_formatter2=None)

    ax1 = fig.add_subplot(
        rect, axes_class=floating_axes.FloatingAxes, grid_helper=grid_helper)

    # adjust axis
    ax1.axis["left"].set_axis_direction("bottom")
    ax1.axis["right"].set_axis_direction("top")

    ax1.axis["bottom"].set_visible(False)
    ax1.axis["top"].set_axis_direction("bottom")
    ax1.axis["top"].toggle(ticklabels=True, label=True)
    ax1.axis["top"].major_ticklabels.set_axis_direction("top")
    ax1.axis["top"].label.set_axis_direction("top")

    ax1.axis["left"].label.set_text(r"cz [km$^{-1}$]")
    ax1.axis["top"].label.set_text(r"$\alpha_{1950}$")
    ax1.grid()

    # create a parasite Axes whose transData in RA, cz
    aux_ax = ax1.get_aux_axes(tr)

    aux_ax.patch = ax1.patch  # for aux_ax to have a clip path as in ax
    ax1.patch.zorder = 0.9  # but this has a side effect that the patch is
    # drawn twice, and possibly over some other
    # artists. So, we decrease the zorder a bit to
    # prevent this.

    return ax1, aux_ax
```

```
fig = plt.figure(figsize=(8, 4))
```

(continues on next page)

(continued from previous page)

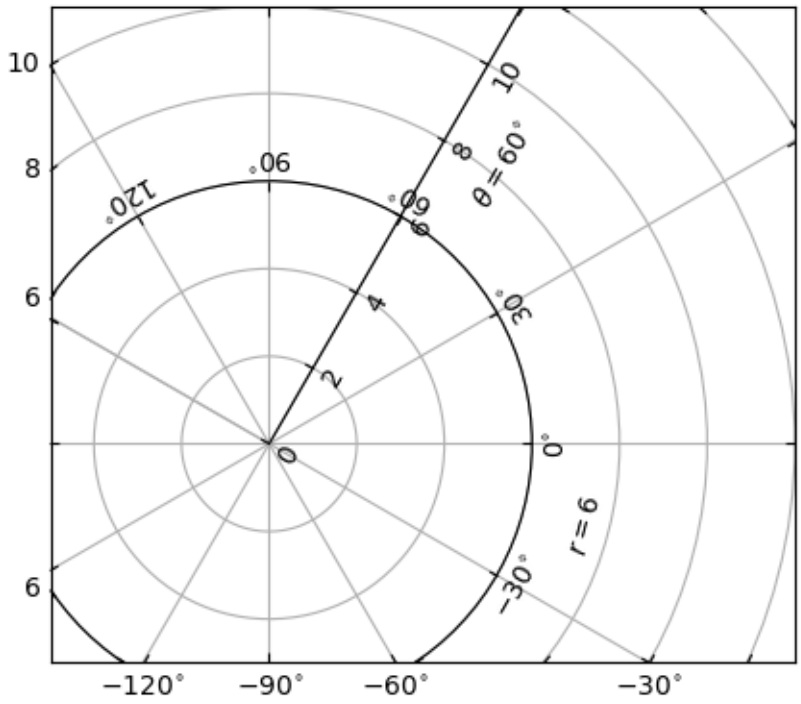
```
fig.subplots_adjust(wspace=0.3, left=0.05, right=0.95)

ax1, aux_ax1 = setup_axes1(fig, 131)
aux_ax1.bar([0, 1, 2, 3], [3, 2, 1, 3])

ax2, aux_ax2 = setup_axes2(fig, 132)
theta = np.random.rand(10)*.5*np.pi
radius = np.random.rand(10) + 1.
aux_ax2.scatter(theta, radius)

ax3, aux_ax3 = setup_axes3(fig, 133)

theta = (8 + np.random.rand(10)*(14 - 8))*15.  # in degrees
radius = np.random.rand(10)*14000.
aux_ax3.scatter(theta, radius)

plt.show()
```



### floating_axis demo

Axis within rectangular frame.

The following code demonstrates how to put a floating polar curve within a rectangular box. In order to get a better sense of polar curves, please look at *Curvilinear grid demo*.




```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.projections import PolarAxes
from matplotlib.transforms import Affine2D
from mpl_toolkits.axisartist import GridHelperCurveLinear, HostAxes
import mpl_toolkits.axisartist.angle_helper as angle_helper


def curvelinear_test2(fig):
    """Polar projection, but in a rectangular box."""
    # see demo_curvelinear_grid.py for details
    tr = Affine2D().scale(np.pi / 180., 1.) + PolarAxes.PolarTransform(
        apply_theta_transforms=False)
```

(continues on next page)

(continued from previous page)

```
    extreme_finder = angle_helper.ExtremeFinderCycle(20,
                                                     20,
                                                     lon_cycle=360,
                                                     lat_cycle=None,
                                                     lon_minmax=None,
                                                     lat_minmax=(0, np.inf),
                                                     )

    grid_locator1 = angle_helper.LocatorDMS(12)

    tick_formatter1 = angle_helper.FormatterDMS()

    grid_helper = GridHelperCurveLinear(tr,
                                        extreme_finder=extreme_finder,
                                        grid_locator1=grid_locator1,
                                        tick_formatter1=tick_formatter1
                                        )

    ax1 = fig.add_subplot(axes_class=HostAxes, grid_helper=grid_helper)

    # Now creates floating axis

    # floating axis whose first coordinate (theta) is fixed at 60
    ax1.axis["lat"] = axis = ax1.new_floating_axis(0, 60)
    axis.label.set_text(r"$\theta = 60^{\circ}$")
    axis.label.set_visible(True)

    # floating axis whose second coordinate (r) is fixed at 6
    ax1.axis["lon"] = axis = ax1.new_floating_axis(1, 6)
    axis.label.set_text(r"$r = 6$")

    ax1.set_aspect(1.)
    ax1.set_xlim(-5, 12)
    ax1.set_ylim(-5, 10)

    ax1.grid(True)


fig = plt.figure(figsize=(5, 5))
curvelinear_test2(fig)
plt.show()
```

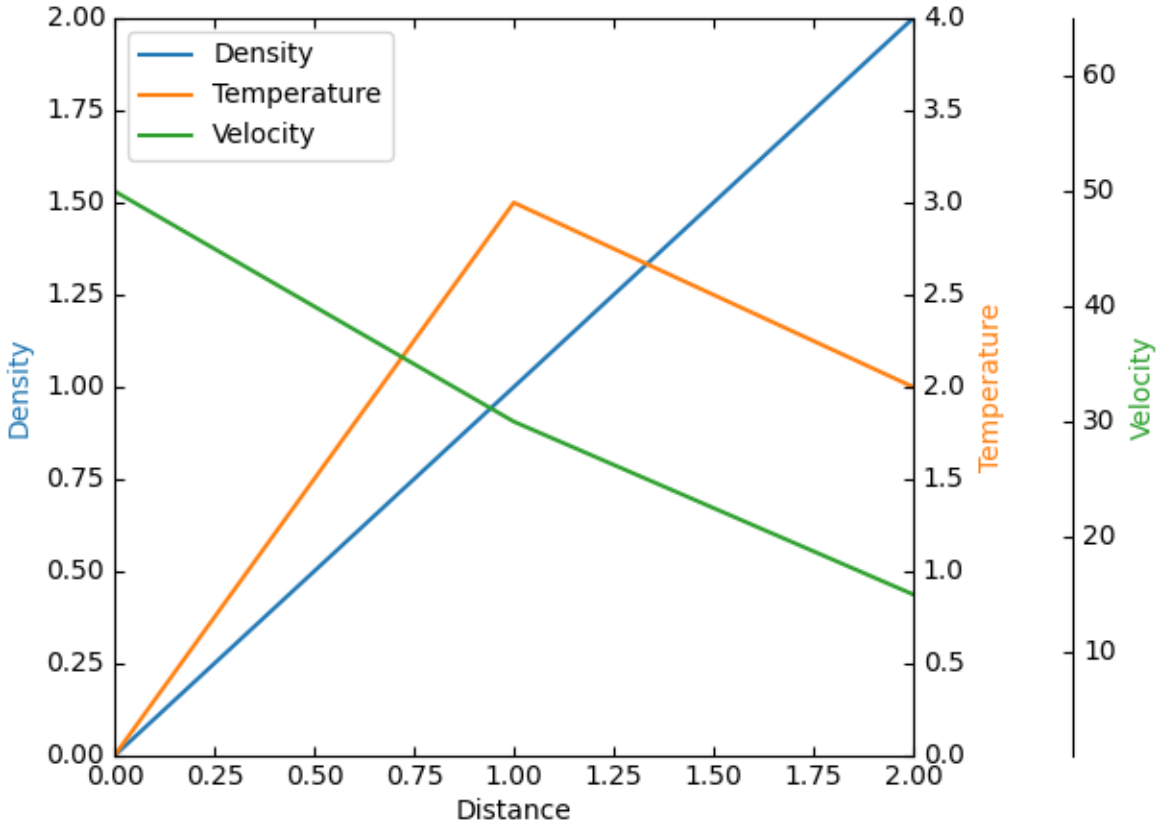
### Parasite Axes demo

Create a parasite Axes. Such Axes would share the x scale with a host Axes, but show a different scale in y direction.

This approach uses `mpl_toolkits.axes_grid1.parasite_axes.HostAxes` and `mpl_toolkits.axes_grid1.parasite_axes.ParasiteAxes`.

The standard and recommended approach is to use instead standard Matplotlib axes, as shown in the *Multiple y-axis with Spines* example.

An alternative approach using `mpl_toolkits.axes_grid1` and `mpl_toolkits.axisartist` is shown in the *Parasite axis demo* example.




```
import matplotlib.pyplot as plt

from mpl_toolkits.axisartist.parasite_axes import HostAxes

fig = plt.figure()

host = fig.add_axes([0.15, 0.1, 0.65, 0.8], axes_class=HostAxes)
par1 = host.get_aux_axes(viewlim_mode=None, sharex=host)
par2 = host.get_aux_axes(viewlim_mode=None, sharex=host)
```

(continues on next page)

(continued from previous page)

```
host.axis["right"].set_visible(False)

par1.axis["right"].set_visible(True)
par1.axis["right"].major_ticklabels.set_visible(True)
par1.axis["right"].label.set_visible(True)

par2.axis["right2"] = par2.new_fixed_axis(loc="right", offset=(60, 0))

p1, = host.plot([0, 1, 2], [0, 1, 2], label="Density")
p2, = par1.plot([0, 1, 2], [0, 3, 2], label="Temperature")
p3, = par2.plot([0, 1, 2], [50, 30, 15], label="Velocity")

host.set(xlim=(0, 2), ylim=(0, 2), xlabel="Distance", ylabel="Density")
par1.set(ylim=(0, 4), ylabel="Temperature")
par2.set(ylim=(1, 65), ylabel="Velocity")

host.legend()

host.axis["left"].label.set_color(p1.get_color())
par1.axis["right"].label.set_color(p2.get_color())
par2.axis["right2"].label.set_color(p3.get_color())

plt.show()
```

### Parasite axis demo

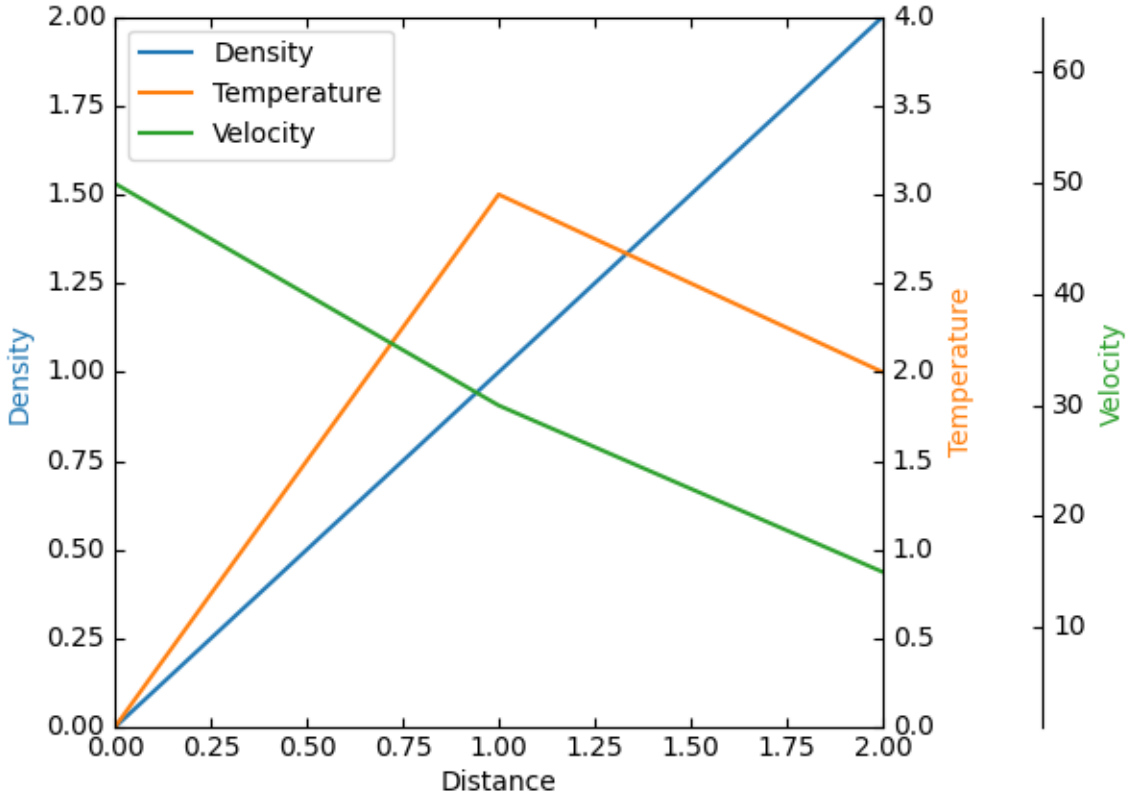
This example demonstrates the use of parasite axis to plot multiple datasets onto one single plot.

Notice how in this example, *par1* and *par2* are both obtained by calling `twinx()`, which ties their x-limits with the host's x-axis. From there, each of those two axis behave separately from each other: different datasets can be plotted, and the y-limits are adjusted separately.

This approach uses *mpl_toolkits.axes_grid1.parasite_axes.host_subplot* and *mpl_toolkits.axisartist.axislines.Axes*.

The standard and recommended approach is to use instead standard Matplotlib axes, as shown in the *Multiple y-axis with Spines* example.

An alternative approach using *mpl_toolkits.axes_grid1.parasite_axes.HostAxes* and *mpl_toolkits.axes_grid1.parasite_axes.ParasiteAxes* is shown in the *Parasite Axes demo* example.



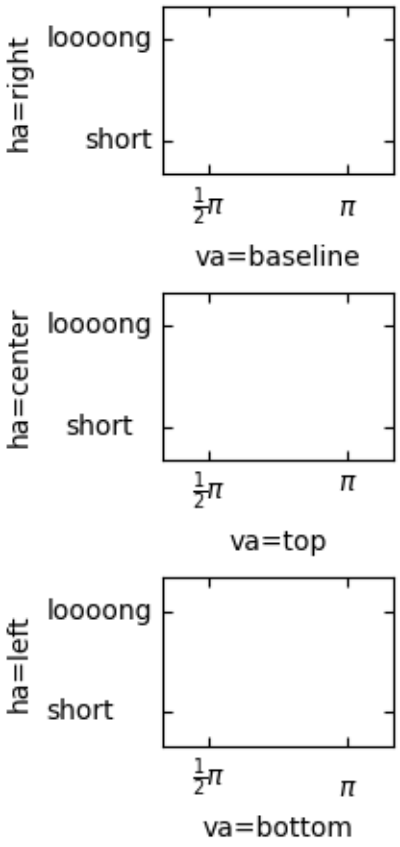

```
import matplotlib.pyplot as plt

from mpl_toolkits import axisartist
from mpl_toolkits.axes_grid1 import host_subplot

host = host_subplot(111, axes_class=axisartist.Axes)
plt.subplots_adjust(right=0.75)

par1 = host.twinx()
par2 = host.twinx()

par2.axis["right"] = par2.new_fixed_axis(loc="right", offset=(60, 0))

par1.axis["right"].toggle(all=True)
par2.axis["right"].toggle(all=True)

p1, = host.plot([0, 1, 2], [0, 1, 2], label="Density")
p2, = par1.plot([0, 1, 2], [0, 3, 2], label="Temperature")
p3, = par2.plot([0, 1, 2], [50, 30, 15], label="Velocity")

host.set(xlim=(0, 2), ylim=(0, 2), xlabel="Distance", ylabel="Density")
par1.set(ylim=(0, 4), ylabel="Temperature")
par2.set(ylim=(1, 65), ylabel="Velocity")
```

(continues on next page)

(continued from previous page)

```
host.legend()

host.axis["left"].label.set_color(p1.get_color())
par1.axis["right"].label.set_color(p2.get_color())
par2.axis["right"].label.set_color(p3.get_color())

plt.show()
```

### Ticklabel alignment



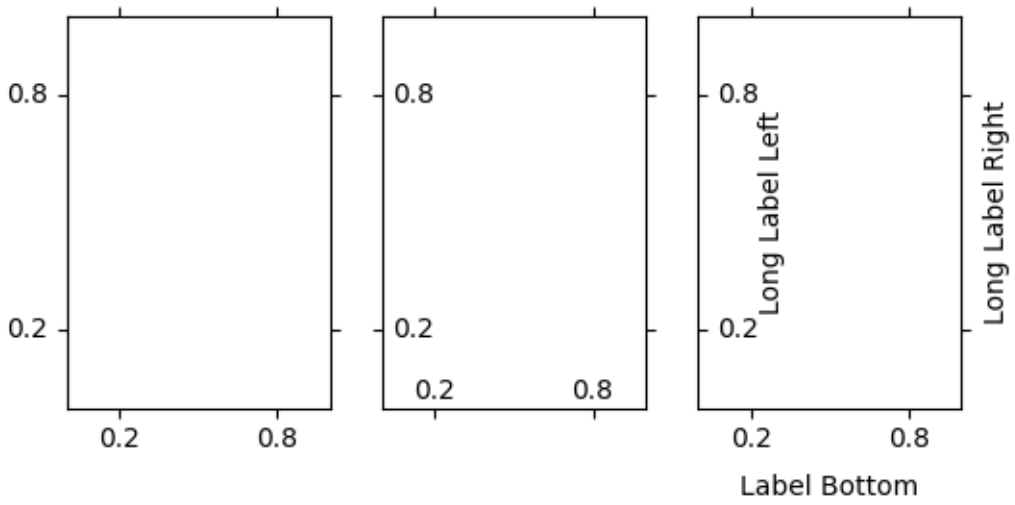

```
import matplotlib.pyplot as plt

import mpl_toolkits.axisartist as axisartist


def setup_axes(fig, pos):
    ax = fig.add_subplot(pos, axes_class=axisartist.Axes)
    ax.set_yticks([0.2, 0.8], labels=["short", "loooong"])
```

(continues on next page)

(continued from previous page)

```
    ax.set_xticks([0.2, 0.8], labels=[r"$\frac{1}{2}\pi$", r"$\pi$"])
    return ax


fig = plt.figure(figsize=(3, 5))
fig.subplots_adjust(left=0.5, hspace=0.7)

ax = setup_axes(fig, 311)
ax.set_ylabel("ha=right")
ax.set_xlabel("va=baseline")

ax = setup_axes(fig, 312)
ax.axis["left"].major_ticklabels.set_ha("center")
ax.axis["bottom"].major_ticklabels.set_va("top")
ax.set_ylabel("ha=center")
ax.set_xlabel("va=top")

ax = setup_axes(fig, 313)
ax.axis["left"].major_ticklabels.set_ha("left")
ax.axis["bottom"].major_ticklabels.set_va("bottom")
ax.set_ylabel("ha=left")
ax.set_xlabel("va=bottom")

plt.show()
```

### Ticklabel direction



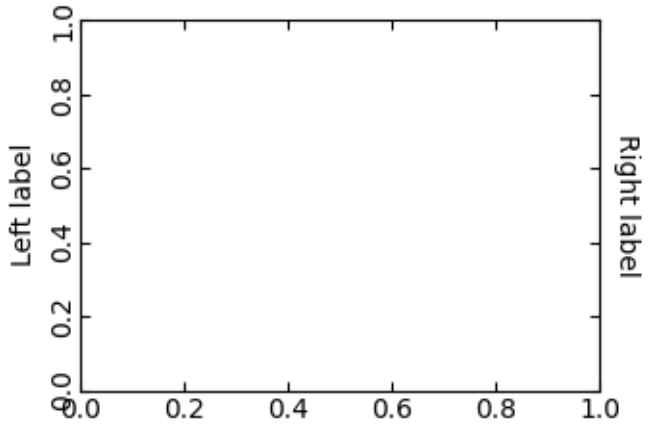

```
import matplotlib.pyplot as plt

import mpl_toolkits.axisartist.axislines as axislines
```

(continues on next page)

(continued from previous page)

```
def setup_axes(fig, pos):
    ax = fig.add_subplot(pos, axes_class=axislines.Axes)
    ax.set_yticks([0.2, 0.8])
    ax.set_xticks([0.2, 0.8])
    return ax


fig = plt.figure(figsize=(6, 3))
fig.subplots_adjust(bottom=0.2)

ax = setup_axes(fig, 131)
for axis in ax.axis.values():
    axis.major_ticks.set_tick_out(True)
# or you can simply do "ax.axis[:].major_ticks.set_tick_out(True)"

ax = setup_axes(fig, 132)
ax.axis["left"].set_axis_direction("right")
ax.axis["bottom"].set_axis_direction("top")
ax.axis["right"].set_axis_direction("left")
ax.axis["top"].set_axis_direction("bottom")

ax = setup_axes(fig, 133)
ax.axis["left"].set_axis_direction("right")
ax.axis[:].major_ticks.set_tick_out(True)

ax.axis["left"].label.set_text("Long Label Left")
ax.axis["bottom"].label.set_text("Label Bottom")
ax.axis["right"].label.set_text("Long Label Right")
ax.axis["right"].label.set_visible(True)
ax.axis["left"].label.set_pad(0)
ax.axis["bottom"].label.set_pad(10)

plt.show()
```

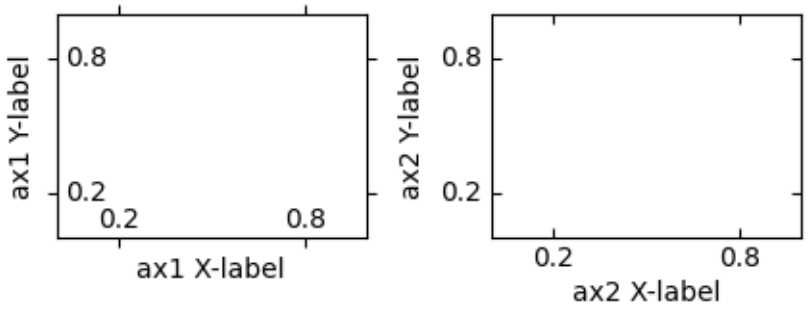
### Simple axis direction




```
import matplotlib.pyplot as plt

import mpl_toolkits.axisartist as axisartist

fig = plt.figure(figsize=(4, 2.5))
ax1 = fig.add_subplot(axes_class=axisartist.Axes)
fig.subplots_adjust(right=0.8)

ax1.axis["left"].major_ticklabels.set_axis_direction("top")
ax1.axis["left"].label.set_text("Left label")

ax1.axis["right"].label.set_visible(True)
ax1.axis["right"].label.set_text("Right label")
ax1.axis["right"].label.set_axis_direction("left")

plt.show()
```

### Simple axis tick label and tick directions

First subplot moves the tick labels to inside the spines. Second subplot moves the ticks to inside the spines. These effects can be obtained for a standard Axes by *tick_params*.




```
import matplotlib.pyplot as plt

import mpl_toolkits.axisartist as axisartist


def setup_axes(fig, pos):
    ax = fig.add_subplot(pos, axes_class=axisartist.Axes)
    ax.set_yticks([0.2, 0.8])
    ax.set_xticks([0.2, 0.8])
    return ax


fig = plt.figure(figsize=(5, 2))
fig.subplots_adjust(wspace=0.4, bottom=0.3)

ax1 = setup_axes(fig, 121)
ax1.set_xlabel("ax1 X-label")
ax1.set_ylabel("ax1 Y-label")

ax1.axis[:].invert_ticklabel_direction()

ax2 = setup_axes(fig, 122)
ax2.set_xlabel("ax2 X-label")
ax2.set_ylabel("ax2 Y-label")

ax2.axis[:].major_ticks.set_tick_out(False)

plt.show()
```

### Simple Axis Pad

```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.projections import PolarAxes
from matplotlib.transforms import Affine2D
import mpl_toolkits.axisartist as axisartist
import mpl_toolkits.axisartist.angle_helper as angle_helper
import mpl_toolkits.axisartist.grid_finder as grid_finder
from mpl_toolkits.axisartist.grid_helper_curvelinear import \
    GridHelperCurveLinear


def setup_axes(fig, rect):
    """Polar projection, but in a rectangular box."""

    # see demo_curvelinear_grid.py for details
    tr = Affine2D().scale(np.pi/180., 1.) + PolarAxes.PolarTransform(
        apply_theta_transforms=False)

    extreme_finder = angle_helper.ExtremeFinderCycle(20, 20,
                                                     lon_cycle=360,
                                                     lat_cycle=None,
                                                     lon_minmax=None,
                                                     lat_minmax=(0, np.inf),
                                                     )

    grid_locator1 = angle_helper.LocatorDMS(12)
    grid_locator2 = grid_finder.MaxNLocator(5)

    tick_formatter1 = angle_helper.FormatterDMS()

    grid_helper = GridHelperCurveLinear(tr,
                                        extreme_finder=extreme_finder,
                                        grid_locator1=grid_locator1,
                                        grid_locator2=grid_locator2,
                                        tick_formatter1=tick_formatter1
```

(continues on next page)

(continued from previous page)

```
                                        )

    ax1 = fig.add_subplot(
        rect, axes_class=axisartist.Axes, grid_helper=grid_helper)
    ax1.axis[:].set_visible(False)
    ax1.set_aspect(1.)
    ax1.set_xlim(-5, 12)
    ax1.set_ylim(-5, 10)

    return ax1


def add_floating_axis1(ax1):
    ax1.axis["lat"] = axis = ax1.new_floating_axis(0, 30)
    axis.label.set_text(r"$\theta = 30^{\circ}$")
    axis.label.set_visible(True)

    return axis


def add_floating_axis2(ax1):
    ax1.axis["lon"] = axis = ax1.new_floating_axis(1, 6)
    axis.label.set_text(r"$r = 6$")
    axis.label.set_visible(True)

    return axis


fig = plt.figure(figsize=(9, 3.))
fig.subplots_adjust(left=0.01, right=0.99, bottom=0.01, top=0.99,
                    wspace=0.01, hspace=0.01)


def ann(ax1, d):
    if plt.rcParams["text.usetex"]:
        d = d.replace("_", r"\_")

    ax1.annotate(d, (0.5, 1), (5, -5),
                 xycoords="axes fraction", textcoords="offset points",
                 va="top", ha="center")


ax1 = setup_axes(fig, rect=141)
axis = add_floating_axis1(ax1)
ann(ax1, r"default")

ax1 = setup_axes(fig, rect=142)
axis = add_floating_axis1(ax1)
axis.major_ticklabels.set_pad(10)
ann(ax1, r"ticklabels.set_pad(10)")

ax1 = setup_axes(fig, rect=143)
```

(continues on next page)

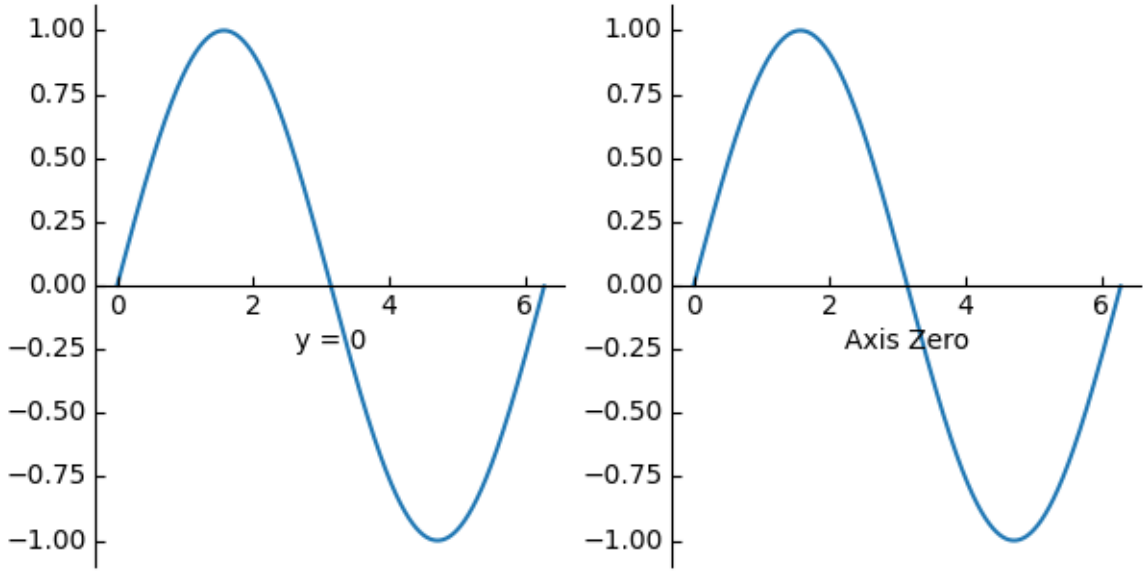
(continued from previous page)

```
axis = add_floating_axis1(ax1)
axis.label.set_pad(20)
ann(ax1, r"label.set_pad(20)")

ax1 = setup_axes(fig, rect=144)
axis = add_floating_axis1(ax1)
axis.major_ticks.set_tick_out(True)
ann(ax1, "ticks.set_tick_out(True)")

plt.show()
```

### Custom spines with axisartist

This example showcases the use of *axisartist* to draw spines at custom positions (here, at `y = 0`).

Note, however, that it is simpler to achieve this effect using standard *Spine* methods, as demonstrated in *Centered spines with arrows*.



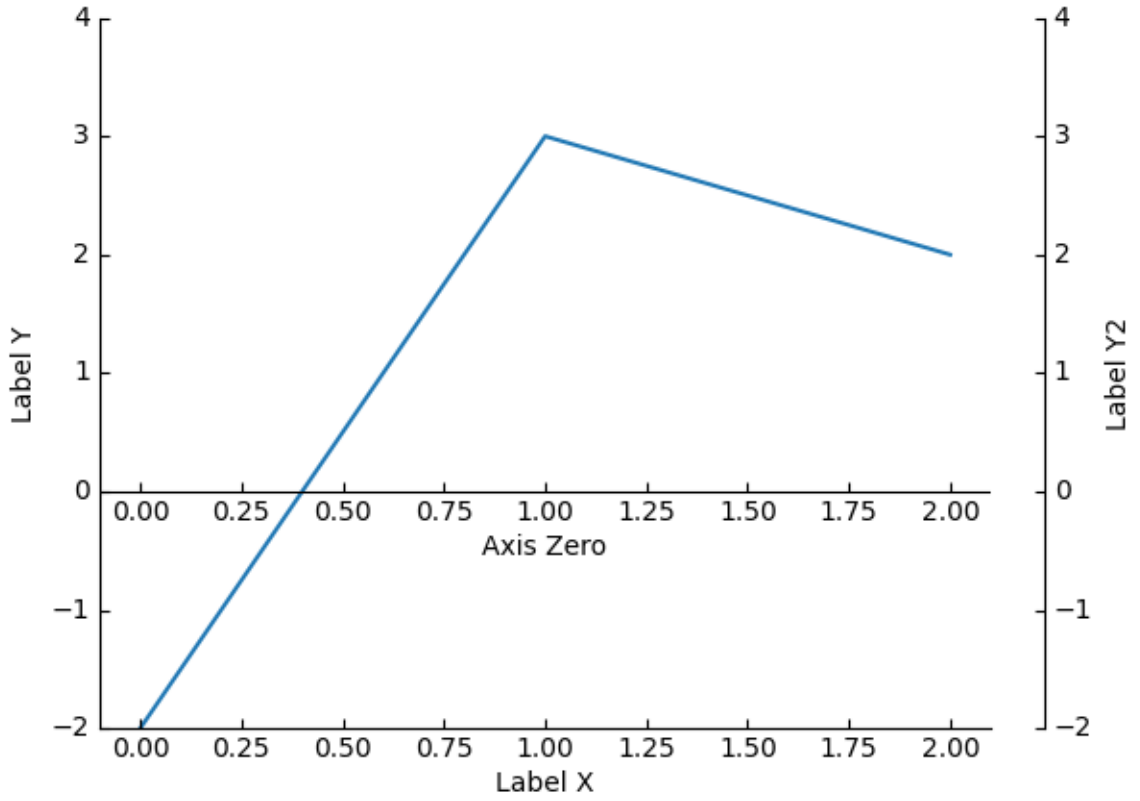

```
import matplotlib.pyplot as plt
import numpy as np

from mpl_toolkits import axisartist

fig = plt.figure(figsize=(6, 3), layout="constrained")
# To construct Axes of two different classes, we need to use gridspec (or
# MATLAB-style add_subplot calls).
gs = fig.add_gridspec(1, 2)


ax0 = fig.add_subplot(gs[0, 0], axes_class=axisartist.Axes)
# Make a new axis along the first (x) axis which passes through y=0.
```

(continues on next page)

(continued from previous page)

```
ax0.axis["y=0"] = ax0.new_floating_axis(nth_coord=0, value=0,
                                        axis_direction="bottom")
ax0.axis["y=0"].toggle(all=True)
ax0.axis["y=0"].label.set_text("y = 0")
# Make other axis invisible.
ax0.axis["bottom", "top", "right"].set_visible(False)


# Alternatively, one can use AxesZero, which automatically sets up two
# additional axis, named "xzero" (the y=0 axis) and "yzero" (the x=0 axis).
ax1 = fig.add_subplot(gs[0, 1], axes_class=axisartist.axislines.AxesZero)
# "xzero" and "yzero" default to invisible; make xzero axis visible.
ax1.axis["xzero"].set_visible(True)
ax1.axis["xzero"].label.set_text("Axis Zero")
# Make other axis invisible.
ax1.axis["bottom", "top", "right"].set_visible(False)


# Draw some sample data.
x = np.arange(0, 2*np.pi, 0.01)
ax0.plot(x, np.sin(x))
ax1.plot(x, np.sin(x))

plt.show()
```

### Simple Axisline



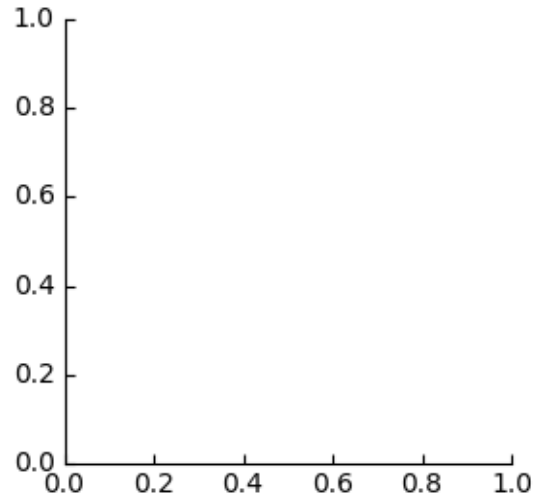

```
import matplotlib.pyplot as plt

from mpl_toolkits.axisartist.axislines import AxesZero

fig = plt.figure()
fig.subplots_adjust(right=0.85)
ax = fig.add_subplot(axes_class=AxesZero)

# make right and top axis invisible
ax.axis["right"].set_visible(False)
ax.axis["top"].set_visible(False)

# make xzero axis (horizontal axis line through y=0) visible.
ax.axis["xzero"].set_visible(True)
ax.axis["xzero"].label.set_text("Axis Zero")

ax.set_ylim(-2, 4)
ax.set_xlabel("Label X")
ax.set_ylabel("Label Y")
# Or:
# ax.axis["bottom"].label.set_text("Label X")
```

(continues on next page)

(continued from previous page)

```
# ax.axis["left"].label.set_text("Label Y")

# make new (right-side) yaxis, but with some offset
ax.axis["right2"] = ax.new_fixed_axis(loc="right", offset=(20, 0))
ax.axis["right2"].label.set_text("Label Y2")

ax.plot([-2, 3, 2])
plt.show()
```

### Simple Axisline3




```
import matplotlib.pyplot as plt

from mpl_toolkits.axisartist.axislines import Axes

fig = plt.figure(figsize=(3, 3))

ax = fig.add_subplot(axes_class=Axes)

ax.axis["right"].set_visible(False)
ax.axis["top"].set_visible(False)

plt.show()
```

### 6.25.13 Showcase

#### Anatomy of a figure

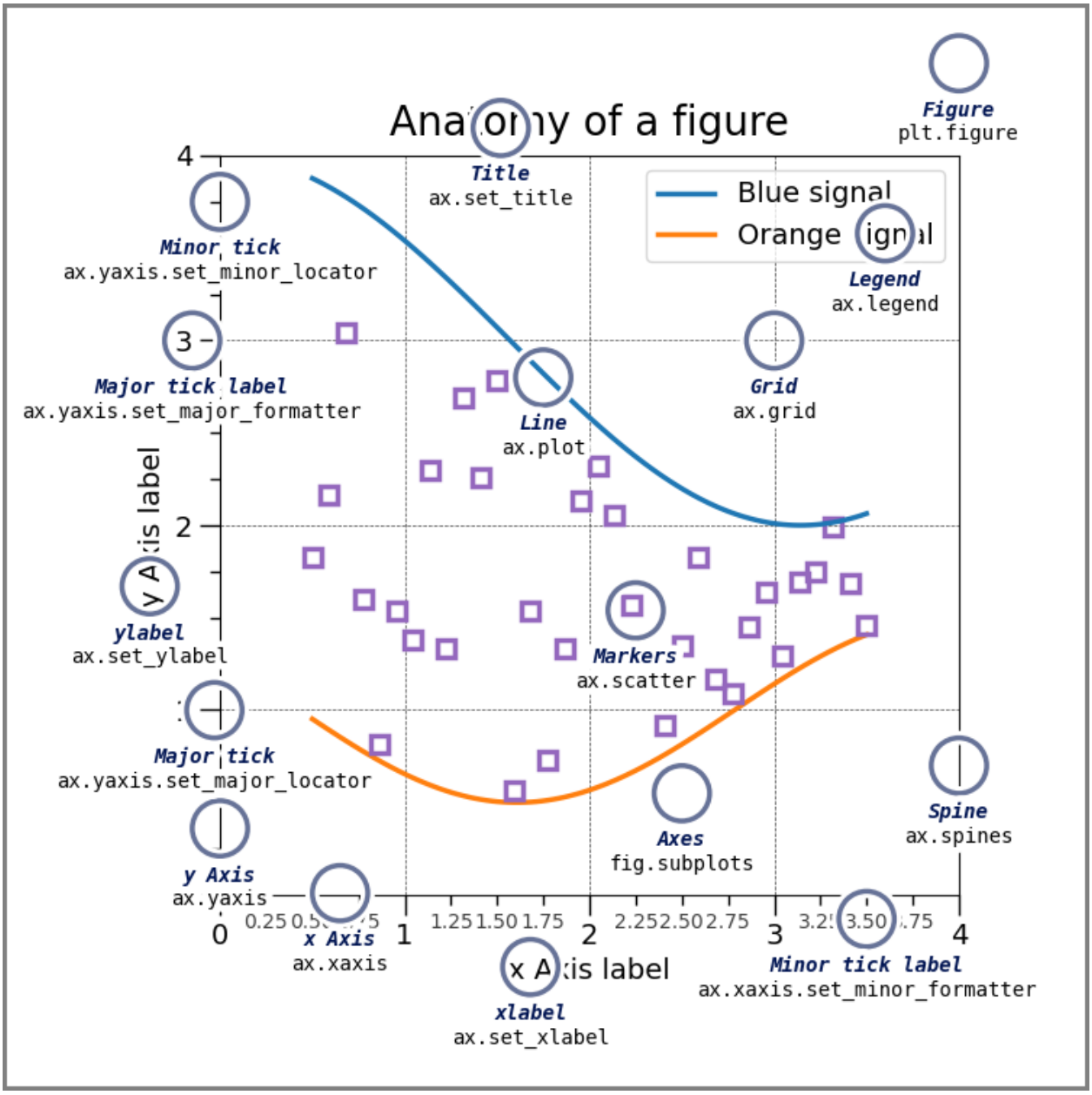
This figure shows the name of several matplotlib elements composing a figure

```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.patches import Circle
from matplotlib.patheffects import withStroke
from matplotlib.ticker import AutoMinorLocator, MultipleLocator

royal_blue = [0, 20/256, 82/256]


# make the figure

np.random.seed(19680801)

X = np.linspace(0.5, 3.5, 100)
Y1 = 3+np.cos(X)
Y2 = 1+np.cos(1+X/0.75)/2
Y3 = np.random.uniform(Y1, Y2, len(X))

fig = plt.figure(figsize=(7.5, 7.5))
ax = fig.add_axes([0.2, 0.17, 0.68, 0.7], aspect=1)

ax.xaxis.set_major_locator(MultipleLocator(1.000))
ax.xaxis.set_minor_locator(AutoMinorLocator(4))
ax.yaxis.set_major_locator(MultipleLocator(1.000))
ax.yaxis.set_minor_locator(AutoMinorLocator(4))
ax.xaxis.set_minor_formatter("{x:.2f}")

ax.set_xlim(0, 4)
ax.set_ylim(0, 4)

ax.tick_params(which='major', width=1.0, length=10, labelsize=14)
ax.tick_params(which='minor', width=1.0, length=5, labelsize=10,
               labelcolor='0.25')

ax.grid(linestyle="--", linewidth=0.5, color='.25', zorder=-10)

ax.plot(X, Y1, c='C0', lw=2.5, label="Blue signal", zorder=10)
ax.plot(X, Y2, c='C1', lw=2.5, label="Orange signal")
ax.plot(X[::3], Y3[::3], linewidth=0, markersize=9,
        marker='s', markerfacecolor='none', markeredgecolor='C4',
        markeredgewidth=2.5)

ax.set_title("Anatomy of a figure", fontsize=20, verticalalignment='bottom')
ax.set_xlabel("x Axis label", fontsize=14)
ax.set_ylabel("y Axis label", fontsize=14)
```

(continues on next page)

(continued from previous page)

```
ax.legend(loc="upper right", fontsize=14)


# Annotate the figure

def annotate(x, y, text, code):
    # Circle marker
    c = Circle((x, y), radius=0.15, clip_on=False, zorder=10, linewidth=2.5,
               edgecolor=royal_blue + [0.6], facecolor='none',
               path_effects=[withStroke(linewidth=7, foreground='white')])
    ax.add_artist(c)

    # use path_effects as a background for the texts
    # draw the path_effects and the colored text separately so that the
    # path_effects cannot clip other texts
    for path_effects in [[withStroke(linewidth=7, foreground='white')], []]:
        color = 'white' if path_effects else royal_blue
        ax.text(x, y-0.2, text, zorder=100,
                ha='center', va='top', weight='bold', color=color,
                style='italic', fontfamily='monospace',
                path_effects=path_effects)

        color = 'white' if path_effects else 'black'
        ax.text(x, y-0.33, code, zorder=100,
                ha='center', va='top', weight='normal', color=color,
                fontfamily='monospace', fontsize='medium',
                path_effects=path_effects)


annotate(3.5, -0.13, "Minor tick label", "ax.xaxis.set_minor_formatter")
annotate(-0.03, 1.0, "Major tick", "ax.yaxis.set_major_locator")
annotate(0.00, 3.75, "Minor tick", "ax.yaxis.set_minor_locator")
annotate(-0.15, 3.00, "Major tick label", "ax.yaxis.set_major_formatter")
annotate(1.68, -0.39, "xlabel", "ax.set_xlabel")
annotate(-0.38, 1.67, "ylabel", "ax.set_ylabel")
annotate(1.52, 4.15, "Title", "ax.set_title")
annotate(1.75, 2.80, "Line", "ax.plot")
annotate(2.25, 1.54, "Markers", "ax.scatter")
annotate(3.00, 3.00, "Grid", "ax.grid")
annotate(3.60, 3.58, "Legend", "ax.legend")
annotate(2.5, 0.55, "Axes", "fig.subplots")
annotate(4, 4.5, "Figure", "plt.figure")
annotate(0.65, 0.01, "x Axis", "ax.xaxis")
annotate(0, 0.36, "y Axis", "ax.yaxis")
annotate(4.0, 0.7, "Spine", "ax.spines")

# frame around figure
fig.patch.set(linewidth=4, edgecolor='0.5')
plt.show()
```




**References**

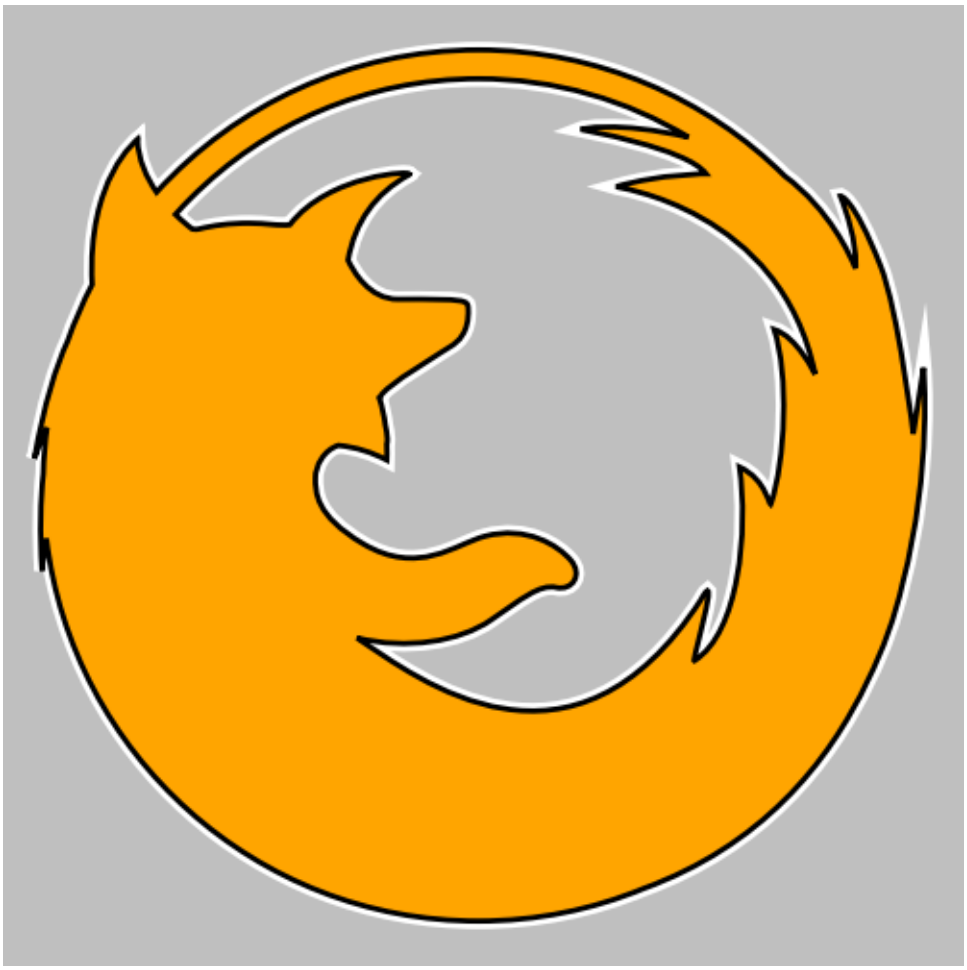
The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.pyplot.figure*
- *matplotlib.axes.Axes.text*
- *matplotlib.axis.Axis.set_minor_formatter*
- *matplotlib.axis.Axis.set_major_locator*
- *matplotlib.axis.Axis.set_minor_locator*
- *matplotlib.patches.Circle*

- *matplotlib.patheffects.withStroke*
- *matplotlib.ticker.FuncFormatter*

### Firefox

This example shows how to create the Firefox logo with path and patches.



```
import re

import matplotlib.pyplot as plt
import numpy as np

import matplotlib.patches as patches
from matplotlib.path import Path

# From: https://dmitrybaranovskiy.github.io/raphael/icons/#firefox
firefox = "M28.4,22.469c0.479-0.964,0.851-1.991,1.095-3.066c0.953-3.661,0.666-
 ↪6.854,0.666-6.854l-0.327,2.104c0,0-0.469-3.896-1.044-5.353c-0.881-2.231-1.
 ↪273-2.214-1.274-2.21c0.542,1.379,0.494,2.169,0.483,2.288c-0.01-0.016-0.019-
```

(continues on next page)

(continued from previous page)

```
↪0.032-0.027-0.047c-0.131-0.324-0.797-1.819-2.225-2.878c-2.502-2.481-5.943-4.
↪014-9.745-4.015c-4.056,0-7.705,1.745-10.238,4.525C5.444,6.5,5.183,5.938,5.
↪159,5.317c0,0-0.002,0.002-0.006,0.005c0-0.011-0.003-0.021-0.003-0.031c0,0-1.
↪61,1.247-1.436,4.612c-0.299,0.574-0.56,1.172-0.777,1.791c-0.375,0.817-0.75,
↪2.004-1.059,3.746c0,0,0.133-0.422,0.399-0.988c-0.064,0.482-0.103,0.971-0.
↪116,1.467c-0.09,0.845-0.118,1.865-0.039,3.088c0,0,0.032-0.406,0.136-1.021c0.
↪834,6.854,6.667,12.165,13.743,12.165l0,0c1.86,0,3.636-0.37,5.256-1.036C24.
↪938,27.771,27.116,25.196,28.4,22.469zM16.002,3.356c2.446,0,4.73,0.68,6.68,1.
↪86c-2.274-0.528-3.433-0.261-3.423-0.248c0.013,0.015,3.384,0.589,3.981,1.
↪411c0,0-1.431,0-2.856,0.41c-0.065,0.019,5.242,0.663,6.327,5.966c0,0-0.582-1.
↪213-1.301-1.42c0.473,1.439,0.351,4.17-0.1,5.528c-0.058,0.174-0.118-0.755-1.
↪004-1.155c0.284,2.037-0.018,5.268-1.432,6.158c-0.109,0.07,0.887-3.189,0.201-
↪1.93c-4.093,6.276-8.959,2.539-10.934,1.208c1.585,0.388,3.267,0.108,4.242-0.
↪559c0.982-0.672,1.564-1.162,2.087-1.047c0.522,0.117,0.87-0.407,0.464-0.872c-
↪0.405-0.466-1.392-1.105-2.725-0.757c-0.94,0.247-2.107,1.287-3.886,0.233c-1.
↪518-0.899-1.507-1.63-1.507-2.095c0-0.366,0.257-0.88,0.734-1.028c0.58,0.062,
↪1.044,0.214,1.537,0.466c0.005-0.135,0.006-0.315-0.001-0.519c0.039-0.077,0.
↪015-0.311-0.047-0.596c-0.036-0.287-0.097-0.582-0.19-0.851c0.01-0.002,0.017-
↪0.007,0.021-0.021c0.076-0.344,2.147-1.544,2.299-1.659c0.153-0.114,0.55-0.
↪378,0.506-1.183c-0.015-0.265-0.058-0.294-2.232-0.286c-0.917,0.003-1.425-0.
↪894-1.589-1.245c0.222-1.231,0.863-2.11,1.919-2.704c0.02-0.011,0.015-0.021-0.
↪008-0.027c0.219-0.127-2.524-0.006-3.76,1.604C9.674,8.045,9.219,7.95,8.71,7.
↪95c-0.638,0-1.139,0.07-1.603,0.187c-0.05,0.013-0.122,0.011-0.208-0.001C6.
↪769,8.04,6.575,7.88,6.365,7.672c0.161-0.18,0.324-0.356,0.495-0.526C9.201,4.
↪804,12.43,3.357,16.002,3.356z"  # noqa


def svg_parse(path):
    commands = {'M': (Path.MOVETO,),
                'L': (Path.LINETO,),
                'Q': (Path.CURVE3,)*2,
                'C': (Path.CURVE4,)*3,
                'Z': (Path.CLOSEPOLY,)}
    vertices = []
    codes = []
    cmd_values = re.split("([A-Za-z])", path)[1:]  # Split over commands.
    for cmd, values in zip(cmd_values[::2], cmd_values[1::2]):
        # Numbers are separated either by commas, or by +/- signs (but not at
        # the beginning of the string).
        points = ([*map(float, re.split(",|(?<!^)(?=[+-])", values))] if␣
↪values
                  else [(0., 0.)])  # Only for "z/Z" (CLOSEPOLY).
        points = np.reshape(points, (-1, 2))
        if cmd.islower():
            points += vertices[-1][-1]
        codes.extend(commands[cmd.upper()])
        vertices.append(points)
    return np.array(codes), np.concatenate(vertices)


# SVG to Matplotlib
codes, verts = svg_parse(firefox)
```

(continues on next page)

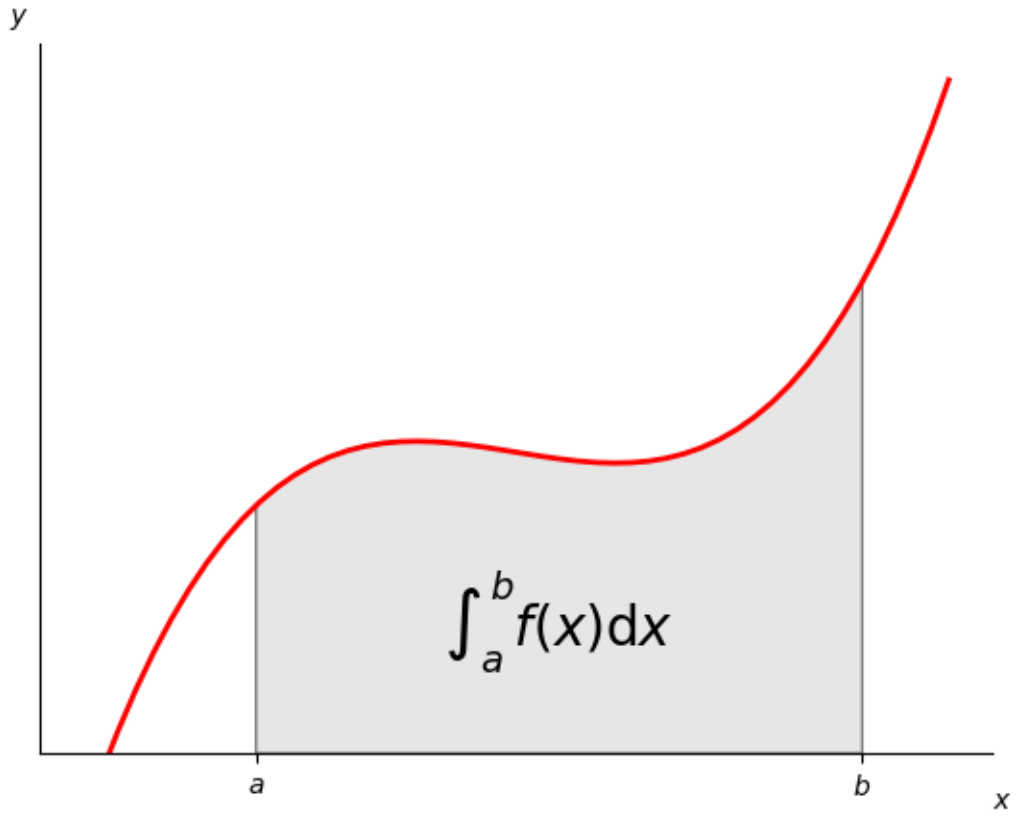
(continued from previous page)

```
path = Path(verts, codes)

xmin, ymin = verts.min(axis=0) - 1
xmax, ymax = verts.max(axis=0) + 1

fig = plt.figure(figsize=(5, 5), facecolor="0.75")  # gray background
ax = fig.add_axes([0, 0, 1, 1], frameon=False, aspect=1,
                  xlim=(xmin, xmax),  # centering
                  ylim=(ymax, ymin),  # centering, upside down
                  xticks=[], yticks=[])  # no ticks

# White outline (width = 6)
ax.add_patch(patches.PathPatch(path, facecolor='none', edgecolor='w', lw=6))
# Actual shape with black outline
ax.add_patch(patches.PathPatch(path, facecolor='orange', edgecolor='k', lw=2))

plt.show()  # Display
```

### Integral as the area under a curve

Although this is a simple example, it demonstrates some important tweaks:

- A simple line plot with custom color and line width.
- A shaded region created using a Polygon patch.
- A text label with mathtext rendering.
- figtext calls to label the x- and y-axes.
- Use of axis spines to hide the top and right spines.
- Custom tick placement and labels.




```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.patches import Polygon


def func(x):
    return (x - 3) * (x - 5) * (x - 7) + 85


a, b = 2, 9  # integral limits
x = np.linspace(0, 10)
y = func(x)

fig, ax = plt.subplots()
ax.plot(x, y, 'r', linewidth=2)
ax.set_ylim(bottom=0)

# Make the shaded region
ix = np.linspace(a, b)
iy = func(ix)
verts = [(a, 0), *zip(ix, iy), (b, 0)]
poly = Polygon(verts, facecolor='0.9', edgecolor='0.5')
```

(continues on next page)

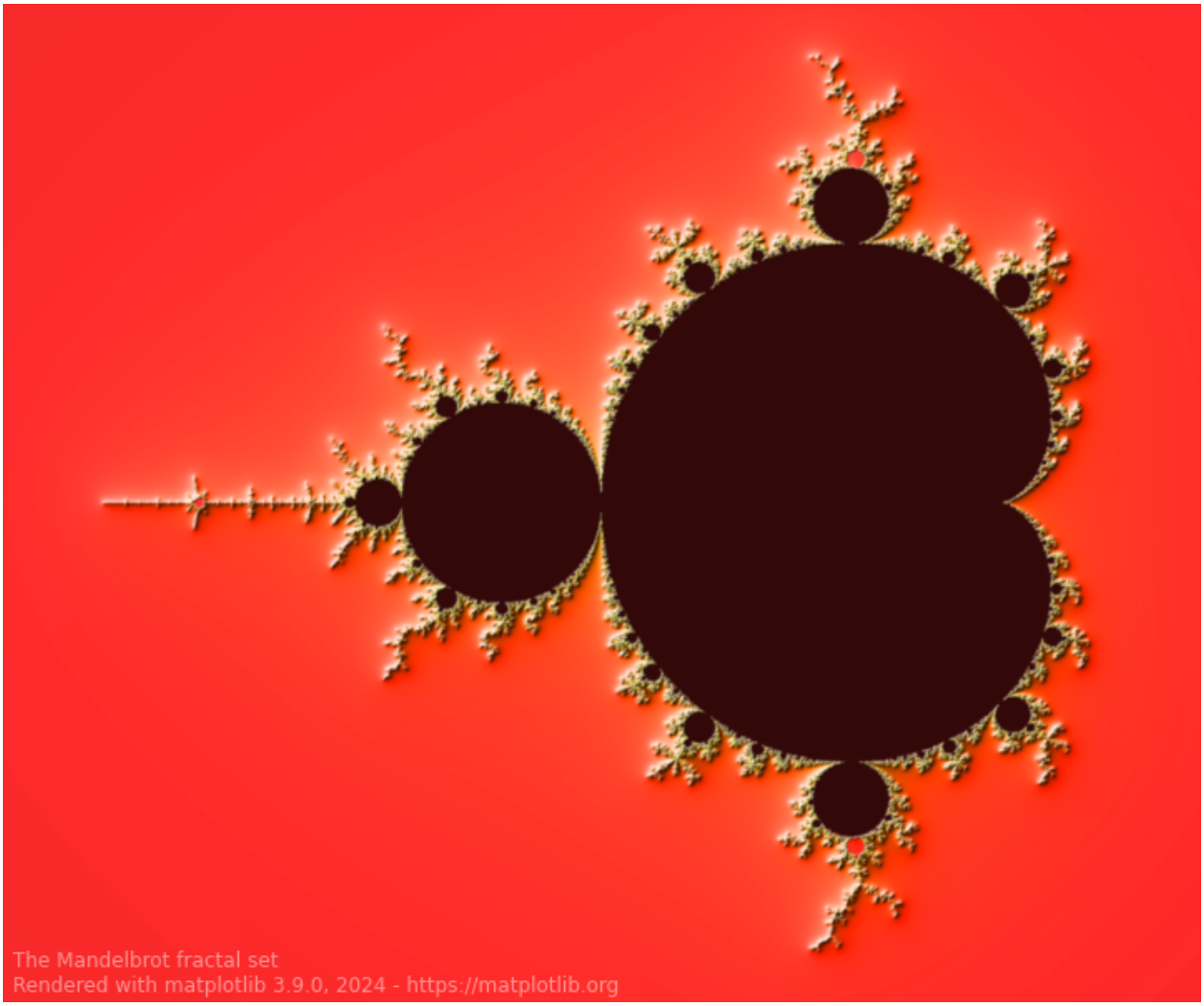
(continued from previous page)

```
ax.add_patch(poly)

ax.text(0.5 * (a + b), 30, r"$\int_a^b f(x)\mathrm{d}x$",
        horizontalalignment='center', fontsize=20)

fig.text(0.9, 0.05, '$x$')
fig.text(0.1, 0.9, '$y$')

ax.spines[['top', 'right']].set_visible(False)
ax.set_xticks([a, b], labels=['$a$', '$b$'])
ax.set_yticks([])

plt.show()
```

### Shaded & power normalized rendering

The Mandelbrot set rendering can be improved by using a normalized recount associated with a power normalized colormap (gamma=0.3). Rendering can be further enhanced thanks to shading.

The `maxiter` gives the precision of the computation. `maxiter=200` should take a few seconds on most modern laptops.




```
import numpy as np


def mandelbrot_set(xmin, xmax, ymin, ymax, xn, yn, maxiter, horizon=2.0):
    X = np.linspace(xmin, xmax, xn).astype(np.float32)
    Y = np.linspace(ymin, ymax, yn).astype(np.float32)
    C = X + Y[:, None] * 1j
    N = np.zeros_like(C, dtype=int)
    Z = np.zeros_like(C)
    for n in range(maxiter):
        I = abs(Z) < horizon
        N[I] = n
        Z[I] = Z[I]**2 + C[I]
    N[N == maxiter-1] = 0
    return Z, N


if __name__ == '__main__':
    import time
```

(continues on next page)

(continued from previous page)

```
    import matplotlib.pyplot as plt

    import matplotlib
    from matplotlib import colors

    xmin, xmax, xn = -2.25, +0.75, 3000 // 2
    ymin, ymax, yn = -1.25, +1.25, 2500 // 2
    maxiter = 200
    horizon = 2.0 ** 40
    log_horizon = np.log2(np.log(horizon))
    Z, N = mandelbrot_set(xmin, xmax, ymin, ymax, xn, yn, maxiter, horizon)

    # Normalized recount as explained in:
    # https://linas.org/art-gallery/escape/smooth.html
    # https://web.archive.org/web/20160331171238/https://www.ibm.com/
↪developerworks/community/blogs/jfp/entry/My_Christmas_Gift?lang=en

    # This line will generate warnings for null values, but it is faster to
    # process them afterwards using the nan_to_num
    with np.errstate(invalid='ignore'):
        M = np.nan_to_num(N + 1 - np.log2(np.log(abs(Z))) + log_horizon)

    dpi = 72
    width = 10
    height = 10*yn/xn
    fig = plt.figure(figsize=(width, height), dpi=dpi)
    ax = fig.add_axes([0, 0, 1, 1], frameon=False, aspect=1)

    # Shaded rendering
    light = colors.LightSource(azdeg=315, altdeg=10)
    M = light.shade(M, cmap=plt.cm.hot, vert_exag=1.5,
                    norm=colors.PowerNorm(0.3), blend_mode='hsv')
    ax.imshow(M, extent=[xmin, xmax, ymin, ymax], interpolation="bicubic")
    ax.set_xticks([])
    ax.set_yticks([])

    # Some advertisement for matplotlib
    year = time.strftime("%Y")
    text = ("The Mandelbrot fractal set\n"
            "Rendered with matplotlib %s, %s - https://matplotlib.org"
            % (matplotlib.__version__, year))
    ax.text(xmin+.025, ymin+.025, text, color="white", fontsize=12, alpha=0.5)

    plt.show()
```

**Total running time of the script:** (0 minutes 4.670 seconds)

### Pan/zoom events of overlapping axes

Example to illustrate how pan/zoom events of overlapping axes are treated.

The default is the following:

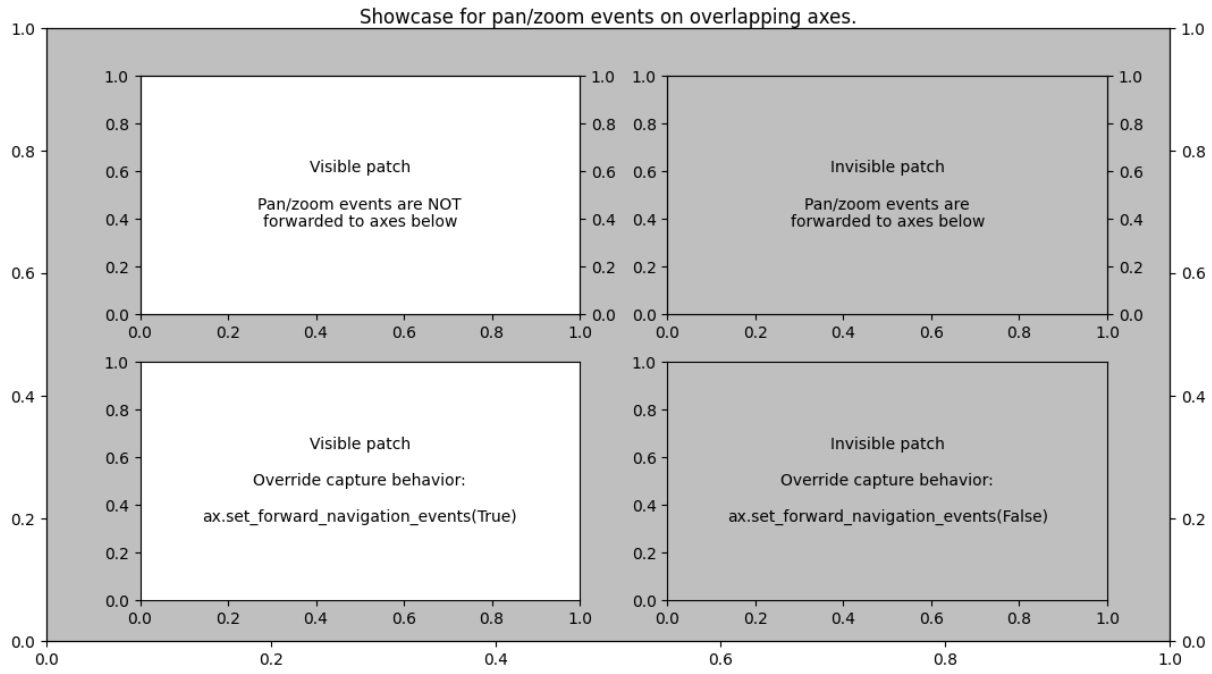
- Axes with a visible patch capture pan/zoom events
- Axes with an invisible patch forward pan/zoom events to axes below
- Shared axes always trigger with their parent axes (irrespective of the patch visibility)

Note: The visibility of the patch hereby refers to the value of `ax.patch.get_visible()`. The color and transparency of a patch have no effect on the treatment of pan/zoom events!

`ax.set_forward_navigation_events(val)` can be used to override the default behaviour:

- `True`: Forward navigation events to axes below.
- `False`: Execute navigation events only on this axes.
- `"auto"`: Use the default behaviour.

To disable pan/zoom events completely, use `ax.set_navigate(False)`




```
import matplotlib.pyplot as plt

fig = plt.figure(figsize=(11, 6))
fig.suptitle("Showcase for pan/zoom events on overlapping axes.")

ax = fig.add_axes((.05, .05, .9, .9))
ax.patch.set_color(".75")
ax_twin = ax.twinx()
```

(continues on next page)

(continued from previous page)

```
ax1 = fig.add_subplot(221)
ax1_twin = ax1.twinx()
ax1.text(.5, .5,
         "Visible patch\n\n"
         "Pan/zoom events are NOT\n"
         "forwarded to axes below",
         ha="center", va="center", transform=ax1.transAxes)

ax11 = fig.add_subplot(223, sharex=ax1, sharey=ax1)
ax11.set_forward_navigation_events(True)
ax11.text(.5, .5,
          "Visible patch\n\n"
          "Override capture behavior:\n\n"
          "ax.set_forward_navigation_events(True)",
          ha="center", va="center", transform=ax11.transAxes)

ax2 = fig.add_subplot(222)
ax2_twin = ax2.twinx()
ax2.patch.set_visible(False)
ax2.text(.5, .5,
         "Invisible patch\n\n"
         "Pan/zoom events are\n"
         "forwarded to axes below",
         ha="center", va="center", transform=ax2.transAxes)

ax22 = fig.add_subplot(224, sharex=ax2, sharey=ax2)
ax22.patch.set_visible(False)
ax22.set_forward_navigation_events(False)
ax22.text(.5, .5,
          "Invisible patch\n\n"
          "Override capture behavior:\n\n"
          "ax.set_forward_navigation_events(False)",
          ha="center", va="center", transform=ax22.transAxes)
```

### Stock prices over 32 years

A graph of multiple time series that demonstrates custom styling of plot frame, tick lines, tick labels, and line graph properties. It also uses custom placement of text labels along the right edge as an alternative to a conventional legend.

Note: The third-party mpl style dufte produces similar-looking plots with less code.

```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.cbook import get_sample_data
import matplotlib.transforms as mtransforms

with get_sample_data('Stocks.csv') as file:
```

(continues on next page)

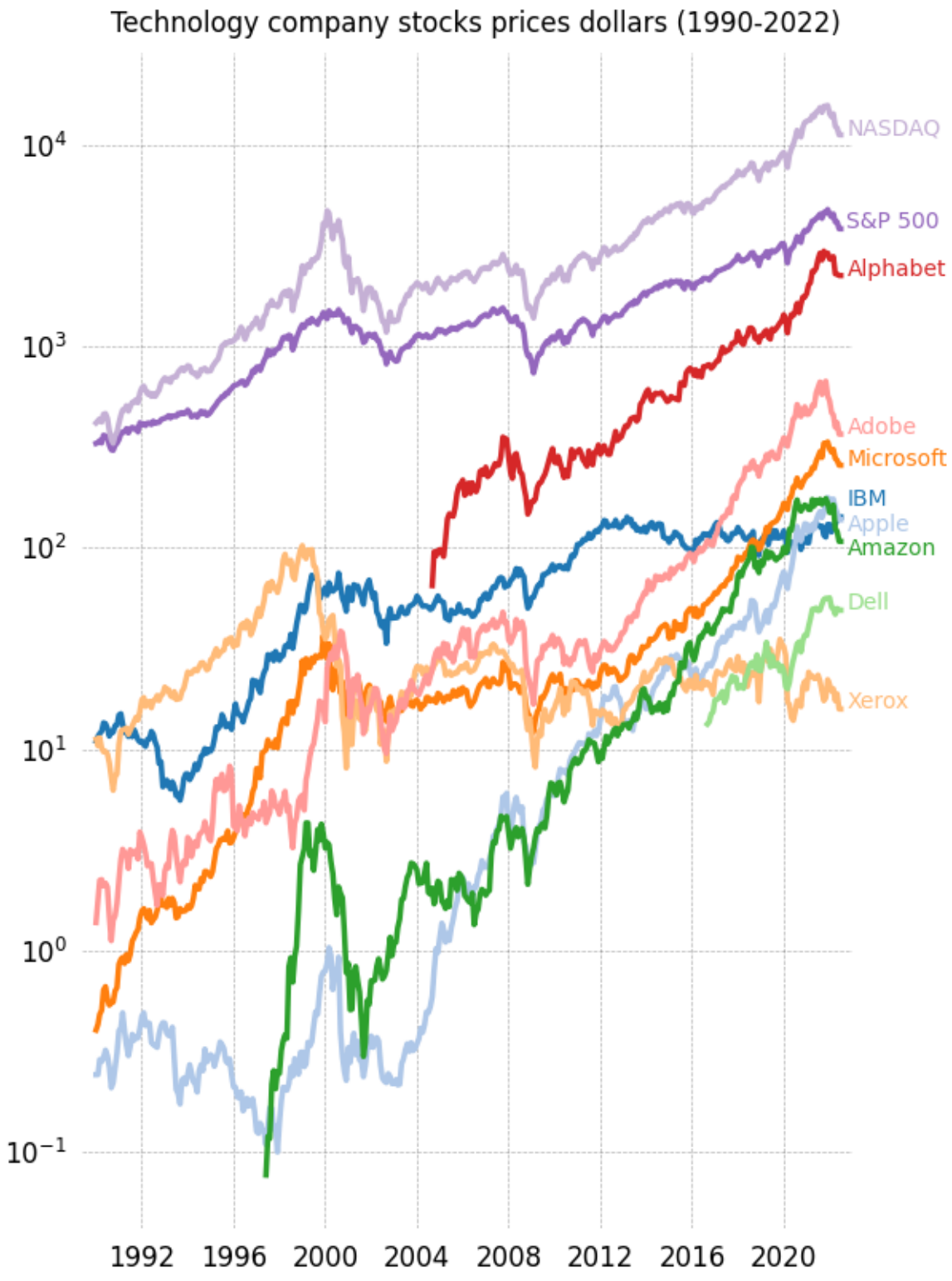
(continued from previous page)

```
    stock_data = np.genfromtxt(
        file, delimiter=',', names=True, dtype=None,
        converters={0: lambda x: np.datetime64(x, 'D')}, skip_header=1)

fig, ax = plt.subplots(1, 1, figsize=(6, 8), layout='constrained')

# These are the colors that will be used in the plot
ax.set_prop_cycle(color=[
    '#1f77b4', '#aec7e8', '#ff7f0e', '#ffbb78', '#2ca02c', '#98df8a',
    '#d62728', '#ff9896', '#9467bd', '#c5b0d5', '#8c564b', '#c49c94',
    '#e377c2', '#f7b6d2', '#7f7f7f', '#c7c7c7', '#bcbd22', '#dbdb8d',
    '#17becf', '#9edae5'])

stocks_name = ['IBM', 'Apple', 'Microsoft', 'Xerox', 'Amazon', 'Dell',
               'Alphabet', 'Adobe', 'S&P 500', 'NASDAQ']
stocks_ticker = ['IBM', 'AAPL', 'MSFT', 'XRX', 'AMZN', 'DELL', 'GOOGL',
                 'ADBE', 'GSPC', 'IXIC']

# Manually adjust the label positions vertically (units are points = 1/72␣
 ↪inch)
y_offsets = {k: 0 for k in stocks_ticker}
y_offsets['IBM'] = 5
y_offsets['AAPL'] = -5
y_offsets['AMZN'] = -6

for nn, column in enumerate(stocks_ticker):
    # Plot each line separately with its own color.
    # don't include any data with NaN.
    good = np.nonzero(np.isfinite(stock_data[column]))
    line, = ax.plot(stock_data['Date'][good], stock_data[column][good], lw=2.
 ↪5)

    # Add a text label to the right end of every line. Most of the code below
    # is adding specific offsets y position because some labels overlapped.
    y_pos = stock_data[column][-1]

    # Use an offset transform, in points, for any text that needs to be nudged
    # up or down.
    offset = y_offsets[column] / 72
    trans = mtransforms.ScaledTranslation(0, offset, fig.dpi_scale_trans)
    trans = ax.transData + trans

    # Again, make sure that all labels are large enough to be easily read
    # by the viewer.
    ax.text(np.datetime64('2022-10-01'), y_pos, stocks_name[nn],
            color=line.get_color(), transform=trans)

ax.set_xlim(np.datetime64('1989-06-01'), np.datetime64('2023-01-01'))

fig.suptitle("Technology company stocks prices dollars (1990-2022)",
             ha="center")
```

(continues on next page)

(continued from previous page)

```
# Remove the plot frame lines. They are unnecessary here.
ax.spines[:].set_visible(False)

# Ensure that the axis ticks only show up on the bottom and left of the plot.
# Ticks on the right and top of the plot are generally unnecessary.
ax.xaxis.tick_bottom()
ax.yaxis.tick_left()
ax.set_yscale('log')

# Provide tick lines across the plot to help your viewers trace along
# the axis ticks. Make sure that the lines are light and small so they
# don't obscure the primary data lines.
ax.grid(True, 'major', 'both', ls='--', lw=.5, c='k', alpha=.3)

# Remove the tick marks; they are unnecessary with the tick lines we just
# plotted. Make sure your axis ticks are large enough to be easily read.
# You don't want your viewers squinting to read your plot.
ax.tick_params(axis='both', which='both', labelsize='large',
               bottom=False, top=False, labelbottom=True,
               left=False, right=False, labelleft=True)

# Finally, save the figure as a PNG.
# You can also save it as a PDF, JPEG, etc.
# Just change the file extension in this call.
# fig.savefig('stock-prices.png', bbox_inches='tight')
plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.pyplot.subplots*
- *matplotlib.axes.Axes.text*
- *matplotlib.axis.XAxis.tick_bottom*
- *matplotlib.axis.YAxis.tick_left*
- *matplotlib.artist.Artist.set_visible*

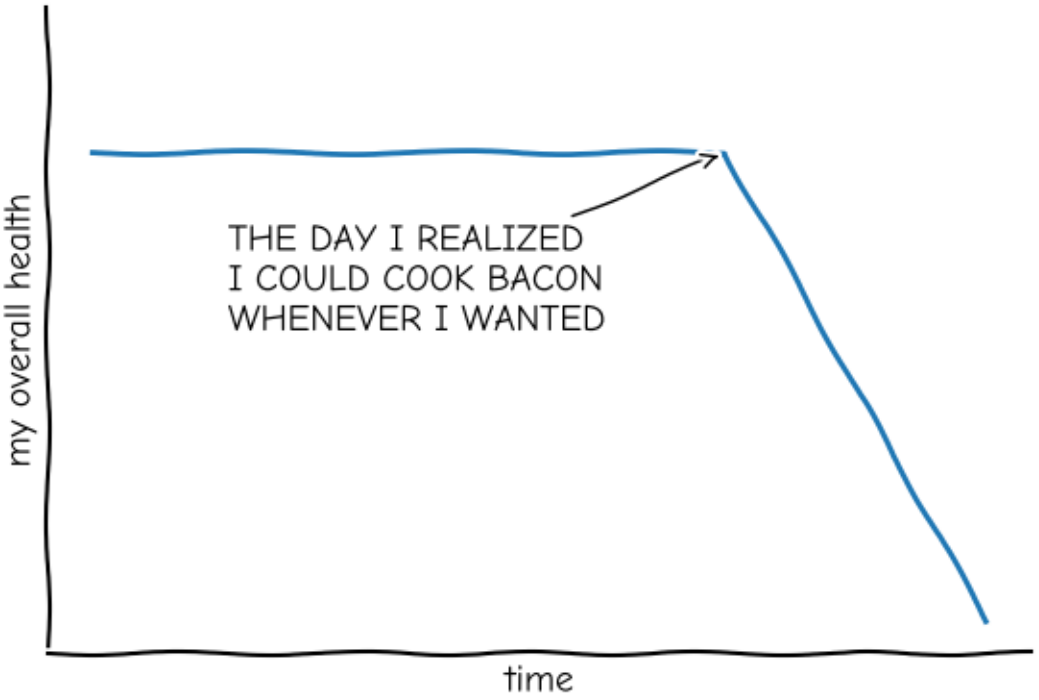
**Total running time of the script:** (0 minutes 1.246 seconds)

## XKCD

Shows how to create an xkcd-like plot.

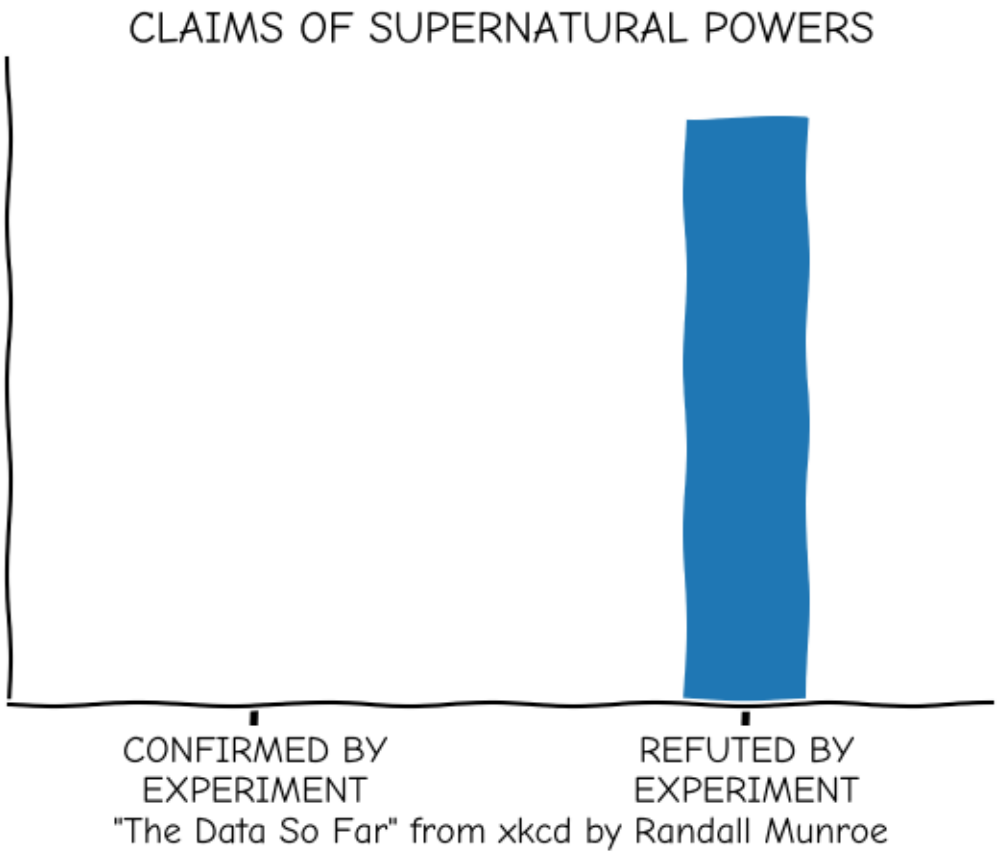
```
import matplotlib.pyplot as plt
import numpy as np
```

```
with plt.xkcd():
    # Based on "Stove Ownership" from XKCD by Randall Munroe
    # https://xkcd.com/418/

    fig = plt.figure()
    ax = fig.add_axes((0.1, 0.2, 0.8, 0.7))
    ax.spines[['top', 'right']].set_visible(False)
    ax.set_xticks([])
    ax.set_yticks([])
    ax.set_ylim([-30, 10])

    data = np.ones(100)
    data[70:] -= np.arange(30)

    ax.annotate(
        'THE DAY I REALIZED\nI COULD COOK BACON\nWHENEVER I WANTED',
        xy=(70, 1), arrowprops=dict(arrowstyle='->'), xytext=(15, -10))

    ax.plot(data)

    ax.set_xlabel('time')
    ax.set_ylabel('my overall health')
    fig.text(
        0.5, 0.05,
        '"Stove Ownership" from xkcd by Randall Munroe',
        ha='center')
```




"Stove Ownership" from xkcd by Randall Munroe

```
with plt.xkcd():
    # Based on "The Data So Far" from XKCD by Randall Munroe
    # https://xkcd.com/373/

    fig = plt.figure()
    ax = fig.add_axes((0.1, 0.2, 0.8, 0.7))
    ax.bar([0, 1], [0, 100], 0.25)
    ax.spines[['top', 'right']].set_visible(False)
    ax.xaxis.set_ticks_position('bottom')
    ax.set_xticks([0, 1])
    ax.set_xticklabels(['CONFIRMED BY\nEXPERIMENT', 'REFUTED BY\nEXPERIMENT'])
    ax.set_xlim([-0.5, 1.5])
    ax.set_yticks([])
    ax.set_ylim([0, 110])

    ax.set_title("CLAIMS OF SUPERNATURAL POWERS")

    fig.text(
        0.5, 0.05,
        '"The Data So Far" from xkcd by Randall Munroe',
        ha='center')

plt.show()
```




### 6.25.14 Animation

#### Decay

This example showcases:

- using a generator to drive an animation,
- changing axes limits during an animation.

Output generated via *matplotlib.animation.Animation.to_jshtml*.

```
import itertools

import matplotlib.pyplot as plt
import numpy as np

import matplotlib.animation as animation


def data_gen():
    for cnt in itertools.count():
```

(continues on next page)

(continued from previous page)

```
        t = cnt / 10
        yield t, np.sin(2*np.pi*t) * np.exp(-t/10.)


def init():
    ax.set_ylim(-1.1, 1.1)
    ax.set_xlim(0, 1)
    del xdata[:]
    del ydata[:]
    line.set_data(xdata, ydata)
    return line,

fig, ax = plt.subplots()
line, = ax.plot([], [], lw=2)
ax.grid()
xdata, ydata = [], []


def run(data):
    # update the data
    t, y = data
    xdata.append(t)
    ydata.append(y)
    xmin, xmax = ax.get_xlim()

    if t >= xmax:
        ax.set_xlim(xmin, 2*xmax)
        ax.figure.canvas.draw()
    line.set_data(xdata, ydata)

    return line,

# Only save last 100 frames, but run forever
ani = animation.FuncAnimation(fig, run, data_gen, interval=100, init_
 ↪func=init,
                              save_count=100)
plt.show()
```

**Total running time of the script:** (0 minutes 11.849 seconds)

### Animated histogram

Use histogram's *BarContainer* to draw a bunch of rectangles for an animated histogram.

```
import functools

import matplotlib.pyplot as plt
import numpy as np

import matplotlib.animation as animation
```

(continues on next page)

(continued from previous page)

```
# Setting up a random number generator with a fixed state for reproducibility.
rng = np.random.default_rng(seed=19680801)
# Fixing bin edges.
HIST_BINS = np.linspace(-4, 4, 100)

# Histogram our data with numpy.
data = rng.standard_normal(1000)
n, _ = np.histogram(data, HIST_BINS)
```

To animate the histogram, we need an `animate` function, which generates a random set of numbers and updates the heights of rectangles. The `animate` function updates the *Rectangle* patches on an instance of *BarContainer*.

```
def animate(frame_number, bar_container):
    # Simulate new data coming in.
    data = rng.standard_normal(1000)
    n, _ = np.histogram(data, HIST_BINS)
    for count, rect in zip(n, bar_container.patches):
        rect.set_height(count)

    return bar_container.patches
```

Using *hist()* allows us to get an instance of *BarContainer*, which is a collection of *Rectangle* instances. Since *FuncAnimation* will only pass the frame number parameter to the animation function, we use `functools.partial` to fix the `bar_container` parameter.

```
# Output generated via `matplotlib.animation.Animation.to_jshtml`.

fig, ax = plt.subplots()
_, _, bar_container = ax.hist(data, HIST_BINS, lw=1,
                              ec="yellow", fc="green", alpha=0.5)
ax.set_ylim(top=55)  # set safe limit to ensure that all data is visible.

anim = functools.partial(animate, bar_container=bar_container)
ani = animation.FuncAnimation(fig, anim, 50, repeat=False, blit=True)
plt.show()
```

Tags: plot-type: histogram animation

**Total running time of the script:** (0 minutes 7.885 seconds)

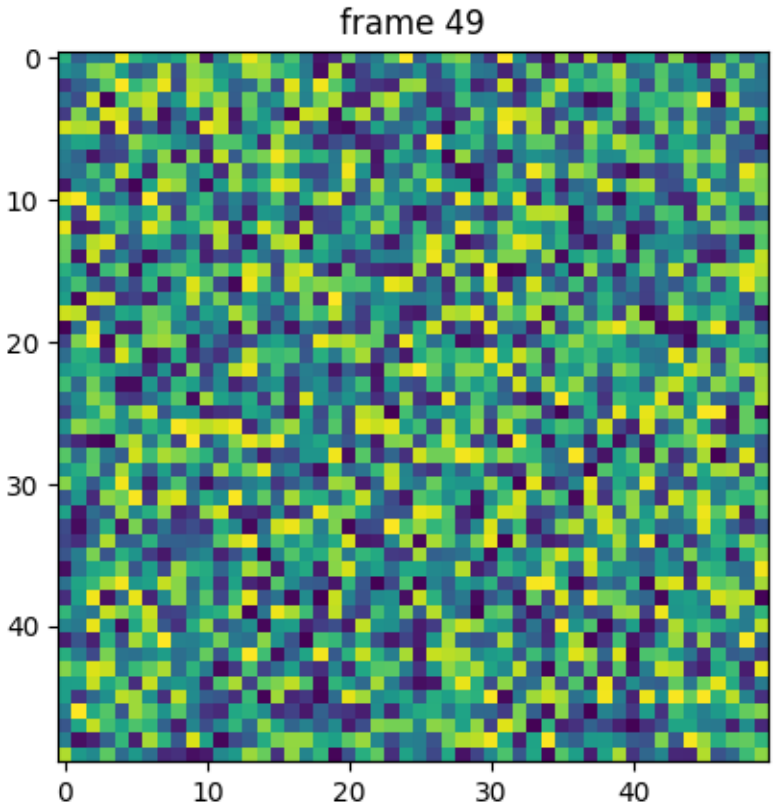
### pyplot animation

Generating an animation by calling *pause* between plotting commands.

The method shown here is only suitable for simple, low-performance use. For more demanding applications, look at the *animation* module and the examples that use it.

Note that calling time.sleep instead of *pause* would *not* work.

Output generated via *matplotlib.animation.Animation.to_jshtml*.




```
import matplotlib.pyplot as plt
import numpy as np

np.random.seed(19680801)
data = np.random.random((50, 50, 50))

fig, ax = plt.subplots()

for i, img in enumerate(data):
    ax.clear()
    ax.imshow(img)
    ax.set_title(f"frame {i}")
    # Note that using time.sleep does *not* work here!
```

(continues on next page)

(continued from previous page)

```
    plt.pause(0.1)
```

**Total running time of the script:** (0 minutes 10.333 seconds)

### The Bayes update

This animation displays the posterior estimate updates as it is refitted when new data arrives. The vertical line represents the theoretical value to which the plotted distribution should converge.

Output generated via *matplotlib.animation.Animation.to_jshtml*.

```
import math

import matplotlib.pyplot as plt
import numpy as np

from matplotlib.animation import FuncAnimation


def beta_pdf(x, a, b):
    return (x**(a-1) * (1-x)**(b-1) * math.gamma(a + b)
            / (math.gamma(a) * math.gamma(b)))


class UpdateDist:
    def __init__(self, ax, prob=0.5):
        self.success = 0
        self.prob = prob
        self.line, = ax.plot([], [], 'k-')
        self.x = np.linspace(0, 1, 200)
        self.ax = ax

        # Set up plot parameters
        self.ax.set_xlim(0, 1)
        self.ax.set_ylim(0, 10)
        self.ax.grid(True)

        # This vertical line represents the theoretical value, to
        # which the plotted distribution should converge.
        self.ax.axvline(prob, linestyle='--', color='black')

    def start(self):
        # Used for the *init_func* parameter of FuncAnimation; this is called␣
↪when
        # initializing the animation, and also after resizing the figure.
        return self.line,

    def __call__(self, i):
        # This way the plot can continuously run and we just keep
        # watching new realizations of the process
```

(continues on next page)

(continued from previous page)

```
        if i == 0:
            self.success = 0
            self.line.set_data([], [])
            return self.line,

        # Choose success based on exceed a threshold with a uniform pick
        if np.random.rand() < self.prob:
            self.success += 1
        y = beta_pdf(self.x, self.success + 1, (i - self.success) + 1)
        self.line.set_data(self.x, y)
        return self.line,

# Fixing random state for reproducibility
np.random.seed(19680801)


fig, ax = plt.subplots()
ud = UpdateDist(ax, prob=0.7)
anim = FuncAnimation(fig, ud, init_func=ud.start, frames=100, interval=100,␣
 ↪blit=True)
plt.show()
```

**Total running time of the script:** (0 minutes 8.899 seconds)

### The double pendulum problem

This animation illustrates the double pendulum problem.

Double pendulum formula translated from the C code at http://www.physics.usyd.edu.au/~wheat/dpend_html/solve_dpend.c

Output generated via *matplotlib.animation.Animation.to_jshtml*.

```
import matplotlib.pyplot as plt
import numpy as np
from numpy import cos, sin

import matplotlib.animation as animation

G = 9.8  # acceleration due to gravity, in m/s^2
L1 = 1.0  # length of pendulum 1 in m
L2 = 1.0  # length of pendulum 2 in m
L = L1 + L2  # maximal length of the combined pendulum
M1 = 1.0  # mass of pendulum 1 in kg
M2 = 1.0  # mass of pendulum 2 in kg
t_stop = 2.5  # how many seconds to simulate
history_len = 500  # how many trajectory points to display


def derivs(t, state):
    dydx = np.zeros_like(state)
```

(continues on next page)

(continued from previous page)

```
    dydx[0] = state[1]

    delta = state[2] - state[0]
    den1 = (M1+M2) * L1 - M2 * L1 * cos(delta) * cos(delta)
    dydx[1] = ((M2 * L1 * state[1] * state[1] * sin(delta) * cos(delta)
                + M2 * G * sin(state[2]) * cos(delta)
                + M2 * L2 * state[3] * state[3] * sin(delta)
                - (M1+M2) * G * sin(state[0]))
               / den1)

    dydx[2] = state[3]

    den2 = (L2/L1) * den1
    dydx[3] = ((- M2 * L2 * state[3] * state[3] * sin(delta) * cos(delta)
                + (M1+M2) * G * sin(state[0]) * cos(delta)
                - (M1+M2) * L1 * state[1] * state[1] * sin(delta)
                - (M1+M2) * G * sin(state[2]))
               / den2)

    return dydx

# create a time array from 0..t_stop sampled at 0.02 second steps
dt = 0.01
t = np.arange(0, t_stop, dt)

# th1 and th2 are the initial angles (degrees)
# w10 and w20 are the initial angular velocities (degrees per second)
th1 = 120.0
w1 = 0.0
th2 = -10.0
w2 = 0.0

# initial state
state = np.radians([th1, w1, th2, w2])

# integrate the ODE using Euler's method
y = np.empty((len(t), 4))
y[0] = state
for i in range(1, len(t)):
    y[i] = y[i - 1] + derivs(t[i - 1], y[i - 1]) * dt

# A more accurate estimate could be obtained e.g. using scipy:
#
#   y = scipy.integrate.solve_ivp(derivs, t[[0, -1]], state, t_eval=t).y.T

x1 = L1*sin(y[:, 0])
y1 = -L1*cos(y[:, 0])

x2 = L2*sin(y[:, 2]) + x1
y2 = -L2*cos(y[:, 2]) + y1
```

(continues on next page)

(continued from previous page)

```
fig = plt.figure(figsize=(5, 4))
ax = fig.add_subplot(autoscale_on=False, xlim=(-L, L), ylim=(-L, 1.))
ax.set_aspect('equal')
ax.grid()

line, = ax.plot([], [], 'o-', lw=2)
trace, = ax.plot([], [], '.-', lw=1, ms=2)
time_template = 'time = %.1fs'
time_text = ax.text(0.05, 0.9, '', transform=ax.transAxes)


def animate(i):
    thisx = [0, x1[i], x2[i]]
    thisy = [0, y1[i], y2[i]]

    history_x = x2[:i]
    history_y = y2[:i]

    line.set_data(thisx, thisy)
    trace.set_data(history_x, history_y)
    time_text.set_text(time_template % (i*dt))
    return line, trace, time_text


ani = animation.FuncAnimation(
    fig, animate, len(y), interval=dt*1000, blit=True)
plt.show()
```

**Total running time of the script:** (0 minutes 26.275 seconds)

### Animated image using a precomputed list of images

Output generated via *matplotlib.animation.Animation.to_jshtml*.

```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib.animation as animation

fig, ax = plt.subplots()


def f(x, y):
    return np.sin(x) + np.cos(y)

x = np.linspace(0, 2 * np.pi, 120)
y = np.linspace(0, 2 * np.pi, 100).reshape(-1, 1)

# ims is a list of lists, each row is a list of artists to draw in the
# current frame; here we are just animating one artist, the image, in
```

(continues on next page)

(continued from previous page)

```
# each frame
ims = []
for i in range(60):
    x += np.pi / 15
    y += np.pi / 30
    im = ax.imshow(f(x, y), animated=True)
    if i == 0:
        ax.imshow(f(x, y))  # show an initial one first
    ims.append([im])

ani = animation.ArtistAnimation(fig, ims, interval=50, blit=True,
                                repeat_delay=1000)

# To save the animation, use e.g.
#
# ani.save("movie.mp4")
#
# or
#
# writer = animation.FFMpegWriter(
#     fps=15, metadata=dict(artist='Me'), bitrate=1800)
# ani.save("movie.mp4", writer=writer)

plt.show()
```

**Total running time of the script:** (0 minutes 8.014 seconds)

### Frame grabbing

Use a MovieWriter directly to grab individual frames and write them to a file. This avoids any event loop integration, and thus works even with the Agg backend. This is not recommended for use in an interactive setting.

Output generated via *matplotlib.animation.Animation.to_jshtml*.

```
import numpy as np

import matplotlib

matplotlib.use("Agg")
import matplotlib.pyplot as plt

from matplotlib.animation import FFMpegWriter

# Fixing random state for reproducibility
np.random.seed(19680801)


metadata = dict(title='Movie Test', artist='Matplotlib',
                comment='Movie support!')
```

(continues on next page)

(continued from previous page)

```
writer = FFMpegWriter(fps=15, metadata=metadata)

fig = plt.figure()
l, = plt.plot([], [], 'k-o')

plt.xlim(-5, 5)
plt.ylim(-5, 5)

x0, y0 = 0, 0

with writer.saving(fig, "writer_test.mp4", 100):
    for i in range(100):
        x0 += 0.1 * np.random.randn()
        y0 += 0.1 * np.random.randn()
        l.set_data(x0, y0)
        writer.grab_frame()
```

### Multiple Axes animation

This example showcases:

- how animation across multiple subplots works,
- using a figure artist in the animation.

Output generated via *matplotlib.animation.Animation.to_jshtml*.

```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib.animation as animation
from matplotlib.patches import ConnectionPatch

fig, (axl, axr) = plt.subplots(
    ncols=2,
    sharey=True,
    figsize=(6, 2),
    gridspec_kw=dict(width_ratios=[1, 3], wspace=0),
)
axl.set_aspect(1)
axr.set_box_aspect(1 / 3)
axr.yaxis.set_visible(False)
axr.xaxis.set_ticks([0, np.pi, 2 * np.pi], ["0", r"$\pi$", r"$2\pi$"])

# draw circle with initial point in left Axes
x = np.linspace(0, 2 * np.pi, 50)
axl.plot(np.cos(x), np.sin(x), "k", lw=0.3)
point, = axl.plot(0, 0, "o")

# draw full curve to set view limits in right Axes
sine, = axr.plot(x, np.sin(x))
```

(continues on next page)

(continued from previous page)

```
# draw connecting line between both graphs
con = ConnectionPatch(
    (1, 0),
    (0, 0),
    "data",
    "data",
    axesA=ax1,
    axesB=ax2,
    color="C0",
    ls="dotted",
)
fig.add_artist(con)


def animate(i):
    x = np.linspace(0, i, int(i * 25 / np.pi))
    sine.set_data(x, np.sin(x))
    x, y = np.cos(i), np.sin(i)
    point.set_data([x], [y])
    con.xy1 = x, y
    con.xy2 = i, y
    return point, sine, con


ani = animation.FuncAnimation(
    fig,
    animate,
    interval=50,
    blit=False,  # blitting can't be used with Figure artists
    frames=x,
    repeat_delay=100,
)

plt.show()
```

**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.patches.ConnectionPatch*
- *matplotlib.animation.FuncAnimation*

Tags: component: axes animation

**Total running time of the script:** (0 minutes 4.060 seconds)

### Pausing and Resuming an Animation

This example showcases:

- using the Animation.pause() method to pause an animation.
- using the Animation.resume() method to resume an animation.

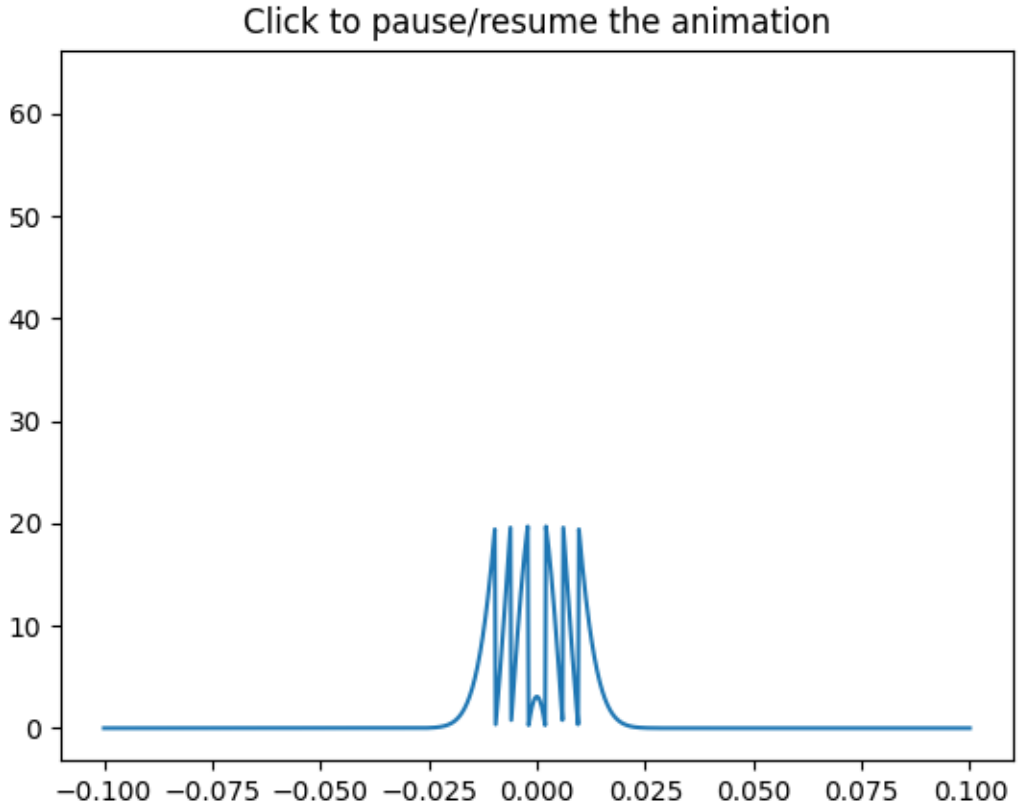
---

**Note:** This example exercises the interactive capabilities of Matplotlib, and this will not appear in the static documentation. Please run this code on your machine to see the interactivity.

You can copy and paste individual parts, or download the entire example using the link at the bottom of the page.

---

Output generated via *matplotlib.animation.Animation.to_jshtml*.




```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib.animation as animation
```

(continues on next page)

(continued from previous page)

```
class PauseAnimation:
    def __init__(self):
        fig, ax = plt.subplots()
        ax.set_title('Click to pause/resume the animation')
        x = np.linspace(-0.1, 0.1, 1000)

        # Start with a normal distribution
        self.n0 = (1.0 / ((4 * np.pi * 2e-4 * 0.1) ** 0.5)
                   * np.exp(-x ** 2 / (4 * 2e-4 * 0.1)))
        self.p, = ax.plot(x, self.n0)

        self.animation = animation.FuncAnimation(
            fig, self.update, frames=200, interval=50, blit=True)
        self.paused = False

        fig.canvas.mpl_connect('button_press_event', self.toggle_pause)

    def toggle_pause(self, *args, **kwargs):
        if self.paused:
            self.animation.resume()
        else:
            self.animation.pause()
        self.paused = not self.paused

    def update(self, i):
        self.n0 += i / 100 % 5
        self.p.set_ydata(self.n0 % 20)
        return (self.p,)


pa = PauseAnimation()
plt.show()
```

### Rain simulation

Simulates rain drops on a surface by animating the scale and opacity of 50 scatter points.

Author: Nicolas P. Rougier

Output generated via *matplotlib.animation.Animation.to_jshtml*.

```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.animation import FuncAnimation

# Fixing random state for reproducibility
np.random.seed(19680801)


# Create new Figure and an Axes which fills it.
```

(continues on next page)

(continued from previous page)

```
fig = plt.figure(figsize=(7, 7))
ax = fig.add_axes([0, 0, 1, 1], frameon=False)
ax.set_xlim(0, 1), ax.set_xticks([])
ax.set_ylim(0, 1), ax.set_yticks([])

# Create rain data
n_drops = 50
rain_drops = np.zeros(n_drops, dtype=[('position', float, (2,)),
                                      ('size',     float),
                                      ('growth',   float),
                                      ('color',    float, (4,))])

# Initialize the raindrops in random positions and with
# random growth rates.
rain_drops['position'] = np.random.uniform(0, 1, (n_drops, 2))
rain_drops['growth'] = np.random.uniform(50, 200, n_drops)

# Construct the scatter which we will update during animation
# as the raindrops develop.
scat = ax.scatter(rain_drops['position'][:, 0], rain_drops['position'][:, 1],
                  s=rain_drops['size'], lw=0.5, edgecolors=rain_drops['color
 ↪'],
                  facecolors='none')


def update(frame_number):
    # Get an index which we can use to re-spawn the oldest raindrop.
    current_index = frame_number % n_drops

    # Make all colors more transparent as time progresses.
    rain_drops['color'][:, 3] -= 1.0/len(rain_drops)
    rain_drops['color'][:, 3] = np.clip(rain_drops['color'][:, 3], 0, 1)

    # Make all circles bigger.
    rain_drops['size'] += rain_drops['growth']

    # Pick a new position for oldest rain drop, resetting its size,
    # color and growth factor.
    rain_drops['position'][current_index] = np.random.uniform(0, 1, 2)
    rain_drops['size'][current_index] = 5
    rain_drops['color'][current_index] = (0, 0, 0, 1)
    rain_drops['growth'][current_index] = np.random.uniform(50, 200)

    # Update the scatter collection, with the new colors, sizes and positions.
    scat.set_edgecolors(rain_drops['color'])
    scat.set_sizes(rain_drops['size'])
    scat.set_offsets(rain_drops['position'])


# Construct the animation, using the update function as the animation␣
 ↪director.
animation = FuncAnimation(fig, update, interval=10, save_count=100)
```

(continues on next page)

(continued from previous page)

```
plt.show()
```

**Total running time of the script:** (0 minutes 4.958 seconds)

### Animated 3D random walk

Output generated via *matplotlib.animation.Animation.to_jshtml*.

```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib.animation as animation

# Fixing random state for reproducibility
np.random.seed(19680801)


def random_walk(num_steps, max_step=0.05):
    """Return a 3D random walk as (num_steps, 3) array."""
    start_pos = np.random.random(3)
    steps = np.random.uniform(-max_step, max_step, size=(num_steps, 3))
    walk = start_pos + np.cumsum(steps, axis=0)
    return walk


def update_lines(num, walks, lines):
    for line, walk in zip(lines, walks):
        line.set_data_3d(walk[:num, :].T)
    return lines


# Data: 40 random walks as (num_steps, 3) arrays
num_steps = 30
walks = [random_walk(num_steps) for index in range(40)]

# Attaching 3D axis to the figure
fig = plt.figure()
ax = fig.add_subplot(projection="3d")

# Create lines initially without data
lines = [ax.plot([], [], [])[0] for _ in walks]

# Setting the Axes properties
ax.set(xlim3d=(0, 1), xlabel='X')
ax.set(ylim3d=(0, 1), ylabel='Y')
ax.set(zlim3d=(0, 1), zlabel='Z')

# Creating the Animation object
ani = animation.FuncAnimation(
    fig, update_lines, num_steps, fargs=(walks, lines), interval=100)
```

(continues on next page)

(continued from previous page)

```
plt.show()
```

**Total running time of the script:** (0 minutes 4.761 seconds)

### Animated line plot

Output generated via *matplotlib.animation.Animation.to_jshtml*.

```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib.animation as animation

fig, ax = plt.subplots()

x = np.arange(0, 2*np.pi, 0.01)
line, = ax.plot(x, np.sin(x))


def animate(i):
    line.set_ydata(np.sin(x + i / 50))  # update the data.
    return line,


ani = animation.FuncAnimation(
    fig, animate, interval=20, blit=True, save_count=50)

# To save the animation, use e.g.
#
# ani.save("movie.mp4")
#
# or
#
# writer = animation.FFMpegWriter(
#     fps=15, metadata=dict(artist='Me'), bitrate=1800)
# ani.save("movie.mp4", writer=writer)

plt.show()
```

**Total running time of the script:** (0 minutes 4.990 seconds)

### Animated scatter saved as GIF

Output generated via *matplotlib.animation.Animation.to_jshtml*.

```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib.animation as animation

fig, ax = plt.subplots()
ax.set_xlim([0, 10])

scat = ax.scatter(1, 0)
x = np.linspace(0, 10)


def animate(i):
    scat.set_offsets((x[i], 0))
    return scat,

ani = animation.FuncAnimation(fig, animate, repeat=True,
                                    frames=len(x) - 1, interval=50)

# To save the animation using Pillow as a gif
# writer = animation.PillowWriter(fps=15,
#                                 metadata=dict(artist='Me'),
#                                 bitrate=1800)
# ani.save('scatter.gif', writer=writer)

plt.show()
```

**Total running time of the script:** (0 minutes 4.235 seconds)

### Oscilloscope

Emulates an oscilloscope.

Output generated via *matplotlib.animation.Animation.to_jshtml*.

```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib.animation as animation
from matplotlib.lines import Line2D


class Scope:
    def __init__(self, ax, maxt=2, dt=0.02):
        self.ax = ax
        self.dt = dt
        self.maxt = maxt
        self.tdata = [0]
```

(continues on next page)

(continued from previous page)

```
        self.ydata = [0]
        self.line = Line2D(self.tdata, self.ydata)
        self.ax.add_line(self.line)
        self.ax.set_ylim(-.1, 1.1)
        self.ax.set_xlim(0, self.maxt)

    def update(self, y):
        lastt = self.tdata[-1]
        if lastt >= self.tdata[0] + self.maxt:  # reset the arrays
            self.tdata = [self.tdata[-1]]
            self.ydata = [self.ydata[-1]]
            self.ax.set_xlim(self.tdata[0], self.tdata[0] + self.maxt)
            self.ax.figure.canvas.draw()

        # This slightly more complex calculation avoids floating-point issues
        # from just repeatedly adding `self.dt` to the previous value.
        t = self.tdata[0] + len(self.tdata) * self.dt

        self.tdata.append(t)
        self.ydata.append(y)
        self.line.set_data(self.tdata, self.ydata)
        return self.line,


def emitter(p=0.1):
    """Return a random value in [0, 1) with probability p, else 0."""
    while True:
        v = np.random.rand()
        if v > p:
            yield 0.
        else:
            yield np.random.rand()


# Fixing random state for reproducibility
np.random.seed(19680801 // 10)


fig, ax = plt.subplots()
scope = Scope(ax)

# pass a generator in "emitter" to produce data for the update func
ani = animation.FuncAnimation(fig, scope.update, emitter, interval=50,
                              blit=True, save_count=100)

plt.show()
```

**Total running time of the script:** (0 minutes 9.587 seconds)

### MATPLOTLIB UNCHAINED

Comparative path demonstration of frequency from a fake signal of a pulsar (mostly known because of the cover for Joy Division's Unknown Pleasures).

Author: Nicolas P. Rougier

Output generated via *matplotlib.animation.Animation.to_jshtml*.

```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib.animation as animation

# Fixing random state for reproducibility
np.random.seed(19680801)


# Create new Figure with black background
fig = plt.figure(figsize=(8, 8), facecolor='black')

# Add a subplot with no frame
ax = plt.subplot(frameon=False)

# Generate random data
data = np.random.uniform(0, 1, (64, 75))
X = np.linspace(-1, 1, data.shape[-1])
G = 1.5 * np.exp(-4 * X ** 2)

# Generate line plots
lines = []
for i in range(len(data)):
    # Small reduction of the X extents to get a cheap perspective effect
    xscale = 1 - i / 200.
    # Same for linewidth (thicker strokes on bottom)
    lw = 1.5 - i / 100.0
    line, = ax.plot(xscale * X, i + G * data[i], color="w", lw=lw)
    lines.append(line)

# Set y limit (or first line is cropped because of thickness)
ax.set_ylim(-1, 70)

# No ticks
ax.set_xticks([])
ax.set_yticks([])

# 2 part titles to get different font weights
ax.text(0.5, 1.0, "MATPLOTLIB ", transform=ax.transAxes,
        ha="right", va="bottom", color="w",
        family="sans-serif", fontweight="light", fontsize=16)
ax.text(0.5, 1.0, "UNCHAINED", transform=ax.transAxes,
        ha="left", va="bottom", color="w",
        family="sans-serif", fontweight="bold", fontsize=16)
```

(continues on next page)

(continued from previous page)

```
def update(*args):
    # Shift all data to the right
    data[:, 1:] = data[:, :-1]

    # Fill-in new values
    data[:, 0] = np.random.uniform(0, 1, len(data))

    # Update data
    for i in range(len(data)):
        lines[i].set_ydata(i + G * data[i])

    # Return modified artists
    return lines

# Construct the animation, using the update function as the animation␣
 ↪director.
anim = animation.FuncAnimation(fig, update, interval=10, save_count=100)
plt.show()
```

**Total running time of the script:** (0 minutes 8.695 seconds)

### 6.25.15 Event handling

Matplotlib supports *event handling* with a GUI neutral event model, so you can connect to Matplotlib events without knowledge of what user interface Matplotlib will ultimately be plugged in to. This has two advantages: the code you write will be more portable, and Matplotlib events are aware of things like data coordinate space and which axes the event occurs in so you don't have to mess with low level transformation details to go from canvas space to data space. Object picking examples are also included.

#### Close Event

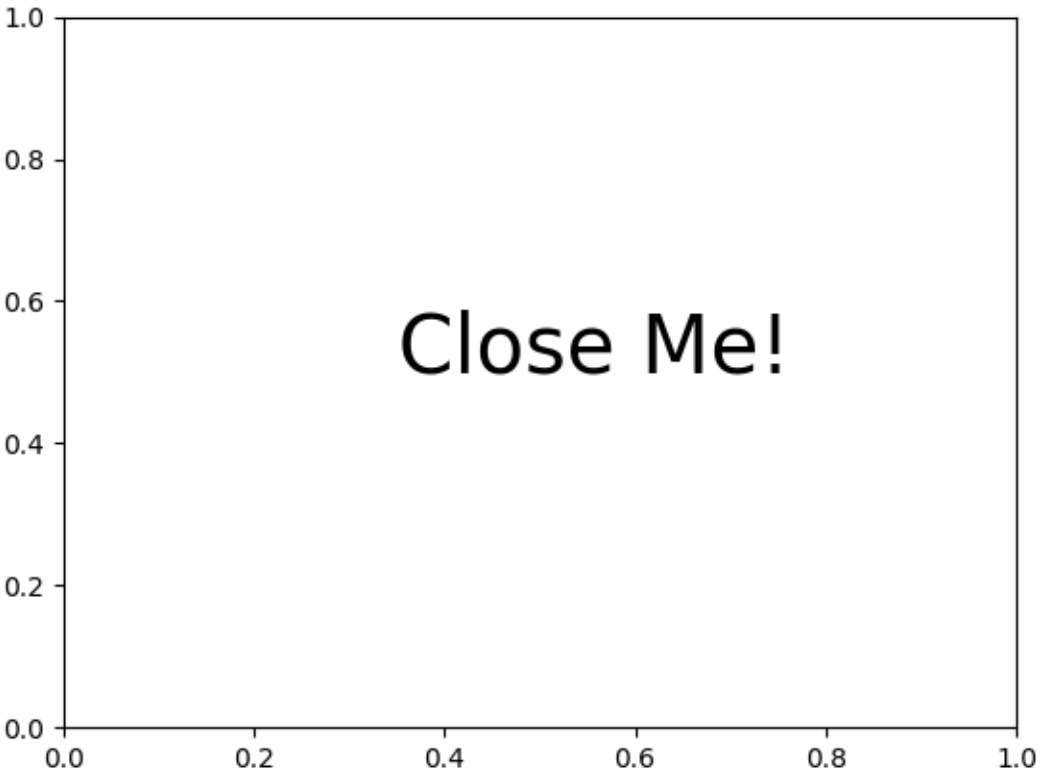
Example to show connecting events that occur when the figure closes.

---

**Note:** This example exercises the interactive capabilities of Matplotlib, and this will not appear in the static documentation. Please run this code on your machine to see the interactivity.

You can copy and paste individual parts, or download the entire example using the link at the bottom of the page.

---




```
import matplotlib.pyplot as plt


def on_close(event):
    print('Closed Figure!')

fig = plt.figure()
fig.canvas.mpl_connect('close_event', on_close)

plt.text(0.35, 0.5, 'Close Me!', dict(size=30))
plt.show()
```

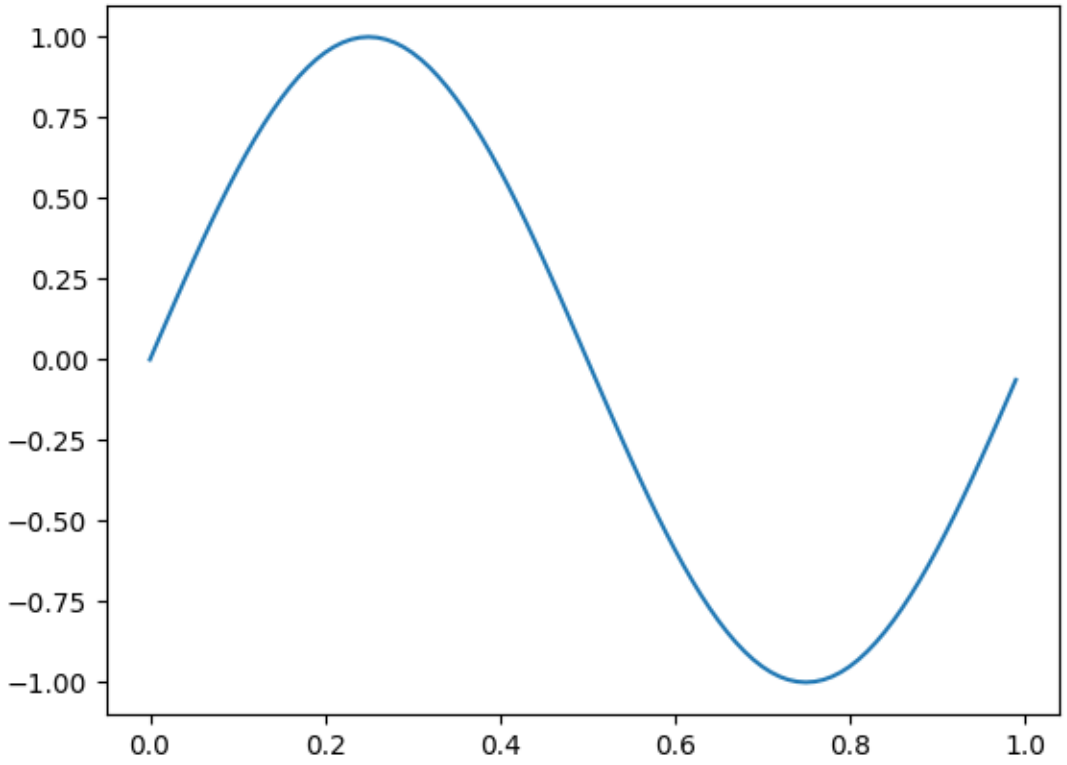
### Mouse move and click events

An example of how to interact with the plotting canvas by connecting to move and click events.

**Note:** This example exercises the interactive capabilities of Matplotlib, and this will not appear in the static documentation. Please run this code on your machine to see the interactivity.

You can copy and paste individual parts, or download the entire example using the link at the bottom of the

page.




```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.backend_bases import MouseButton

t = np.arange(0.0, 1.0, 0.01)
s = np.sin(2 * np.pi * t)
fig, ax = plt.subplots()
ax.plot(t, s)


def on_move(event):
    if event.inaxes:
        print(f'data coords {event.xdata} {event.ydata},',
              f'pixel coords {event.x} {event.y}')


def on_click(event):
    if event.button is MouseButton.LEFT:
        print('disconnecting callback')
```

(continues on next page)

(continued from previous page)

```
        plt.disconnect(binding_id)


binding_id = plt.connect('motion_notify_event', on_move)
plt.connect('button_press_event', on_click)

plt.show()
```

### Cross-hair cursor

This example adds a cross-hair as a data cursor. The cross-hair is implemented as regular line objects that are updated on mouse move.

We show three implementations:

1) A simple cursor implementation that redraws the figure on every mouse move. This is a bit slow, and you may notice some lag of the cross-hair movement.
2) A cursor that uses blitting for speedup of the rendering.
3) A cursor that snaps to data points.

Faster cursoring is possible using native GUI drawing, as in *Adding a cursor in WX*.

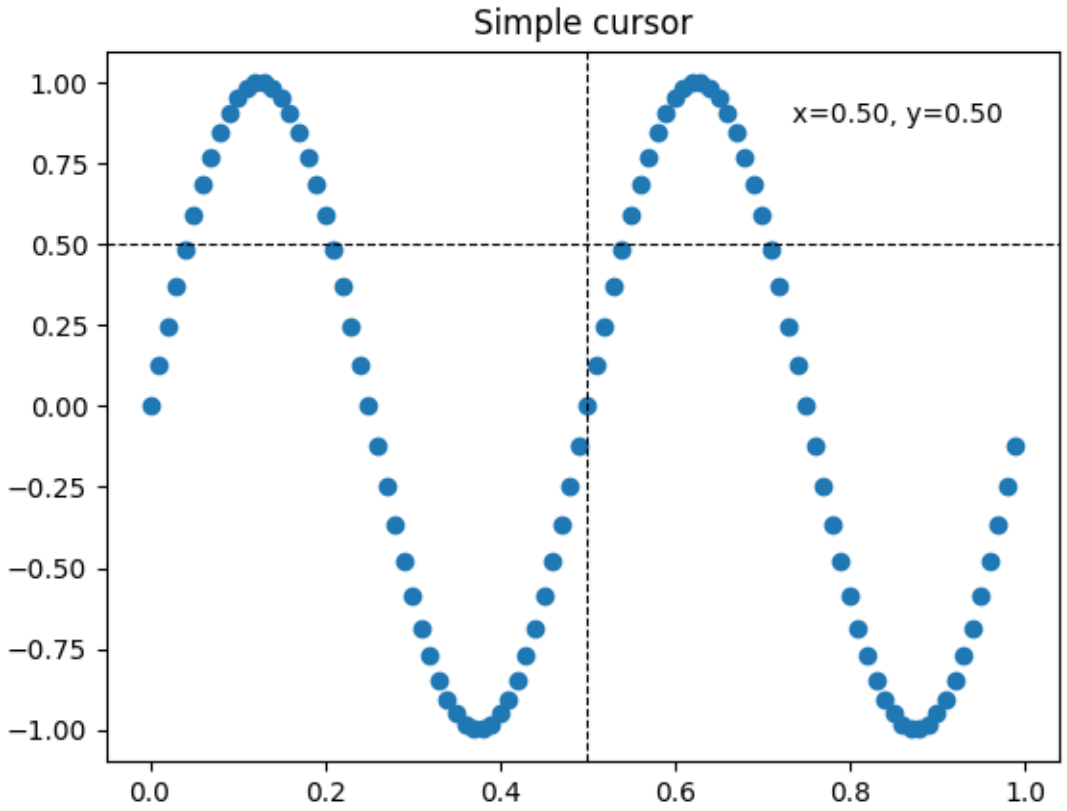
The mpldatacursor and mplcursors third-party packages can be used to achieve a similar effect.

```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.backend_bases import MouseEvent


class Cursor:
    """
    A cross hair cursor.
    """
    def __init__(self, ax):
        self.ax = ax
        self.horizontal_line = ax.axhline(color='k', lw=0.8, ls='--')
        self.vertical_line = ax.axvline(color='k', lw=0.8, ls='--')
        # text location in axes coordinates
        self.text = ax.text(0.72, 0.9, '', transform=ax.transAxes)

    def set_cross_hair_visible(self, visible):
        need_redraw = self.horizontal_line.get_visible() != visible
        self.horizontal_line.set_visible(visible)
        self.vertical_line.set_visible(visible)
        self.text.set_visible(visible)
        return need_redraw

    def on_mouse_move(self, event):
```

(continues on next page)

(continued from previous page)

```
        if not event.inaxes:
            need_redraw = self.set_cross_hair_visible(False)
            if need_redraw:
                self.ax.figure.canvas.draw()
        else:
            self.set_cross_hair_visible(True)
            x, y = event.xdata, event.ydata
            # update the line positions
            self.horizontal_line.set_ydata([y])
            self.vertical_line.set_xdata([x])
            self.text.set_text(f'x={x:1.2f}, y={y:1.2f}')
            self.ax.figure.canvas.draw()


x = np.arange(0, 1, 0.01)
y = np.sin(2 * 2 * np.pi * x)

fig, ax = plt.subplots()
ax.set_title('Simple cursor')
ax.plot(x, y, 'o')
cursor = Cursor(ax)
fig.canvas.mpl_connect('motion_notify_event', cursor.on_mouse_move)

# Simulate a mouse move to (0.5, 0.5), needed for online docs
t = ax.transData
MouseEvent(
    "motion_notify_event", ax.figure.canvas, *t.transform((0.5, 0.5))
)._process()
```




### Faster redrawing using blitting

This technique stores the rendered plot as a background image. Only the changed artists (cross-hair lines and text) are rendered anew. They are combined with the background using blitting.

This technique is significantly faster. It requires a bit more setup because the background has to be stored without the cross-hair lines (see `create_new_background()`). Additionally, a new background has to be created whenever the figure changes. This is achieved by connecting to the `'draw_event'`.

```
class BlittedCursor:
    """
    A cross-hair cursor using blitting for faster redraw.
    """
    def __init__(self, ax):
        self.ax = ax
        self.background = None
        self.horizontal_line = ax.axhline(color='k', lw=0.8, ls='--')
        self.vertical_line = ax.axvline(color='k', lw=0.8, ls='--')
        # text location in axes coordinates
        self.text = ax.text(0.72, 0.9, '', transform=ax.transAxes)
        self._creating_background = False
```

(continues on next page)

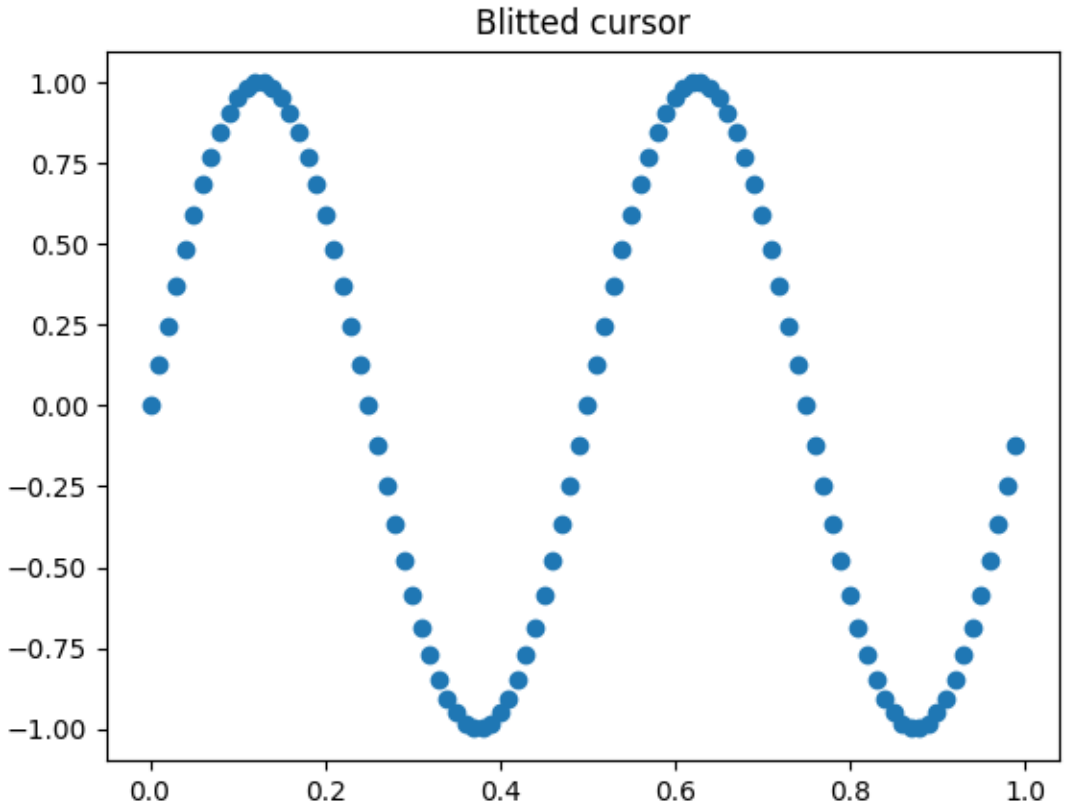
(continued from previous page)

```
        ax.figure.canvas.mpl_connect('draw_event', self.on_draw)

    def on_draw(self, event):
        self.create_new_background()

    def set_cross_hair_visible(self, visible):
        need_redraw = self.horizontal_line.get_visible() != visible
        self.horizontal_line.set_visible(visible)
        self.vertical_line.set_visible(visible)
        self.text.set_visible(visible)
        return need_redraw

    def create_new_background(self):
        if self._creating_background:
            # discard calls triggered from within this function
            return
        self._creating_background = True
        self.set_cross_hair_visible(False)
        self.ax.figure.canvas.draw()
        self.background = self.ax.figure.canvas.copy_from_bbox(self.ax.bbox)
        self.set_cross_hair_visible(True)
        self._creating_background = False

    def on_mouse_move(self, event):
        if self.background is None:
            self.create_new_background()
        if not event.inaxes:
            need_redraw = self.set_cross_hair_visible(False)
            if need_redraw:
                self.ax.figure.canvas.restore_region(self.background)
                self.ax.figure.canvas.blit(self.ax.bbox)
        else:
            self.set_cross_hair_visible(True)
            # update the line positions
            x, y = event.xdata, event.ydata
            self.horizontal_line.set_ydata([y])
            self.vertical_line.set_xdata([x])
            self.text.set_text(f'x={x:1.2f}, y={y:1.2f}')

            self.ax.figure.canvas.restore_region(self.background)
            self.ax.draw_artist(self.horizontal_line)
            self.ax.draw_artist(self.vertical_line)
            self.ax.draw_artist(self.text)
            self.ax.figure.canvas.blit(self.ax.bbox)


x = np.arange(0, 1, 0.01)
y = np.sin(2 * 2 * np.pi * x)

fig, ax = plt.subplots()
ax.set_title('Blitted cursor')
ax.plot(x, y, 'o')
```

(continues on next page)

(continued from previous page)

```
blitted_cursor = BlittedCursor(ax)
fig.canvas.mpl_connect('motion_notify_event', blitted_cursor.on_mouse_move)

# Simulate a mouse move to (0.5, 0.5), needed for online docs
t = ax.transData
MouseEvent(
    "motion_notify_event", ax.figure.canvas, *t.transform((0.5, 0.5))
)._process()
```




## Snapping to data points

The following cursor snaps its position to the data points of a *Line2D* object.

To save unnecessary redraws, the index of the last indicated data point is saved in `self._last_index`. A redraw is only triggered when the mouse moves far enough so that another data point must be selected. This reduces the lag due to many redraws. Of course, blitting could still be added on top for additional speedup.

```
class SnappingCursor:
    """
    A cross-hair cursor that snaps to the data point of a line, which is
```

(continues on next page)

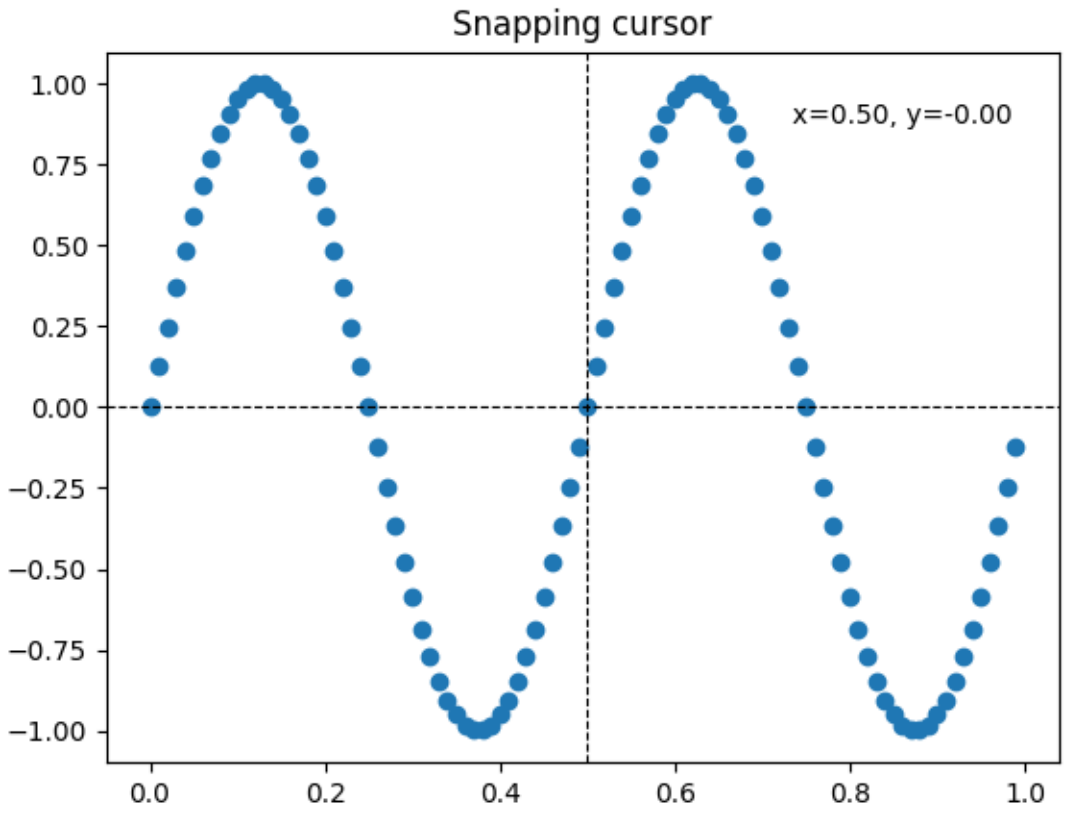
(continued from previous page)

```
    closest to the *x* position of the cursor.

    For simplicity, this assumes that *x* values of the data are sorted.
    """
    def __init__(self, ax, line):
        self.ax = ax
        self.horizontal_line = ax.axhline(color='k', lw=0.8, ls='--')
        self.vertical_line = ax.axvline(color='k', lw=0.8, ls='--')
        self.x, self.y = line.get_data()
        self._last_index = None
        # text location in axes coords
        self.text = ax.text(0.72, 0.9, '', transform=ax.transAxes)

    def set_cross_hair_visible(self, visible):
        need_redraw = self.horizontal_line.get_visible() != visible
        self.horizontal_line.set_visible(visible)
        self.vertical_line.set_visible(visible)
        self.text.set_visible(visible)
        return need_redraw

    def on_mouse_move(self, event):
        if not event.inaxes:
            self._last_index = None
            need_redraw = self.set_cross_hair_visible(False)
            if need_redraw:
                self.ax.figure.canvas.draw()
        else:
            self.set_cross_hair_visible(True)
            x, y = event.xdata, event.ydata
            index = min(np.searchsorted(self.x, x), len(self.x) - 1)
            if index == self._last_index:
                return  # still on the same data point. Nothing to do.
            self._last_index = index
            x = self.x[index]
            y = self.y[index]
            # update the line positions
            self.horizontal_line.set_ydata([y])
            self.vertical_line.set_xdata([x])
            self.text.set_text(f'x={x:1.2f}, y={y:1.2f}')
            self.ax.figure.canvas.draw()


x = np.arange(0, 1, 0.01)
y = np.sin(2 * 2 * np.pi * x)

fig, ax = plt.subplots()
ax.set_title('Snapping cursor')
line, = ax.plot(x, y, 'o')
snap_cursor = SnappingCursor(ax, line)
fig.canvas.mpl_connect('motion_notify_event', snap_cursor.on_mouse_move)

# Simulate a mouse move to (0.5, 0.5), needed for online docs
```

(continues on next page)

(continued from previous page)

```
t = ax.transData
MouseEvent(
    "motion_notify_event", ax.figure.canvas, *t.transform((0.5, 0.5))
)._process()

plt.show()
```




## Data browser

Connecting data between multiple canvases.

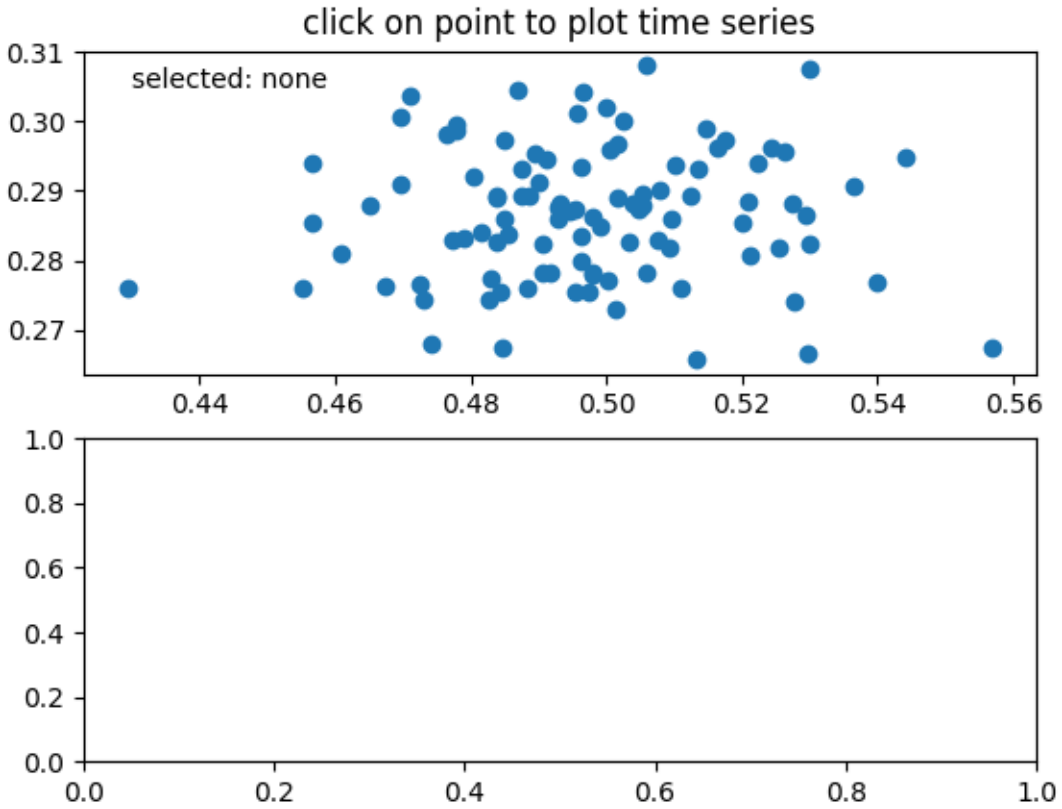
This example covers how to interact data with multiple canvases. This lets you select and highlight a point on one axis, and generating the data of that point on the other axis.

---

**Note:** This example exercises the interactive capabilities of Matplotlib, and this will not appear in the static documentation. Please run this code on your machine to see the interactivity.

You can copy and paste individual parts, or download the entire example using the link at the bottom of the page.

---




```
import numpy as np


class PointBrowser:
    """
    Click on a point to select and highlight it -- the data that
    generated the point will be shown in the lower Axes.  Use the 'n'
    and 'p' keys to browse through the next and previous points
    """

    def __init__(self):
        self.lastind = 0

        self.text = ax.text(0.05, 0.95, 'selected: none',
                            transform=ax.transAxes, va='top')
        self.selected, = ax.plot([xs[0]], [ys[0]], 'o', ms=12, alpha=0.4,
                                 color='yellow', visible=False)

    def on_press(self, event):
        if self.lastind is None:
            return
        if event.key not in ('n', 'p'):
            return
```

(continues on next page)

(continued from previous page)

```
        if event.key == 'n':
            inc = 1
        else:
            inc = -1

        self.lastind += inc
        self.lastind = np.clip(self.lastind, 0, len(xs) - 1)
        self.update()

    def on_pick(self, event):

        if event.artist != line:
            return True

        N = len(event.ind)
        if not N:
            return True

        # the click locations
        x = event.mouseevent.xdata
        y = event.mouseevent.ydata

        distances = np.hypot(x - xs[event.ind], y - ys[event.ind])
        indmin = distances.argmin()
        dataind = event.ind[indmin]

        self.lastind = dataind
        self.update()

    def update(self):
        if self.lastind is None:
            return

        dataind = self.lastind

        ax2.clear()
        ax2.plot(X[dataind])

        ax2.text(0.05, 0.9, f'mu={xs[dataind]:1.3f}\nsigma={ys[dataind]:1.3f}
↪',
                 transform=ax2.transAxes, va='top')
        ax2.set_ylim(-0.5, 1.5)
        self.selected.set_visible(True)
        self.selected.set_data(xs[dataind], ys[dataind])

        self.text.set_text('selected: %d' % dataind)
        fig.canvas.draw()


if __name__ == '__main__':
    import matplotlib.pyplot as plt
```

(continues on next page)

(continued from previous page)

```
    # Fixing random state for reproducibility
    np.random.seed(19680801)

    X = np.random.rand(100, 200)
    xs = np.mean(X, axis=1)
    ys = np.std(X, axis=1)

    fig, (ax, ax2) = plt.subplots(2, 1)
    ax.set_title('click on point to plot time series')
    line, = ax.plot(xs, ys, 'o', picker=True, pickradius=5)

    browser = PointBrowser()

    fig.canvas.mpl_connect('pick_event', browser.on_pick)
    fig.canvas.mpl_connect('key_press_event', browser.on_press)

    plt.show()
```

### Figure/Axes enter and leave events

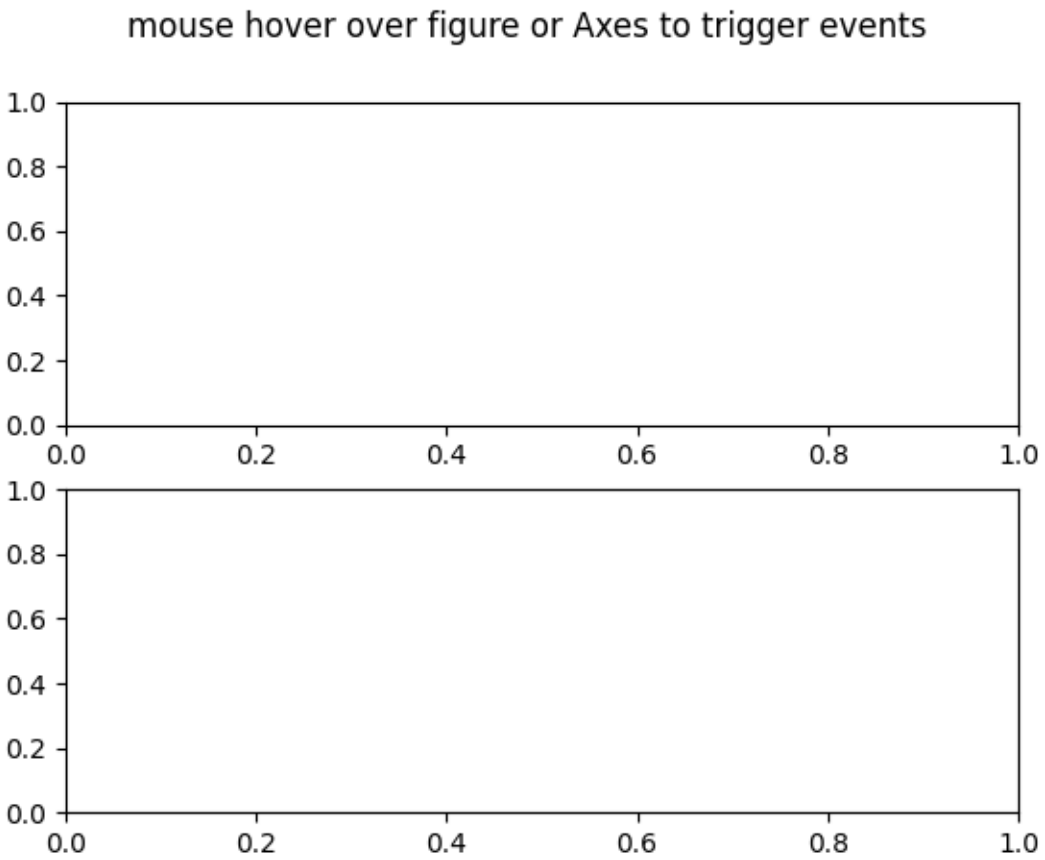
Illustrate the figure and Axes enter and leave events by changing the frame colors on enter and leave.

---

**Note:** This example exercises the interactive capabilities of Matplotlib, and this will not appear in the static documentation. Please run this code on your machine to see the interactivity.

You can copy and paste individual parts, or download the entire example using the link at the bottom of the page.

---




```
import matplotlib.pyplot as plt


def on_enter_axes(event):
    print('enter_axes', event.inaxes)
    event.inaxes.patch.set_facecolor('yellow')
    event.canvas.draw()


def on_leave_axes(event):
    print('leave_axes', event.inaxes)
    event.inaxes.patch.set_facecolor('white')
    event.canvas.draw()


def on_enter_figure(event):
    print('enter_figure', event.canvas.figure)
    event.canvas.figure.patch.set_facecolor('red')
    event.canvas.draw()


def on_leave_figure(event):
    print('leave_figure', event.canvas.figure)
```

(continues on next page)

(continued from previous page)

```
    event.canvas.figure.patch.set_facecolor('grey')
    event.canvas.draw()


fig, axs = plt.subplots(2, 1)
fig.suptitle('mouse hover over figure or Axes to trigger events')

fig.canvas.mpl_connect('figure_enter_event', on_enter_figure)
fig.canvas.mpl_connect('figure_leave_event', on_leave_figure)
fig.canvas.mpl_connect('axes_enter_event', on_enter_axes)
fig.canvas.mpl_connect('axes_leave_event', on_leave_axes)

plt.show()
```

### Interactive functions

This provides examples of uses of interactive functions, such as ginput, waitforbuttonpress and manual clabel placement.

---

**Note:** This example exercises the interactive capabilities of Matplotlib, and this will not appear in the static documentation. Please run this code on your machine to see the interactivity.

You can copy and paste individual parts, or download the entire example using the link at the bottom of the page.

---

```
import time

import matplotlib.pyplot as plt
import numpy as np


def tellme(s):
    print(s)
    plt.title(s, fontsize=16)
    plt.draw()
```

Define a triangle by clicking three points

```
plt.figure()
plt.xlim(0, 1)
plt.ylim(0, 1)

tellme('You will define a triangle, click to begin')

plt.waitforbuttonpress()

while True:
    pts = []
```

(continues on next page)

(continued from previous page)

```
    while len(pts) < 3:
        tellme('Select 3 corners with mouse')
        pts = np.asarray(plt.ginput(3, timeout=-1))
        if len(pts) < 3:
            tellme('Too few points, starting over')
            time.sleep(1)  # Wait a second

    ph = plt.fill(pts[:, 0], pts[:, 1], 'r', lw=2)

    tellme('Happy? Key click for yes, mouse click for no')

    if plt.waitforbuttonpress():
        break

    # Get rid of fill
    for p in ph:
        p.remove()
```

Now contour according to distance from triangle corners - just an example

```
# Define a nice function of distance from individual pts
def f(x, y, pts):
    z = np.zeros_like(x)
    for p in pts:
        z = z + 1/(np.sqrt((x - p[0])**2 + (y - p[1])**2))
    return 1/z


X, Y = np.meshgrid(np.linspace(-1, 1, 51), np.linspace(-1, 1, 51))
Z = f(X, Y, pts)

CS = plt.contour(X, Y, Z, 20)

tellme('Use mouse to select contour label locations, middle button to finish')
CL = plt.clabel(CS, manual=True)
```

Now do a zoom

```
tellme('Now do a nested zoom, click to begin')
plt.waitforbuttonpress()

while True:
    tellme('Select two corners of zoom, middle mouse button to finish')
    pts = plt.ginput(2, timeout=-1)
    if len(pts) < 2:
        break
    (x0, y0), (x1, y1) = pts
    xmin, xmax = sorted([x0, x1])
    ymin, ymax = sorted([y0, y1])
    plt.xlim(xmin, xmax)
    plt.ylim(ymin, ymax)
```

(continues on next page)

(continued from previous page)

```
tellme('All Done!')
plt.show()
```

### Scroll event

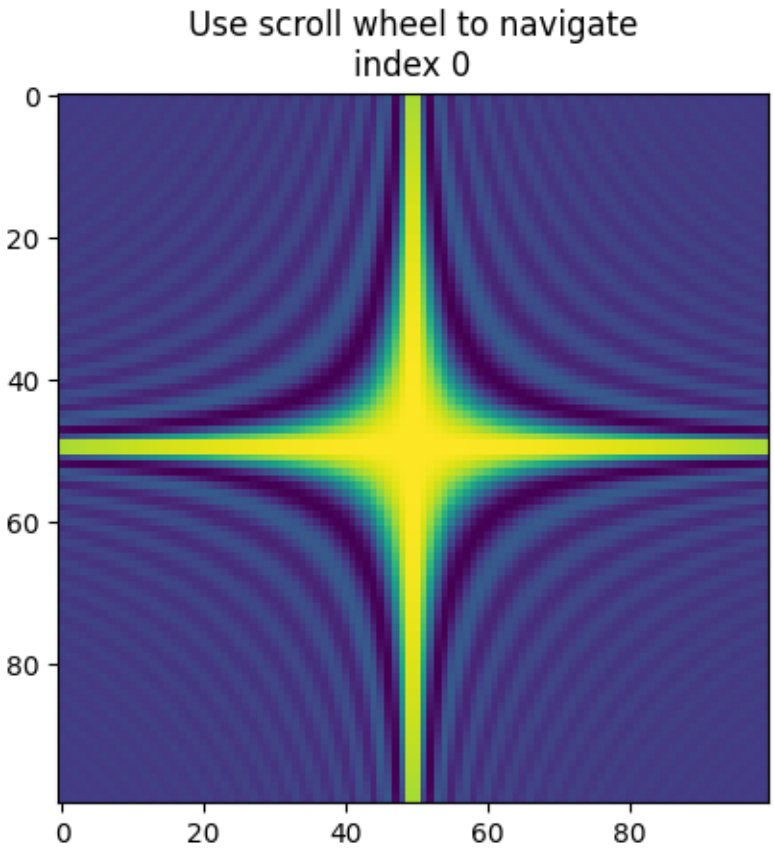
In this example a scroll wheel event is used to scroll through 2D slices of 3D data.

---

**Note:** This example exercises the interactive capabilities of Matplotlib, and this will not appear in the static documentation. Please run this code on your machine to see the interactivity.

You can copy and paste individual parts, or download the entire example using the link at the bottom of the page.

---




```
import matplotlib.pyplot as plt
import numpy as np
```

(continues on next page)

(continued from previous page)

```
class IndexTracker:
    def __init__(self, ax, X):
        self.index = 0
        self.X = X
        self.ax = ax
        self.im = ax.imshow(self.X[:, :, self.index])
        self.update()

    def on_scroll(self, event):
        print(event.button, event.step)
        increment = 1 if event.button == 'up' else -1
        max_index = self.X.shape[-1] - 1
        self.index = np.clip(self.index + increment, 0, max_index)
        self.update()

    def update(self):
        self.im.set_data(self.X[:, :, self.index])
        self.ax.set_title(
            f'Use scroll wheel to navigate\nindex {self.index}')
        self.im.axes.figure.canvas.draw()


x, y, z = np.ogrid[-10:10:100j, -10:10:100j, 1:10:20j]
X = np.sin(x * y * z) / (x * y * z)

fig, ax = plt.subplots()
# create an IndexTracker and make sure it lives during the whole
# lifetime of the figure by assigning it to a variable
tracker = IndexTracker(ax, X)

fig.canvas.mpl_connect('scroll_event', tracker.on_scroll)
plt.show()
```

### Keypress event

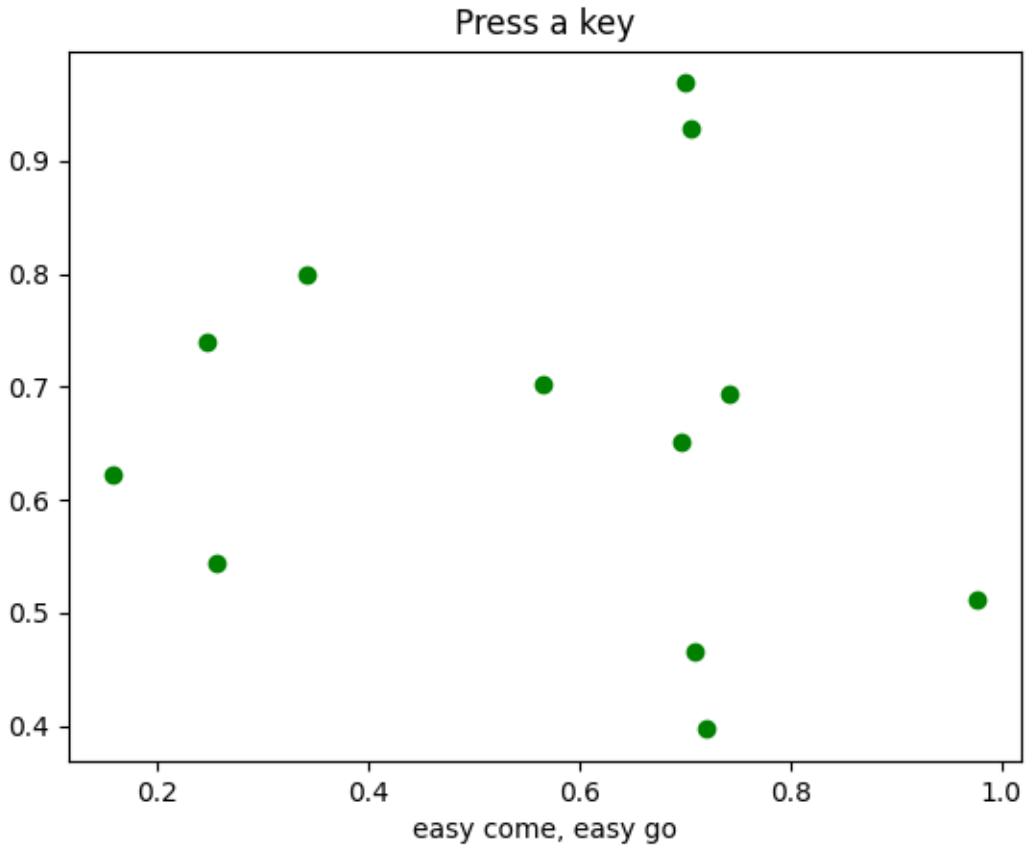
Show how to connect to keypress events.

---

**Note:** This example exercises the interactive capabilities of Matplotlib, and this will not appear in the static documentation. Please run this code on your machine to see the interactivity.

You can copy and paste individual parts, or download the entire example using the link at the bottom of the page.

---




```
import sys

import matplotlib.pyplot as plt
import numpy as np


def on_press(event):
    print('press', event.key)
    sys.stdout.flush()
    if event.key == 'x':
        visible = xl.get_visible()
        xl.set_visible(not visible)
        fig.canvas.draw()


# Fixing random state for reproducibility
np.random.seed(19680801)

fig, ax = plt.subplots()

fig.canvas.mpl_connect('key_press_event', on_press)

ax.plot(np.random.rand(12), np.random.rand(12), 'go')
```

(continues on next page)

(continued from previous page)

```
xl = ax.set_xlabel('easy come, easy go')
ax.set_title('Press a key')
plt.show()
```

### Lasso Demo

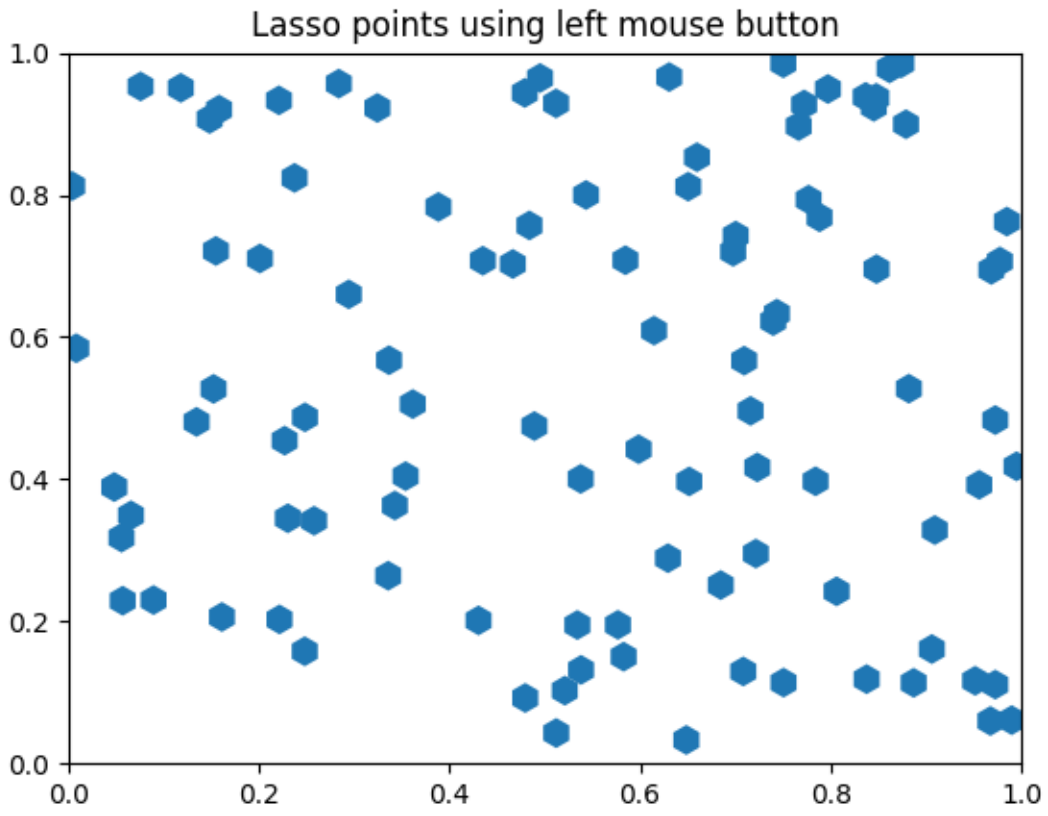
Use a lasso to select a set of points and get the indices of the selected points. A callback is used to change the color of the selected points.

---

**Note:** This example exercises the interactive capabilities of Matplotlib, and this will not appear in the static documentation. Please run this code on your machine to see the interactivity.

You can copy and paste individual parts, or download the entire example using the link at the bottom of the page.

---




```
import matplotlib.pyplot as plt
import numpy as np
```

(continues on next page)

(continued from previous page)

```
from matplotlib import colors as mcolors
from matplotlib import path
from matplotlib.collections import RegularPolyCollection
from matplotlib.widgets import Lasso


class LassoManager:
    def __init__(self, ax, data):
        # The information of whether a point has been selected or not is␣
 ↪stored in the
        # collection's array (0 = out, 1 = in), which then gets colormapped␣
 ↪to blue
        # (out) and red (in).
        self.collection = RegularPolyCollection(
            6, sizes=(100,), offset_transform=ax.transData,
            offsets=data, array=np.zeros(len(data)),
            clim=(0, 1), cmap=mcolors.ListedColormap(["tab:blue", "tab:red"]))
        ax.add_collection(self.collection)
        canvas = ax.figure.canvas
        canvas.mpl_connect('button_press_event', self.on_press)
        canvas.mpl_connect('button_release_event', self.on_release)

    def callback(self, verts):
        data = self.collection.get_offsets()
        self.collection.set_array(path.Path(verts).contains_points(data))
        canvas = self.collection.figure.canvas
        canvas.draw_idle()
        del self.lasso

    def on_press(self, event):
        canvas = self.collection.figure.canvas
        if event.inaxes is not self.collection.axes or canvas.widgetlock.
 ↪locked():
            return
        self.lasso = Lasso(event.inaxes, (event.xdata, event.ydata), self.
 ↪callback)
        canvas.widgetlock(self.lasso)  # acquire a lock on the widget drawing

    def on_release(self, event):
        canvas = self.collection.figure.canvas
        if hasattr(self, 'lasso') and canvas.widgetlock.isowner(self.lasso):
            canvas.widgetlock.release(self.lasso)


if __name__ == '__main__':
    np.random.seed(19680801)
    ax = plt.figure().add_subplot(
        xlim=(0, 1), ylim=(0, 1), title='Lasso points using left mouse button
 ↪')
    manager = LassoManager(ax, np.random.rand(100, 2))
    plt.show()
```

### Legend picking

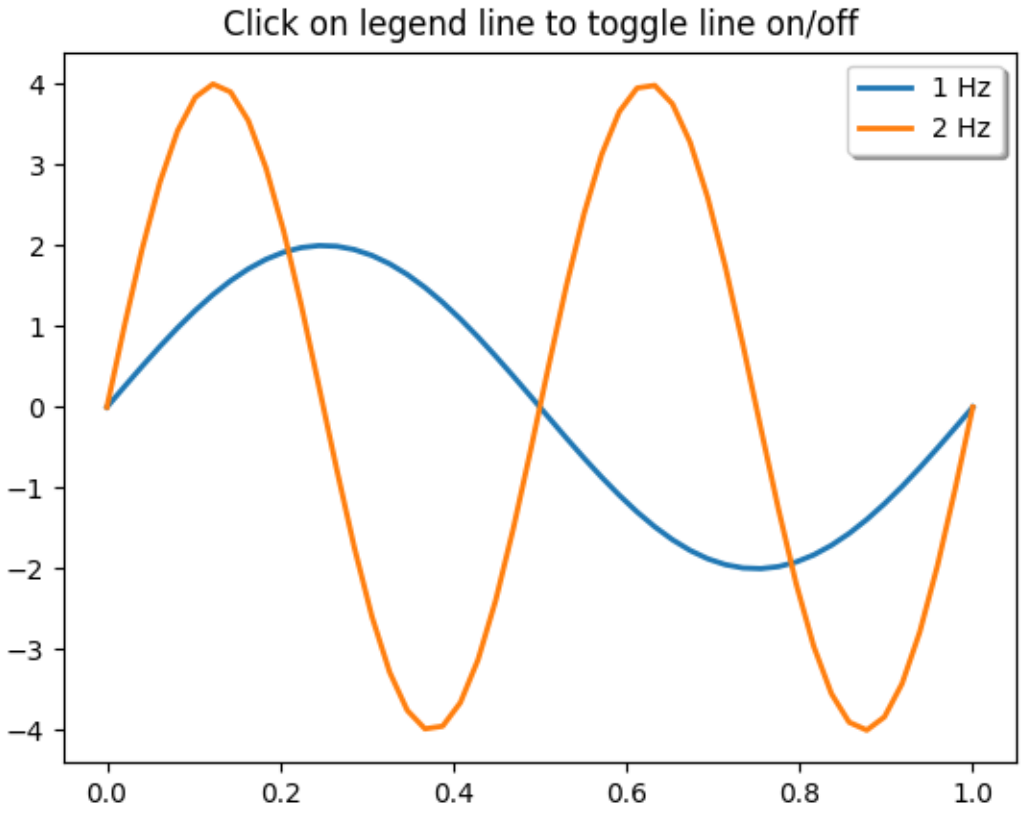
Enable picking on the legend to toggle the original line on and off

---

**Note:** This example exercises the interactive capabilities of Matplotlib, and this will not appear in the static documentation. Please run this code on your machine to see the interactivity.

You can copy and paste individual parts, or download the entire example using the link at the bottom of the page.

---




```
import matplotlib.pyplot as plt
import numpy as np

t = np.linspace(0, 1)
y1 = 2 * np.sin(2 * np.pi * t)
y2 = 4 * np.sin(2 * np.pi * 2 * t)

fig, ax = plt.subplots()
ax.set_title('Click on legend line to toggle line on/off')
(line1, ) = ax.plot(t, y1, lw=2, label='1 Hz')
(line2, ) = ax.plot(t, y2, lw=2, label='2 Hz')
```

(continues on next page)

(continued from previous page)

```
leg = ax.legend(fancybox=True, shadow=True)

lines = [line1, line2]
map_legend_to_ax = {}  # Will map legend lines to original lines.

pickradius = 5  # Points (Pt). How close the click needs to be to trigger an␣
 ↪event.

for legend_line, ax_line in zip(leg.get_lines(), lines):
    legend_line.set_picker(pickradius)  # Enable picking on the legend line.
    map_legend_to_ax[legend_line] = ax_line


def on_pick(event):
    # On the pick event, find the original line corresponding to the legend
    # proxy line, and toggle its visibility.
    legend_line = event.artist

    # Do nothing if the source of the event is not a legend line.
    if legend_line not in map_legend_to_ax:
        return

    ax_line = map_legend_to_ax[legend_line]
    visible = not ax_line.get_visible()
    ax_line.set_visible(visible)
    # Change the alpha on the line in the legend, so we can see what lines
    # have been toggled.
    legend_line.set_alpha(1.0 if visible else 0.2)
    fig.canvas.draw()


fig.canvas.mpl_connect('pick_event', on_pick)

# Works even if the legend is draggable. This is independent from picking␣
 ↪legend lines.
leg.set_draggable(True)

plt.show()
```

### Looking Glass

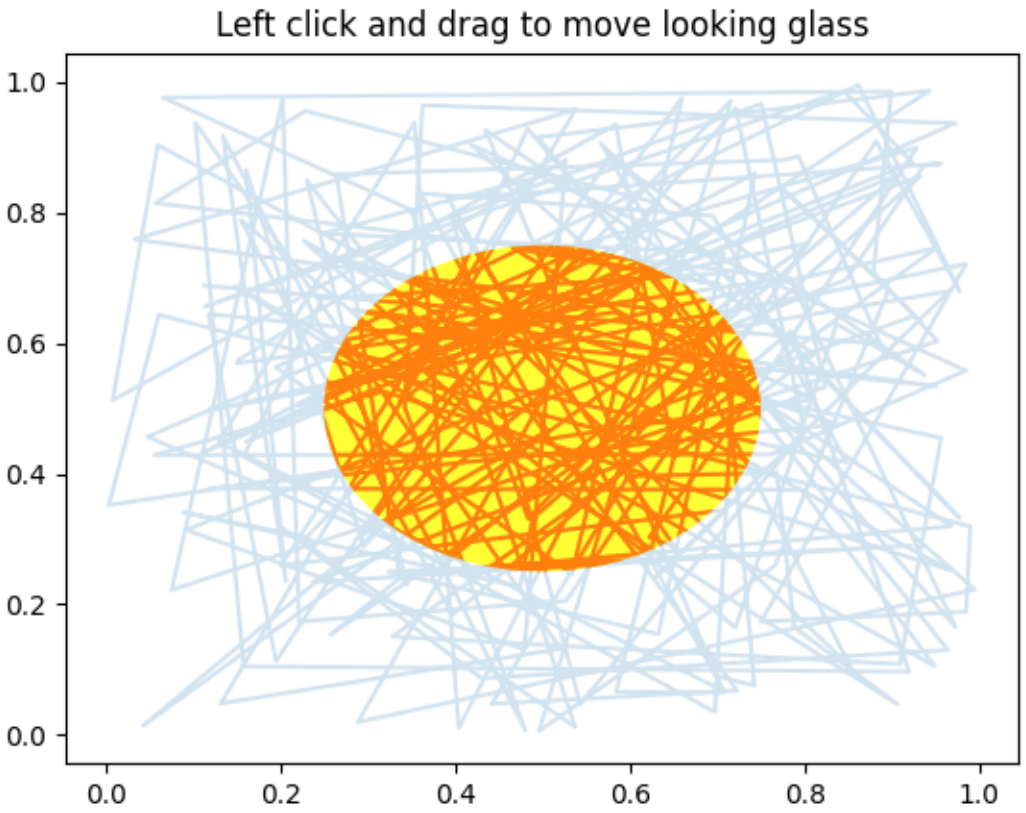
Example using mouse events to simulate a looking glass for inspecting data.

---

**Note:** This example exercises the interactive capabilities of Matplotlib, and this will not appear in the static documentation. Please run this code on your machine to see the interactivity.

You can copy and paste individual parts, or download the entire example using the link at the bottom of the page.

---




```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib.patches as patches

# Fixing random state for reproducibility
np.random.seed(19680801)

x, y = np.random.rand(2, 200)

fig, ax = plt.subplots()
circ = patches.Circle((0.5, 0.5), 0.25, alpha=0.8, fc='yellow')
ax.add_patch(circ)


ax.plot(x, y, alpha=0.2)
line, = ax.plot(x, y, alpha=1.0, clip_path=circ)
ax.set_title("Left click and drag to move looking glass")


class EventHandler:
    def __init__(self):
        fig.canvas.mpl_connect('button_press_event', self.on_press)
```

(continues on next page)

(continued from previous page)

```
        fig.canvas.mpl_connect('button_release_event', self.on_release)
        fig.canvas.mpl_connect('motion_notify_event', self.on_move)
        self.x0, self.y0 = circ.center
        self.pressevent = None

    def on_press(self, event):
        if event.inaxes != ax:
            return

        if not circ.contains(event)[0]:
            return

        self.pressevent = event

    def on_release(self, event):
        self.pressevent = None
        self.x0, self.y0 = circ.center

    def on_move(self, event):
        if self.pressevent is None or event.inaxes != self.pressevent.inaxes:
            return

        dx = event.xdata - self.pressevent.xdata
        dy = event.ydata - self.pressevent.ydata
        circ.center = self.x0 + dx, self.y0 + dy
        line.set_clip_path(circ)
        fig.canvas.draw()

handler = EventHandler()
plt.show()
```

### Path editor

Sharing events across GUIs.

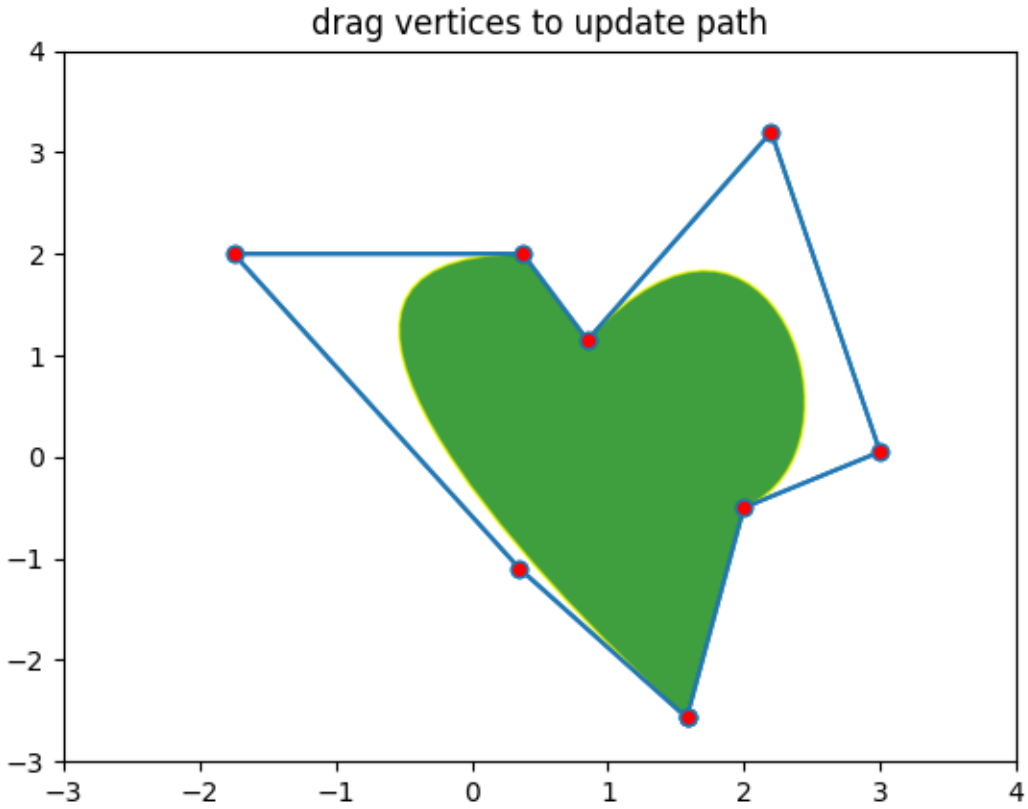
This example demonstrates a cross-GUI application using Matplotlib event handling to interact with and modify objects on the canvas.

---

**Note:** This example exercises the interactive capabilities of Matplotlib, and this will not appear in the static documentation. Please run this code on your machine to see the interactivity.

You can copy and paste individual parts, or download the entire example using the link at the bottom of the page.

---




```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.backend_bases import MouseButton
from matplotlib.patches import PathPatch
from matplotlib.path import Path

fig, ax = plt.subplots()

pathdata = [
    (Path.MOVETO, (1.58, -2.57)),
    (Path.CURVE4, (0.35, -1.1)),
    (Path.CURVE4, (-1.75, 2.0)),
    (Path.CURVE4, (0.375, 2.0)),
    (Path.LINETO, (0.85, 1.15)),
    (Path.CURVE4, (2.2, 3.2)),
    (Path.CURVE4, (3, 0.05)),
    (Path.CURVE4, (2.0, -0.5)),
    (Path.CLOSEPOLY, (1.58, -2.57)),
]

codes, verts = zip(*pathdata)
path = Path(verts, codes)
```

(continues on next page)

(continued from previous page)

```
patch = PathPatch(
    path, facecolor='green', edgecolor='yellow', alpha=0.5)
ax.add_patch(patch)


class PathInteractor:
    """
    A path editor.

    Press 't' to toggle vertex markers on and off.  When vertex markers are␣
 ↪on,
    they can be dragged with the mouse.
    """

    showverts = True
    epsilon = 5  # max pixel distance to count as a vertex hit

    def __init__(self, pathpatch):

        self.ax = pathpatch.axes
        canvas = self.ax.figure.canvas
        self.pathpatch = pathpatch
        self.pathpatch.set_animated(True)

        x, y = zip(*self.pathpatch.get_path().vertices)

        self.line, = ax.plot(
            x, y, marker='o', markerfacecolor='r', animated=True)

        self._ind = None  # the active vertex

        canvas.mpl_connect('draw_event', self.on_draw)
        canvas.mpl_connect('button_press_event', self.on_button_press)
        canvas.mpl_connect('key_press_event', self.on_key_press)
        canvas.mpl_connect('button_release_event', self.on_button_release)
        canvas.mpl_connect('motion_notify_event', self.on_mouse_move)
        self.canvas = canvas

    def get_ind_under_point(self, event):
        """
        Return the index of the point closest to the event position or *None*
        if no point is within ``self.epsilon`` to the event position.
        """
        xy = self.pathpatch.get_path().vertices
        xyt = self.pathpatch.get_transform().transform(xy)  # to display␣
 ↪coords
        xt, yt = xyt[:, 0], xyt[:, 1]
        d = np.sqrt((xt - event.x)**2 + (yt - event.y)**2)
        ind = d.argmin()
        return ind if d[ind] < self.epsilon else None

    def on_draw(self, event):
```

(continues on next page)

(continued from previous page)

```
        """Callback for draws."""
        self.background = self.canvas.copy_from_bbox(self.ax.bbox)
        self.ax.draw_artist(self.pathpatch)
        self.ax.draw_artist(self.line)
        self.canvas.blit(self.ax.bbox)

    def on_button_press(self, event):
        """Callback for mouse button presses."""
        if (event.inaxes is None
                or event.button != MouseButton.LEFT
                or not self.showverts):
            return
        self._ind = self.get_ind_under_point(event)

    def on_button_release(self, event):
        """Callback for mouse button releases."""
        if (event.button != MouseButton.LEFT
                or not self.showverts):
            return
        self._ind = None

    def on_key_press(self, event):
        """Callback for key presses."""
        if not event.inaxes:
            return
        if event.key == 't':
            self.showverts = not self.showverts
            self.line.set_visible(self.showverts)
            if not self.showverts:
                self._ind = None
        self.canvas.draw()

    def on_mouse_move(self, event):
        """Callback for mouse movements."""
        if (self._ind is None
                or event.inaxes is None
                or event.button != MouseButton.LEFT
                or not self.showverts):
            return

        vertices = self.pathpatch.get_path().vertices

        vertices[self._ind] = event.xdata, event.ydata
        self.line.set_data(zip(*vertices))

        self.canvas.restore_region(self.background)
        self.ax.draw_artist(self.pathpatch)
        self.ax.draw_artist(self.line)
        self.canvas.blit(self.ax.bbox)


interactor = PathInteractor(patch)
```

(continues on next page)

(continued from previous page)

```
ax.set_title('drag vertices to update path')
ax.set_xlim(-3, 4)
ax.set_ylim(-3, 4)

plt.show()
```

### Pick event demo

You can enable picking by setting the "picker" property of an artist (for example, a Matplotlib Line2D, Text, Patch, Polygon, AxesImage, etc.)

There are a variety of meanings of the picker property:

- *None* - picking is disabled for this artist (default)
- bool - if *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist.

  Setting `pickradius` will add an epsilon tolerance in points and the artist will fire off an event if its data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, for example, the indices of the data within epsilon of the pick event
- function - if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event.

  ```
  hit, props = picker(artist, mouseevent)
  ```

  to determine the hit test. If the mouse event is over the artist, return hit=True and props is a dictionary of properties you want added to the PickEvent attributes.

After you have enabled an artist for picking by setting the "picker" property, you need to connect to the figure canvas pick_event to get pick callbacks on mouse press events. For example,

```
def pick_handler(event):
    mouseevent = event.mouseevent
    artist = event.artist
    # now do something with this...
```

The pick event (matplotlib.backend_bases.PickEvent) which is passed to your callback is always fired with two attributes:

**mouseevent**

the mouse event that generate the pick event.

The mouse event in turn has attributes like x and y (the coordinates in display space, such as pixels from left, bottom) and xdata, ydata (the coords in data space). Additionally, you can get information about which buttons were pressed, which keys were pressed, which Axes the mouse is over, etc. See matplotlib.backend_bases.MouseEvent for details.

**artist**

the matplotlib.artist that generated the pick event.

Additionally, certain artists like Line2D and PatchCollection may attach additional metadata like the indices into the data that meet the picker criteria (for example, all the points in the line that are within the specified epsilon tolerance)

The examples below illustrate each of these methods.

---

**Note:** These examples exercises the interactive capabilities of Matplotlib, and this will not appear in the static documentation. Please run this code on your machine to see the interactivity.

You can copy and paste individual parts, or download the entire example using the link at the bottom of the page.

---

```
import matplotlib.pyplot as plt
import numpy as np
from numpy.random import rand

from matplotlib.image import AxesImage
from matplotlib.lines import Line2D
from matplotlib.patches import Rectangle
from matplotlib.text import Text

# Fixing random state for reproducibility
np.random.seed(19680801)
```

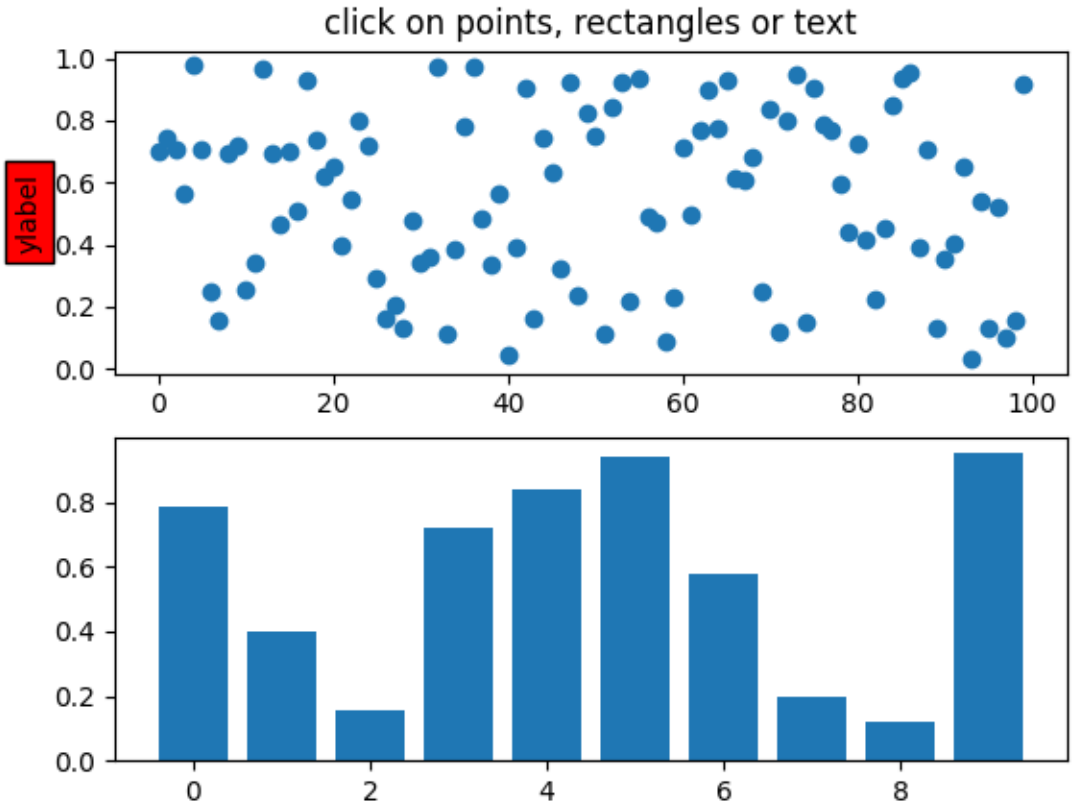
### Simple picking, lines, rectangles and text

```
fig, (ax1, ax2) = plt.subplots(2, 1)
ax1.set_title('click on points, rectangles or text', picker=True)
ax1.set_ylabel('ylabel', picker=True, bbox=dict(facecolor='red'))
line, = ax1.plot(rand(100), 'o', picker=True, pickradius=5)

# Pick the rectangle.
ax2.bar(range(10), rand(10), picker=True)
for label in ax2.get_xticklabels():  # Make the xtick labels pickable.
    label.set_picker(True)


def onpick1(event):
    if isinstance(event.artist, Line2D):
        thisline = event.artist
        xdata = thisline.get_xdata()
        ydata = thisline.get_ydata()
        ind = event.ind
        print('onpick1 line:', np.column_stack([xdata[ind], ydata[ind]]))
    elif isinstance(event.artist, Rectangle):
```

(continues on next page)

(continued from previous page)

```
        patch = event.artist
        print('onpick1 patch:', patch.get_path())
    elif isinstance(event.artist, Text):
        text = event.artist
        print('onpick1 text:', text.get_text())


fig.canvas.mpl_connect('pick_event', onpick1)
```




### Picking with a custom hit test function

You can define custom pickers by setting picker to a callable function. The function has the signature:

```
hit, props = func(artist, mouseevent)
```

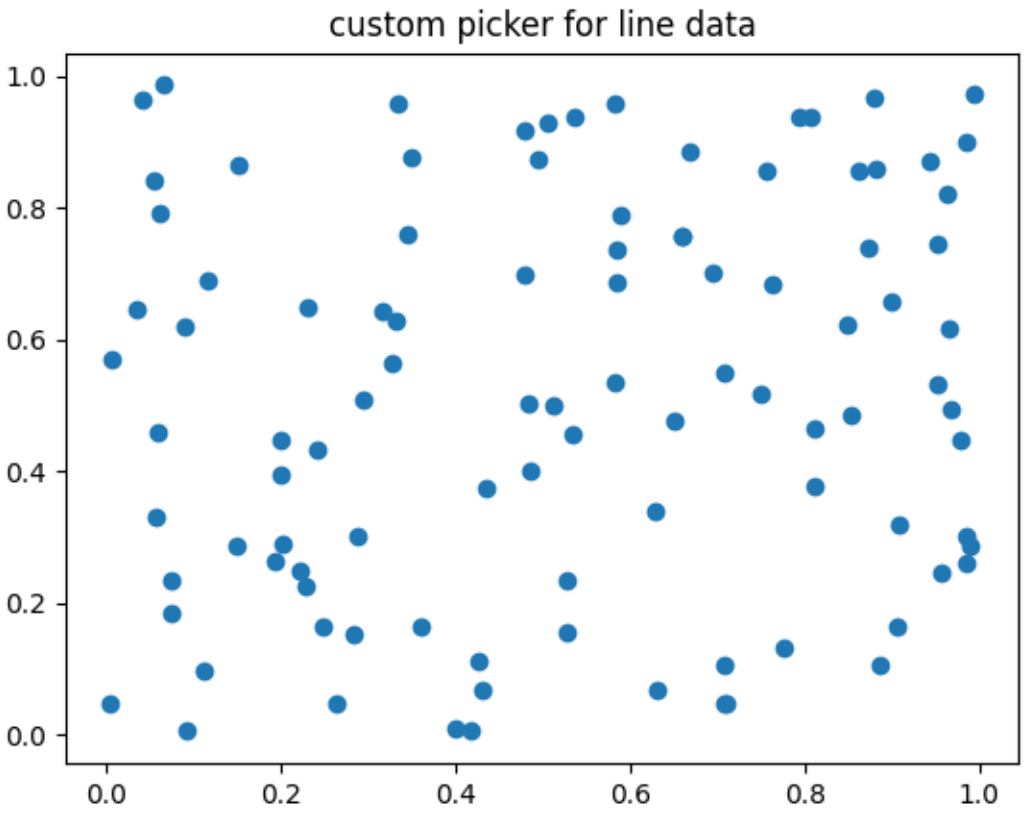
to determine the hit test. If the mouse event is over the artist, return `hit=True` and `props` is a dictionary of properties you want added to the *`PickEvent`* attributes.

```
def line_picker(line, mouseevent):
    """
```

(continues on next page)

(continued from previous page)

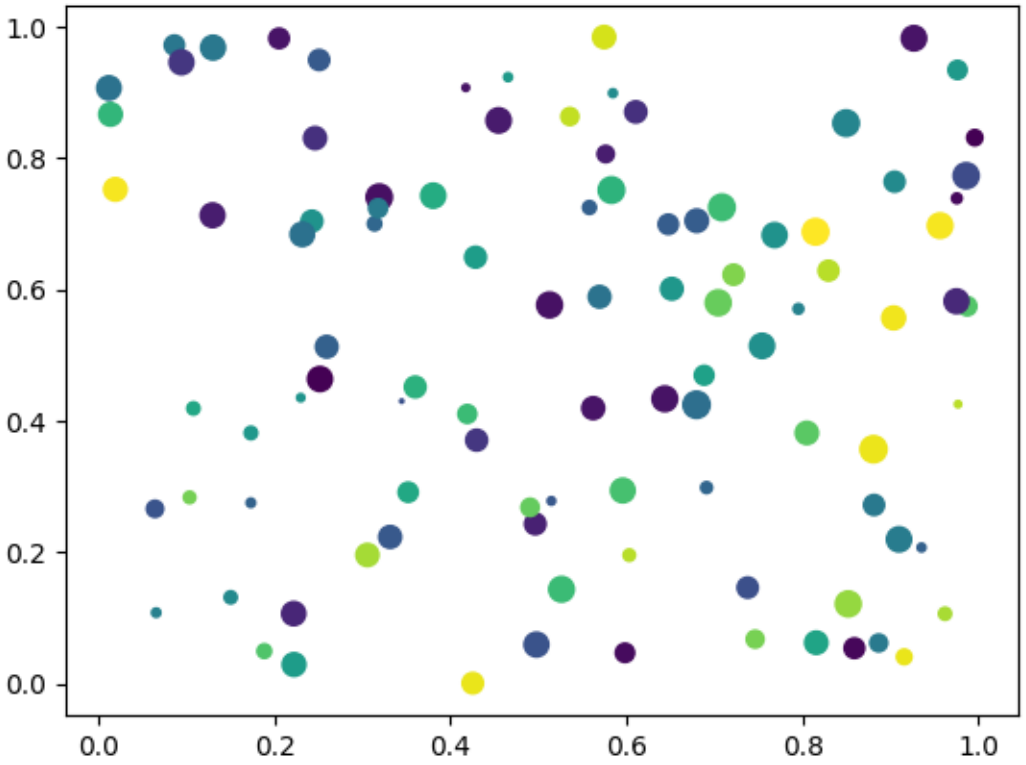
```
    Find the points within a certain distance from the mouseclick in
    data coords and attach some extra attributes, pickx and picky
    which are the data points that were picked.
    """
    if mouseevent.xdata is None:
        return False, dict()
    xdata = line.get_xdata()
    ydata = line.get_ydata()
    maxd = 0.05
    d = np.sqrt(
        (xdata - mouseevent.xdata)**2 + (ydata - mouseevent.ydata)**2)

    ind, = np.nonzero(d <= maxd)
    if len(ind):
        pickx = xdata[ind]
        picky = ydata[ind]
        props = dict(ind=ind, pickx=pickx, picky=picky)
        return True, props
    else:
        return False, dict()


def onpick2(event):
    print('onpick2 line:', event.pickx, event.picky)


fig, ax = plt.subplots()
ax.set_title('custom picker for line data')
line, = ax.plot(rand(100), rand(100), 'o', picker=line_picker)
fig.canvas.mpl_connect('pick_event', onpick2)
```




### Picking on a scatter plot

A scatter plot is backed by a *PathCollection*.

```
x, y, c, s = rand(4, 100)


def onpick3(event):
    ind = event.ind
    print('onpick3 scatter:', ind, x[ind], y[ind])


fig, ax = plt.subplots()
ax.scatter(x, y, 100*s, c, picker=True)
fig.canvas.mpl_connect('pick_event', onpick3)
```




### Picking images

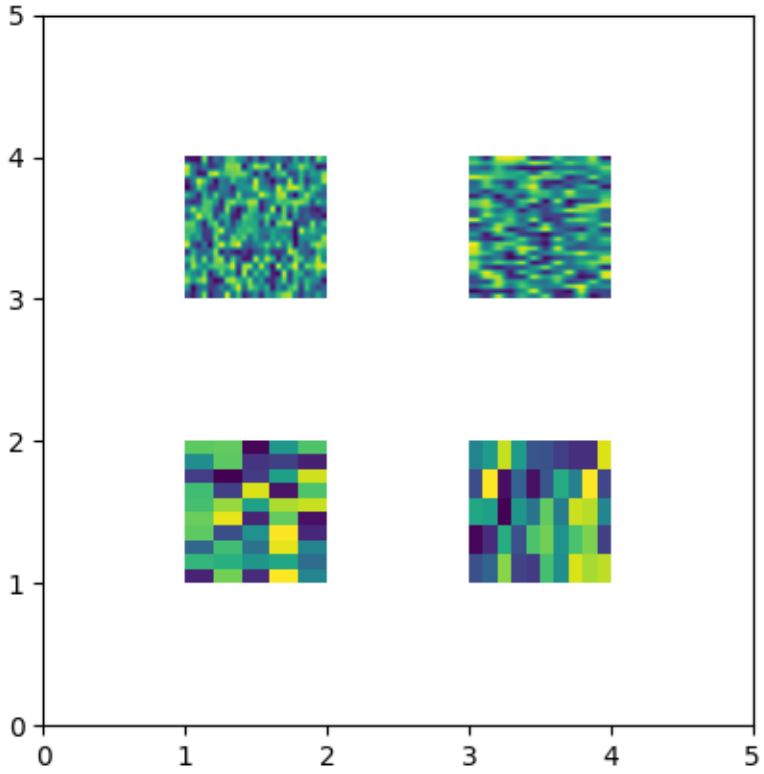
Images plotted using *Axes.imshow* are *AxesImage* objects.

```
fig, ax = plt.subplots()
ax.imshow(rand(10, 5), extent=(1, 2, 1, 2), picker=True)
ax.imshow(rand(5, 10), extent=(3, 4, 1, 2), picker=True)
ax.imshow(rand(20, 25), extent=(1, 2, 3, 4), picker=True)
ax.imshow(rand(30, 12), extent=(3, 4, 3, 4), picker=True)
ax.set(xlim=(0, 5), ylim=(0, 5))


def onpick4(event):
    artist = event.artist
    if isinstance(artist, AxesImage):
        im = artist
        A = im.get_array()
        print('onpick4 image', A.shape)


fig.canvas.mpl_connect('pick_event', onpick4)
```

(continues on next page)

(continued from previous page)

```
plt.show()
```




**Total running time of the script:** (0 minutes 1.162 seconds)

### Pick event demo 2

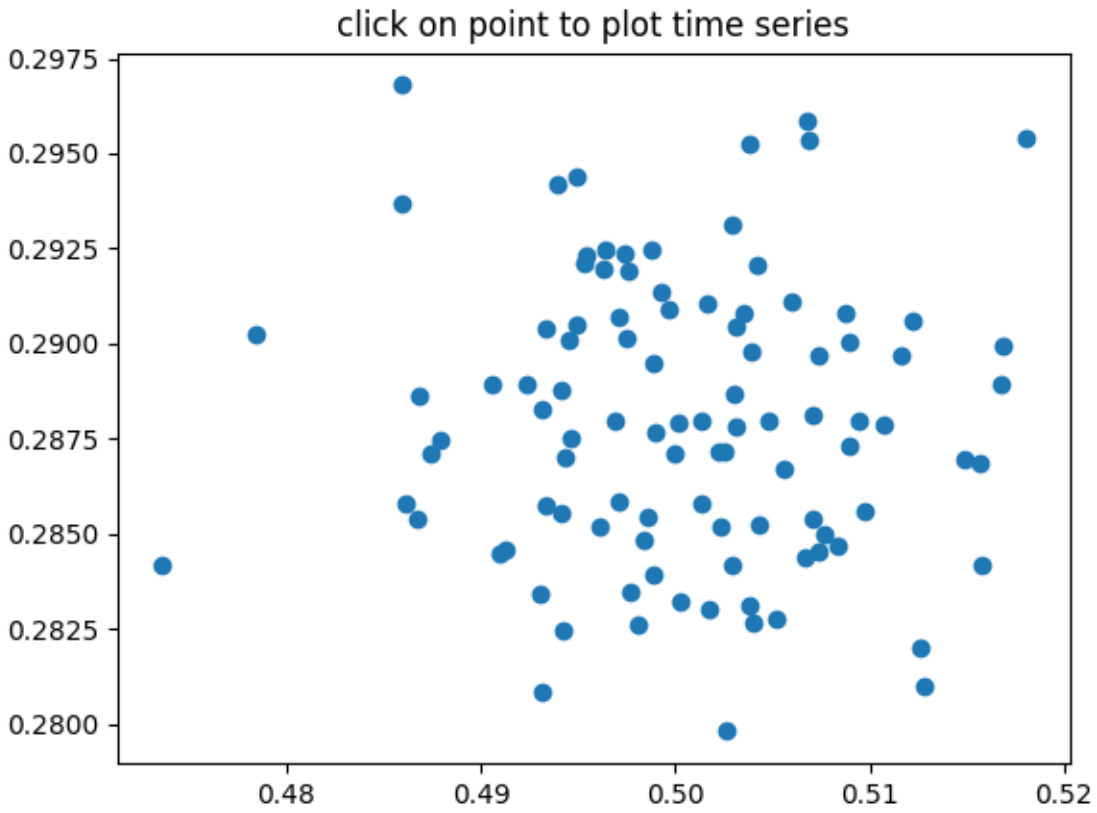
Compute the mean (mu) and standard deviation (sigma) of 100 data sets and plot mu vs. sigma. When you click on one of the (mu, sigma) points, plot the raw data from the dataset that generated this point.

---

**Note:** This example exercises the interactive capabilities of Matplotlib, and this will not appear in the static documentation. Please run this code on your machine to see the interactivity.

You can copy and paste individual parts, or download the entire example using the link at the bottom of the page.

---




```
import matplotlib.pyplot as plt
import numpy as np

# Fixing random state for reproducibility
np.random.seed(19680801)

X = np.random.rand(100, 1000)
xs = np.mean(X, axis=1)
ys = np.std(X, axis=1)

fig, ax = plt.subplots()
ax.set_title('click on point to plot time series')
line, = ax.plot(xs, ys, 'o', picker=True, pickradius=5)


def onpick(event):

    if event.artist != line:
        return

    N = len(event.ind)
    if not N:
        return
```

(continues on next page)

(continued from previous page)

```
    figi, axs = plt.subplots(N, squeeze=False)
    for ax, dataind in zip(axs.flat, event.ind):
        ax.plot(X[dataind])
        ax.text(.05, .9, f'mu={xs[dataind]:1.3f}\nsigma={ys[dataind]:1.3f}',
                transform=ax.transAxes, va='top')
        ax.set_ylim(-0.5, 1.5)
    figi.show()


fig.canvas.mpl_connect('pick_event', onpick)

plt.show()
```

### Poly Editor

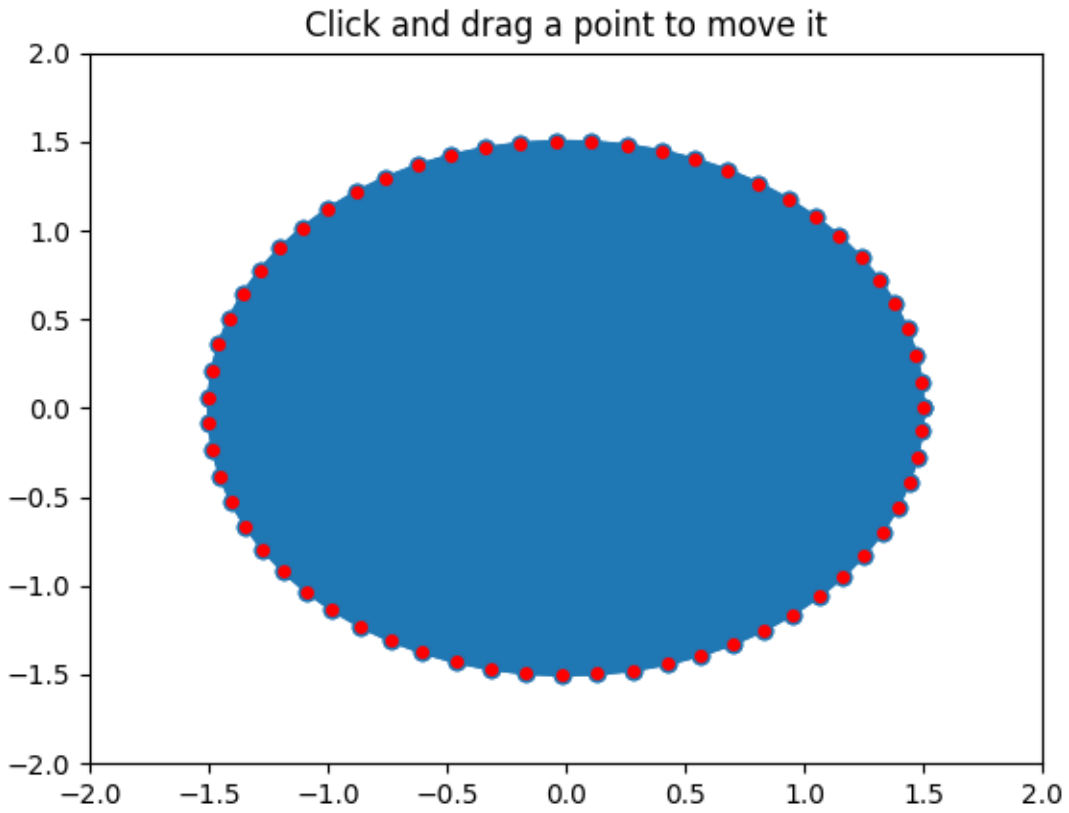
This is an example to show how to build cross-GUI applications using Matplotlib event handling to interact with objects on the canvas.

---

**Note:** This example exercises the interactive capabilities of Matplotlib, and this will not appear in the static documentation. Please run this code on your machine to see the interactivity.

You can copy and paste individual parts, or download the entire example using the link at the bottom of the page.

---




```
import numpy as np

from matplotlib.artist import Artist
from matplotlib.lines import Line2D


def dist_point_to_segment(p, s0, s1):
    """
    Get the distance from the point *p* to the segment (*s0*, *s1*), where
    *p*, *s0*, *s1* are ``[x, y]`` arrays.
    """
    s01 = s1 - s0
    s0p = p - s0
    if (s01 == 0).all():
        return np.hypot(*s0p)
    # Project onto segment, without going past segment ends.
    p1 = s0 + np.clip((s0p @ s01) / (s01 @ s01), 0, 1) * s01
    return np.hypot(*(p - p1))


class PolygonInteractor:
    """
    A polygon editor.
```

(continues on next page)

(continued from previous page)

```
    Key-bindings

      't' toggle vertex markers on and off.  When vertex markers are on,
          you can move them, delete them

      'd' delete the vertex under point

      'i' insert a vertex at point.  You must be within epsilon of the
          line connecting two existing vertices

    """

    showverts = True
    epsilon = 5  # max pixel distance to count as a vertex hit

    def __init__(self, ax, poly):
        if poly.figure is None:
            raise RuntimeError('You must first add the polygon to a figure '
                               'or canvas before defining the interactor')
        self.ax = ax
        canvas = poly.figure.canvas
        self.poly = poly

        x, y = zip(*self.poly.xy)
        self.line = Line2D(x, y,
                           marker='o', markerfacecolor='r',
                           animated=True)
        self.ax.add_line(self.line)

        self.cid = self.poly.add_callback(self.poly_changed)
        self._ind = None  # the active vert

        canvas.mpl_connect('draw_event', self.on_draw)
        canvas.mpl_connect('button_press_event', self.on_button_press)
        canvas.mpl_connect('key_press_event', self.on_key_press)
        canvas.mpl_connect('button_release_event', self.on_button_release)
        canvas.mpl_connect('motion_notify_event', self.on_mouse_move)
        self.canvas = canvas

    def on_draw(self, event):
        self.background = self.canvas.copy_from_bbox(self.ax.bbox)
        self.ax.draw_artist(self.poly)
        self.ax.draw_artist(self.line)
        # do not need to blit here, this will fire before the screen is
        # updated

    def poly_changed(self, poly):
        """This method is called whenever the pathpatch object is called."""
        # only copy the artist props to the line (except visibility)
        vis = self.line.get_visible()
        Artist.update_from(self.line, poly)
```

(continues on next page)

(continued from previous page)

```
        self.line.set_visible(vis)  # don't use the poly visibility state

    def get_ind_under_point(self, event):
        """
        Return the index of the point closest to the event position or *None*
        if no point is within ``self.epsilon`` to the event position.
        """
        # display coords
        xy = np.asarray(self.poly.xy)
        xyt = self.poly.get_transform().transform(xy)
        xt, yt = xyt[:, 0], xyt[:, 1]
        d = np.hypot(xt - event.x, yt - event.y)
        indseq, = np.nonzero(d == d.min())
        ind = indseq[0]

        if d[ind] >= self.epsilon:
            ind = None

        return ind

    def on_button_press(self, event):
        """Callback for mouse button presses."""
        if not self.showverts:
            return
        if event.inaxes is None:
            return
        if event.button != 1:
            return
        self._ind = self.get_ind_under_point(event)

    def on_button_release(self, event):
        """Callback for mouse button releases."""
        if not self.showverts:
            return
        if event.button != 1:
            return
        self._ind = None

    def on_key_press(self, event):
        """Callback for key presses."""
        if not event.inaxes:
            return
        if event.key == 't':
            self.showverts = not self.showverts
            self.line.set_visible(self.showverts)
            if not self.showverts:
                self._ind = None
        elif event.key == 'd':
            ind = self.get_ind_under_point(event)
            if ind is not None:
                self.poly.xy = np.delete(self.poly.xy,
                                         ind, axis=0)
```

(continues on next page)

(continued from previous page)

```
                self.line.set_data(zip(*self.poly.xy))
        elif event.key == 'i':
            xys = self.poly.get_transform().transform(self.poly.xy)
            p = event.x, event.y  # display coords
            for i in range(len(xys) - 1):
                s0 = xys[i]
                s1 = xys[i + 1]
                d = dist_point_to_segment(p, s0, s1)
                if d <= self.epsilon:
                    self.poly.xy = np.insert(
                        self.poly.xy, i+1,
                        [event.xdata, event.ydata],
                        axis=0)
                    self.line.set_data(zip(*self.poly.xy))
                    break
        if self.line.stale:
            self.canvas.draw_idle()

    def on_mouse_move(self, event):
        """Callback for mouse movements."""
        if not self.showverts:
            return
        if self._ind is None:
            return
        if event.inaxes is None:
            return
        if event.button != 1:
            return
        x, y = event.xdata, event.ydata

        self.poly.xy[self._ind] = x, y
        if self._ind == 0:
            self.poly.xy[-1] = x, y
        elif self._ind == len(self.poly.xy) - 1:
            self.poly.xy[0] = x, y
        self.line.set_data(zip(*self.poly.xy))

        self.canvas.restore_region(self.background)
        self.ax.draw_artist(self.poly)
        self.ax.draw_artist(self.line)
        self.canvas.blit(self.ax.bbox)


if __name__ == '__main__':
    import matplotlib.pyplot as plt

    from matplotlib.patches import Polygon

    theta = np.arange(0, 2*np.pi, 0.1)
    r = 1.5

    xs = r * np.cos(theta)
```

(continues on next page)

(continued from previous page)

```
    ys = r * np.sin(theta)

    poly = Polygon(np.column_stack([xs, ys]), animated=True)

    fig, ax = plt.subplots()
    ax.add_patch(poly)
    p = PolygonInteractor(ax, poly)

    ax.set_title('Click and drag a point to move it')
    ax.set_xlim((-2, 2))
    ax.set_ylim((-2, 2))
    plt.show()
```

## Pong

A Matplotlib based game of Pong illustrating one way to write interactive animations that are easily ported to multiple backends.

---

**Note:** This example exercises the interactive capabilities of Matplotlib, and this will not appear in the static documentation. Please run this code on your machine to see the interactivity.

You can copy and paste individual parts, or download the entire example using the link at the bottom of the page.

---

```
import time

import matplotlib.pyplot as plt
import numpy as np
from numpy.random import randint, randn

from matplotlib.font_manager import FontProperties

instructions = """
Player A:       Player B:
  'e'      up     'i'
  'd'     down    'k'

press 't' -- close these instructions
             (animation will be much faster)
press 'a' -- add a puck
press 'A' -- remove a puck
press '1' -- slow down all pucks
press '2' -- speed up all pucks
press '3' -- slow down distractors
press '4' -- speed up distractors
press ' ' -- reset the first puck
press 'n' -- toggle distractors on/off
press 'g' -- toggle the game on/off
```

(continues on next page)

(continued from previous page)

```
  """


class Pad:
    def __init__(self, disp, x, y, type='l'):
        self.disp = disp
        self.x = x
        self.y = y
        self.w = .3
        self.score = 0
        self.xoffset = 0.3
        self.yoffset = 0.1
        if type == 'r':
            self.xoffset *= -1.0

        if type == 'l' or type == 'r':
            self.signx = -1.0
            self.signy = 1.0
        else:
            self.signx = 1.0
            self.signy = -1.0

    def contains(self, loc):
        return self.disp.get_bbox().contains(loc.x, loc.y)


class Puck:
    def __init__(self, disp, pad, field):
        self.vmax = .2
        self.disp = disp
        self.field = field
        self._reset(pad)

    def _reset(self, pad):
        self.x = pad.x + pad.xoffset
        if pad.y < 0:
            self.y = pad.y + pad.yoffset
        else:
            self.y = pad.y - pad.yoffset
        self.vx = pad.x - self.x
        self.vy = pad.y + pad.w/2 - self.y
        self._speedlimit()
        self._slower()
        self._slower()

    def update(self, pads):
        self.x += self.vx
        self.y += self.vy
        for pad in pads:
            if pad.contains(self):
                self.vx *= 1.2 * pad.signx
```

(continues on next page)

(continued from previous page)

```
                self.vy *= 1.2 * pad.signy
        fudge = .001
        # probably cleaner with something like...
        if self.x < fudge:
            pads[1].score += 1
            self._reset(pads[0])
            return True
        if self.x > 7 - fudge:
            pads[0].score += 1
            self._reset(pads[1])
            return True
        if self.y < -1 + fudge or self.y > 1 - fudge:
            self.vy *= -1.0
            # add some randomness, just to make it interesting
            self.vy -= (randn()/300.0 + 1/300.0) * np.sign(self.vy)
        self._speedlimit()
        return False

    def _slower(self):
        self.vx /= 5.0
        self.vy /= 5.0

    def _faster(self):
        self.vx *= 5.0
        self.vy *= 5.0

    def _speedlimit(self):
        if self.vx > self.vmax:
            self.vx = self.vmax
        if self.vx < -self.vmax:
            self.vx = -self.vmax

        if self.vy > self.vmax:
            self.vy = self.vmax
        if self.vy < -self.vmax:
            self.vy = -self.vmax


class Game:
    def __init__(self, ax):
        # create the initial line
        self.ax = ax
        ax.xaxis.set_visible(False)
        ax.set_xlim([0, 7])
        ax.yaxis.set_visible(False)
        ax.set_ylim([-1, 1])
        pad_a_x = 0
        pad_b_x = .50
        pad_a_y = pad_b_y = .30
        pad_b_x += 6.3

        # pads
```

(continues on next page)

(continued from previous page)

```
        pA, = self.ax.barh(pad_a_y, .2,
                           height=.3, color='k', alpha=.5, edgecolor='b',
                           lw=2, label="Player B",
                           animated=True)
        pB, = self.ax.barh(pad_b_y, .2,
                           height=.3, left=pad_b_x, color='k', alpha=.5,
                           edgecolor='r', lw=2, label="Player A",
                           animated=True)

        # distractors
        self.x = np.arange(0, 2.22*np.pi, 0.01)
        self.line, = self.ax.plot(self.x, np.sin(self.x), "r",
                                  animated=True, lw=4)
        self.line2, = self.ax.plot(self.x, np.cos(self.x), "g",
                                   animated=True, lw=4)
        self.line3, = self.ax.plot(self.x, np.cos(self.x), "g",
                                   animated=True, lw=4)
        self.line4, = self.ax.plot(self.x, np.cos(self.x), "r",
                                   animated=True, lw=4)

        # center line
        self.centerline, = self.ax.plot([3.5, 3.5], [1, -1], 'k',
                                        alpha=.5, animated=True, lw=8)

        # puck (s)
        self.puckdisp = self.ax.scatter([1], [1], label='_nolegend_',
                                        s=200, c='g',
                                        alpha=.9, animated=True)

        self.canvas = self.ax.figure.canvas
        self.background = None
        self.cnt = 0
        self.distract = True
        self.res = 100.0
        self.on = False
        self.inst = True    # show instructions from the beginning
        self.pads = [Pad(pA, pad_a_x, pad_a_y),
                     Pad(pB, pad_b_x, pad_b_y, 'r')]
        self.pucks = []
        self.i = self.ax.annotate(instructions, (.5, 0.5),
                                  name='monospace',
                                  verticalalignment='center',
                                  horizontalalignment='center',
                                  multialignment='left',
                                  xycoords='axes fraction',
                                  animated=False)
        self.canvas.mpl_connect('key_press_event', self.on_key_press)

    def draw(self):
        draw_artist = self.ax.draw_artist
        if self.background is None:
            self.background = self.canvas.copy_from_bbox(self.ax.bbox)
```

(continues on next page)

(continued from previous page)

```
        # restore the clean slate background
        self.canvas.restore_region(self.background)

        # show the distractors
        if self.distract:
            self.line.set_ydata(np.sin(self.x + self.cnt/self.res))
            self.line2.set_ydata(np.cos(self.x - self.cnt/self.res))
            self.line3.set_ydata(np.tan(self.x + self.cnt/self.res))
            self.line4.set_ydata(np.tan(self.x - self.cnt/self.res))
            draw_artist(self.line)
            draw_artist(self.line2)
            draw_artist(self.line3)
            draw_artist(self.line4)

        # pucks and pads
        if self.on:
            self.ax.draw_artist(self.centerline)
            for pad in self.pads:
                pad.disp.set_y(pad.y)
                pad.disp.set_x(pad.x)
                self.ax.draw_artist(pad.disp)

            for puck in self.pucks:
                if puck.update(self.pads):
                    # we only get here if someone scored
                    self.pads[0].disp.set_label(f"   {self.pads[0].score}")
                    self.pads[1].disp.set_label(f"   {self.pads[1].score}")
                    self.ax.legend(loc='center', framealpha=.2,
                                   facecolor='0.5',
                                   prop=FontProperties(size='xx-large',
                                                       weight='bold'))

                    self.background = None
                    self.ax.figure.canvas.draw_idle()
                    return
                puck.disp.set_offsets([[puck.x, puck.y]])
                self.ax.draw_artist(puck.disp)

        # just redraw the Axes rectangle
        self.canvas.blit(self.ax.bbox)
        self.canvas.flush_events()
        if self.cnt == 50000:
            # just so we don't get carried away
            print("...and you've been playing for too long!!!")
            plt.close()

        self.cnt += 1

    def on_key_press(self, event):
        if event.key == '3':
            self.res *= 5.0
```

(continues on next page)

(continued from previous page)

```
        if event.key == '4':
            self.res /= 5.0

        if event.key == 'e':
            self.pads[0].y += .1
            if self.pads[0].y > 1 - .3:
                self.pads[0].y = 1 - .3
        if event.key == 'd':
            self.pads[0].y -= .1
            if self.pads[0].y < -1:
                self.pads[0].y = -1

        if event.key == 'i':
            self.pads[1].y += .1
            if self.pads[1].y > 1 - .3:
                self.pads[1].y = 1 - .3
        if event.key == 'k':
            self.pads[1].y -= .1
            if self.pads[1].y < -1:
                self.pads[1].y = -1

        if event.key == 'a':
            self.pucks.append(Puck(self.puckdisp,
                                   self.pads[randint(2)],
                                   self.ax.bbox))
        if event.key == 'A' and len(self.pucks):
            self.pucks.pop()
        if event.key == ' ' and len(self.pucks):
            self.pucks[0]._reset(self.pads[randint(2)])
        if event.key == '1':
            for p in self.pucks:
                p._slower()
        if event.key == '2':
            for p in self.pucks:
                p._faster()

        if event.key == 'n':
            self.distract = not self.distract

        if event.key == 'g':
            self.on = not self.on
        if event.key == 't':
            self.inst = not self.inst
            self.i.set_visible(not self.i.get_visible())
            self.background = None
            self.canvas.draw_idle()
        if event.key == 'q':
            plt.close()


fig, ax = plt.subplots()
canvas = ax.figure.canvas
```

(continues on next page)

(continued from previous page)

```
animation = Game(ax)

# disable the default key bindings
if fig.canvas.manager.key_press_handler_id is not None:
    canvas.mpl_disconnect(fig.canvas.manager.key_press_handler_id)


# reset the blitting background on redraw
def on_redraw(event):
    animation.background = None


# bootstrap after the first draw
def start_anim(event):
    canvas.mpl_disconnect(start_anim.cid)

    start_anim.timer.add_callback(animation.draw)
    start_anim.timer.start()
    canvas.mpl_connect('draw_event', on_redraw)


start_anim.cid = canvas.mpl_connect('draw_event', start_anim)
start_anim.timer = animation.canvas.new_timer(interval=1)

tstart = time.time()

plt.show()
print('FPS: %f' % (animation.cnt/(time.time() - tstart)))
```

### Resampling Data

Downsampling lowers the sample rate or sample size of a signal. In this tutorial, the signal is downsampled when the plot is adjusted through dragging and zooming.

---

**Note:** This example exercises the interactive capabilities of Matplotlib, and this will not appear in the static documentation. Please run this code on your machine to see the interactivity.

You can copy and paste individual parts, or download the entire example using the link at the bottom of the page.

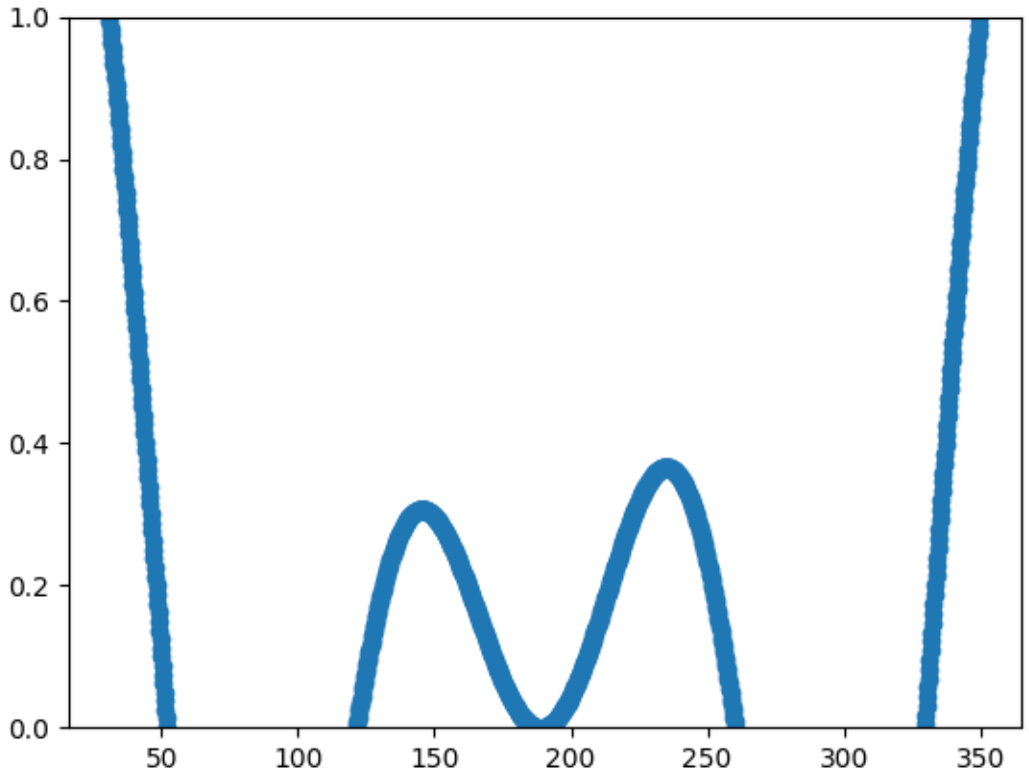
---

```
import matplotlib.pyplot as plt
import numpy as np


# A class that will downsample the data and recompute when zoomed.
class DataDisplayDownsampler:
    def __init__(self, xdata, ydata):
        self.origYData = ydata
```

(continues on next page)

(continued from previous page)

```
        self.origXData = xdata
        self.max_points = 50
        self.delta = xdata[-1] - xdata[0]

    def downsample(self, xstart, xend):
        # get the points in the view range
        mask = (self.origXData > xstart) & (self.origXData < xend)
        # dilate the mask by one to catch the points just outside
        # of the view range to not truncate the line
        mask = np.convolve([1, 1, 1], mask, mode='same').astype(bool)
        # sort out how many points to drop
        ratio = max(np.sum(mask) // self.max_points, 1)

        # mask data
        xdata = self.origXData[mask]
        ydata = self.origYData[mask]

        # downsample data
        xdata = xdata[::ratio]
        ydata = ydata[::ratio]

        print(f"using {len(ydata)} of {np.sum(mask)} visible points")

        return xdata, ydata

    def update(self, ax):
        # Update the line
        lims = ax.viewLim
        if abs(lims.width - self.delta) > 1e-8:
            self.delta = lims.width
            xstart, xend = lims.intervalx
            self.line.set_data(*self.downsample(xstart, xend))
            ax.figure.canvas.draw_idle()


# Create a signal
xdata = np.linspace(16, 365, (365-16)*4)
ydata = np.sin(2*np.pi*xdata/153) + np.cos(2*np.pi*xdata/127)

d = DataDisplayDownsampler(xdata, ydata)

fig, ax = plt.subplots()

# Hook up the line
d.line, = ax.plot(xdata, ydata, 'o-')
ax.set_autoscale_on(False)  # Otherwise, infinite loop

# Connect for changing the view limits
ax.callbacks.connect('xlim_changed', d.update)
ax.set_xlim(16, 365)
plt.show()
```




Tags: interactivity: zoom event-handling

### Timers

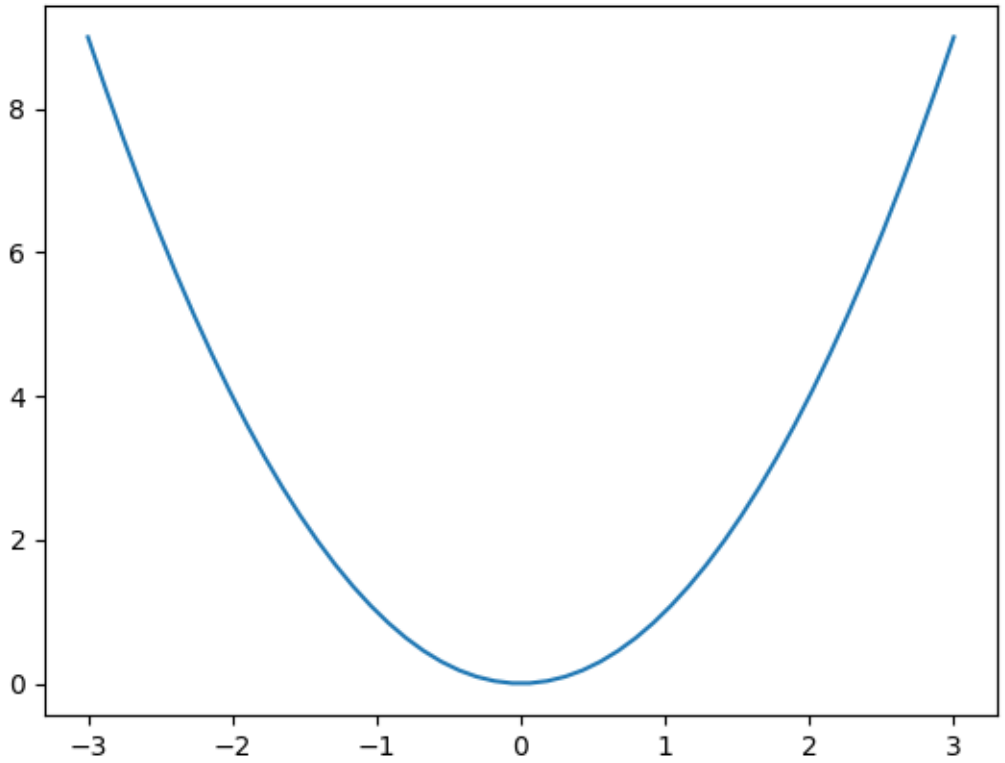
Simple example of using general timer objects. This is used to update the time placed in the title of the figure.

---

**Note:** This example exercises the interactive capabilities of Matplotlib, and this will not appear in the static documentation. Please run this code on your machine to see the interactivity.

You can copy and paste individual parts, or download the entire example using the link at the bottom of the page.

---




```
from datetime import datetime

import matplotlib.pyplot as plt
import numpy as np


def update_title(axes):
    axes.set_title(datetime.now())
    axes.figure.canvas.draw()

fig, ax = plt.subplots()

x = np.linspace(-3, 3)
ax.plot(x, x ** 2)

# Create a new timer object. Set the interval to 100 milliseconds
# (1000 is default) and tell the timer what function should be called.
timer = fig.canvas.new_timer(interval=100)
timer.add_callback(update_title, ax)
timer.start()

# Or could start the timer on first figure draw:
# def start_timer(event):
```

(continues on next page)

(continued from previous page)

```
#     timer.start()
#     fig.canvas.mpl_disconnect(drawid)
# drawid = fig.canvas.mpl_connect('draw_event', start_timer)

plt.show()
```

### Trifinder Event Demo

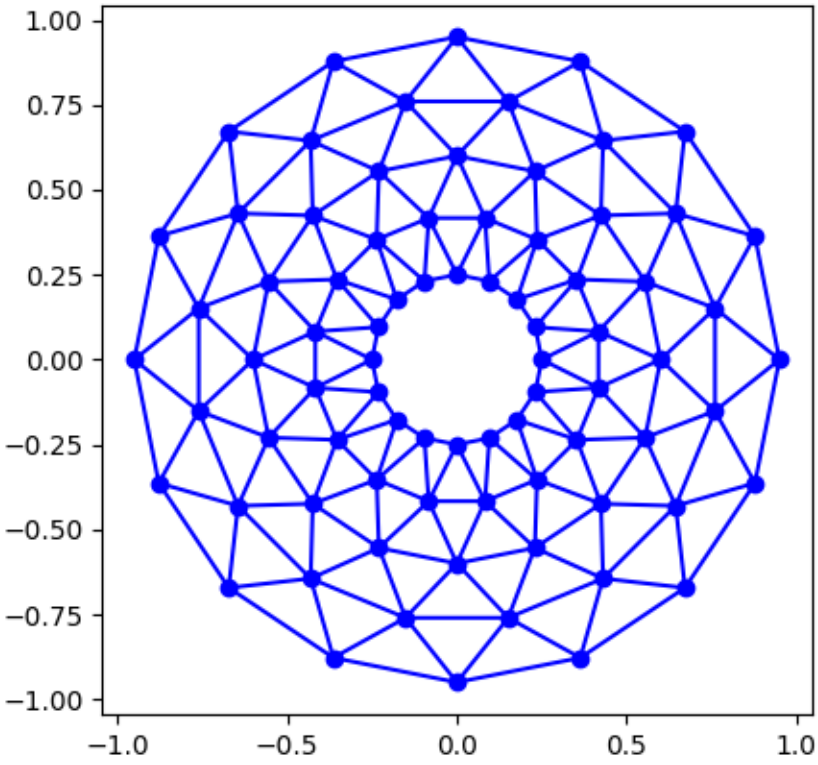
Example showing the use of a TriFinder object. As the mouse is moved over the triangulation, the triangle under the cursor is highlighted and the index of the triangle is displayed in the plot title.

---

**Note:** This example exercises the interactive capabilities of Matplotlib, and this will not appear in the static documentation. Please run this code on your machine to see the interactivity.

You can copy and paste individual parts, or download the entire example using the link at the bottom of the page.

---



```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.patches import Polygon
from matplotlib.tri import Triangulation


def update_polygon(tri):
    if tri == -1:
        points = [0, 0, 0]
    else:
        points = triang.triangles[tri]
    xs = triang.x[points]
    ys = triang.y[points]
    polygon.set_xy(np.column_stack([xs, ys]))


def on_mouse_move(event):
    if event.inaxes is None:
        tri = -1
    else:
        tri = trifinder(event.xdata, event.ydata)
    update_polygon(tri)
    ax.set_title(f'In triangle {tri}')
    event.canvas.draw()


# Create a Triangulation.
n_angles = 16
n_radii = 5
min_radius = 0.25
radii = np.linspace(min_radius, 0.95, n_radii)
angles = np.linspace(0, 2 * np.pi, n_angles, endpoint=False)
angles = np.repeat(angles[..., np.newaxis], n_radii, axis=1)
angles[:, 1::2] += np.pi / n_angles
x = (radii*np.cos(angles)).flatten()
y = (radii*np.sin(angles)).flatten()
triang = Triangulation(x, y)
triang.set_mask(np.hypot(x[triang.triangles].mean(axis=1),
                         y[triang.triangles].mean(axis=1))
                < min_radius)

# Use the triangulation's default TriFinder object.
trifinder = triang.get_trifinder()

# Setup plot and callbacks.
fig, ax = plt.subplots(subplot_kw={'aspect': 'equal'})
ax.triplot(triang, 'bo-')
polygon = Polygon([[0, 0], [0, 0]], facecolor='y')  # dummy data for (xs, ys)
update_polygon(-1)
ax.add_patch(polygon)
fig.canvas.mpl_connect('motion_notify_event', on_mouse_move)
plt.show()
```

### Viewlims

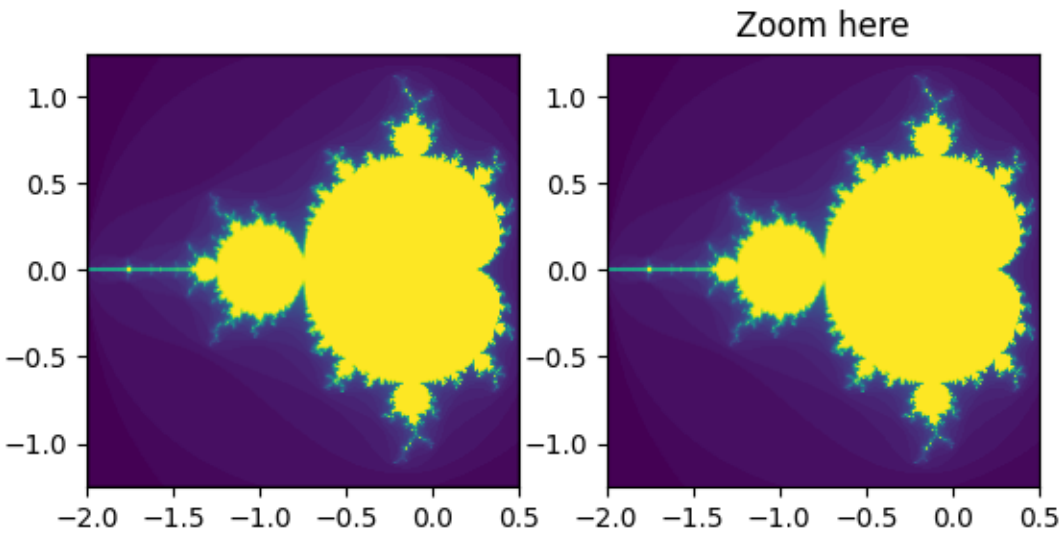
Creates two identical panels. Zooming in on the right panel will show a rectangle in the first panel, denoting the zoomed region.

---

**Note:** This example exercises the interactive capabilities of Matplotlib, and this will not appear in the static documentation. Please run this code on your machine to see the interactivity.

You can copy and paste individual parts, or download the entire example using the link at the bottom of the page.

---




```
import functools

import matplotlib.pyplot as plt
import numpy as np

from matplotlib.patches import Rectangle


# A class that will regenerate a fractal set as we zoom in, so that you
# can actually see the increasing detail.  A box in the left panel will show
```

(continues on next page)

(continued from previous page)

```
# the area to which we are zoomed.
class MandelbrotDisplay:
    def __init__(self, h=500, w=500, niter=50, radius=2., power=2):
        self.height = h
        self.width = w
        self.niter = niter
        self.radius = radius
        self.power = power

    def compute_image(self, xlim, ylim):
        self.x = np.linspace(*xlim, self.width)
        self.y = np.linspace(*ylim, self.height).reshape(-1, 1)
        c = self.x + 1.0j * self.y
        threshold_time = np.zeros((self.height, self.width))
        z = np.zeros(threshold_time.shape, dtype=complex)
        mask = np.ones(threshold_time.shape, dtype=bool)
        for i in range(self.niter):
            z[mask] = z[mask]**self.power + c[mask]
            mask = (np.abs(z) < self.radius)
            threshold_time += mask
        return threshold_time

    def ax_update(self, ax):
        ax.set_autoscale_on(False)  # Otherwise, infinite loop
        # Get the number of points from the number of pixels in the window
        self.width, self.height = ax.patch.get_window_extent().size.round().
 ↪astype(int)
        # Update the image object with our new data and extent
        ax.images[-1].set(data=self.compute_image(ax.get_xlim(), ax.get_
 ↪ylim()),
                          extent=(*ax.get_xlim(), *ax.get_ylim()))
        ax.figure.canvas.draw_idle()


md = MandelbrotDisplay()

fig1, (ax_full, ax_zoom) = plt.subplots(1, 2)
ax_zoom.imshow([[0]], origin="lower")  # Empty initial image.
ax_zoom.set_title("Zoom here")

rect = Rectangle(
    [0, 0], 0, 0, facecolor="none", edgecolor="black", linewidth=1.0)
ax_full.add_patch(rect)


def update_rect(rect, ax):  # Let the rectangle track the bounds of the zoom␣
 ↪axes.
    xlo, xhi = ax.get_xlim()
    ylo, yhi = ax.get_ylim()
    rect.set_bounds((xlo, ylo, xhi - xlo, yhi - ylo))
    ax.figure.canvas.draw_idle()
```

(continues on next page)

(continued from previous page)

```
# Connect for changing the view limits.
ax_zoom.callbacks.connect("xlim_changed", functools.partial(update_rect,␣
 ↪rect))
ax_zoom.callbacks.connect("ylim_changed", functools.partial(update_rect,␣
 ↪rect))

ax_zoom.callbacks.connect("xlim_changed", md.ax_update)
ax_zoom.callbacks.connect("ylim_changed", md.ax_update)

# Initialize: trigger image computation by setting view limits; set colormap␣
 ↪limits;
# copy image to full view.
ax_zoom.set(xlim=(-2, .5), ylim=(-1.25, 1.25))
im = ax_zoom.images[0]
ax_zoom.images[0].set(clim=(im.get_array().min(), im.get_array().max()))
ax_full.imshow(im.get_array(), extent=im.get_extent(), origin="lower")

plt.show()
```

### Zoom Window

This example shows how to connect events in one window, for example, a mouse press, to another figure window.

If you click on a point in the first window, the z and y limits of the second will be adjusted so that the center of the zoom in the second window will be the (x, y) coordinates of the clicked point.

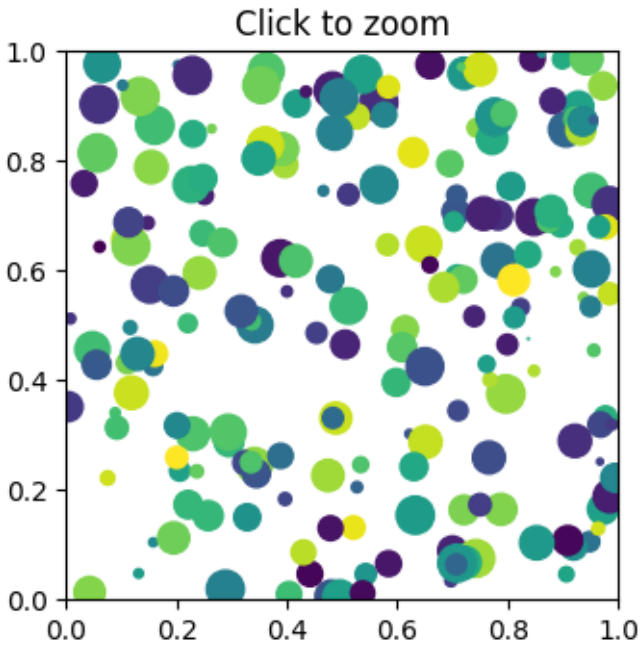
Note the diameter of the circles in the scatter are defined in points**2, so their size is independent of the zoom.

---

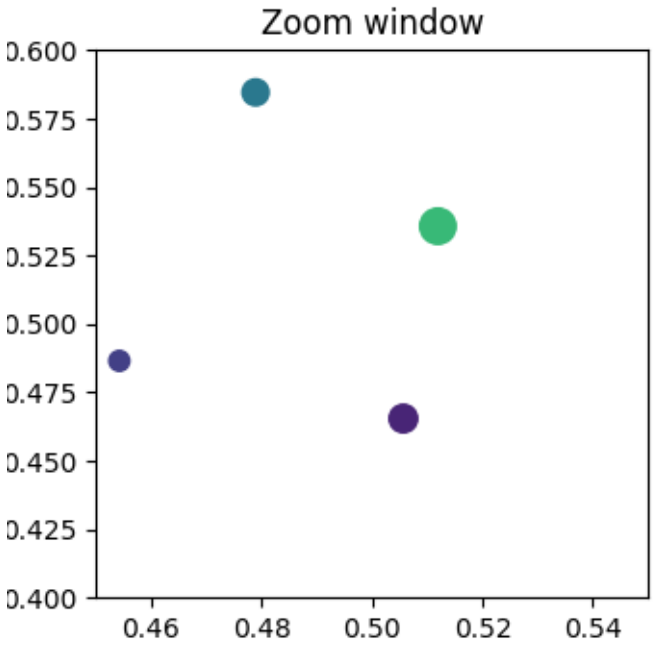
**Note:** This example exercises the interactive capabilities of Matplotlib, and this will not appear in the static documentation. Please run this code on your machine to see the interactivity.

You can copy and paste individual parts, or download the entire example using the link at the bottom of the page.

---




- 




- 

```
import matplotlib.pyplot as plt
import numpy as np

# Fixing random state for reproducibility
np.random.seed(19680801)

figsrc, axsrc = plt.subplots(figsize=(3.7, 3.7))
```

(continues on next page)

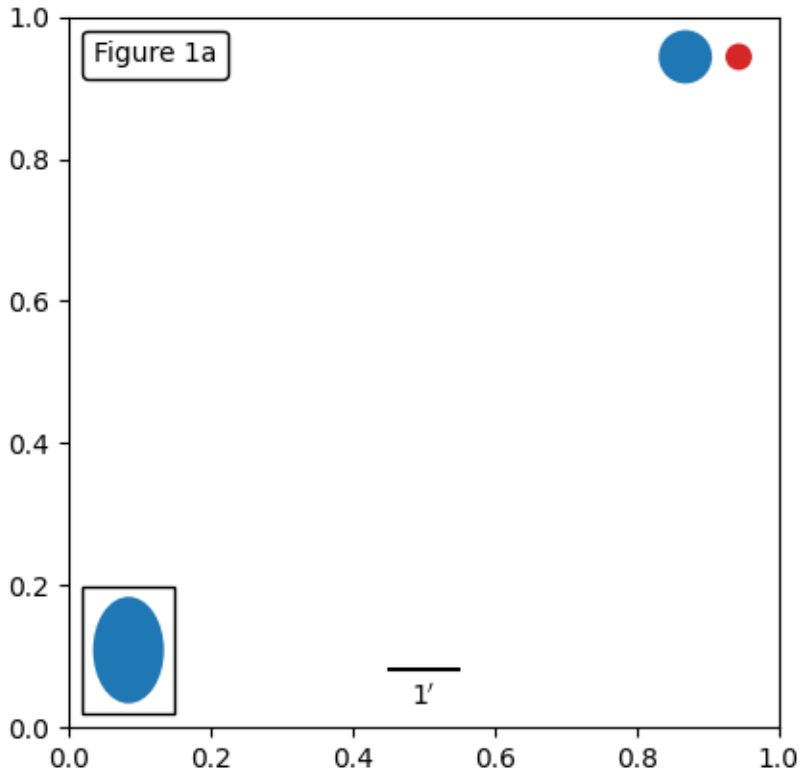
(continued from previous page)

```
figzoom, axzoom = plt.subplots(figsize=(3.7, 3.7))
axsrc.set(xlim=(0, 1), ylim=(0, 1), autoscale_on=False,
          title='Click to zoom')
axzoom.set(xlim=(0.45, 0.55), ylim=(0.4, 0.6), autoscale_on=False,
           title='Zoom window')

x, y, s, c = np.random.rand(4, 200)
s *= 200

axsrc.scatter(x, y, s, c)
axzoom.scatter(x, y, s, c)


def on_press(event):
    if event.button != 1:
        return
    x, y = event.xdata, event.ydata
    axzoom.set_xlim(x - 0.1, x + 0.1)
    axzoom.set_ylim(y - 0.1, y + 0.1)
    figzoom.canvas.draw()

figsrc.canvas.mpl_connect('button_press_event', on_press)
plt.show()
```

### 6.25.16 Miscellaneous

#### Anchored Artists

This example illustrates the use of the anchored objects without the helper classes found in *mpl_toolkits.axes_grid1*. This version of the figure is similar to the one found in *Simple Anchored Artists*, but it is implemented using only the matplotlib namespace, without the help of additional toolkits.




```
from matplotlib import pyplot as plt
from matplotlib.lines import Line2D
from matplotlib.offsetbox import (AnchoredOffsetbox, AuxTransformBox,
                                  DrawingArea, TextArea, VPacker)
from matplotlib.patches import Circle, Ellipse


def draw_text(ax):
    """Draw a text-box anchored to the upper-left corner of the figure."""
    box = AnchoredOffsetbox(child=TextArea("Figure 1a"),
                            loc="upper left", frameon=True)
    box.patch.set_boxstyle("round,pad=0.,rounding_size=0.2")
    ax.add_artist(box)


def draw_circles(ax):
    """Draw circles in axes coordinates."""
    area = DrawingArea(width=40, height=20)
    area.add_artist(Circle((10, 10), 10, fc="tab:blue"))
    area.add_artist(Circle((30, 10), 5, fc="tab:red"))
    box = AnchoredOffsetbox(
        child=area, loc="upper right", pad=0, frameon=False)
    ax.add_artist(box)
```

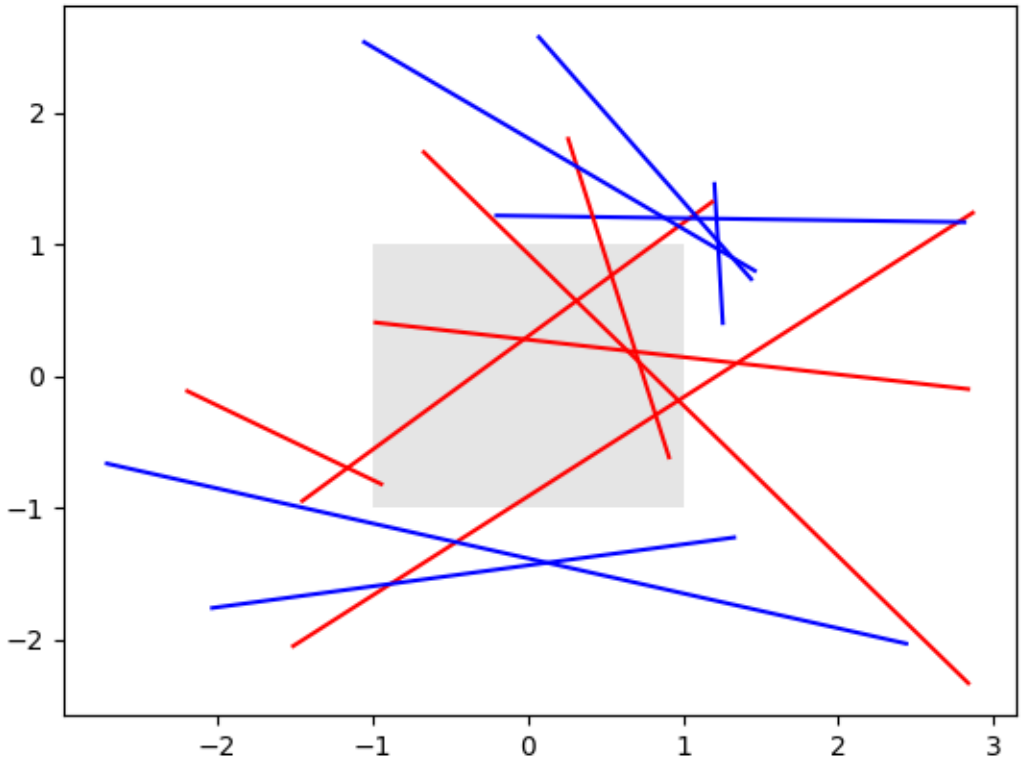
(continues on next page)

(continued from previous page)

```
def draw_ellipse(ax):
    """Draw an ellipse of width=0.1, height=0.15 in data coordinates."""
    aux_tr_box = AuxTransformBox(ax.transData)
    aux_tr_box.add_artist(Ellipse((0, 0), width=0.1, height=0.15))
    box = AnchoredOffsetbox(child=aux_tr_box, loc="lower left", frameon=True)
    ax.add_artist(box)


def draw_sizebar(ax):
    """
    Draw a horizontal bar with length of 0.1 in data coordinates,
    with a fixed label center-aligned underneath.
    """
    size = 0.1
    text = r"1$^{\prime}$"
    sizebar = AuxTransformBox(ax.transData)
    sizebar.add_artist(Line2D([0, size], [0, 0], color="black"))
    text = TextArea(text)
    packer = VPacker(
        children=[sizebar, text], align="center", sep=5)  # separation in␣
 ↪points.
    ax.add_artist(AnchoredOffsetbox(
        child=packer, loc="lower center", frameon=False,
        pad=0.1, borderpad=0.5))  # paddings relative to the legend fontsize.


fig, ax = plt.subplots()
ax.set_aspect(1)

draw_text(ax)
draw_circles(ax)
draw_ellipse(ax)
draw_sizebar(ax)

plt.show()
```

### Changing colors of lines intersecting a box

The lines intersecting the rectangle are colored in red, while the others are left as blue lines. This example showcases the *intersects_bbox* function.




```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.path import Path
from matplotlib.transforms import Bbox

# Fixing random state for reproducibility
np.random.seed(19680801)


left, bottom, width, height = (-1, -1, 2, 2)
rect = plt.Rectangle((left, bottom), width, height,
                     facecolor="black", alpha=0.1)

fig, ax = plt.subplots()
ax.add_patch(rect)

bbox = Bbox.from_bounds(left, bottom, width, height)

for i in range(12):
    vertices = (np.random.random((2, 2)) - 0.5) * 6.0
    path = Path(vertices)
    if path.intersects_bbox(bbox):
```

(continues on next page)

(continued from previous page)

```
        color = 'r'
    else:
        color = 'b'
    ax.plot(vertices[:, 0], vertices[:, 1], color=color)

plt.show()
```

### Manual Contour

Example of displaying your own contour lines and polygons using ContourSet.

```
import matplotlib.pyplot as plt

import matplotlib.cm as cm
from matplotlib.contour import ContourSet
from matplotlib.path import Path
```

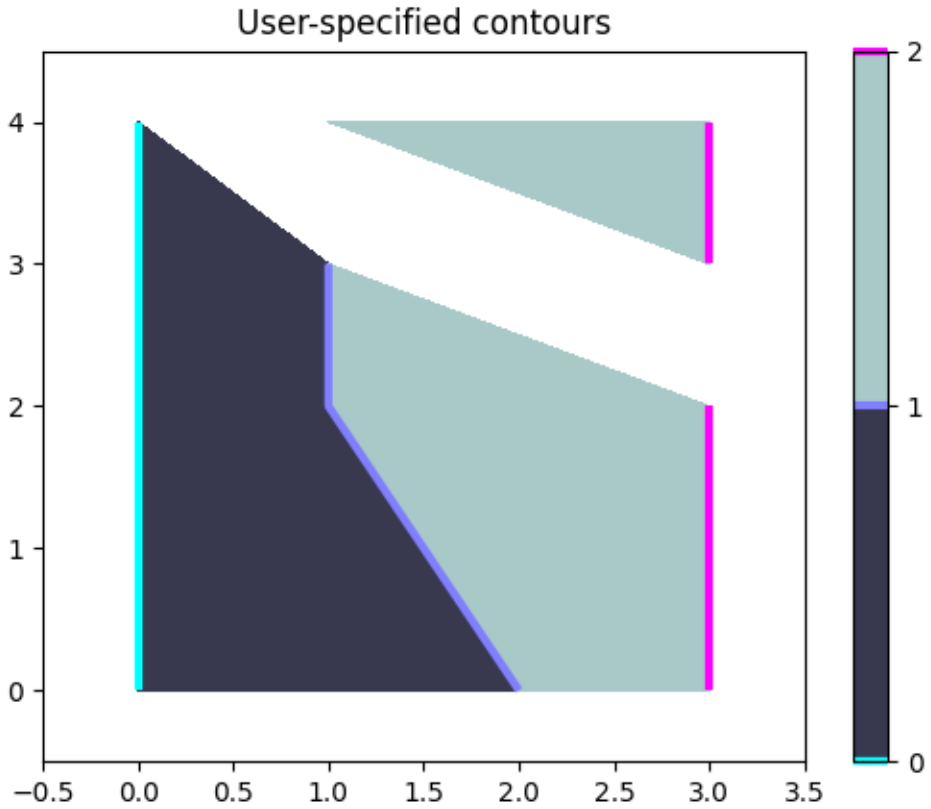
Contour lines for each level are a list/tuple of polygons.

```
lines0 = [[[0, 0], [0, 4]]]
lines1 = [[[2, 0], [1, 2], [1, 3]]]
lines2 = [[[3, 0], [3, 2]], [[3, 3], [3, 4]]]  # Note two lines.
```

Filled contours between two levels are also a list/tuple of polygons. Points can be ordered clockwise or anticlockwise.

```
filled01 = [[[0, 0], [0, 4], [1, 3], [1, 2], [2, 0]]]
filled12 = [[[2, 0], [3, 0], [3, 2], [1, 3], [1, 2]],   # Note two polygons.
            [[1, 4], [3, 4], [3, 3]]]
```

```
fig, ax = plt.subplots()

# Filled contours using filled=True.
cs = ContourSet(ax, [0, 1, 2], [filled01, filled12], filled=True, cmap=cm.
 ↪bone)
cbar = fig.colorbar(cs)

# Contour lines (non-filled).
lines = ContourSet(
    ax, [0, 1, 2], [lines0, lines1, lines2], cmap=cm.cool, linewidths=3)
cbar.add_lines(lines)

ax.set(xlim=(-0.5, 3.5), ylim=(-0.5, 4.5),
       title='User-specified contours')
```

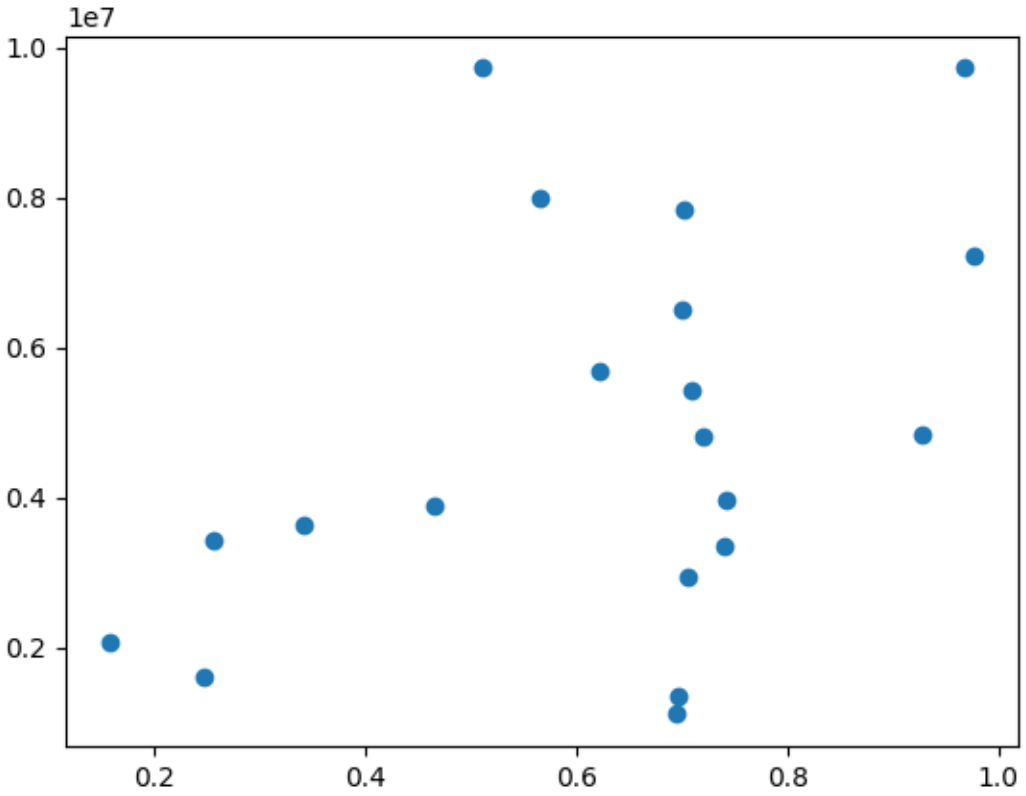



Multiple filled contour lines can be specified in a single list of polygon vertices along with a list of vertex kinds (code types) as described in the Path class. This is particularly useful for polygons with holes.

```
fig, ax = plt.subplots()
filled01 = [[[0, 0], [3, 0], [3, 3], [0, 3], [1, 1], [1, 2], [2, 2], [2, 1]]]
M = Path.MOVETO
L = Path.LINETO
kinds01 = [[M, L, L, L, M, L, L, L]]
cs = ContourSet(ax, [0, 1], [filled01], [kinds01], filled=True)
cbar = fig.colorbar(cs)

ax.set(xlim=(-0.5, 3.5), ylim=(-0.5, 3.5),
       title='User specified filled contours with holes')

plt.show()
```

### Coords Report

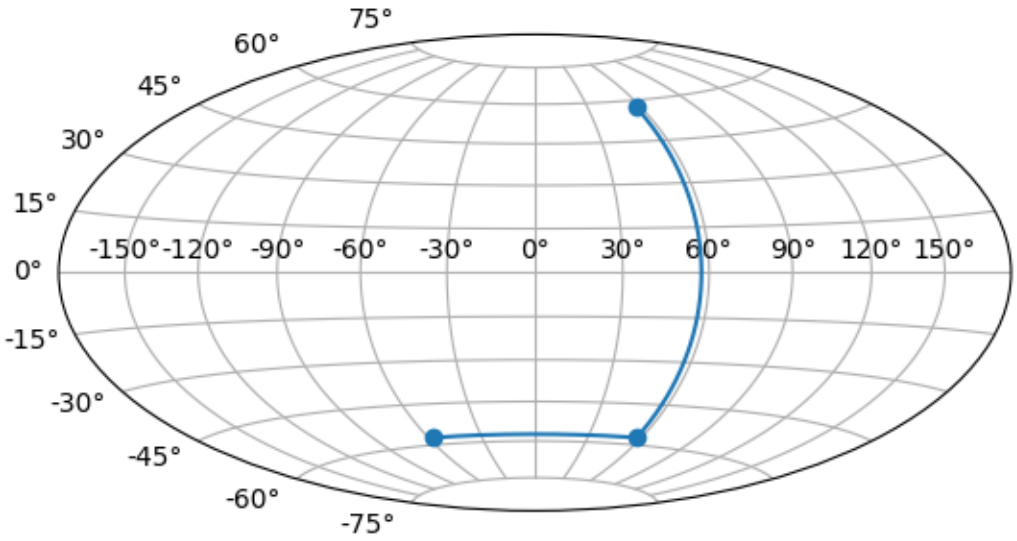
Override the default reporting of coords as the mouse moves over the Axes in an interactive backend.




```
import matplotlib.pyplot as plt
import numpy as np


def millions(x):
    return '$%1.1fM' % (x * 1e-6)


# Fixing random state for reproducibility
np.random.seed(19680801)

x = np.random.rand(20)
y = 1e7 * np.random.rand(20)

fig, ax = plt.subplots()
ax.fmt_ydata = millions
plt.plot(x, y, 'o')

plt.show()
```

### Custom projection

Showcase Hammer projection by alleviating many features of Matplotlib.




```
import numpy as np

import matplotlib
from matplotlib.axes import Axes
import matplotlib.axis as maxis
from matplotlib.patches import Circle
from matplotlib.path import Path
from matplotlib.projections import register_projection
import matplotlib.spines as mspines
from matplotlib.ticker import FixedLocator, Formatter, NullLocator
from matplotlib.transforms import Affine2D, BboxTransformTo, Transform

rcParams = matplotlib.rcParams

# This example projection class is rather long, but it is designed to
# illustrate many features, not all of which will be used every time.
# It is also common to factor out a lot of these methods into common
# code used by a number of projections with similar characteristics
# (see geo.py).
```

(continues on next page)

(continued from previous page)

```
class GeoAxes(Axes):
    """
    An abstract base class for geographic projections
    """
    class ThetaFormatter(Formatter):
        """
        Used to format the theta tick labels.  Converts the native
        unit of radians into degrees and adds a degree symbol.
        """
        def __init__(self, round_to=1.0):
            self._round_to = round_to

        def __call__(self, x, pos=None):
            degrees = round(np.rad2deg(x) / self._round_to) * self._round_to
            return f"{degrees:0.0f}\N{DEGREE SIGN}"

    RESOLUTION = 75

    def _init_axis(self):
        self.xaxis = maxis.XAxis(self)
        self.yaxis = maxis.YAxis(self)
        # Do not register xaxis or yaxis with spines -- as done in
        # Axes._init_axis() -- until GeoAxes.xaxis.clear() works.
        # self.spines['geo'].register_axis(self.yaxis)

    def clear(self):
        # docstring inherited
        super().clear()

        self.set_longitude_grid(30)
        self.set_latitude_grid(15)
        self.set_longitude_grid_ends(75)
        self.xaxis.set_minor_locator(NullLocator())
        self.yaxis.set_minor_locator(NullLocator())
        self.xaxis.set_ticks_position('none')
        self.yaxis.set_ticks_position('none')
        self.yaxis.set_tick_params(label1On=True)
        # Why do we need to turn on yaxis tick labels, but
        # xaxis tick labels are already on?

        self.grid(rcParams['axes.grid'])

        Axes.set_xlim(self, -np.pi, np.pi)
        Axes.set_ylim(self, -np.pi / 2.0, np.pi / 2.0)

    def _set_lim_and_transforms(self):
        # A (possibly non-linear) projection on the (already scaled) data

        # There are three important coordinate spaces going on here:
        #
```

(continues on next page)

(continued from previous page)

```
        # 1. Data space: The space of the data itself
        #
        # 2. Axes space: The unit rectangle (0, 0) to (1, 1)
        #    covering the entire plot area.
        #
        # 3. Display space: The coordinates of the resulting image,
        #    often in pixels or dpi/inch.

        # This function makes heavy use of the Transform classes in
        # ``lib/matplotlib/transforms.py.`` For more information, see
        # the inline documentation there.

        # The goal of the first two transformations is to get from the
        # data space (in this case longitude and latitude) to Axes
        # space.  It is separated into a non-affine and affine part so
        # that the non-affine part does not have to be recomputed when
        # a simple affine change to the figure has been made (such as
        # resizing the window or changing the dpi).

        # 1) The core transformation from data space into
        # rectilinear space defined in the HammerTransform class.
        self.transProjection = self._get_core_transform(self.RESOLUTION)

        # 2) The above has an output range that is not in the unit
        # rectangle, so scale and translate it so it fits correctly
        # within the Axes.  The peculiar calculations of xscale and
        # yscale are specific to an Aitoff-Hammer projection, so don't
        # worry about them too much.
        self.transAffine = self._get_affine_transform()

        # 3) This is the transformation from Axes space to display
        # space.
        self.transAxes = BboxTransformTo(self.bbox)

        # Now put these 3 transforms together -- from data all the way
        # to display coordinates.  Using the '+' operator, these
        # transforms will be applied "in order".  The transforms are
        # automatically simplified, if possible, by the underlying
        # transformation framework.
        self.transData = \
            self.transProjection + \
            self.transAffine + \
            self.transAxes

        # The main data transformation is set up.  Now deal with
        # gridlines and tick labels.

        # Longitude gridlines and ticklabels.  The input to these
        # transforms are in display space in x and Axes space in y.
        # Therefore, the input values will be in range (-xmin, 0),
        # (xmax, 1).  The goal of these transforms is to go from that
        # space to display space.  The tick labels will be offset 4
```

(continues on next page)

(continued from previous page)

```
        # pixels from the equator.
        self._xaxis_pretransform = \
            Affine2D() \
            .scale(1.0, self._longitude_cap * 2.0) \
            .translate(0.0, -self._longitude_cap)
        self._xaxis_transform = \
            self._xaxis_pretransform + \
            self.transData
        self._xaxis_text1_transform = \
            Affine2D().scale(1.0, 0.0) + \
            self.transData + \
            Affine2D().translate(0.0, 4.0)
        self._xaxis_text2_transform = \
            Affine2D().scale(1.0, 0.0) + \
            self.transData + \
            Affine2D().translate(0.0, -4.0)

        # Now set up the transforms for the latitude ticks.  The input to
        # these transforms are in Axes space in x and display space in
        # y.  Therefore, the input values will be in range (0, -ymin),
        # (1, ymax).  The goal of these transforms is to go from that
        # space to display space.  The tick labels will be offset 4
        # pixels from the edge of the Axes ellipse.
        yaxis_stretch = Affine2D().scale(np.pi*2, 1).translate(-np.pi, 0)
        yaxis_space = Affine2D().scale(1.0, 1.1)
        self._yaxis_transform = \
            yaxis_stretch + \
            self.transData
        yaxis_text_base = \
            yaxis_stretch + \
            self.transProjection + \
            (yaxis_space +
             self.transAffine +
             self.transAxes)
        self._yaxis_text1_transform = \
            yaxis_text_base + \
            Affine2D().translate(-8.0, 0.0)
        self._yaxis_text2_transform = \
            yaxis_text_base + \
            Affine2D().translate(8.0, 0.0)

    def _get_affine_transform(self):
        transform = self._get_core_transform(1)
        xscale, _ = transform.transform((np.pi, 0))
        _, yscale = transform.transform((0, np.pi/2))
        return Affine2D() \
            .scale(0.5 / xscale, 0.5 / yscale) \
            .translate(0.5, 0.5)

    def get_xaxis_transform(self, which='grid'):
        """
        Override this method to provide a transformation for the
```

(continues on next page)

(continued from previous page)

```
        x-axis tick labels.

        Returns a tuple of the form (transform, valign, halign)
        """
        if which not in ['tick1', 'tick2', 'grid']:
            raise ValueError(
                "'which' must be one of 'tick1', 'tick2', or 'grid'")
        return self._xaxis_transform

    def get_xaxis_text1_transform(self, pad):
        return self._xaxis_text1_transform, 'bottom', 'center'

    def get_xaxis_text2_transform(self, pad):
        """
        Override this method to provide a transformation for the
        secondary x-axis tick labels.

        Returns a tuple of the form (transform, valign, halign)
        """
        return self._xaxis_text2_transform, 'top', 'center'

    def get_yaxis_transform(self, which='grid'):
        """
        Override this method to provide a transformation for the
        y-axis grid and ticks.
        """
        if which not in ['tick1', 'tick2', 'grid']:
            raise ValueError(
                "'which' must be one of 'tick1', 'tick2', or 'grid'")
        return self._yaxis_transform

    def get_yaxis_text1_transform(self, pad):
        """
        Override this method to provide a transformation for the
        y-axis tick labels.

        Returns a tuple of the form (transform, valign, halign)
        """
        return self._yaxis_text1_transform, 'center', 'right'

    def get_yaxis_text2_transform(self, pad):
        """
        Override this method to provide a transformation for the
        secondary y-axis tick labels.

        Returns a tuple of the form (transform, valign, halign)
        """
        return self._yaxis_text2_transform, 'center', 'left'

    def _gen_axes_patch(self):
        """
        Override this method to define the shape that is used for the
```

(continues on next page)

(continued from previous page)

```
        background of the plot.  It should be a subclass of Patch.

        In this case, it is a Circle (that may be warped by the Axes
        transform into an ellipse).  Any data and gridlines will be
        clipped to this shape.
        """
        return Circle((0.5, 0.5), 0.5)

    def _gen_axes_spines(self):
        return {'geo': mspines.Spine.circular_spine(self, (0.5, 0.5), 0.5)}

    def set_yscale(self, *args, **kwargs):
        if args[0] != 'linear':
            raise NotImplementedError

    # Prevent the user from applying scales to one or both of the
    # axes.  In this particular case, scaling the axes wouldn't make
    # sense, so we don't allow it.
    set_xscale = set_yscale

    # Prevent the user from changing the axes limits.  In our case, we
    # want to display the whole sphere all the time, so we override
    # set_xlim and set_ylim to ignore any input.  This also applies to
    # interactive panning and zooming in the GUI interfaces.
    def set_xlim(self, *args, **kwargs):
        raise TypeError("Changing axes limits of a geographic projection is "
                        "not supported.  Please consider using Cartopy.")

    set_ylim = set_xlim

    def format_coord(self, lon, lat):
        """
        Override this method to change how the values are displayed in
        the status bar.

        In this case, we want them to be displayed in degrees N/S/E/W.
        """
        lon, lat = np.rad2deg([lon, lat])
        ns = 'N' if lat >= 0.0 else 'S'
        ew = 'E' if lon >= 0.0 else 'W'
        return ('%f\N{DEGREE SIGN}%s, %f\N{DEGREE SIGN}%s'
                % (abs(lat), ns, abs(lon), ew))

    def set_longitude_grid(self, degrees):
        """
        Set the number of degrees between each longitude grid.

        This is an example method that is specific to this projection
        class -- it provides a more convenient interface to set the
        ticking than set_xticks would.
        """
        # Skip -180 and 180, which are the fixed limits.
```

(continues on next page)

(continued from previous page)

```
        grid = np.arange(-180 + degrees, 180, degrees)
        self.xaxis.set_major_locator(FixedLocator(np.deg2rad(grid)))
        self.xaxis.set_major_formatter(self.ThetaFormatter(degrees))

    def set_latitude_grid(self, degrees):
        """
        Set the number of degrees between each longitude grid.

        This is an example method that is specific to this projection
        class -- it provides a more convenient interface than
        set_yticks would.
        """
        # Skip -90 and 90, which are the fixed limits.
        grid = np.arange(-90 + degrees, 90, degrees)
        self.yaxis.set_major_locator(FixedLocator(np.deg2rad(grid)))
        self.yaxis.set_major_formatter(self.ThetaFormatter(degrees))

    def set_longitude_grid_ends(self, degrees):
        """
        Set the latitude(s) at which to stop drawing the longitude grids.

        Often, in geographic projections, you wouldn't want to draw
        longitude gridlines near the poles.  This allows the user to
        specify the degree at which to stop drawing longitude grids.

        This is an example method that is specific to this projection
        class -- it provides an interface to something that has no
        analogy in the base Axes class.
        """
        self._longitude_cap = np.deg2rad(degrees)
        self._xaxis_pretransform \
            .clear() \
            .scale(1.0, self._longitude_cap * 2.0) \
            .translate(0.0, -self._longitude_cap)

    def get_data_ratio(self):
        """
        Return the aspect ratio of the data itself.

        This method should be overridden by any Axes that have a
        fixed data ratio.
        """
        return 1.0

    # Interactive panning and zooming is not supported with this projection,
    # so we override all of the following methods to disable it.
    def can_zoom(self):
        """
        Return whether this Axes supports the zoom box button functionality.

        This Axes object does not support interactive zoom box.
        """
```

(continues on next page)

(continued from previous page)

```
        return False

    def can_pan(self):
        """
        Return whether this Axes supports the pan/zoom button functionality.

        This Axes object does not support interactive pan/zoom.
        """
        return False

    def start_pan(self, x, y, button):
        pass

    def end_pan(self):
        pass

    def drag_pan(self, button, key, x, y):
        pass


class HammerAxes(GeoAxes):
    """
    A custom class for the Aitoff-Hammer projection, an equal-area map
    projection.

    https://en.wikipedia.org/wiki/Hammer_projection
    """

    # The projection must specify a name.  This will be used by the
    # user to select the projection,
    # i.e. ``subplot(projection='custom_hammer')``.
    name = 'custom_hammer'

    class HammerTransform(Transform):
        """The base Hammer transform."""
        input_dims = output_dims = 2

        def __init__(self, resolution):
            """
            Create a new Hammer transform.  Resolution is the number of steps
            to interpolate between each input line segment to approximate its
            path in curved Hammer space.
            """
            Transform.__init__(self)
            self._resolution = resolution

        def transform_non_affine(self, ll):
            longitude, latitude = ll.T

            # Pre-compute some values
            half_long = longitude / 2
            cos_latitude = np.cos(latitude)
```

(continues on next page)

(continued from previous page)

```
            sqrt2 = np.sqrt(2)

            alpha = np.sqrt(1 + cos_latitude * np.cos(half_long))
            x = (2 * sqrt2) * (cos_latitude * np.sin(half_long)) / alpha
            y = (sqrt2 * np.sin(latitude)) / alpha
            return np.column_stack([x, y])

        def transform_path_non_affine(self, path):
            # vertices = path.vertices
            ipath = path.interpolated(self._resolution)
            return Path(self.transform(ipath.vertices), ipath.codes)

        def inverted(self):
            return HammerAxes.InvertedHammerTransform(self._resolution)

    class InvertedHammerTransform(Transform):
        input_dims = output_dims = 2

        def __init__(self, resolution):
            Transform.__init__(self)
            self._resolution = resolution

        def transform_non_affine(self, xy):
            x, y = xy.T
            z = np.sqrt(1 - (x / 4) ** 2 - (y / 2) ** 2)
            longitude = 2 * np.arctan((z * x) / (2 * (2 * z ** 2 - 1)))
            latitude = np.arcsin(y*z)
            return np.column_stack([longitude, latitude])

        def inverted(self):
            return HammerAxes.HammerTransform(self._resolution)

    def __init__(self, *args, **kwargs):
        self._longitude_cap = np.pi / 2.0
        super().__init__(*args, **kwargs)
        self.set_aspect(0.5, adjustable='box', anchor='C')
        self.clear()

    def _get_core_transform(self, resolution):
        return self.HammerTransform(resolution)


# Now register the projection with Matplotlib so the user can select it.
register_projection(HammerAxes)


if __name__ == '__main__':
    import matplotlib.pyplot as plt

    # Now make a simple example using the custom projection.
    fig, ax = plt.subplots(subplot_kw={'projection': 'custom_hammer'})
    ax.plot([-1, 1, 1], [-1, -1, 1], "o-")
```

(continues on next page)

(continued from previous page)

```
    ax.grid()

    plt.show()
```

### Customize Rc

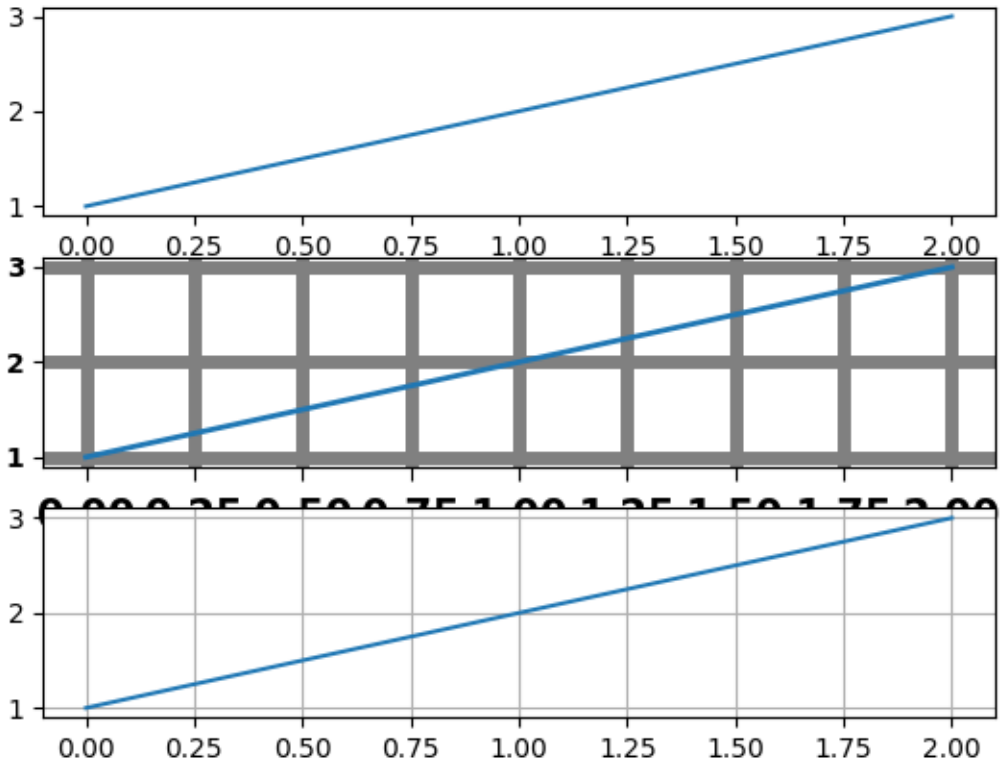
I'm not trying to make a good-looking figure here, but just to show some examples of customizing *rcParams* on the fly.

If you like to work interactively, and need to create different sets of defaults for figures (e.g., one set of defaults for publication, one set for interactive exploration), you may want to define some functions in a custom module that set the defaults, e.g.,:

```
def set_pub():
    rcParams.update({
        "font.weight": "bold",   # bold fonts
        "tick.labelsize": 15,    # large tick labels
        "lines.linewidth": 1,    # thick lines
        "lines.color": "k",      # black lines
        "grid.color": "0.5",     # gray gridlines
        "grid.linestyle": "-",   # solid gridlines
        "grid.linewidth": 0.5,   # thin gridlines
        "savefig.dpi": 300,      # higher resolution output.
    })
```

Then as you are working interactively, you just need to do:

```
>>> set_pub()
>>> plot([1, 2, 3])
>>> savefig('myfig')
>>> rcdefaults()  # restore the defaults
```




```
import matplotlib.pyplot as plt

plt.subplot(311)
plt.plot([1, 2, 3])

# the axes attributes need to be set before the call to subplot
plt.rcParams.update({
    "font.weight": "bold",
    "xtick.major.size": 5,
    "xtick.major.pad": 7,
    "xtick.labelsize": 15,
    "grid.color": "0.5",
    "grid.linestyle": "-",
    "grid.linewidth": 5,
    "lines.linewidth": 2,
    "lines.color": "g",
})
plt.subplot(312)
plt.plot([1, 2, 3])
plt.grid(True)

plt.rcdefaults()
plt.subplot(313)
```

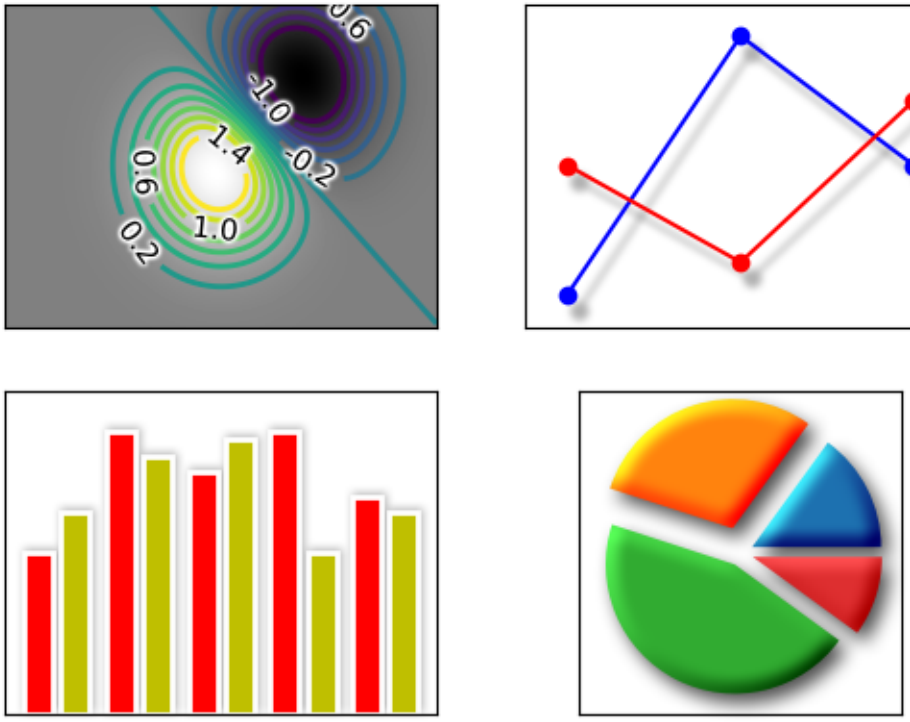
(continues on next page)

(continued from previous page)

```
plt.plot([1, 2, 3])
plt.grid(True)
plt.show()
```

### AGG filter

Most pixel-based backends in Matplotlib use Anti-Grain Geometry (AGG) for rendering. You can modify the rendering of Artists by applying a filter via *Artist.set_agg_filter*.




```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.artist import Artist
import matplotlib.cm as cm
from matplotlib.colors import LightSource
import matplotlib.transforms as mtransforms


def smooth1d(x, window_len):
    # copied from https://scipy-cookbook.readthedocs.io/items/SignalSmooth.
```

(continues on next page)

(continued from previous page)

```
↪html
    s = np.r_[2*x[0] - x[window_len:1:-1], x, 2*x[-1] - x[-1:-window_len:-1]]
    w = np.hanning(window_len)
    y = np.convolve(w/w.sum(), s, mode='same')
    return y[window_len-1:-window_len+1]


def smooth2d(A, sigma=3):
    window_len = max(int(sigma), 3) * 2 + 1
    A = np.apply_along_axis(smooth1d, 0, A, window_len)
    A = np.apply_along_axis(smooth1d, 1, A, window_len)
    return A


class BaseFilter:

    def get_pad(self, dpi):
        return 0

    def process_image(self, padded_src, dpi):
        raise NotImplementedError("Should be overridden by subclasses")

    def __call__(self, im, dpi):
        pad = self.get_pad(dpi)
        padded_src = np.pad(im, [(pad, pad), (pad, pad), (0, 0)], "constant")
        tgt_image = self.process_image(padded_src, dpi)
        return tgt_image, -pad, -pad


class OffsetFilter(BaseFilter):

    def __init__(self, offsets=(0, 0)):
        self.offsets = offsets

    def get_pad(self, dpi):
        return int(max(self.offsets) / 72 * dpi)

    def process_image(self, padded_src, dpi):
        ox, oy = self.offsets
        a1 = np.roll(padded_src, int(ox / 72 * dpi), axis=1)
        a2 = np.roll(a1, -int(oy / 72 * dpi), axis=0)
        return a2


class GaussianFilter(BaseFilter):
    """Simple Gaussian filter."""

    def __init__(self, sigma, alpha=0.5, color=(0, 0, 0)):
        self.sigma = sigma
        self.alpha = alpha
        self.color = color
```

(continues on next page)

(continued from previous page)

```
    def get_pad(self, dpi):
        return int(self.sigma*3 / 72 * dpi)

    def process_image(self, padded_src, dpi):
        tgt_image = np.empty_like(padded_src)
        tgt_image[:, :, :3] = self.color
        tgt_image[:, :, 3] = smooth2d(padded_src[:, :, 3] * self.alpha,
                                      self.sigma / 72 * dpi)
        return tgt_image


class DropShadowFilter(BaseFilter):

    def __init__(self, sigma, alpha=0.3, color=(0, 0, 0), offsets=(0, 0)):
        self.gauss_filter = GaussianFilter(sigma, alpha, color)
        self.offset_filter = OffsetFilter(offsets)

    def get_pad(self, dpi):
        return max(self.gauss_filter.get_pad(dpi),
                   self.offset_filter.get_pad(dpi))

    def process_image(self, padded_src, dpi):
        t1 = self.gauss_filter.process_image(padded_src, dpi)
        t2 = self.offset_filter.process_image(t1, dpi)
        return t2


class LightFilter(BaseFilter):
    """Apply LightSource filter"""

    def __init__(self, sigma, fraction=1):
        """
        Parameters
        ----------
        sigma : float
            sigma for gaussian filter
        fraction: number, default: 1
            Increases or decreases the contrast of the hillshade.
            See `matplotlib.colors.LightSource`

        """
        self.gauss_filter = GaussianFilter(sigma, alpha=1)
        self.light_source = LightSource()
        self.fraction = fraction

    def get_pad(self, dpi):
        return self.gauss_filter.get_pad(dpi)

    def process_image(self, padded_src, dpi):
        t1 = self.gauss_filter.process_image(padded_src, dpi)
        elevation = t1[:, :, 3]
        rgb = padded_src[:, :, :3]
```

(continues on next page)

(continued from previous page)

```
        alpha = padded_src[:, :, 3:]
        rgb2 = self.light_source.shade_rgb(rgb, elevation,
                                           fraction=self.fraction,
                                           blend_mode="overlay")
        return np.concatenate([rgb2, alpha], -1)


class GrowFilter(BaseFilter):
    """Enlarge the area."""

    def __init__(self, pixels, color=(1, 1, 1)):
        self.pixels = pixels
        self.color = color

    def __call__(self, im, dpi):
        alpha = np.pad(im[..., 3], self.pixels, "constant")
        alpha2 = np.clip(smooth2d(alpha, self.pixels / 72 * dpi) * 5, 0, 1)
        new_im = np.empty((*alpha2.shape, 4))
        new_im[:, :, :3] = self.color
        new_im[:, :, 3] = alpha2
        offsetx, offsety = -self.pixels, -self.pixels
        return new_im, offsetx, offsety


class FilteredArtistList(Artist):
    """A simple container to filter multiple artists at once."""

    def __init__(self, artist_list, filter):
        super().__init__()
        self._artist_list = artist_list
        self._filter = filter

    def draw(self, renderer):
        renderer.start_rasterizing()
        renderer.start_filter()
        for a in self._artist_list:
            a.draw(renderer)
        renderer.stop_filter(self._filter)
        renderer.stop_rasterizing()


def filtered_text(ax):
    # mostly copied from contour_demo.py

    # prepare image
    delta = 0.025
    x = np.arange(-3.0, 3.0, delta)
    y = np.arange(-2.0, 2.0, delta)
    X, Y = np.meshgrid(x, y)
    Z1 = np.exp(-X**2 - Y**2)
    Z2 = np.exp(-(X - 1)**2 - (Y - 1)**2)
    Z = (Z1 - Z2) * 2
```

(continues on next page)

(continued from previous page)

```
    # draw
    ax.imshow(Z, interpolation='bilinear', origin='lower',
              cmap=cm.gray, extent=(-3, 3, -2, 2), aspect='auto')
    levels = np.arange(-1.2, 1.6, 0.2)
    CS = ax.contour(Z, levels,
                    origin='lower',
                    linewidths=2,
                    extent=(-3, 3, -2, 2))

    # contour label
    cl = ax.clabel(CS, levels[1::2],  # label every second level
                   inline=True,
                   fmt='%1.1f',
                   fontsize=11)

    # change clabel color to black
    from matplotlib.patheffects import Normal
    for t in cl:
        t.set_color("k")
        # to force TextPath (i.e., same font in all backends)
        t.set_path_effects([Normal()])

    # Add white glows to improve visibility of labels.
    white_glows = FilteredArtistList(cl, GrowFilter(3))
    ax.add_artist(white_glows)
    white_glows.set_zorder(cl[0].get_zorder() - 0.1)

    ax.xaxis.set_visible(False)
    ax.yaxis.set_visible(False)


def drop_shadow_line(ax):
    # copied from examples/misc/svg_filter_line.py

    # draw lines
    l1, = ax.plot([0.1, 0.5, 0.9], [0.1, 0.9, 0.5], "bo-")
    l2, = ax.plot([0.1, 0.5, 0.9], [0.5, 0.2, 0.7], "ro-")

    gauss = DropShadowFilter(4)

    for l in [l1, l2]:

        # draw shadows with same lines with slight offset.
        xx = l.get_xdata()
        yy = l.get_ydata()
        shadow, = ax.plot(xx, yy)
        shadow.update_from(l)

        # offset transform
        transform = mtransforms.offset_copy(l.get_transform(), ax.figure,
                                            x=4.0, y=-6.0, units='points')
```

(continues on next page)

(continued from previous page)

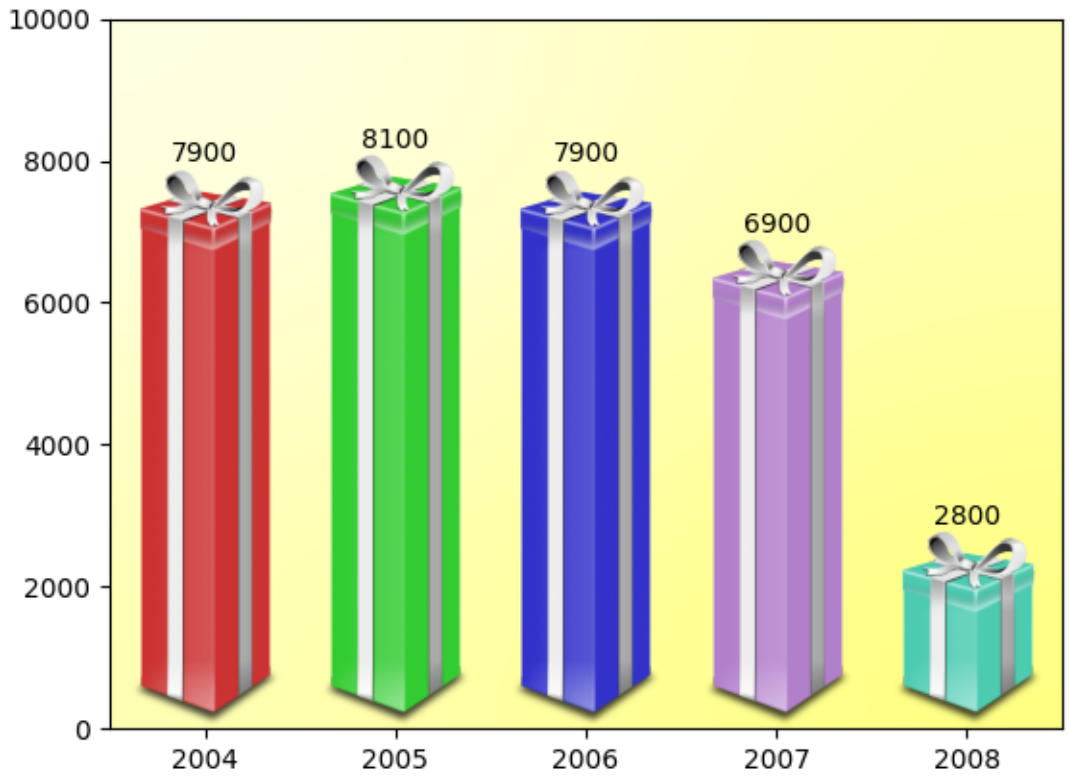
```
        shadow.set_transform(transform)

        # adjust zorder of the shadow lines so that it is drawn below the
        # original lines
        shadow.set_zorder(l.get_zorder() - 0.5)
        shadow.set_agg_filter(gauss)
        shadow.set_rasterized(True)  # to support mixed-mode renderers

    ax.set_xlim(0., 1.)
    ax.set_ylim(0., 1.)

    ax.xaxis.set_visible(False)
    ax.yaxis.set_visible(False)


def drop_shadow_patches(ax):
    # Copied from barchart_demo.py
    N = 5
    group1_means = [20, 35, 30, 35, 27]

    ind = np.arange(N)  # the x locations for the groups
    width = 0.35  # the width of the bars

    rects1 = ax.bar(ind, group1_means, width, color='r', ec="w", lw=2)

    group2_means = [25, 32, 34, 20, 25]
    rects2 = ax.bar(ind + width + 0.1, group2_means, width,
                    color='y', ec="w", lw=2)

    drop = DropShadowFilter(5, offsets=(1, 1))
    shadow = FilteredArtistList(rects1 + rects2, drop)
    ax.add_artist(shadow)
    shadow.set_zorder(rects1[0].get_zorder() - 0.1)

    ax.set_ylim(0, 40)

    ax.xaxis.set_visible(False)
    ax.yaxis.set_visible(False)


def light_filter_pie(ax):
    fracs = [15, 30, 45, 10]
    explode = (0.1, 0.2, 0.1, 0.1)
    pies = ax.pie(fracs, explode=explode)

    light_filter = LightFilter(9)
    for p in pies[0]:
        p.set_agg_filter(light_filter)
        p.set_rasterized(True)  # to support mixed-mode renderers
        p.set(ec="none",
              lw=2)
```

(continues on next page)

(continued from previous page)

```
    gauss = DropShadowFilter(9, offsets=(3, -4), alpha=0.7)
    shadow = FilteredArtistList(pies[0], gauss)
    ax.add_artist(shadow)
    shadow.set_zorder(pies[0][0].get_zorder() - 0.1)


if __name__ == "__main__":

    fix, axs = plt.subplots(2, 2)

    filtered_text(axs[0, 0])
    drop_shadow_line(axs[0, 1])
    drop_shadow_patches(axs[1, 0])
    light_filter_pie(axs[1, 1])
    axs[1, 1].set_frame_on(True)

    plt.show()
```

### Ribbon Box



```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib import cbook
from matplotlib import colors as mcolors
from matplotlib.image import AxesImage
from matplotlib.transforms import Bbox, BboxTransformTo, TransformedBbox


class RibbonBox:

    original_image = plt.imread(
        cbook.get_sample_data("Minduka_Present_Blue_Pack.png"))
    cut_location = 70
    b_and_h = original_image[:, :, 2:3]
    color = original_image[:, :, 2:3] - original_image[:, :, 0:1]
    alpha = original_image[:, :, 3:4]
    nx = original_image.shape[1]

    def __init__(self, color):
        rgb = mcolors.to_rgb(color)
        self.im = np.dstack(
            [self.b_and_h - self.color * (1 - np.array(rgb)), self.alpha])

    def get_stretched_image(self, stretch_factor):
        stretch_factor = max(stretch_factor, 1)
        ny, nx, nch = self.im.shape
        ny2 = int(ny*stretch_factor)
        return np.vstack(
            [self.im[:self.cut_location],
             np.broadcast_to(
                 self.im[self.cut_location], (ny2 - ny, nx, nch)),
             self.im[self.cut_location:]])


class RibbonBoxImage(AxesImage):
    zorder = 1

    def __init__(self, ax, bbox, color, *, extent=(0, 1, 0, 1), **kwargs):
        super().__init__(ax, extent=extent, **kwargs)
        self._bbox = bbox
        self._ribbonbox = RibbonBox(color)
        self.set_transform(BboxTransformTo(bbox))

    def draw(self, renderer):
        stretch_factor = self._bbox.height / self._bbox.width

        ny = int(stretch_factor*self._ribbonbox.nx)
        if self.get_array() is None or self.get_array().shape[0] != ny:
            arr = self._ribbonbox.get_stretched_image(stretch_factor)
            self.set_array(arr)
```

(continues on next page)

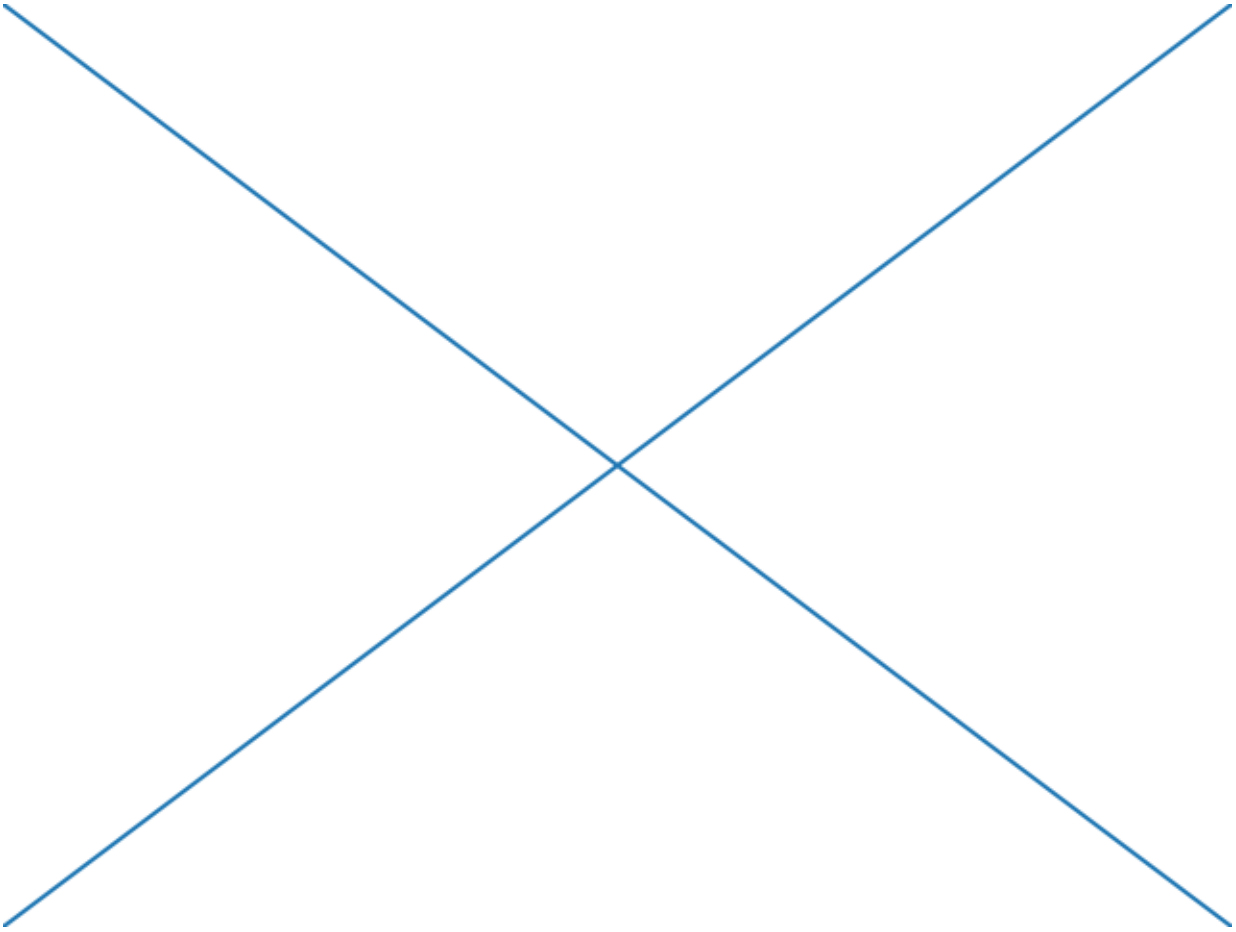
(continued from previous page)

```
        super().draw(renderer)


def main():
    fig, ax = plt.subplots()

    years = np.arange(2004, 2009)
    heights = [7900, 8100, 7900, 6900, 2800]
    box_colors = [
        (0.8, 0.2, 0.2),
        (0.2, 0.8, 0.2),
        (0.2, 0.2, 0.8),
        (0.7, 0.5, 0.8),
        (0.3, 0.8, 0.7),
    ]

    for year, h, bc in zip(years, heights, box_colors):
        bbox0 = Bbox.from_extents(year - 0.4, 0., year + 0.4, h)
        bbox = TransformedBbox(bbox0, ax.transData)
        ax.add_artist(RibbonBoxImage(ax, bbox, bc, interpolation="bicubic"))
        ax.annotate(str(h), (year, h), va="bottom", ha="center")

    ax.set_xlim(years[0] - 0.5, years[-1] + 0.5)
    ax.set_ylim(0, 10000)

    background_gradient = np.zeros((2, 2, 4))
    background_gradient[:, :, :3] = [1, 1, 0]
    background_gradient[:, :, 3] = [[0.1, 0.3], [0.3, 0.5]]  # alpha channel
    ax.imshow(background_gradient, interpolation="bicubic", zorder=0.1,
              extent=(0, 1, 0, 1), transform=ax.transAxes)

    plt.show()


main()
```

### Adding lines to figures

Adding lines to a figure without any Axes.

```
import matplotlib.pyplot as plt

import matplotlib.lines as lines

fig = plt.figure()
fig.add_artist(lines.Line2D([0, 1], [0, 1]))
fig.add_artist(lines.Line2D([0, 1], [1, 0]))
plt.show()
```

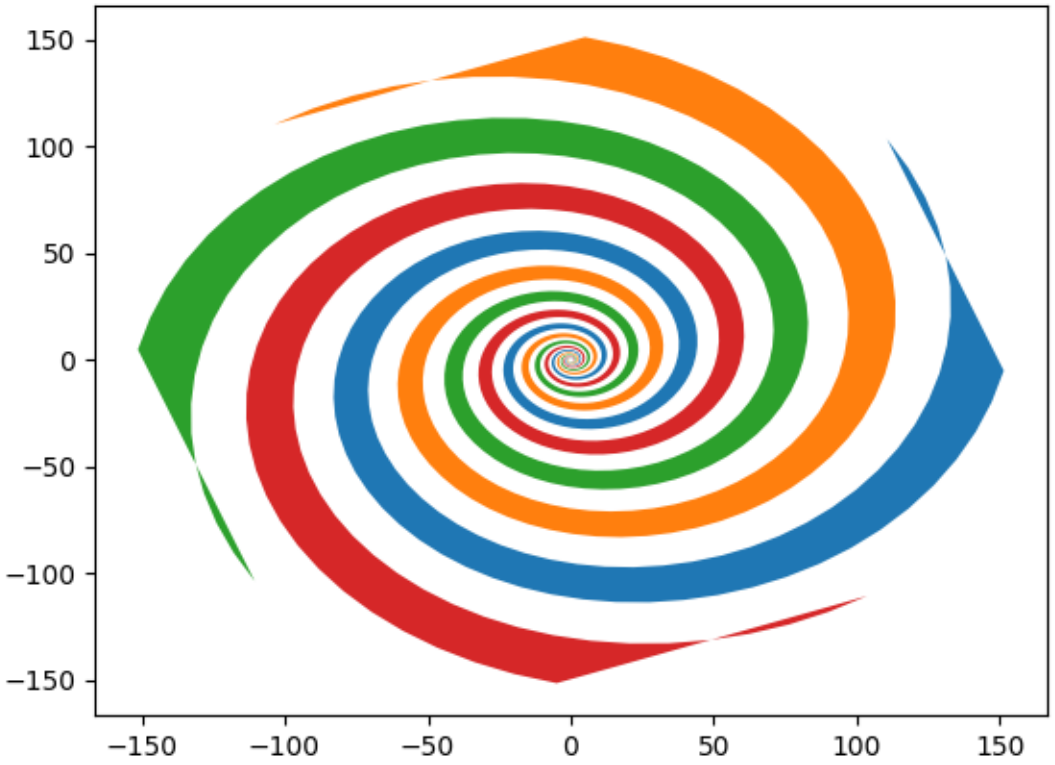


**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.pyplot.figure*
- *matplotlib.lines*
- *matplotlib.lines.Line2D*

### Fill Spiral




```
import matplotlib.pyplot as plt
import numpy as np

theta = np.arange(0, 8*np.pi, 0.1)
a = 1
b = .2

for dt in np.arange(0, 2*np.pi, np.pi/2.0):

    x = a*np.cos(theta + dt)*np.exp(b*theta)
    y = a*np.sin(theta + dt)*np.exp(b*theta)

    dt = dt + np.pi/4.0

    x2 = a*np.cos(theta + dt)*np.exp(b*theta)
    y2 = a*np.sin(theta + dt)*np.exp(b*theta)

    xf = np.concatenate((x, x2[::-1]))
    yf = np.concatenate((y, y2[::-1]))

    p1 = plt.fill(xf, yf)
```

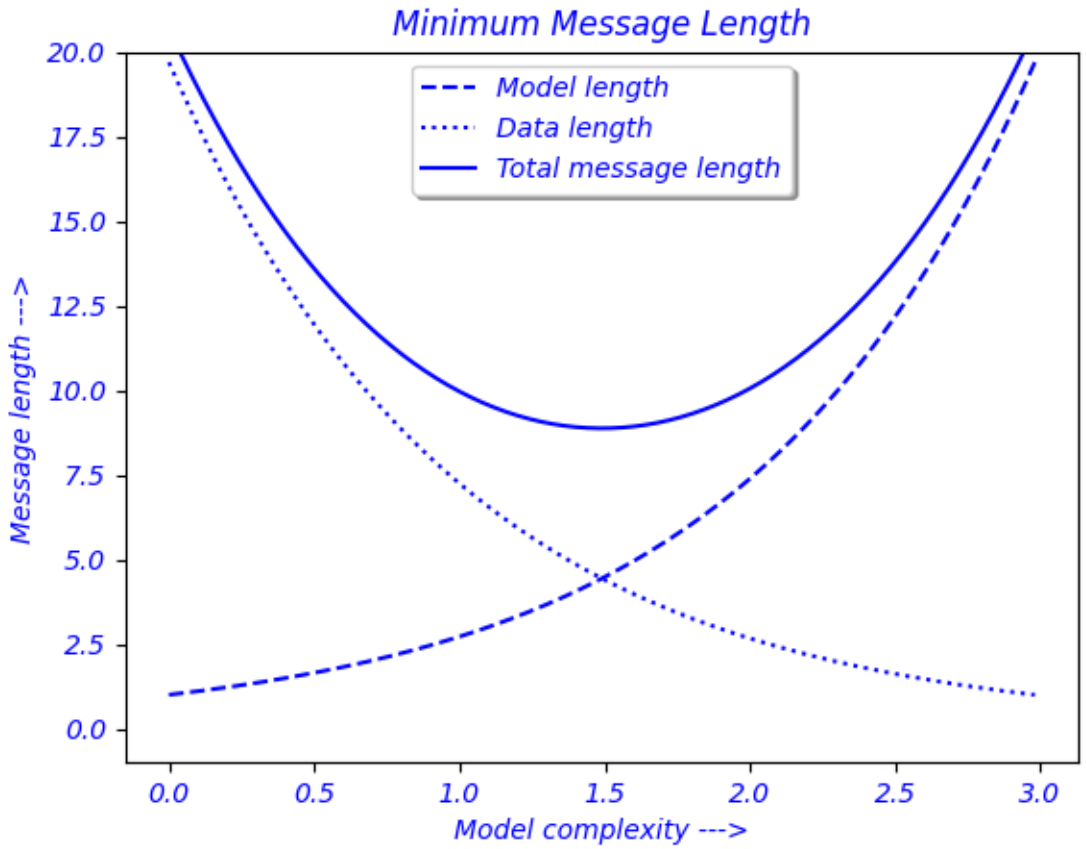
(continues on next page)

(continued from previous page)

```
plt.show()
```

### Findobj Demo

Recursively find all objects that match some criteria




```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib.text as text

a = np.arange(0, 3, .02)
b = np.arange(0, 3, .02)
c = np.exp(a)
d = c[::-1]

fig, ax = plt.subplots()
plt.plot(a, c, 'k--', a, d, 'k:', a, c + d, 'k')
plt.legend(('Model length', 'Data length', 'Total message length'),
```

(continues on next page)

(continued from previous page)

```
            loc='upper center', shadow=True)
plt.ylim([-1, 20])
plt.grid(False)
plt.xlabel('Model complexity --->')
plt.ylabel('Message length --->')
plt.title('Minimum Message Length')


# match on arbitrary function
def myfunc(x):
    return hasattr(x, 'set_color') and not hasattr(x, 'set_facecolor')


for o in fig.findobj(myfunc):
    o.set_color('blue')

# match on class instances
for o in fig.findobj(text.Text):
    o.set_fontstyle('italic')


plt.show()
```

### Font indexing

This example shows how the font tables relate to one another.

```
(6, 0, 519, 576)
36 57 65 86
AV -48
AV -49
AV -16
AT -19
```

```
import os

import matplotlib
from matplotlib.ft2font import (KERNING_DEFAULT, KERNING_UNFITTED,
                                KERNING_UNSCALED, FT2Font)

font = FT2Font(
    os.path.join(matplotlib.get_data_path(), 'fonts/ttf/DejaVuSans.ttf'))
font.set_charmap(0)
```

(continues on next page)

(continued from previous page)

```
codes = font.get_charmap().items()

# make a charname to charcode and glyphind dictionary
coded = {}
glyphd = {}
for ccode, glyphind in codes:
    name = font.get_glyph_name(glyphind)
    coded[name] = ccode
    glyphd[name] = glyphind
    # print(glyphind, ccode, hex(int(ccode)), name)

code = coded['A']
glyph = font.load_char(code)
print(glyph.bbox)
print(glyphd['A'], glyphd['V'], coded['A'], coded['V'])
print('AV', font.get_kerning(glyphd['A'], glyphd['V'], KERNING_DEFAULT))
print('AV', font.get_kerning(glyphd['A'], glyphd['V'], KERNING_UNFITTED))
print('AV', font.get_kerning(glyphd['A'], glyphd['V'], KERNING_UNSCALED))
print('AT', font.get_kerning(glyphd['A'], glyphd['T'], KERNING_UNSCALED))
```

## Font properties

This example lists the attributes of an *FT2Font* object, which describe global font properties. For individual character metrics, use the `Glyph` object, as returned by *load_char*.

```
Num faces:   1
Num glyphs:  5343
Family name: DejaVu Sans
Style name:  Oblique
PS name:     DejaVuSans-Oblique
Num fixed:   0
Bbox:                (-2080, -717, 3398, 2187)
EM:                  2048
Ascender:            1901
Descender:           -483
Height:              2384
Max adv width:       3461
Max adv height:      2384
Underline pos:       -175
Underline thickness: 90
Italic:           True
Bold:             False
Scalable:         True
Fixed sizes:      False
Fixed width:      False
SFNT:             False
Horizontal:       False
Vertical:         False
Kerning:          False
Fast glyphs:      False
```

(continues on next page)

(continued from previous page)

```
Multiple masters: False
Glyph names:      False
External stream:  False
```

```
import os

import matplotlib
import matplotlib.ft2font as ft

font = ft.FT2Font(
    # Use a font shipped with Matplotlib.
    os.path.join(matplotlib.get_data_path(),
                 'fonts/ttf/DejaVuSans-Oblique.ttf'))

print('Num faces:  ', font.num_faces)        # number of faces in file
print('Num glyphs: ', font.num_glyphs)       # number of glyphs in the face
print('Family name:', font.family_name)      # face family name
print('Style name: ', font.style_name)       # face style name
print('PS name:    ', font.postscript_name)  # the postscript name
print('Num fixed:  ', font.num_fixed_sizes)  # number of embedded bitmaps

# the following are only available if face.scalable
if font.scalable:
    # the face global bounding box (xmin, ymin, xmax, ymax)
    print('Bbox:               ', font.bbox)
    # number of font units covered by the EM
    print('EM:                 ', font.units_per_EM)
    # the ascender in 26.6 units
    print('Ascender:           ', font.ascender)
    # the descender in 26.6 units
    print('Descender:          ', font.descender)
    # the height in 26.6 units
    print('Height:             ', font.height)
    # maximum horizontal cursor advance
    print('Max adv width:      ', font.max_advance_width)
    # same for vertical layout
    print('Max adv height:     ', font.max_advance_height)
    # vertical position of the underline bar
    print('Underline pos:      ', font.underline_position)
    # vertical thickness of the underline
    print('Underline thickness:', font.underline_thickness)

for style in ('Italic',
              'Bold',
              'Scalable',
              'Fixed sizes',
```

(continues on next page)

(continued from previous page)

```
                'Fixed width',
                'SFNT',
                'Horizontal',
                'Vertical',
                'Kerning',
                'Fast glyphs',
                'Multiple masters',
                'Glyph names',
                'External stream'):
    bitpos = getattr(ft, style.replace(' ', '_').upper()) - 1
    print(f"{style+':':17}", bool(font.style_flags & (1 << bitpos)))
```

### Building histograms using Rectangles and PolyCollections

Using a path patch to draw rectangles.

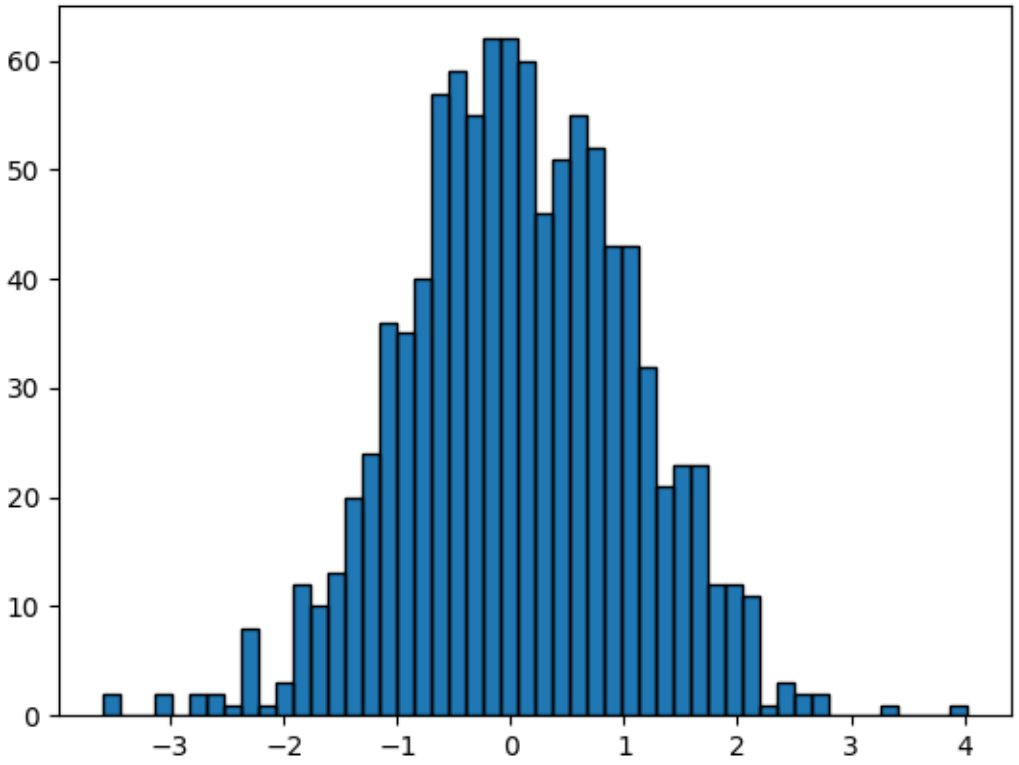
The technique of using lots of *Rectangle* instances, or the faster method of using *PolyCollection*, were implemented before we had proper paths with moveto, lineto, closepoly, etc. in Matplotlib. Now that we have them, we can draw collections of regularly shaped objects with homogeneous properties more efficiently with a PathCollection. This example makes a histogram -- it's more work to set up the vertex arrays at the outset, but it should be much faster for large numbers of objects.

```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib.patches as patches
import matplotlib.path as path

np.random.seed(19680801)  # Fixing random state for reproducibility

# histogram our data with numpy
data = np.random.randn(1000)
n, bins = np.histogram(data, 50)

# get the corners of the rectangles for the histogram
left = bins[:-1]
right = bins[1:]
bottom = np.zeros(len(left))
top = bottom + n

# we need a (numrects x numsides x 2) numpy array for the path helper
# function to build a compound path
XY = np.array([[left, left, right, right], [bottom, top, top, bottom]]).T

# get the Path object
barpath = path.Path.make_compound_path_from_polys(XY)

# make a patch out of it, don't add a margin at y=0
patch = patches.PathPatch(barpath)
patch.sticky_edges.y[:] = [0]
```

(continues on next page)

(continued from previous page)

```
fig, ax = plt.subplots()
ax.add_patch(patch)
ax.autoscale_view()
plt.show()
```



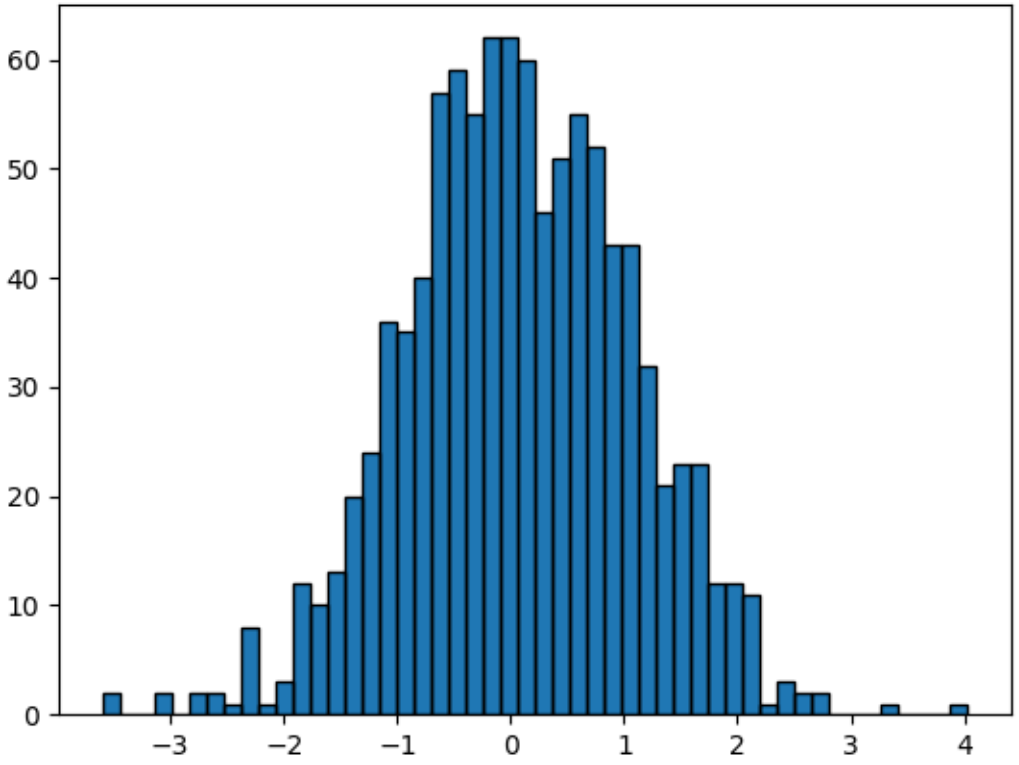

Instead of creating a three-dimensional array and using *make_compound_path_from_polys*, we could as well create the compound path directly using vertices and codes as shown below

```
nrects = len(left)
nverts = nrects*(1+3+1)
verts = np.zeros((nverts, 2))
codes = np.ones(nverts, int) * path.Path.LINETO
codes[0::5] = path.Path.MOVETO
codes[4::5] = path.Path.CLOSEPOLY
verts[0::5, 0] = left
verts[0::5, 1] = bottom
verts[1::5, 0] = left
verts[1::5, 1] = top
verts[2::5, 0] = right
verts[2::5, 1] = top
verts[3::5, 0] = right
```

(continues on next page)

(continued from previous page)

```
verts[3::5, 1] = bottom

barpath = path.Path(verts, codes)

# make a patch out of it, don't add a margin at y=0
patch = patches.PathPatch(barpath)
patch.sticky_edges.y[:] = [0]

fig, ax = plt.subplots()
ax.add_patch(patch)
ax.autoscale_view()
plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.patches*
- *matplotlib.patches.PathPatch*
- *matplotlib.path*

- *matplotlib.path.Path*
- *matplotlib.path.Path.make_compound_path_from_polys*
- *matplotlib.axes.Axes.add_patch*
- *matplotlib.collections.PathCollection*

This example shows an alternative to

- *matplotlib.collections.PolyCollection*
- *matplotlib.axes.Axes.hist*

### Hyperlinks

This example demonstrates how to set a hyperlinks on various kinds of elements.

This currently only works with the SVG backend.

```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib.cm as cm
```

```
fig = plt.figure()
s = plt.scatter([1, 2, 3], [4, 5, 6])
s.set_urls(['https://www.bbc.com/news', 'https://www.google.com/', None])
fig.savefig('scatter.svg')
```

```
fig = plt.figure()
delta = 0.025
x = y = np.arange(-3.0, 3.0, delta)
X, Y = np.meshgrid(x, y)
Z1 = np.exp(-X**2 - Y**2)
Z2 = np.exp(-(X - 1)**2 - (Y - 1)**2)
Z = (Z1 - Z2) * 2

im = plt.imshow(Z, interpolation='bilinear', cmap=cm.gray,
                origin='lower', extent=(-3, 3, -3, 3))

im.set_url('https://www.google.com/')
fig.savefig('image.svg')
```

### Image Thumbnail

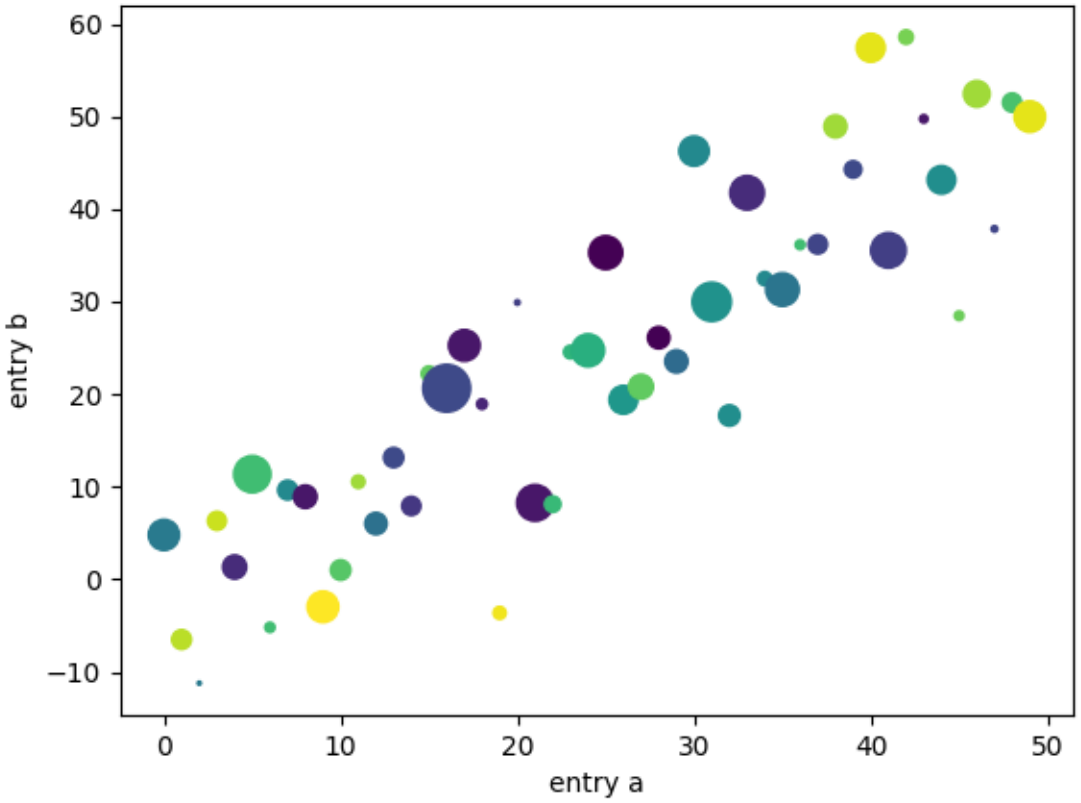
You can use Matplotlib to generate thumbnails from existing images. Matplotlib relies on Pillow for reading images, and thus supports all formats supported by Pillow.

```
from argparse import ArgumentParser
from pathlib import Path
import sys

import matplotlib.image as image

parser = ArgumentParser(
    description="Build thumbnails of all images in a directory.")
parser.add_argument("imagedir", type=Path)
args = parser.parse_args()
if not args.imagedir.is_dir():
    sys.exit(f"Could not find input directory {args.imagedir}")

outdir = Path("thumbs")
outdir.mkdir(parents=True, exist_ok=True)

for path in args.imagedir.glob("*.png"):
    outpath = outdir / path.name
    fig = image.thumbnail(path, outpath, scale=0.15)
    print(f"saved thumbnail of {path} to {outpath}")
```

### Plotting with keywords

Some data structures, like dict, structured numpy array or `pandas.DataFrame` provide access to labelled data via string index access `data[key]`.

For these data types, Matplotlib supports passing the whole datastructure via the `data` keyword argument, and using the string names as plot function parameters, where you'd normally pass in your data.




```
import matplotlib.pyplot as plt
import numpy as np

np.random.seed(19680801)

data = {'a': np.arange(50),
        'c': np.random.randint(0, 50, 50),
        'd': np.random.randn(50)}
data['b'] = data['a'] + 10 * np.random.randn(50)
data['d'] = np.abs(data['d']) * 100

fig, ax = plt.subplots()
ax.scatter('a', 'b', c='c', s='d', data=data)
ax.set(xlabel='entry a', ylabel='entry b')
plt.show()
```

### Matplotlib logo

This example generates the current matplotlib logo.

```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib.cm as cm
import matplotlib.font_manager
from matplotlib.patches import PathPatch, Rectangle
from matplotlib.text import TextPath
import matplotlib.transforms as mtrans

MPL_BLUE = '#11557c'


def get_font_properties():
    # The original font is Calibri, if that is not installed, we fall back
    # to Carlito, which is metrically equivalent.
    if 'Calibri' in matplotlib.font_manager.findfont('Calibri:bold'):
        return matplotlib.font_manager.FontProperties(family='Calibri',
                                                      weight='bold')
    if 'Carlito' in matplotlib.font_manager.findfont('Carlito:bold'):
        print('Original font not found. Falling back to Carlito. '
              'The logo text will not be in the correct font.')
        return matplotlib.font_manager.FontProperties(family='Carlito',
                                                      weight='bold')
    print('Original font not found. '
          'The logo text will not be in the correct font.')
    return None


def create_icon_axes(fig, ax_position, lw_bars, lw_grid, lw_border, rgrid):
    """
    Create a polar Axes containing the matplotlib radar plot.

    Parameters
    ----------
    fig : matplotlib.figure.Figure
        The figure to draw into.
    ax_position : (float, float, float, float)
        The position of the created Axes in figure coordinates as
        (x, y, width, height).
    lw_bars : float
        The linewidth of the bars.
    lw_grid : float
        The linewidth of the grid.
    lw_border : float
        The linewidth of the Axes border.
    rgrid : array-like
        Positions of the radial grid.

    Returns
```

(continues on next page)

(continued from previous page)

```
    -------
    ax : matplotlib.axes.Axes
        The created Axes.
    """
    with plt.rc_context({'axes.edgecolor': MPL_BLUE,
                         'axes.linewidth': lw_border}):
        ax = fig.add_axes(ax_position, projection='polar')
        ax.set_axisbelow(True)

        N = 7
        arc = 2. * np.pi
        theta = np.arange(0.0, arc, arc / N)
        radii = np.array([2, 6, 8, 7, 4, 5, 8])
        width = np.pi / 4 * np.array([0.4, 0.4, 0.6, 0.8, 0.2, 0.5, 0.3])
        bars = ax.bar(theta, radii, width=width, bottom=0.0, align='edge',
                      edgecolor='0.3', lw=lw_bars)
        for r, bar in zip(radii, bars):
            color = *cm.jet(r / 10.)[:3], 0.6  # color from jet with alpha=0.6
            bar.set_facecolor(color)

        ax.tick_params(labelbottom=False, labeltop=False,
                       labelleft=False, labelright=False)

        ax.grid(lw=lw_grid, color='0.9')
        ax.set_rmax(9)
        ax.set_yticks(rgrid)

        # the actual visible background - extends a bit beyond the axis
        ax.add_patch(Rectangle((0, 0), arc, 9.58,
                               facecolor='white', zorder=0,
                               clip_on=False, in_layout=False))
        return ax


def create_text_axes(fig, height_px):
    """Create an Axes in *fig* that contains 'matplotlib' as Text."""
    ax = fig.add_axes((0, 0, 1, 1))
    ax.set_aspect("equal")
    ax.set_axis_off()

    path = TextPath((0, 0), "matplotlib", size=height_px * 0.8,
                    prop=get_font_properties())

    angle = 4.25  # degrees
    trans = mtrans.Affine2D().skew_deg(angle, 0)

    patch = PathPatch(path, transform=trans + ax.transData, color=MPL_BLUE,
                      lw=0)
    ax.add_patch(patch)
    ax.autoscale()
```

(continues on next page)

(continued from previous page)

```
def make_logo(height_px, lw_bars, lw_grid, lw_border, rgrid, with_text=False):
    """
    Create a full figure with the Matplotlib logo.

    Parameters
    ----------
    height_px : int
        Height of the figure in pixel.
    lw_bars : float
        The linewidth of the bar border.
    lw_grid : float
        The linewidth of the grid.
    lw_border : float
        The linewidth of icon border.
    rgrid : sequence of float
        The radial grid positions.
    with_text : bool
        Whether to draw only the icon or to include 'matplotlib' as text.
    """
    dpi = 100
    height = height_px / dpi
    figsize = (5 * height, height) if with_text else (height, height)
    fig = plt.figure(figsize=figsize, dpi=dpi)
    fig.patch.set_alpha(0)

    if with_text:
        create_text_axes(fig, height_px)
    ax_pos = (0.535, 0.12, .17, 0.75) if with_text else (0.03, 0.03, .94, .94)
    ax = create_icon_axes(fig, ax_pos, lw_bars, lw_grid, lw_border, rgrid)

    return fig, ax
```

A large logo:

```
make_logo(height_px=110, lw_bars=0.7, lw_grid=0.5, lw_border=1,
          rgrid=[1, 3, 5, 7])
```

A small 32px logo:

```
make_logo(height_px=32, lw_bars=0.3, lw_grid=0.3, lw_border=0.3, rgrid=[5])
```

A large logo including text, as used on the matplotlib website.

```
make_logo(height_px=110, lw_bars=0.7, lw_grid=0.5, lw_border=1,
          rgrid=[1, 3, 5, 7], with_text=True)
plt.show()
```

### Multipage PDF

This is a demo of creating a pdf file with several pages, as well as adding metadata and annotations to pdf files.

If you want to use a multipage pdf file using LaTeX, you need to use `from matplotlib.backends.backend_pgf import PdfPages`. This version however does not support *attach_note*.

```
import datetime

import matplotlib.pyplot as plt
import numpy as np

from matplotlib.backends.backend_pdf import PdfPages

# Create the PdfPages object to which we will save the pages:
# The with statement makes sure that the PdfPages object is closed properly at
# the end of the block, even if an Exception occurs.
with PdfPages('multipage_pdf.pdf') as pdf:
    plt.figure(figsize=(3, 3))
    plt.plot(range(7), [3, 1, 4, 1, 5, 9, 2], 'r-o')
    plt.title('Page One')
    pdf.savefig()  # saves the current figure into a pdf page
    plt.close()

    # if LaTeX is not installed or error caught, change to `False`
    plt.rcParams['text.usetex'] = True
    plt.figure(figsize=(8, 6))
    x = np.arange(0, 5, 0.1)
    plt.plot(x, np.sin(x), 'b-')
    plt.title('Page Two')
    pdf.attach_note("plot of sin(x)")  # attach metadata (as pdf note) to page
    pdf.savefig()
    plt.close()

    plt.rcParams['text.usetex'] = False
    fig = plt.figure(figsize=(4, 5))
    plt.plot(x, x ** 2, 'ko')
    plt.title('Page Three')
```

(continues on next page)

(continued from previous page)

```
    pdf.savefig(fig)  # or you can pass a Figure object to pdf.savefig
    plt.close()

    # We can also set the file's metadata via the PdfPages object:
    d = pdf.infodict()
    d['Title'] = 'Multipage PDF Example'
    d['Author'] = 'Jouni K. Sepp\xe4nen'
    d['Subject'] = 'How to create a multipage pdf file and set its metadata'
    d['Keywords'] = 'PdfPages multipage keywords author title subject'
    d['CreationDate'] = datetime.datetime(2009, 11, 13)
    d['ModDate'] = datetime.datetime.today()
```

### Multiprocessing

Demo of using multiprocessing for generating data in one process and plotting in another.

Written by Robert Cimrman

```
import multiprocessing as mp
import time

import matplotlib.pyplot as plt
import numpy as np

# Fixing random state for reproducibility
np.random.seed(19680801)
```

### Processing Class

This class plots data it receives from a pipe.

```
class ProcessPlotter:
    def __init__(self):
        self.x = []
        self.y = []

    def terminate(self):
        plt.close('all')

    def call_back(self):
        while self.pipe.poll():
            command = self.pipe.recv()
            if command is None:
                self.terminate()
                return False
            else:
                self.x.append(command[0])
                self.y.append(command[1])
```

(continues on next page)

(continued from previous page)

```
                self.ax.plot(self.x, self.y, 'ro')
        self.fig.canvas.draw()
        return True

    def __call__(self, pipe):
        print('starting plotter...')

        self.pipe = pipe
        self.fig, self.ax = plt.subplots()
        timer = self.fig.canvas.new_timer(interval=1000)
        timer.add_callback(self.call_back)
        timer.start()

        print('...done')
        plt.show()
```

### Plotting class

This class uses multiprocessing to spawn a process to run code from the class above. When initialized, it creates a pipe and an instance of `ProcessPlotter` which will be run in a separate process.

When run from the command line, the parent process sends data to the spawned process which is then plotted via the callback function specified in `ProcessPlotter:__call__`.

```
class NBPlot:
    def __init__(self):
        self.plot_pipe, plotter_pipe = mp.Pipe()
        self.plotter = ProcessPlotter()
        self.plot_process = mp.Process(
            target=self.plotter, args=(plotter_pipe,), daemon=True)
        self.plot_process.start()

    def plot(self, finished=False):
        send = self.plot_pipe.send
        if finished:
            send(None)
        else:
            data = np.random.random(2)
            send(data)


def main():
    pl = NBPlot()
    for _ in range(10):
        pl.plot()
        time.sleep(0.5)
    pl.plot(finished=True)


if __name__ == '__main__':
```

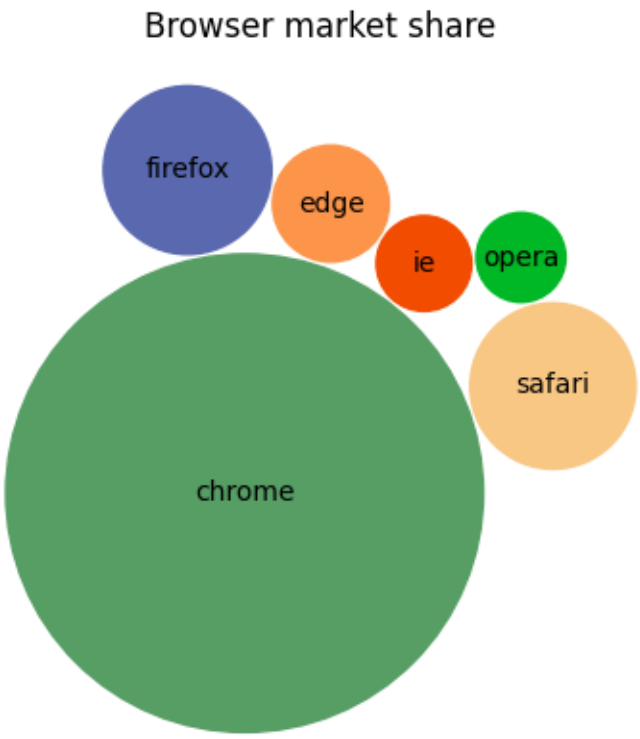
(continues on next page)

(continued from previous page)

```
    if plt.get_backend() == "MacOSX":
        mp.set_start_method("forkserver")
    main()
```

### Packed-bubble chart

Create a packed-bubble chart to represent scalar data. The presented algorithm tries to move all bubbles as close to the center of mass as possible while avoiding some collisions by moving around colliding objects. In this example we plot the market share of different desktop browsers. (source: https://gs.statcounter.com/browser-market-share/desktop/worldwidev)




```
import matplotlib.pyplot as plt
import numpy as np

browser_market_share = {
    'browsers': ['firefox', 'chrome', 'safari', 'edge', 'ie', 'opera'],
    'market_share': [8.61, 69.55, 8.36, 4.12, 2.76, 2.43],
    'color': ['#5A69AF', '#579E65', '#F9C784', '#FC944A', '#F24C00', '#00B825
 ↪']
}
```

(continues on next page)

(continued from previous page)

```
class BubbleChart:
    def __init__(self, area, bubble_spacing=0):
        """
        Setup for bubble collapse.

        Parameters
        ----------
        area : array-like
            Area of the bubbles.
        bubble_spacing : float, default: 0
            Minimal spacing between bubbles after collapsing.

        Notes
        -----
        If "area" is sorted, the results might look weird.
        """
        area = np.asarray(area)
        r = np.sqrt(area / np.pi)

        self.bubble_spacing = bubble_spacing
        self.bubbles = np.ones((len(area), 4))
        self.bubbles[:, 2] = r
        self.bubbles[:, 3] = area
        self.maxstep = 2 * self.bubbles[:, 2].max() + self.bubble_spacing
        self.step_dist = self.maxstep / 2

        # calculate initial grid layout for bubbles
        length = np.ceil(np.sqrt(len(self.bubbles)))
        grid = np.arange(length) * self.maxstep
        gx, gy = np.meshgrid(grid, grid)
        self.bubbles[:, 0] = gx.flatten()[:len(self.bubbles)]
        self.bubbles[:, 1] = gy.flatten()[:len(self.bubbles)]

        self.com = self.center_of_mass()

    def center_of_mass(self):
        return np.average(
            self.bubbles[:, :2], axis=0, weights=self.bubbles[:, 3]
        )

    def center_distance(self, bubble, bubbles):
        return np.hypot(bubble[0] - bubbles[:, 0],
                        bubble[1] - bubbles[:, 1])

    def outline_distance(self, bubble, bubbles):
        center_distance = self.center_distance(bubble, bubbles)
        return center_distance - bubble[2] - \
            bubbles[:, 2] - self.bubble_spacing

    def check_collisions(self, bubble, bubbles):
```

(continues on next page)

(continued from previous page)

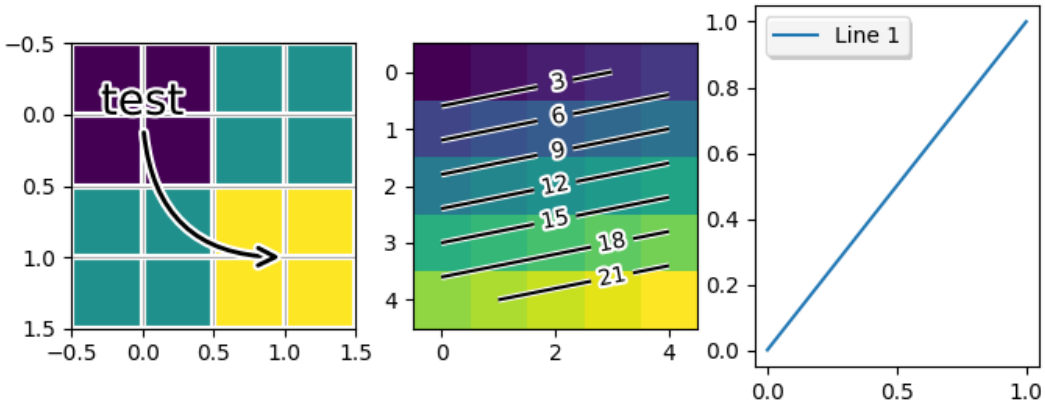
```
        distance = self.outline_distance(bubble, bubbles)
        return len(distance[distance < 0])

    def collides_with(self, bubble, bubbles):
        distance = self.outline_distance(bubble, bubbles)
        return np.argmin(distance, keepdims=True)

    def collapse(self, n_iterations=50):
        """
        Move bubbles to the center of mass.

        Parameters
        ----------
        n_iterations : int, default: 50
            Number of moves to perform.
        """
        for _i in range(n_iterations):
            moves = 0
            for i in range(len(self.bubbles)):
                rest_bub = np.delete(self.bubbles, i, 0)
                # try to move directly towards the center of mass
                # direction vector from bubble to the center of mass
                dir_vec = self.com - self.bubbles[i, :2]

                # shorten direction vector to have length of 1
                dir_vec = dir_vec / np.sqrt(dir_vec.dot(dir_vec))

                # calculate new bubble position
                new_point = self.bubbles[i, :2] + dir_vec * self.step_dist
                new_bubble = np.append(new_point, self.bubbles[i, 2:4])

                # check whether new bubble collides with other bubbles
                if not self.check_collisions(new_bubble, rest_bub):
                    self.bubbles[i, :] = new_bubble
                    self.com = self.center_of_mass()
                    moves += 1
                else:
                    # try to move around a bubble that you collide with
                    # find colliding bubble
                    for colliding in self.collides_with(new_bubble, rest_bub):
                        # calculate direction vector
                        dir_vec = rest_bub[colliding, :2] - self.bubbles[i,␣
↪:2]
                        dir_vec = dir_vec / np.sqrt(dir_vec.dot(dir_vec))
                        # calculate orthogonal vector
                        orth = np.array([dir_vec[1], -dir_vec[0]])
                        # test which direction to go
                        new_point1 = (self.bubbles[i, :2] + orth *
                                      self.step_dist)
                        new_point2 = (self.bubbles[i, :2] - orth *
                                      self.step_dist)
                        dist1 = self.center_distance(
```

(continues on next page)

(continued from previous page)

```
                            self.com, np.array([new_point1]))
                        dist2 = self.center_distance(
                            self.com, np.array([new_point2]))
                        new_point = new_point1 if dist1 < dist2 else new_
↪point2
                        new_bubble = np.append(new_point, self.bubbles[i,␣
↪2:4])
                        if not self.check_collisions(new_bubble, rest_bub):
                            self.bubbles[i, :] = new_bubble
                            self.com = self.center_of_mass()

            if moves / len(self.bubbles) < 0.1:
                self.step_dist = self.step_dist / 2

    def plot(self, ax, labels, colors):
        """
        Draw the bubble plot.

        Parameters
        ----------
        ax : matplotlib.axes.Axes
        labels : list
            Labels of the bubbles.
        colors : list
            Colors of the bubbles.
        """
        for i in range(len(self.bubbles)):
            circ = plt.Circle(
                self.bubbles[i, :2], self.bubbles[i, 2], color=colors[i])
            ax.add_patch(circ)
            ax.text(*self.bubbles[i, :2], labels[i],
                    horizontalalignment='center', verticalalignment='center')


bubble_chart = BubbleChart(area=browser_market_share['market_share'],
                           bubble_spacing=0.1)

bubble_chart.collapse()

fig, ax = plt.subplots(subplot_kw=dict(aspect="equal"))
bubble_chart.plot(
    ax, browser_market_share['browsers'], browser_market_share['color'])
ax.axis("off")
ax.relim()
ax.autoscale_view()
ax.set_title('Browser market share')

plt.show()
```

### Patheffect Demo




```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib import patheffects

fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(8, 3))
ax1.imshow([[1, 2], [2, 3]])
txt = ax1.annotate("test", (1., 1.), (0., 0),
                   arrowprops=dict(arrowstyle="->",
                                   connectionstyle="angle3", lw=2),
                   size=20, ha="center",
                   path_effects=[patheffects.withStroke(linewidth=3,
                                                        foreground="w")])
txt.arrow_patch.set_path_effects([
    patheffects.Stroke(linewidth=5, foreground="w"),
    patheffects.Normal()])

pe = [patheffects.withStroke(linewidth=3,
                             foreground="w")]
ax1.grid(True, linestyle="-", path_effects=pe)

arr = np.arange(25).reshape((5, 5))
ax2.imshow(arr)
cntr = ax2.contour(arr, colors="k")

cntr.set(path_effects=[patheffects.withStroke(linewidth=3, foreground="w")])

clbls = ax2.clabel(cntr, fmt="%2.0f", use_clabeltext=True)
plt.setp(clbls, path_effects=[
    patheffects.withStroke(linewidth=3, foreground="w")])

# shadow as a path effect
p1, = ax3.plot([0, 1], [0, 1])
leg = ax3.legend([p1], ["Line 1"], fancybox=True, loc='upper left')
leg.legendPatch.set_path_effects([patheffects.withSimplePatchShadow()])
```

(continues on next page)

(continued from previous page)

```
plt.show()
```

### Print Stdout

print png to standard out

usage: python print_stdout.py > somefile.png

```
import sys

import matplotlib

matplotlib.use('Agg')
import matplotlib.pyplot as plt

plt.plot([1, 2, 3])
plt.savefig(sys.stdout.buffer)
```

### Rasterization for vector graphics

Rasterization converts vector graphics into a raster image (pixels). It can speed up rendering and produce smaller files for large data sets, but comes at the cost of a fixed resolution.

Whether rasterization should be used can be specified per artist. This can be useful to reduce the file size of large artists, while maintaining the advantages of vector graphics for other artists such as the Axes and text. For instance a complicated *pcolormesh* or *contourf* can be made significantly simpler by rasterizing. Setting rasterization only affects vector backends such as PDF, SVG, or PS.

Rasterization is disabled by default. There are two ways to enable it, which can also be combined:

- Set *set_rasterized* on individual artists, or use the keyword argument *rasterized* when creating the artist.
- Set *Axes.set_rasterization_zorder* to rasterize all artists with a zorder less than the given value.

The storage size and the resolution of the rasterized artist is determined by its physical size and the value of the `dpi` parameter passed to *savefig*.

---

**Note:** The image of this example shown in the HTML documentation is not a vector graphic. Therefore, it cannot illustrate the rasterization effect. Please run this example locally and check the generated graphics files.

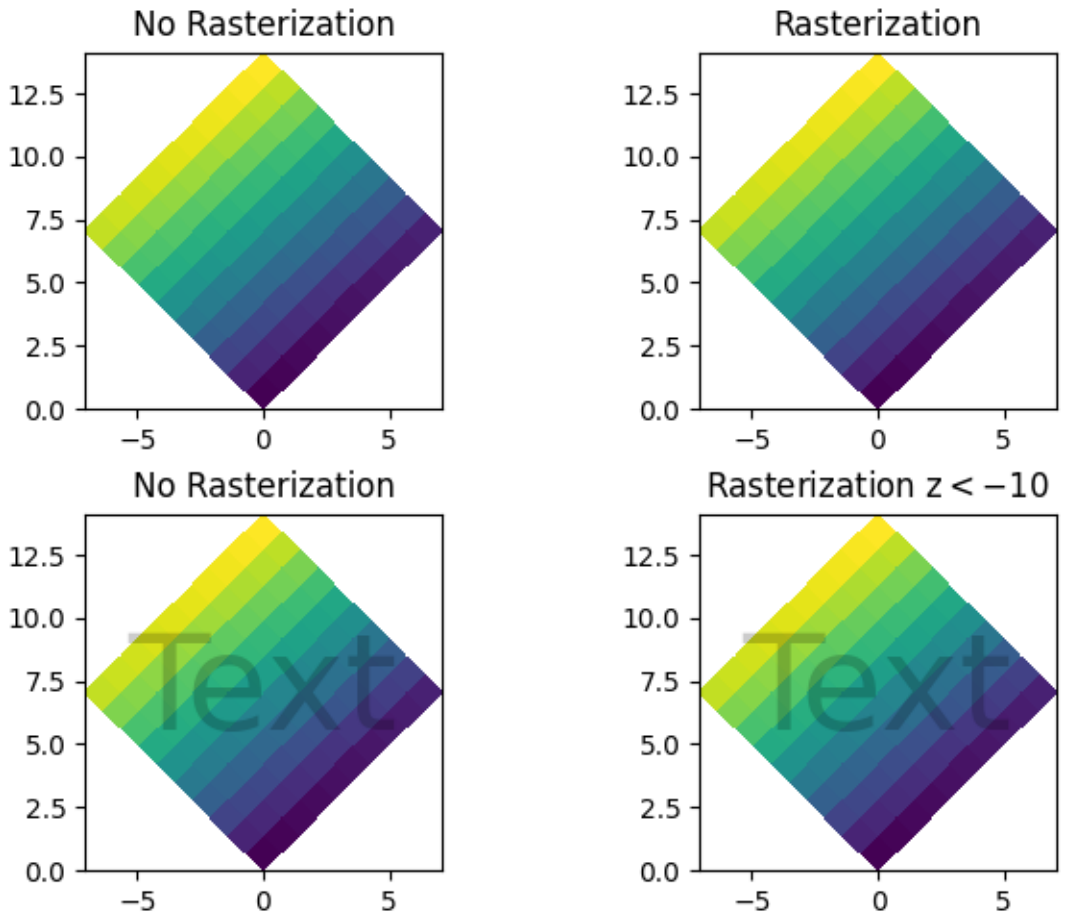
---

```
import matplotlib.pyplot as plt
import numpy as np
```

(continues on next page)

(continued from previous page)

```
d = np.arange(100).reshape(10, 10)  # the values to be color-mapped
x, y = np.meshgrid(np.arange(11), np.arange(11))

theta = 0.25*np.pi
xx = x*np.cos(theta) - y*np.sin(theta)  # rotate x by -theta
yy = x*np.sin(theta) + y*np.cos(theta)  # rotate y by -theta

fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, layout="constrained")

# pcolormesh without rasterization
ax1.set_aspect(1)
ax1.pcolormesh(xx, yy, d)
ax1.set_title("No Rasterization")

# pcolormesh with rasterization; enabled by keyword argument
ax2.set_aspect(1)
ax2.set_title("Rasterization")
ax2.pcolormesh(xx, yy, d, rasterized=True)

# pcolormesh with an overlaid text without rasterization
ax3.set_aspect(1)
ax3.pcolormesh(xx, yy, d)
ax3.text(0.5, 0.5, "Text", alpha=0.2,
         va="center", ha="center", size=50, transform=ax3.transAxes)
ax3.set_title("No Rasterization")

# pcolormesh with an overlaid text without rasterization; enabled by zorder.
# Setting the rasterization zorder threshold to 0 and a negative zorder on the
# pcolormesh rasterizes it. All artists have a non-negative zorder by default,
# so they (e.g. the text here) are not affected.
ax4.set_aspect(1)
m = ax4.pcolormesh(xx, yy, d, zorder=-10)
ax4.text(0.5, 0.5, "Text", alpha=0.2,
         va="center", ha="center", size=50, transform=ax4.transAxes)
ax4.set_rasterization_zorder(0)
ax4.set_title("Rasterization z$<-10$")

# Save files in pdf and eps format
plt.savefig("test_rasterization.pdf", dpi=150)
plt.savefig("test_rasterization.eps", dpi=150)

if not plt.rcParams["text.usetex"]:
    plt.savefig("test_rasterization.svg", dpi=150)
    # svg backend currently ignores the dpi
```




```
The PostScript backend does not support transparency; partially transparent␣
↪artists will be rendered opaque.
```

**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.artist.Artist.set_rasterized*
- *matplotlib.axes.Axes.set_rasterization_zorder*
- *matplotlib.axes.Axes.pcolormesh* / *matplotlib.pyplot.pcolormesh*

**Total running time of the script:** (0 minutes 2.279 seconds)

### Set and get properties

The pyplot interface allows you to use `setp` and `getp` to set and get object properties respectively, as well as to do introspection on the object.

### Setting with `setp`

To set the linestyle of a line to be dashed, you use `setp`:

```
>>> line, = plt.plot([1, 2, 3])
>>> plt.setp(line, linestyle='--')
```

If you want to know the valid types of arguments, you can provide the name of the property you want to set without a value:

```
>>> plt.setp(line, 'linestyle')
    linestyle: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
```

If you want to see all the properties that can be set, and their possible values, you can do:

```
>>> plt.setp(line)
```

`setp` operates on a single instance or a list of instances. If you are in query mode introspecting the possible values, only the first instance in the sequence is used. When actually setting values, all the instances will be set. For example, suppose you have a list of two lines, the following will make both lines thicker and red:

```
>>> x = np.arange(0, 1, 0.01)
>>> y1 = np.sin(2*np.pi*x)
>>> y2 = np.sin(4*np.pi*x)
>>> lines = plt.plot(x, y1, x, y2)
>>> plt.setp(lines, linewidth=2, color='r')
```

### Getting with `getp`

`getp` returns the value of a given attribute. You can use it to query the value of a single attribute:

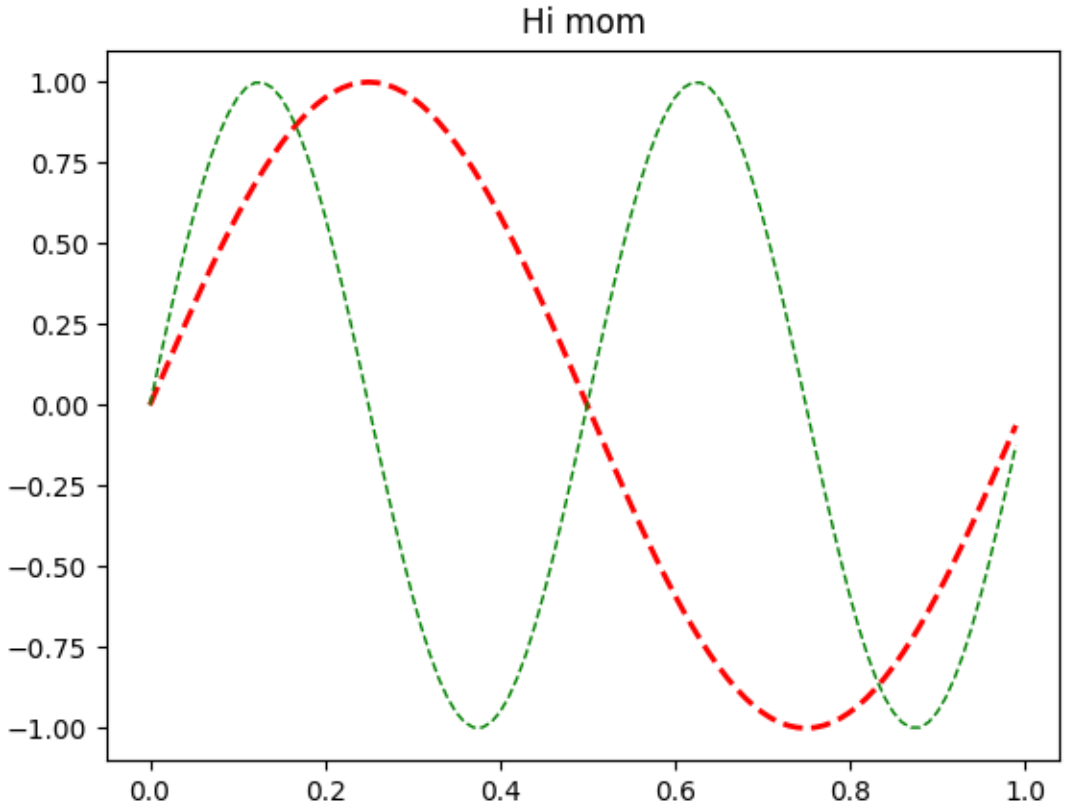
```
>>> plt.getp(line, 'linewidth')
    0.5
```

or all the attribute/value pairs:

```
>>> plt.getp(line)
    aa = True
    alpha = 1.0
    antialiased = True
    c = b
    clip_on = True
    color = b
    ... long listing skipped ...
```

### Aliases

To reduce keystrokes in interactive mode, a number of properties have short aliases, e.g., 'lw' for 'linewidth' and 'mec' for 'markeredgecolor'. When calling set or get in introspection mode, these properties will be listed as 'fullname' or 'aliasname'.




```
Line setters
  agg_filter: a filter function, which takes a (m, n, 3) float array and a␣
 ↪dpi value, and returns a (m, n, 3) array and two offsets from the bottom␣
 ↪left corner of the image
  alpha: scalar or None
  animated: bool
  antialiased or aa: bool
  clip_box: `~matplotlib.transforms.BboxBase` or None
  clip_on: bool
  clip_path: Patch or (Path, Transform) or None
  color or c: :mpltype:`color`
  dash_capstyle: `.CapStyle` or {'butt', 'projecting', 'round'}
  dash_joinstyle: `.JoinStyle` or {'miter', 'round', 'bevel'}
  dashes: sequence of floats (on/off ink in points) or (None, None)
  data: (2, N) array or two 1D arrays
  drawstyle or ds: {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post
 ↪'}, default: 'default'
```

(continues on next page)

(continued from previous page)

```
  figure: `~matplotlib.figure.Figure`
  fillstyle: {'full', 'left', 'right', 'bottom', 'top', 'none'}
  gapcolor: :mpltype:`color` or None
  gid: str
  in_layout: bool
  label: object
  linestyle or ls: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
  linewidth or lw: float
  marker: marker style string, `~.path.Path` or `~.markers.MarkerStyle`
  markeredgecolor or mec: :mpltype:`color`
  markeredgewidth or mew: float
  markerfacecolor or mfc: :mpltype:`color`
  markerfacecoloralt or mfcalt: :mpltype:`color`
  markersize or ms: float
  markevery: None or int or (int, int) or slice or list[int] or float or␣
↪(float, float) or list[bool]
  mouseover: bool
  path_effects: list of `.AbstractPathEffect`
  picker: float or callable[[Artist, Event], tuple[bool, dict]]
  pickradius: float
  rasterized: bool
  sketch_params: (scale: float, length: float, randomness: float)
  snap: bool or None
  solid_capstyle: `.CapStyle` or {'butt', 'projecting', 'round'}
  solid_joinstyle: `.JoinStyle` or {'miter', 'round', 'bevel'}
  transform: `~matplotlib.transforms.Transform`
  url: str
  visible: bool
  xdata: 1D array
  ydata: 1D array
  zorder: float
Line getters
    agg_filter = None
    alpha = None
    animated = False
    antialiased or aa = True
    bbox = Bbox(x0=0.0, y0=-1.0, x1=0.99, y1=1.0)
    children = []
    clip_box = TransformedBbox(     Bbox(x0=0.0, y0=0.0, x1=1.0, ...
    clip_on = True
    clip_path = None
    color or c = r
    dash_capstyle = butt
    dash_joinstyle = round
    data = (array([0.  , 0.01, 0.02, 0.03, 0.04, 0.05, 0.06, ...
    drawstyle or ds = default
    figure = Figure(640x480)
    fillstyle = full
    gapcolor = None
    gid = None
    in_layout = True
    label = _child0
```

(continues on next page)

(continued from previous page)

```
    linestyle or ls = --
    linewidth or lw = 2.0
    marker = None
    markeredgecolor or mec = r
    markeredgewidth or mew = 1.0
    markerfacecolor or mfc = r
    markerfacecoloralt or mfcalt = none
    markersize or ms = 6.0
    markevery = None
    mouseover = False
    path = Path(array([[ 0.00000000e+00,  0.00000000e+00],   ...
    path_effects = []
    picker = None
    pickradius = 5
    rasterized = False
    sketch_params = None
    snap = None
    solid_capstyle = projecting
    solid_joinstyle = round
    tightbbox = Bbox(x0=80.0, y0=52.8, x1=571.04, y1=422.4)
    transform = CompositeGenericTransform(     TransformWrapper(  ...
    transformed_clip_path_and_affine = (None, None)
    url = None
    visible = True
    window_extent = Bbox(x0=80.0, y0=-316.79999999999995, x1=571.04, y...
    xdata = [0.   0.01 0.02 0.03 0.04 0.05]...
    xydata = [[0.         0.        ]  [0.01       0.06279052] ...
    ydata = [0.         0.06279052 0.12533323 0.18738131 0.248...
    zorder = 2
Rectangle setters
  agg_filter: a filter function, which takes a (m, n, 3) float array and a␣
 ↪dpi value, and returns a (m, n, 3) array and two offsets from the bottom␣
 ↪left corner of the image
  alpha: scalar or None
  angle: unknown
  animated: bool
  antialiased or aa: bool or None
  bounds: (left, bottom, width, height)
  capstyle: `.CapStyle` or {'butt', 'projecting', 'round'}
  clip_box: `~matplotlib.transforms.BboxBase` or None
  clip_on: bool
  clip_path: Patch or (Path, Transform) or None
  color: :mpltype:`color`
  edgecolor or ec: :mpltype:`color` or None
  facecolor or fc: :mpltype:`color` or None
  figure: `~matplotlib.figure.Figure`
  fill: bool
  gid: str
  hatch: {'/', '\\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
  height: unknown
  in_layout: bool
  joinstyle: `.JoinStyle` or {'miter', 'round', 'bevel'}
```

(continues on next page)

(continued from previous page)

```
  label: object
  linestyle or ls: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
  linewidth or lw: float or None
  mouseover: bool
  path_effects: list of `.AbstractPathEffect`
  picker: None or bool or float or callable
  rasterized: bool
  sketch_params: (scale: float, length: float, randomness: float)
  snap: bool or None
  transform: `~matplotlib.transforms.Transform`
  url: str
  visible: bool
  width: unknown
  x: unknown
  xy: (float, float)
  y: unknown
  zorder: float
Rectangle getters
    agg_filter = None
    alpha = None
    angle = 0.0
    animated = False
    antialiased or aa = True
    bbox = Bbox(x0=0.0, y0=0.0, x1=1.0, y1=1.0)
    capstyle = butt
    center = [0.5 0.5]
    children = []
    clip_box = None
    clip_on = True
    clip_path = None
    corners = [[0. 0.]  [1. 0.]  [1. 1.]  [0. 1.]]
    data_transform = BboxTransformTo(     TransformedBbox(         Bbox...
    edgecolor or ec = (0.0, 0.0, 0.0, 0.0)
    extents = Bbox(x0=80.0, y0=52.8, x1=576.0, y1=422.4)
    facecolor or fc = (1.0, 1.0, 1.0, 1.0)
    figure = Figure(640x480)
    fill = True
    gid = None
    hatch = None
    height = 1.0
    in_layout = True
    joinstyle = miter
    label =
    linestyle or ls = solid
    linewidth or lw = 0.0
    mouseover = False
    patch_transform = CompositeGenericTransform(     BboxTransformTo(   ...
    path = Path(array([[0., 0.],        [1., 0.],        [1.,...
    path_effects = []
    picker = None
    rasterized = False
    sketch_params = None
```

(continues on next page)

(continued from previous page)

```
    snap = None
    tightbbox = Bbox(x0=80.0, y0=52.8, x1=576.0, y1=422.4)
    transform = CompositeGenericTransform(     CompositeGenericTra...
    transformed_clip_path_and_affine = (None, None)
    url = None
    verts = [[ 80.   52.8]  [576.   52.8]  [576.  422.4]  [ 80...
    visible = True
    width = 1.0
    window_extent = Bbox(x0=80.0, y0=52.8, x1=576.0, y1=422.4)
    x = 0.0
    xy = (0.0, 0.0)
    y = 0.0
    zorder = 1
Text setters
  agg_filter: a filter function, which takes a (m, n, 3) float array and a␣
 ↪dpi value, and returns a (m, n, 3) array and two offsets from the bottom␣
 ↪left corner of the image
  alpha: scalar or None
  animated: bool
  antialiased: bool
  backgroundcolor: :mpltype:`color`
  bbox: dict with properties for `.patches.FancyBboxPatch`
  clip_box: `~matplotlib.transforms.BboxBase` or None
  clip_on: bool
  clip_path: Patch or (Path, Transform) or None
  color or c: :mpltype:`color`
  figure: `~matplotlib.figure.Figure`
  fontfamily or family or fontname: {FONTNAME, 'serif', 'sans-serif', 'cursive
 ↪', 'fantasy', 'monospace'}
  fontproperties or font or font_properties: `.font_manager.FontProperties`␣
 ↪or `str` or `pathlib.Path`
  fontsize or size: float or {'xx-small', 'x-small', 'small', 'medium', 'large
 ↪', 'x-large', 'xx-large'}
  fontstretch or stretch: {a numeric value in range 0-1000, 'ultra-condensed',
 ↪ 'extra-condensed', 'condensed', 'semi-condensed', 'normal', 'semi-expanded
 ↪', 'expanded', 'extra-expanded', 'ultra-expanded'}
  fontstyle or style: {'normal', 'italic', 'oblique'}
  fontvariant or variant: {'normal', 'small-caps'}
  fontweight or weight: {a numeric value in range 0-1000, 'ultralight', 'light
 ↪', 'normal', 'regular', 'book', 'medium', 'roman', 'semibold', 'demibold',
 ↪'demi', 'bold', 'heavy', 'extra bold', 'black'}
  gid: str
  horizontalalignment or ha: {'left', 'center', 'right'}
  in_layout: bool
  label: object
  linespacing: float (multiple of font size)
  math_fontfamily: str
  mouseover: bool
  multialignment or ma: {'left', 'right', 'center'}
  parse_math: bool
  path_effects: list of `.AbstractPathEffect`
  picker: None or bool or float or callable
```

(continues on next page)

(continued from previous page)

```
  position: (float, float)
  rasterized: bool
  rotation: float or {'vertical', 'horizontal'}
  rotation_mode: {None, 'default', 'anchor'}
  sketch_params: (scale: float, length: float, randomness: float)
  snap: bool or None
  text: object
  transform: `~matplotlib.transforms.Transform`
  transform_rotates_text: bool
  url: str
  usetex: bool, default: :rc:`text.usetex`
  verticalalignment or va: {'baseline', 'bottom', 'center', 'center_baseline',
 ↪ 'top'}
  visible: bool
  wrap: bool
  x: float
  y: float
  zorder: float
Text getters
    agg_filter = None
    alpha = None
    animated = False
    antialiased = True
    bbox_patch = None
    children = []
    clip_box = None
    clip_on = True
    clip_path = None
    color or c = black
    figure = Figure(640x480)
    fontfamily or family or fontname = ['sans-serif']
    fontname or name = DejaVu Sans
    fontproperties or font or font_properties = sans\-
 ↪serif:style=normal:variant=normal:weight=nor...
    fontsize or size = 12.0
    fontstyle or style = normal
    fontvariant or variant = normal
    fontweight or weight = normal
    gid = None
    horizontalalignment or ha = center
    in_layout = True
    label =
    math_fontfamily = dejavusans
    mouseover = False
    parse_math = True
    path_effects = []
    picker = None
    position = (0.5, 1.0)
    rasterized = False
    rotation = 0.0
    rotation_mode = default
    sketch_params = None
```

(continues on next page)

(continued from previous page)

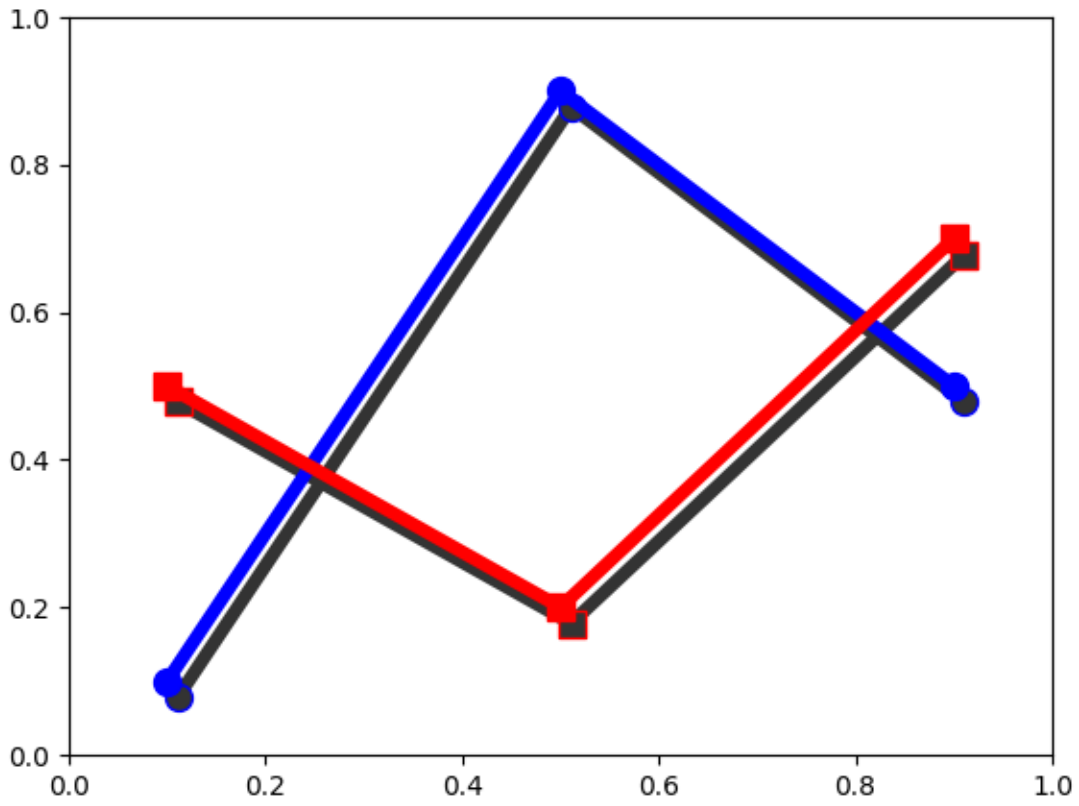
```
    snap = None
    stretch = normal
    text = Hi mom
    tightbbox = Bbox(x0=295.5, y0=426.7333333333333, x1=360.5, y1=...
    transform = CompositeGenericTransform(     BboxTransformTo(   ...
    transform_rotates_text = False
    transformed_clip_path_and_affine = (None, None)
    unitless_position = (0.5, 1.0)
    url = None
    usetex = False
    verticalalignment or va = baseline
    visible = True
    window_extent = Bbox(x0=295.5, y0=426.7333333333333, x1=360.5, y1=...
    wrap = False
    zorder = 3
```

```
import matplotlib.pyplot as plt
import numpy as np

x = np.arange(0, 1.0, 0.01)
y1 = np.sin(2*np.pi*x)
y2 = np.sin(4*np.pi*x)
lines = plt.plot(x, y1, x, y2)
l1, l2 = lines
plt.setp(lines, linestyle='--')       # set both to dashed
plt.setp(l1, linewidth=2, color='r')  # line1 is thick and red
plt.setp(l2, linewidth=1, color='g')  # line2 is thinner and green


print('Line setters')
plt.setp(l1)
print('Line getters')
plt.getp(l1)

print('Rectangle setters')
plt.setp(plt.gca().patch)
print('Rectangle getters')
plt.getp(plt.gca().patch)

t = plt.title('Hi mom')
print('Text setters')
plt.setp(t)
print('Text getters')
plt.getp(t)

plt.show()
```

### SVG Filter Line

Demonstrate SVG filtering effects which might be used with Matplotlib.

Note that the filtering effects are only effective if your SVG renderer support it.




```
Saving 'svg_filter_line.svg'
```

```
import io
import xml.etree.ElementTree as ET

import matplotlib.pyplot as plt

import matplotlib.transforms as mtransforms

fig1 = plt.figure()
ax = fig1.add_axes([0.1, 0.1, 0.8, 0.8])

```

(continues on next page)

(continued from previous page)

```
# draw lines
l1, = ax.plot([0.1, 0.5, 0.9], [0.1, 0.9, 0.5], "bo-",
              mec="b", lw=5, ms=10, label="Line 1")
l2, = ax.plot([0.1, 0.5, 0.9], [0.5, 0.2, 0.7], "rs-",
              mec="r", lw=5, ms=10, label="Line 2")


for l in [l1, l2]:

    # draw shadows with same lines with slight offset and gray colors.

    xx = l.get_xdata()
    yy = l.get_ydata()
    shadow, = ax.plot(xx, yy)
    shadow.update_from(l)

    # adjust color
    shadow.set_color("0.2")
    # adjust zorder of the shadow lines so that it is drawn below the
    # original lines
    shadow.set_zorder(l.get_zorder() - 0.5)

    # offset transform
    transform = mtransforms.offset_copy(l.get_transform(), fig1,
                                        x=4.0, y=-6.0, units='points')
    shadow.set_transform(transform)

    # set the id for a later use
    shadow.set_gid(l.get_label() + "_shadow")


ax.set_xlim(0., 1.)
ax.set_ylim(0., 1.)

# save the figure as a bytes string in the svg format.
f = io.BytesIO()
plt.savefig(f, format="svg")


# filter definition for a gaussian blur
filter_def = """
  <defs xmlns='http://www.w3.org/2000/svg'
        xmlns:xlink='http://www.w3.org/1999/xlink'>
    <filter id='dropshadow' height='1.2' width='1.2'>
      <feGaussianBlur result='blur' stdDeviation='3'/>
    </filter>
  </defs>
"""


# read in the saved svg
tree, xmlid = ET.XMLID(f.getvalue())
```

(continues on next page)

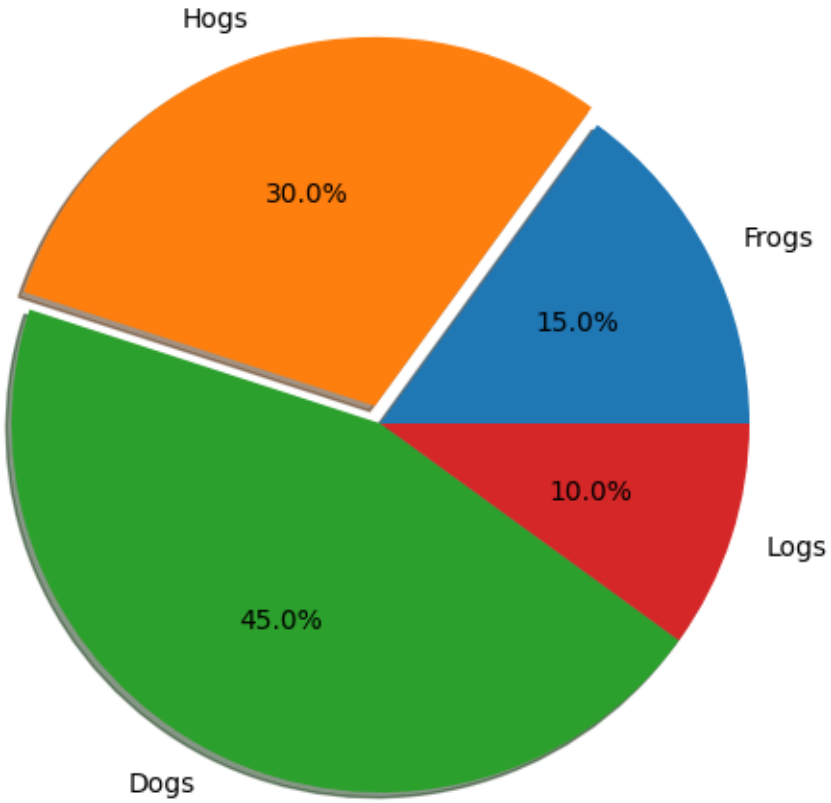
(continued from previous page)

```
# insert the filter definition in the svg dom tree.
tree.insert(0, ET.XML(filter_def))

for l in [l1, l2]:
    # pick up the svg element with given id
    shadow = xmlid[l.get_label() + "_shadow"]
    # apply shadow filter
    shadow.set("filter", 'url(#dropshadow)')

fn = "svg_filter_line.svg"
print(f"Saving '{fn}'")
ET.ElementTree(tree).write(fn)
```

### SVG filter pie

Demonstrate SVG filtering effects which might be used with Matplotlib. The pie chart drawing code is borrowed from pie_demo.py

Note that the filtering effects are only effective if your SVG renderer support it.




```
Saving 'svg_filter_pie.svg'
```

```
import io
import xml.etree.ElementTree as ET

import matplotlib.pyplot as plt

from matplotlib.patches import Shadow

# make a square figure and Axes
```

(continues on next page)

(continued from previous page)

```
fig = plt.figure(figsize=(6, 6))
ax = fig.add_axes([0.1, 0.1, 0.8, 0.8])

labels = 'Frogs', 'Hogs', 'Dogs', 'Logs'
fracs = [15, 30, 45, 10]

explode = (0, 0.05, 0, 0)

# We want to draw the shadow for each pie, but we will not use "shadow"
# option as it doesn't save the references to the shadow patches.
pies = ax.pie(fracs, explode=explode, labels=labels, autopct='%1.1f%%')

for w in pies[0]:
    # set the id with the label.
    w.set_gid(w.get_label())

    # we don't want to draw the edge of the pie
    w.set_edgecolor("none")

for w in pies[0]:
    # create shadow patch
    s = Shadow(w, -0.01, -0.01)
    s.set_gid(w.get_gid() + "_shadow")
    s.set_zorder(w.get_zorder() - 0.1)
    ax.add_patch(s)


# save
f = io.BytesIO()
plt.savefig(f, format="svg")


# Filter definition for shadow using a gaussian blur and lighting effect.
# The lighting filter is copied from http://www.w3.org/TR/SVG/filters.html

# I tested it with Inkscape and Firefox3. "Gaussian blur" is supported
# in both, but the lighting effect only in Inkscape. Also note
# that, Inkscape's exporting also may not support it.

filter_def = """
  <defs xmlns='http://www.w3.org/2000/svg'
        xmlns:xlink='http://www.w3.org/1999/xlink'>
    <filter id='dropshadow' height='1.2' width='1.2'>
      <feGaussianBlur result='blur' stdDeviation='2'/>
    </filter>

    <filter id='MyFilter' filterUnits='objectBoundingBox'
            x='0' y='0' width='1' height='1'>
      <feGaussianBlur in='SourceAlpha' stdDeviation='4%' result='blur'/>
      <feOffset in='blur' dx='4%' dy='4%' result='offsetBlur'/>
      <feSpecularLighting in='blur' surfaceScale='5' specularConstant='.75'
          specularExponent='20' lighting-color='#bbbbbb' result='specOut'>
```

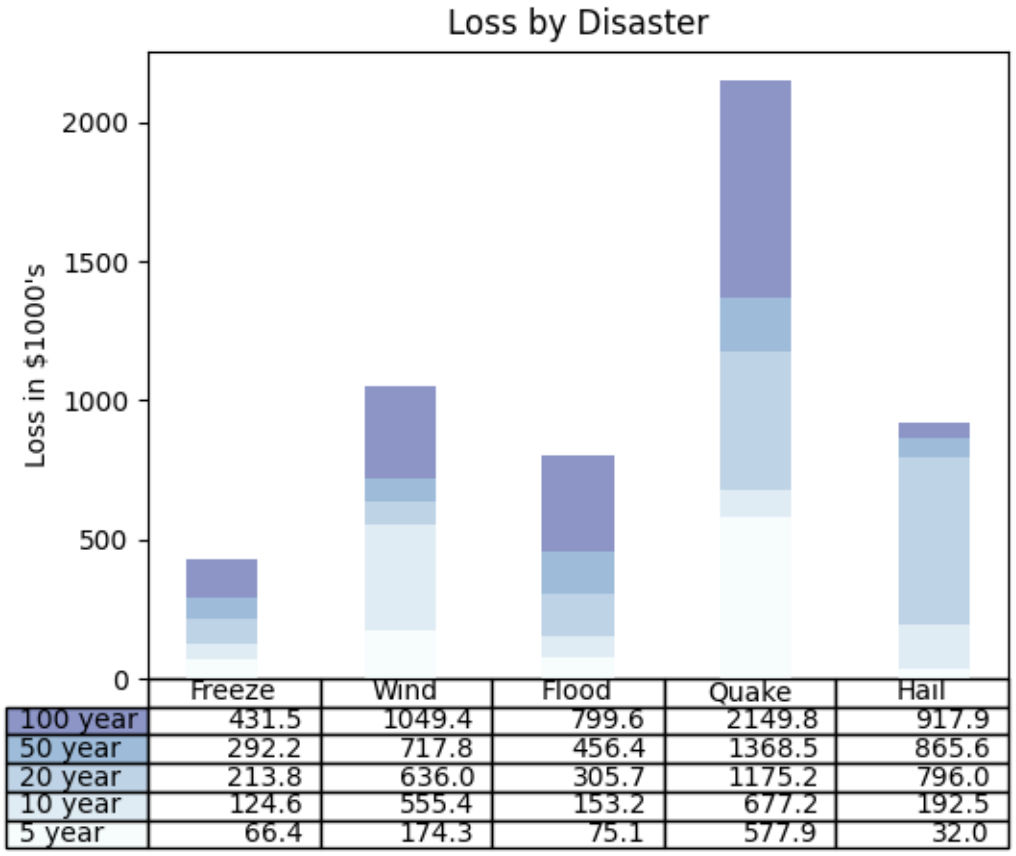
(continues on next page)

(continued from previous page)

```
        <fePointLight x='-5000%' y='-10000%' z='20000%'/>
      </feSpecularLighting>
      <feComposite in='specOut' in2='SourceAlpha'
                   operator='in' result='specOut'/>
      <feComposite in='SourceGraphic' in2='specOut' operator='arithmetic'
    k1='0' k2='1' k3='1' k4='0'/>
    </filter>
  </defs>
"""


tree, xmlid = ET.XMLID(f.getvalue())

# insert the filter definition in the svg dom tree.
tree.insert(0, ET.XML(filter_def))

for i, pie_name in enumerate(labels):
    pie = xmlid[pie_name]
    pie.set("filter", 'url(#MyFilter)')

    shadow = xmlid[pie_name + "_shadow"]
    shadow.set("filter", 'url(#dropshadow)')

fn = "svg_filter_pie.svg"
print(f"Saving '{fn}'")
ET.ElementTree(tree).write(fn)
```

### Table Demo

Demo of table function to display a table within a plot.




```
import matplotlib.pyplot as plt
import numpy as np

data = [[ 66386, 174296,  75131, 577908,  32015],
        [ 58230, 381139,  78045,  99308, 160454],
        [ 89135,  80552, 152558, 497981, 603535],
        [ 78415,  81858, 150656, 193263,  69638],
        [139361, 331509, 343164, 781380,  52269]]

columns = ('Freeze', 'Wind', 'Flood', 'Quake', 'Hail')
rows = ['%d year' % x for x in (100, 50, 20, 10, 5)]

values = np.arange(0, 2500, 500)
value_increment = 1000

# Get some pastel shades for the colors
colors = plt.cm.BuPu(np.linspace(0, 0.5, len(rows)))
n_rows = len(data)

index = np.arange(len(columns)) + 0.3
bar_width = 0.4

# Initialize the vertical-offset for the stacked bar chart.
```

(continues on next page)

(continued from previous page)

```
y_offset = np.zeros(len(columns))

# Plot bars and create text labels for the table
cell_text = []
for row in range(n_rows):
    plt.bar(index, data[row], bar_width, bottom=y_offset, color=colors[row])
    y_offset = y_offset + data[row]
    cell_text.append(['%1.1f' % (x / 1000.0) for x in y_offset])
# Reverse colors and text labels to display the last value at the top.
colors = colors[::-1]
cell_text.reverse()

# Add a table at the bottom of the Axes
the_table = plt.table(cellText=cell_text,
                      rowLabels=rows,
                      rowColours=colors,
                      colLabels=columns,
                      loc='bottom')

# Adjust layout to make room for the table:
plt.subplots_adjust(left=0.2, bottom=0.2)

plt.ylabel(f"Loss in ${value_increment}'s")
plt.yticks(values * value_increment, ['%d' % val for val in values])
plt.xticks([])
plt.title('Loss by Disaster')

plt.show()
```

### TickedStroke patheffect

Matplotlib's *patheffects* can be used to alter the way paths are drawn at a low enough level that they can affect almost anything.

The *patheffects guide* details the use of patheffects.

The *TickedStroke* patheffect illustrated here draws a path with a ticked style. The spacing, length, and angle of ticks can be controlled.

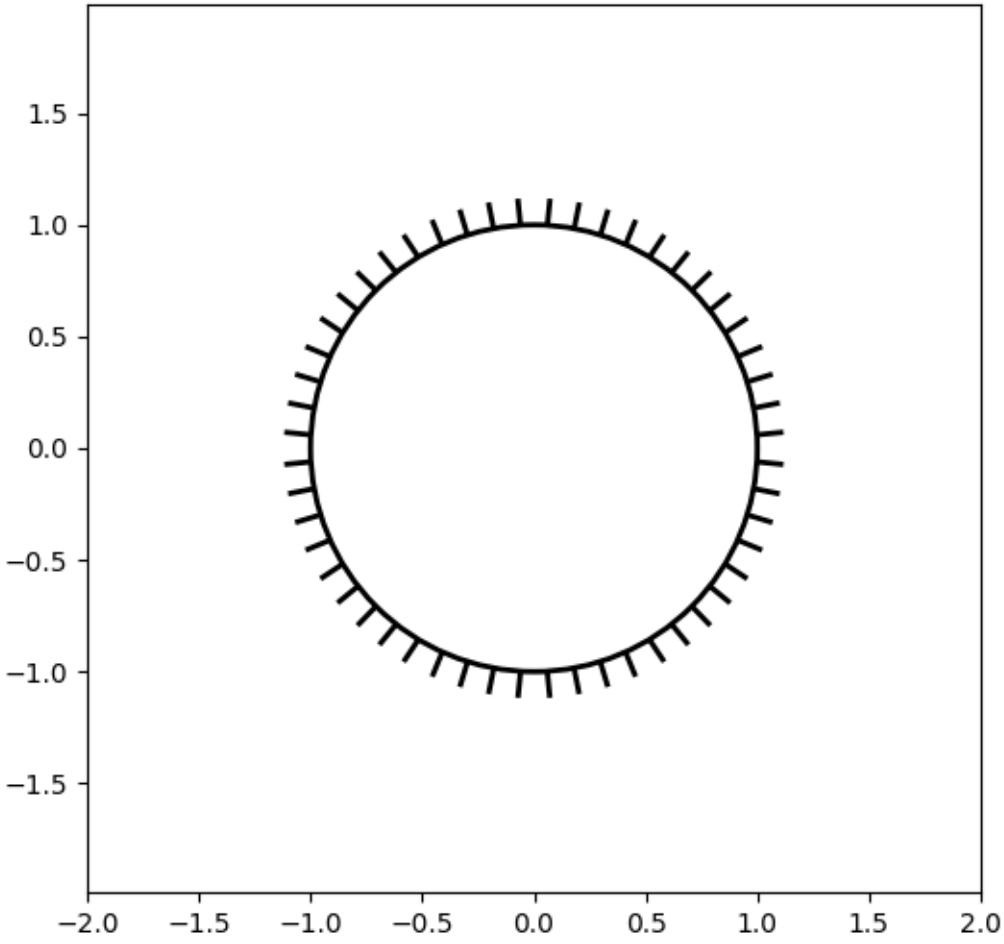
See also the *Lines with a ticked patheffect* example.

See also the *Contouring the solution space of optimizations* example.

```
import matplotlib.pyplot as plt
import numpy as np
```

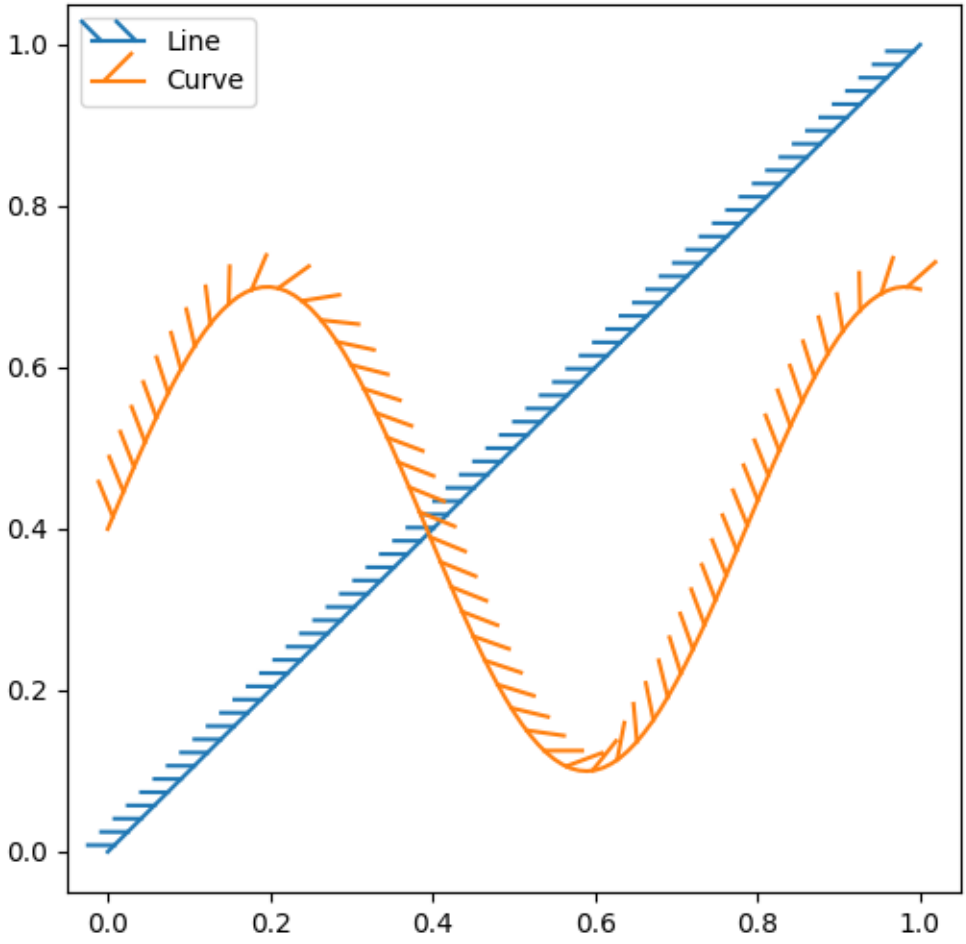
### Applying TickedStroke to paths

```
import matplotlib.patches as patches
from matplotlib.path import Path
import matplotlib.patheffects as patheffects

fig, ax = plt.subplots(figsize=(6, 6))
path = Path.unit_circle()
patch = patches.PathPatch(path, facecolor='none', lw=2, path_effects=[
    patheffects.withTickedStroke(angle=-90, spacing=10, length=1)])

ax.add_patch(patch)
ax.axis('equal')
ax.set_xlim(-2, 2)
ax.set_ylim(-2, 2)

plt.show()
```




### Applying TickedStroke to lines

```
fig, ax = plt.subplots(figsize=(6, 6))
ax.plot([0, 1], [0, 1], label="Line",
        path_effects=[patheffects.withTickedStroke(spacing=7, angle=135)])

nx = 101
x = np.linspace(0.0, 1.0, nx)
y = 0.3*np.sin(x*8) + 0.4
ax.plot(x, y, label="Curve", path_effects=[patheffects.withTickedStroke()])

ax.legend()

plt.show()
```




### Applying TickedStroke to contour plots

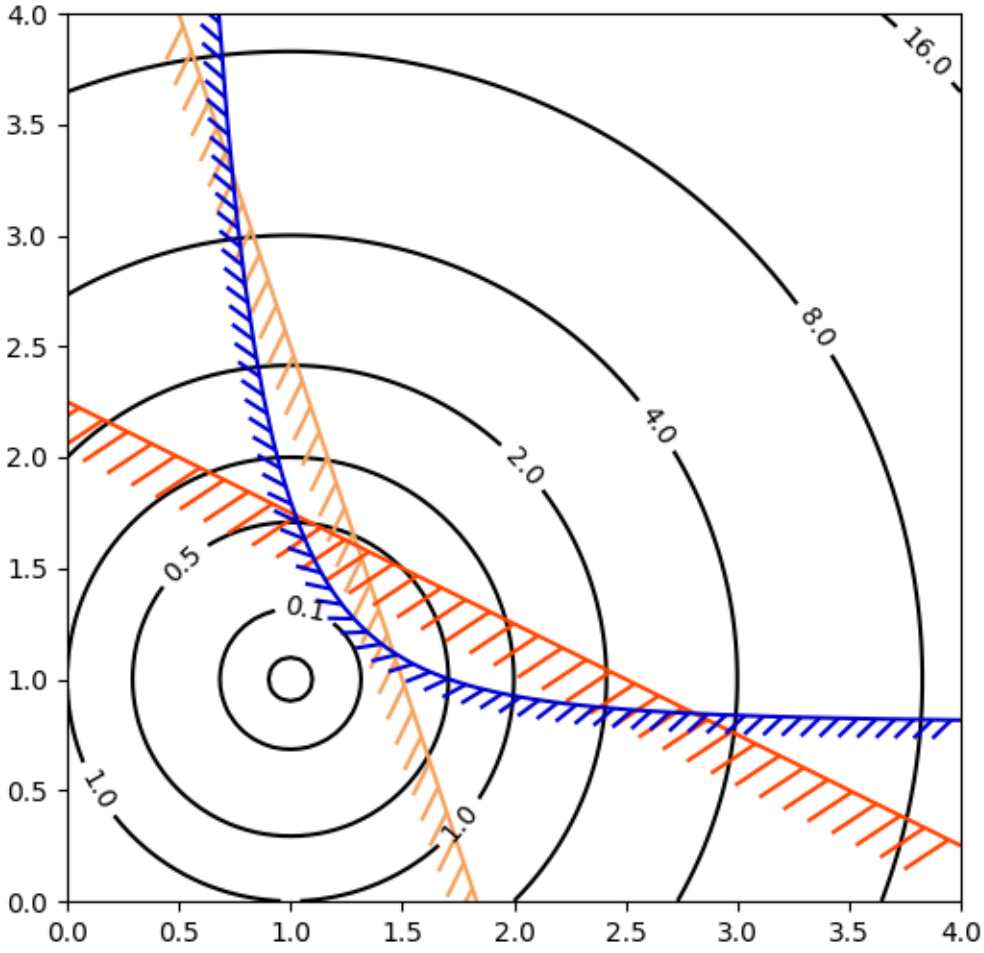
Contour plot with objective and constraints. Curves generated by contour to represent a typical constraint in an optimization problem should be plotted with angles between zero and 180 degrees.

```
fig, ax = plt.subplots(figsize=(6, 6))

nx = 101
ny = 105

# Set up survey vectors
xvec = np.linspace(0.001, 4.0, nx)
yvec = np.linspace(0.001, 4.0, ny)
```

(continues on next page)

(continued from previous page)

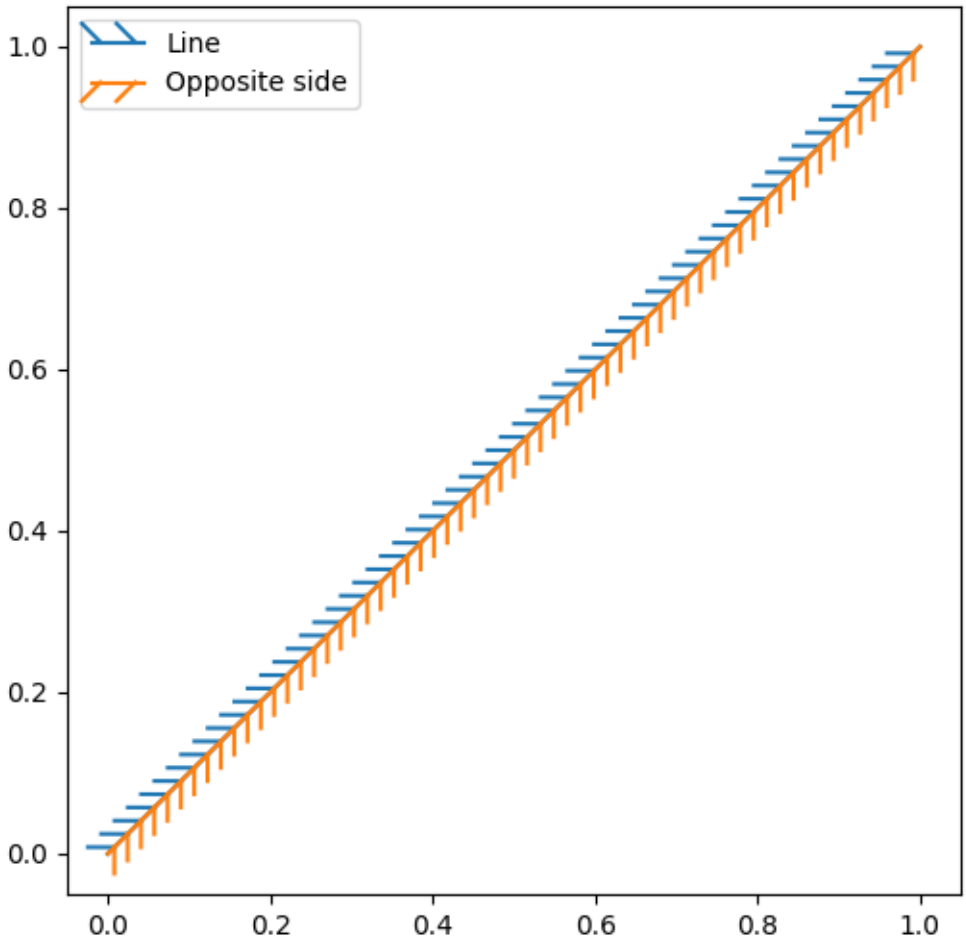
```
# Set up survey matrices.  Design disk loading and gear ratio.
x1, x2 = np.meshgrid(xvec, yvec)

# Evaluate some stuff to plot
obj = x1**2 + x2**2 - 2*x1 - 2*x2 + 2
g1 = -(3*x1 + x2 - 5.5)
g2 = -(x1 + 2*x2 - 4.5)
g3 = 0.8 + x1**-3 - x2

cntr = ax.contour(x1, x2, obj, [0.01, 0.1, 0.5, 1, 2, 4, 8, 16],
                  colors='black')
ax.clabel(cntr, fmt="%2.1f", use_clabeltext=True)

cg1 = ax.contour(x1, x2, g1, [0], colors='sandybrown')
cg1.set(path_effects=[patheffects.withTickedStroke(angle=135)])

cg2 = ax.contour(x1, x2, g2, [0], colors='orangered')
cg2.set(path_effects=[patheffects.withTickedStroke(angle=60, length=2)])

cg3 = ax.contour(x1, x2, g3, [0], colors='mediumblue')
cg3.set(path_effects=[patheffects.withTickedStroke(spacing=7)])

ax.set_xlim(0, 4)
ax.set_ylim(0, 4)

plt.show()
```




### Direction/side of the ticks

To change which side of the line the ticks are drawn, change the sign of the angle.

```
fig, ax = plt.subplots(figsize=(6, 6))
line_x = line_y = [0, 1]
ax.plot(line_x, line_y, label="Line",
        path_effects=[patheffects.withTickedStroke(spacing=7, angle=135)])

ax.plot(line_x, line_y, label="Opposite side",
        path_effects=[patheffects.withTickedStroke(spacing=7, angle=-135)])

ax.legend()
plt.show()
```

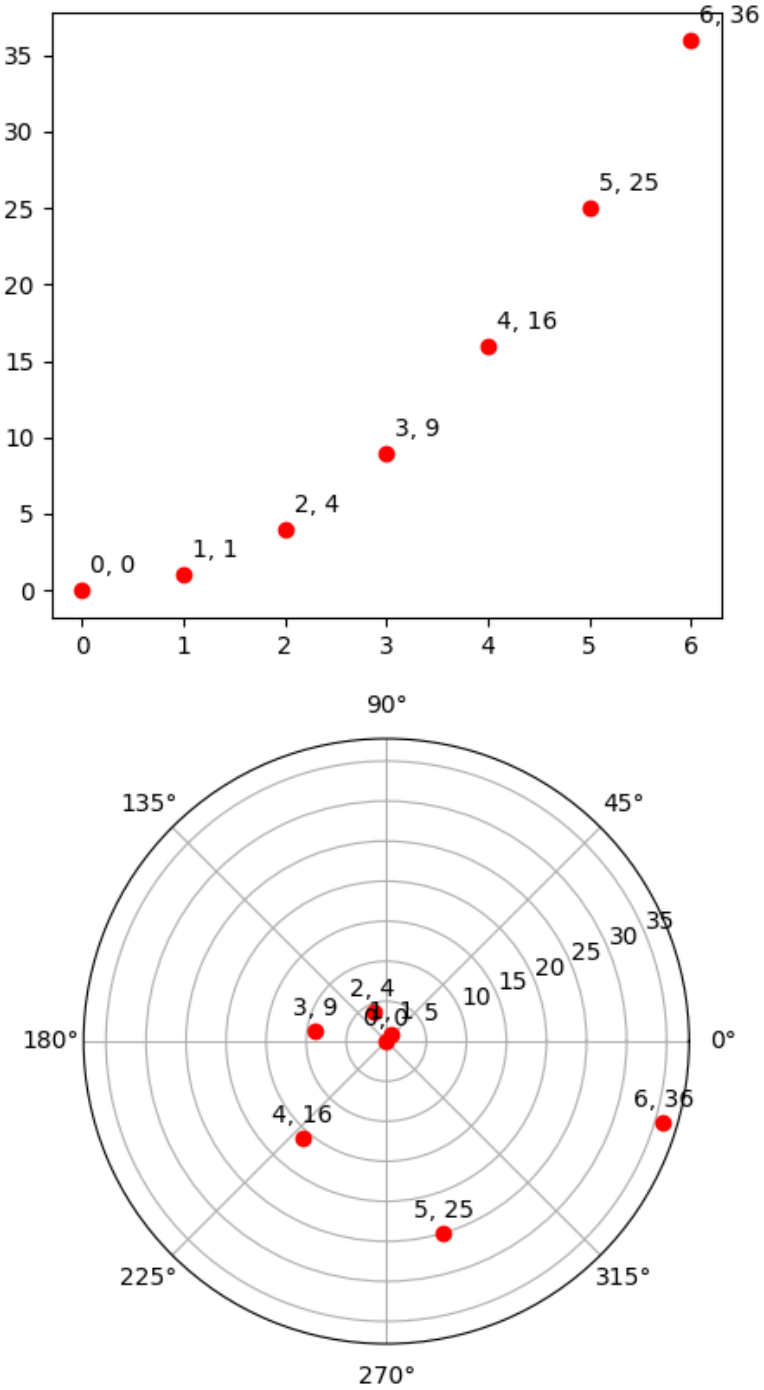



**Total running time of the script:** (0 minutes 1.575 seconds)

### transforms.offset_copy

This illustrates the use of *transforms.offset_copy* to make a transform that positions a drawing element such as a text string at a specified offset in screen coordinates (dots or inches) relative to a location given in any coordinates.

Every Artist (Text, Line2D, etc.) has a transform that can be set when the Artist is created, such as by the corresponding pyplot function. By default, this is usually the Axes.transData transform, going from data units to screen pixels. We can use the *offset_copy* function to make a modified copy of this transform, where the modification consists of an offset.


6, 36
5, 25
4, 16
3, 9
2, 4
1, 1
0, 0
0
5
10
15
20
25
30
35
0
1
2
3
4
5
6
90°
135°
45°
180°
0°
225°
315°
270°
2, 4
3, 9
4, 16
5, 25
6, 36

```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib.transforms as mtransforms

xs = np.arange(7)
ys = xs**2

fig = plt.figure(figsize=(5, 10))
ax = plt.subplot(2, 1, 1)

# If we want the same offset for each text instance,
# we only need to make one transform.  To get the
# transform argument to offset_copy, we need to make the Axes
# first; the subplot function above is one way to do this.
trans_offset = mtransforms.offset_copy(ax.transData, fig=fig,
                                       x=0.05, y=0.10, units='inches')

for x, y in zip(xs, ys):
    plt.plot(x, y, 'ro')
    plt.text(x, y, '%d, %d' % (int(x), int(y)), transform=trans_offset)


# offset_copy works for polar plots also.
ax = plt.subplot(2, 1, 2, projection='polar')

trans_offset = mtransforms.offset_copy(ax.transData, fig=fig,
                                       y=6, units='dots')

for x, y in zip(xs, ys):
    plt.polar(x, y, 'ro')
    plt.text(x, y, '%d, %d' % (int(x), int(y)),
             transform=trans_offset,
             horizontalalignment='center',
             verticalalignment='bottom')

plt.show()
```

### Zorder Demo

The drawing order of artists is determined by their `zorder` attribute, which is a floating point number. Artists with higher `zorder` are drawn on top. You can change the order for individual artists by setting their `zorder`. The default value depends on the type of the Artist:

| Artist | Z-order |
|---|---|
| Images (*AxesImage*, *FigureImage*, *BboxImage*) | 0 |
| *Patch*, *PatchCollection* | 1 |
| *Line2D*, *LineCollection* (including minor ticks, grid lines) | 2 |
| Major ticks | 2.01 |
| *Text* (including Axes labels and titles) | 3 |
| *Legend* | 5 |

Any call to a plotting method can set a value for the zorder of that particular item explicitly.

---

**Note:** *set_axisbelow* and *rcParams["axes.axisbelow"]* (default: `'line'`) are convenient helpers for setting the zorder of ticks and grid lines.

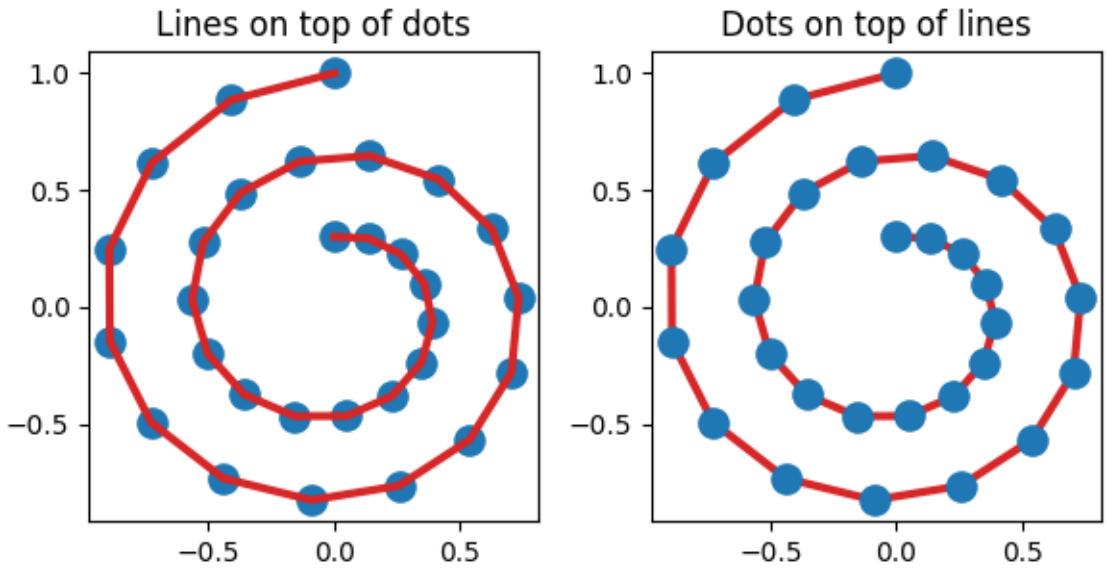
---

Drawing is done per *Axes* at a time. If you have overlapping Axes, all elements of the second Axes are drawn on top of the first Axes, irrespective of their relative zorder.

```
import matplotlib.pyplot as plt
import numpy as np

r = np.linspace(0.3, 1, 30)
theta = np.linspace(0, 4*np.pi, 30)
x = r * np.sin(theta)
y = r * np.cos(theta)
```

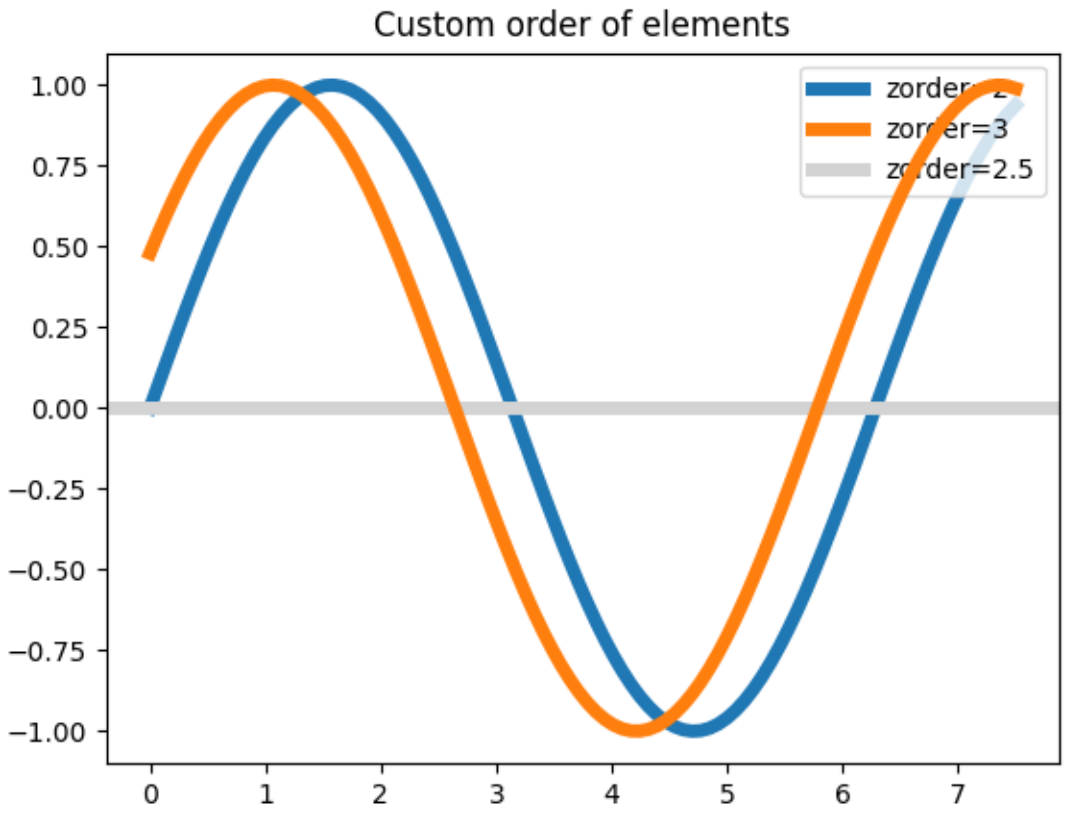
The following example contains a *Line2D* created by *plot()* and the dots (a *PatchCollection*) created by *scatter()*. Hence, by default the dots are below the line (first subplot). In the second subplot, the `zorder` is set explicitly to move the dots on top of the line.

```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(6, 3.2))

ax1.plot(x, y, 'C3', lw=3)
ax1.scatter(x, y, s=120)
ax1.set_title('Lines on top of dots')

ax2.plot(x, y, 'C3', lw=3)
ax2.scatter(x, y, s=120, zorder=2.5)  # move dots on top of line
ax2.set_title('Dots on top of lines')

plt.tight_layout()
```




Many functions that create a visible object accepts a `zorder` parameter. Alternatively, you can call `set_zorder()` on the created object later.

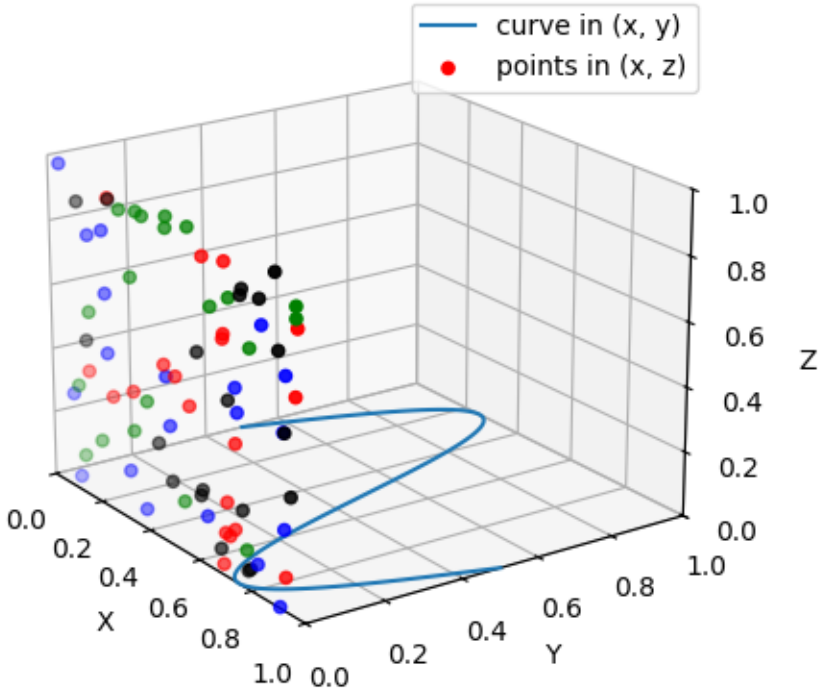
```
x = np.linspace(0, 7.5, 100)
plt.rcParams['lines.linewidth'] = 5
plt.figure()
plt.plot(x, np.sin(x), label='zorder=2', zorder=2)  # bottom
plt.plot(x, np.sin(x+0.5), label='zorder=3',  zorder=3)
plt.axhline(0, label='zorder=2.5', color='lightgrey', zorder=2.5)
plt.title('Custom order of elements')
l = plt.legend(loc='upper right')
l.set_zorder(2.5)  # legend between blue and orange line
plt.show()
```




### 6.25.17 3D plotting

#### Plot 2D data on 3D plot

Demonstrates using ax.plot's *zdir* keyword to plot 2D data on selective axes of a 3D plot.




```
import matplotlib.pyplot as plt
import numpy as np

ax = plt.figure().add_subplot(projection='3d')

# Plot a sin curve using the x and y axes.
x = np.linspace(0, 1, 100)
y = np.sin(x * 2 * np.pi) / 2 + 0.5
ax.plot(x, y, zs=0, zdir='z', label='curve in (x, y)')

# Plot scatterplot data (20 2D points per colour) on the x and z axes.
colors = ('r', 'g', 'b', 'k')

# Fixing random state for reproducibility
np.random.seed(19680801)

x = np.random.sample(20 * len(colors))
y = np.random.sample(20 * len(colors))
c_list = []
for c in colors:
    c_list.extend([c] * 20)
# By using zdir='y', the y value of these points is fixed to the zs value 0
# and the (x, y) points are plotted on the x and z axes.
```

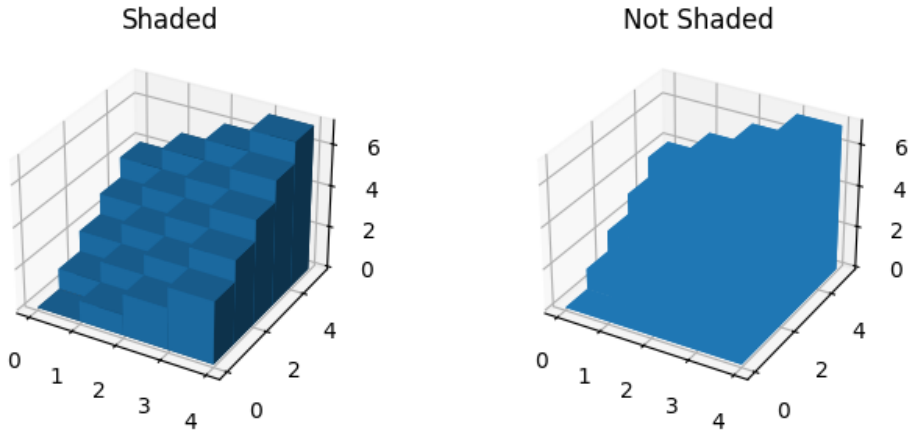
(continues on next page)

(continued from previous page)

```
ax.scatter(x, y, zs=0, zdir='y', c=c_list, label='points in (x, z)')

# Make legend, set axes limits and labels
ax.legend()
ax.set_xlim(0, 1)
ax.set_ylim(0, 1)
ax.set_zlim(0, 1)
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')

# Customize the view angle so it's easier to see that the scatter points lie
# on the plane y=0
ax.view_init(elev=20., azim=-35, roll=0)

plt.show()
```

## Demo of 3D bar charts

A basic demo of how to plot 3D bars with and without shading.




```
import matplotlib.pyplot as plt
import numpy as np

# set up the figure and Axes
fig = plt.figure(figsize=(8, 3))
ax1 = fig.add_subplot(121, projection='3d')
ax2 = fig.add_subplot(122, projection='3d')

# fake data
_x = np.arange(4)
_y = np.arange(5)
_xx, _yy = np.meshgrid(_x, _y)
x, y = _xx.ravel(), _yy.ravel()
```

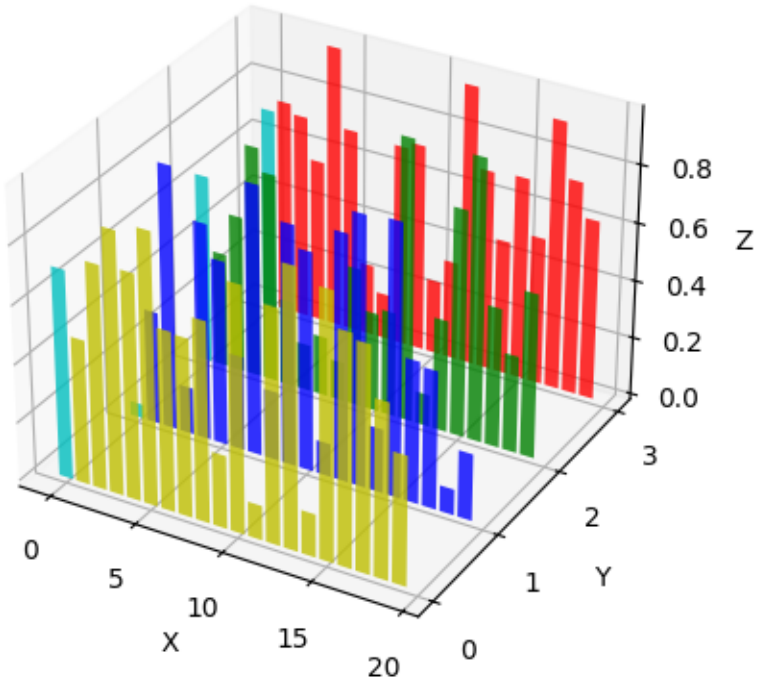
(continues on next page)

(continued from previous page)

```
top = x + y
bottom = np.zeros_like(top)
width = depth = 1

ax1.bar3d(x, y, bottom, width, depth, top, shade=True)
ax1.set_title('Shaded')

ax2.bar3d(x, y, bottom, width, depth, top, shade=False)
ax2.set_title('Not Shaded')

plt.show()
```

### Create 2D bar graphs in different planes

Demonstrates making a 3D plot which has 2D bar graphs projected onto planes y=0, y=1, etc.




```
import matplotlib.pyplot as plt
import numpy as np
```

(continues on next page)

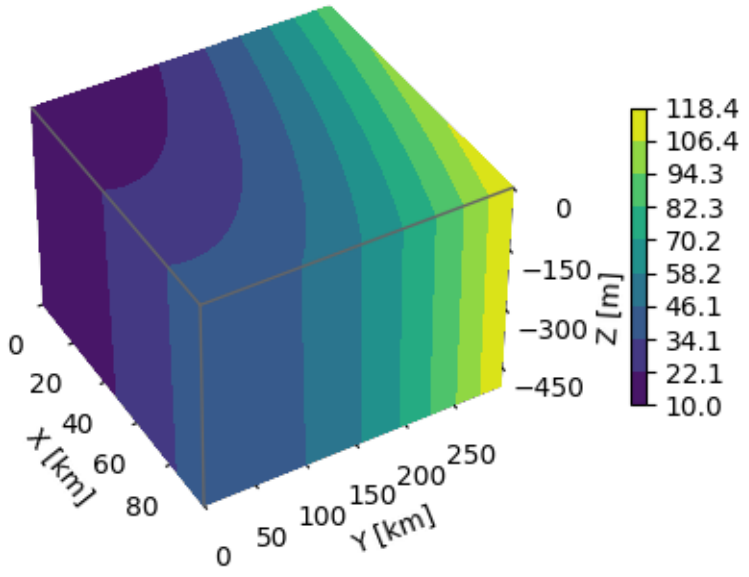
(continued from previous page)

```
# Fixing random state for reproducibility
np.random.seed(19680801)


fig = plt.figure()
ax = fig.add_subplot(projection='3d')

colors = ['r', 'g', 'b', 'y']
yticks = [3, 2, 1, 0]
for c, k in zip(colors, yticks):
    # Generate the random data for the y=k 'layer'.
    xs = np.arange(20)
    ys = np.random.rand(20)

    # You can provide either a single color or an array with the same length␣
 ↪as
    # xs and ys. To demonstrate this, we color the first bar of each set cyan.
    cs = [c] * len(xs)
    cs[0] = 'c'

    # Plot the bar graph given by xs and ys on the plane y=k with 80% opacity.
    ax.bar(xs, ys, zs=k, zdir='y', color=cs, alpha=0.8)

ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')

# On the y-axis let's only label the discrete values that we have data for.
ax.set_yticks(yticks)

plt.show()
```

### 3D box surface plot

Given data on a gridded volume `X`, `Y`, `Z`, this example plots the data values on the volume surfaces.

The strategy is to select the data from each surface and plot contours separately using *axes3d.Axes3D.contourf* with appropriate parameters *zdir* and *offset*.




```
import matplotlib.pyplot as plt
import numpy as np

# Define dimensions
Nx, Ny, Nz = 100, 300, 500
X, Y, Z = np.meshgrid(np.arange(Nx), np.arange(Ny), -np.arange(Nz))

# Create fake data
data = (((X+100)**2 + (Y-20)**2 + 2*Z)/1000+1)

kw = {
    'vmin': data.min(),
    'vmax': data.max(),
    'levels': np.linspace(data.min(), data.max(), 10),
}

# Create a figure with 3D ax
fig = plt.figure(figsize=(5, 4))
ax = fig.add_subplot(111, projection='3d')

# Plot contour surfaces
_ = ax.contourf(
    X[:, :, 0], Y[:, :, 0], data[:, :, 0],
    zdir='z', offset=0, **kw
)
_ = ax.contourf(
    X[0, :, :], data[0, :, :], Z[0, :, :],
    zdir='y', offset=0, **kw
```

(continues on next page)

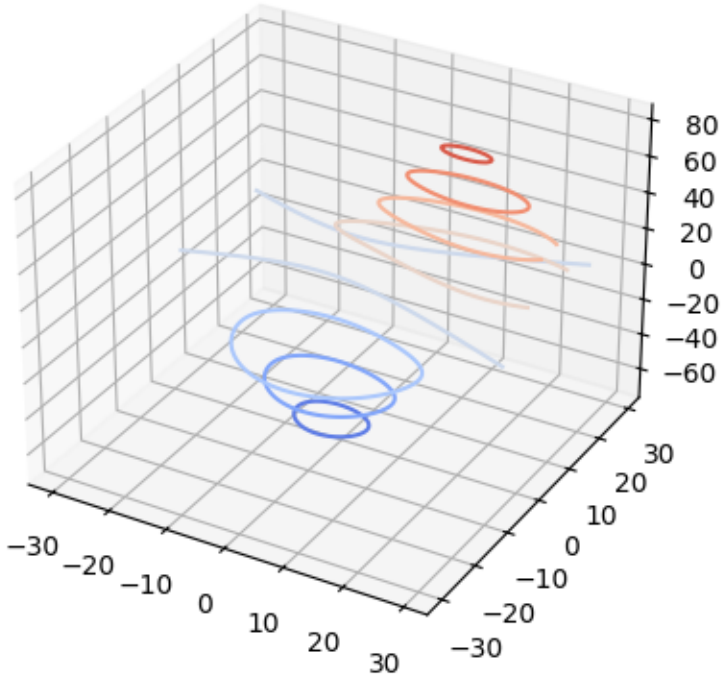
(continued from previous page)

```
)
C = ax.contourf(
    data[:, -1, :], Y[:, -1, :], Z[:, -1, :],
    zdir='x', offset=X.max(), **kw
)
# --


# Set limits of the plot from coord limits
xmin, xmax = X.min(), X.max()
ymin, ymax = Y.min(), Y.max()
zmin, zmax = Z.min(), Z.max()
ax.set(xlim=[xmin, xmax], ylim=[ymin, ymax], zlim=[zmin, zmax])

# Plot edges
edges_kw = dict(color='0.4', linewidth=1, zorder=1e3)
ax.plot([xmax, xmax], [ymin, ymax], 0, **edges_kw)
ax.plot([xmin, xmax], [ymin, ymin], 0, **edges_kw)
ax.plot([xmax, xmax], [ymin, ymin], [zmin, zmax], **edges_kw)

# Set labels and zticks
ax.set(
    xlabel='X [km]',
    ylabel='Y [km]',
    zlabel='Z [m]',
    zticks=[0, -150, -300, -450],
)

# Set zoom and angle view
ax.view_init(40, -30, 0)
ax.set_box_aspect(None, zoom=0.9)

# Colorbar
fig.colorbar(C, ax=ax, fraction=0.02, pad=0.1, label='Name [units]')

# Show Figure
plt.show()
```

**Total running time of the script:** (0 minutes 1.975 seconds)

### Plot contour (level) curves in 3D

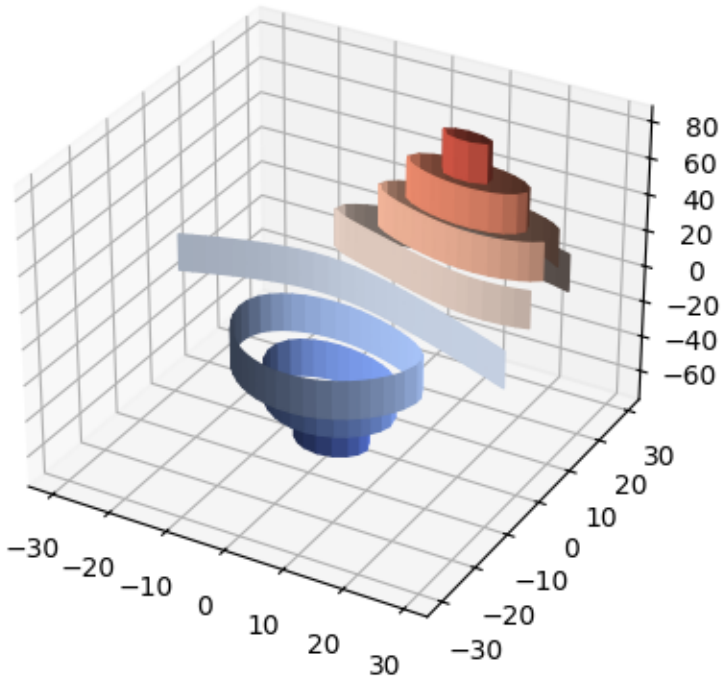
This is like a contour plot in 2D except that the `f(x, y)=c` curve is plotted on the plane `z=c`.




```
import matplotlib.pyplot as plt

from matplotlib import cm
from mpl_toolkits.mplot3d import axes3d

ax = plt.figure().add_subplot(projection='3d')
X, Y, Z = axes3d.get_test_data(0.05)

ax.contour(X, Y, Z, cmap=cm.coolwarm)  # Plot contour curves

plt.show()
```

### Plot contour (level) curves in 3D using the extend3d option

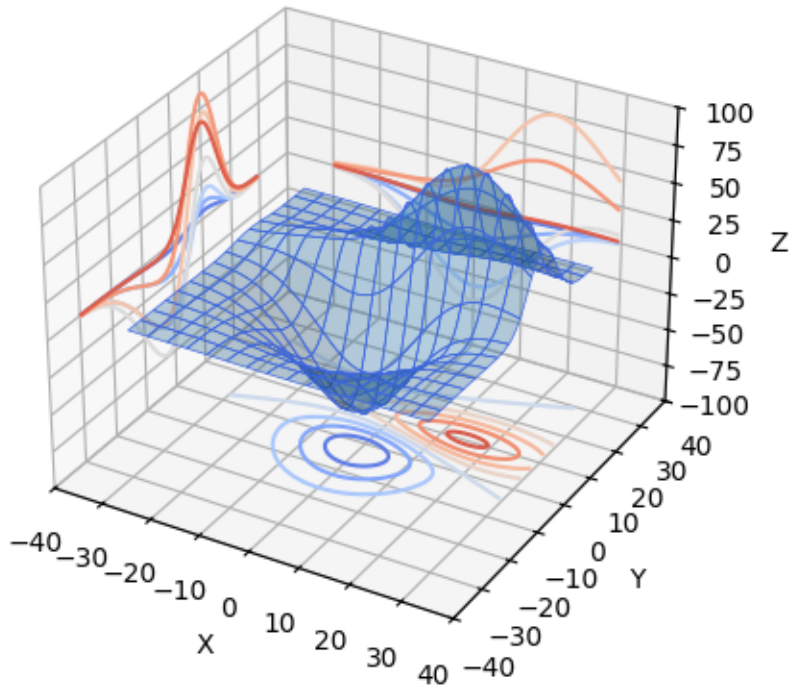
This modification of the *Plot contour (level) curves in 3D* example uses `extend3d=True` to extend the curves vertically into 'ribbons'.




```
import matplotlib.pyplot as plt

from matplotlib import cm
from mpl_toolkits.mplot3d import axes3d

ax = plt.figure().add_subplot(projection='3d')
X, Y, Z = axes3d.get_test_data(0.05)
ax.contour(X, Y, Z, extend3d=True, cmap=cm.coolwarm)

plt.show()
```

### Project contour profiles onto a graph

Demonstrates displaying a 3D surface while also projecting contour 'profiles' onto the 'walls' of the graph. See *Project filled contour onto a graph* for the filled version.




```
import matplotlib.pyplot as plt

from mpl_toolkits.mplot3d import axes3d

ax = plt.figure().add_subplot(projection='3d')
X, Y, Z = axes3d.get_test_data(0.05)

# Plot the 3D surface
ax.plot_surface(X, Y, Z, edgecolor='royalblue', lw=0.5, rstride=8, cstride=8,
                alpha=0.3)

# Plot projections of the contours for each dimension.  By choosing offsets
# that match the appropriate axes limits, the projected contours will sit on
# the 'walls' of the graph.
ax.contour(X, Y, Z, zdir='z', offset=-100, cmap='coolwarm')
ax.contour(X, Y, Z, zdir='x', offset=-40, cmap='coolwarm')
ax.contour(X, Y, Z, zdir='y', offset=40, cmap='coolwarm')
```

(continues on next page)

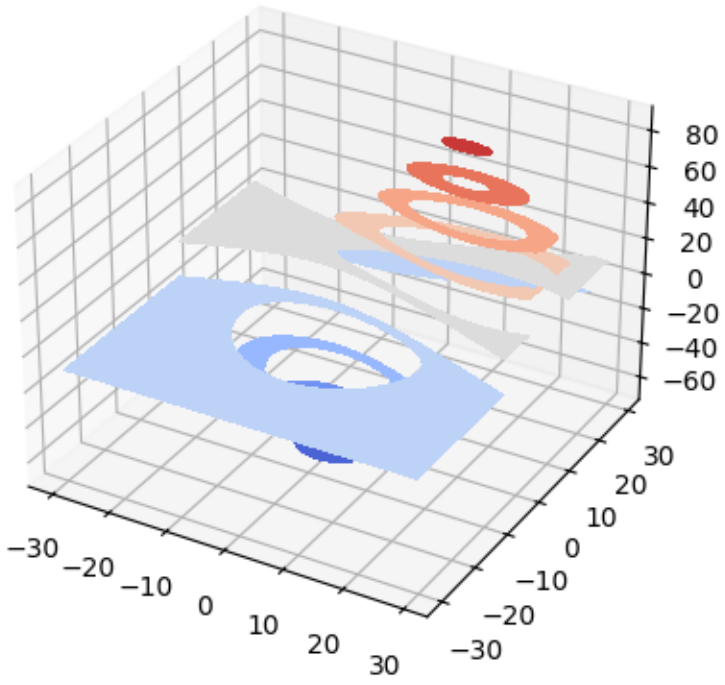
(continued from previous page)

```
ax.set(xlim=(-40, 40), ylim=(-40, 40), zlim=(-100, 100),
       xlabel='X', ylabel='Y', zlabel='Z')

plt.show()
```

### Filled contours

*Axes3D.contourf* differs from *Axes3D.contour* in that it creates filled contours, i.e. a discrete number of colours are used to shade the domain.

This is like a *Axes.contourf* plot in 2D except that the shaded region corresponding to the level c is graphed on the plane z=c.




```
import matplotlib.pyplot as plt

from matplotlib import cm
from mpl_toolkits.mplot3d import axes3d

ax = plt.figure().add_subplot(projection='3d')
X, Y, Z = axes3d.get_test_data(0.05)
```

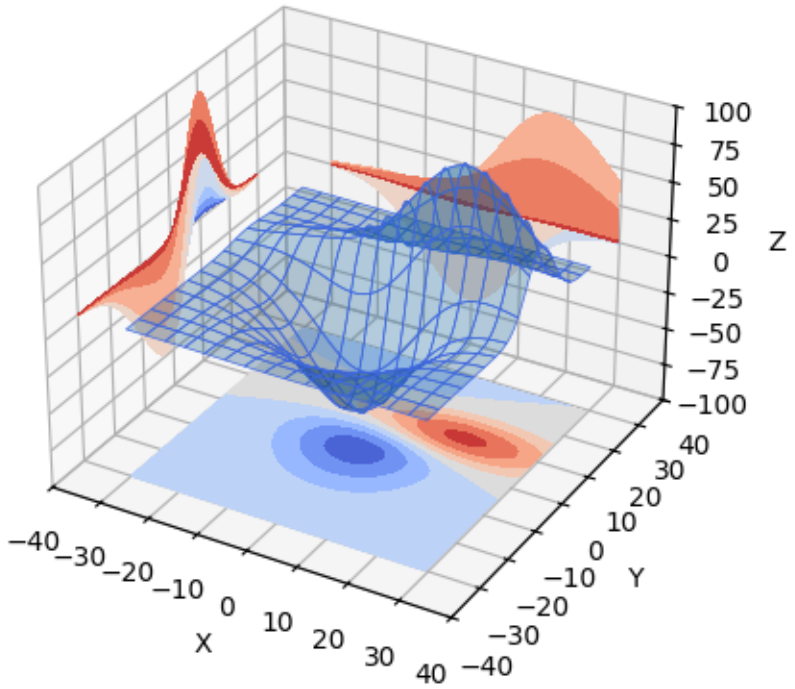
(continues on next page)

(continued from previous page)

```
ax.contourf(X, Y, Z, cmap=cm.coolwarm)

plt.show()
```

### Project filled contour onto a graph

Demonstrates displaying a 3D surface while also projecting filled contour 'profiles' onto the 'walls' of the graph. See *Project contour profiles onto a graph* for the unfilled version.




```
import matplotlib.pyplot as plt

from mpl_toolkits.mplot3d import axes3d

ax = plt.figure().add_subplot(projection='3d')
X, Y, Z = axes3d.get_test_data(0.05)

# Plot the 3D surface
ax.plot_surface(X, Y, Z, edgecolor='royalblue', lw=0.5, rstride=8, cstride=8,
                alpha=0.3)
```

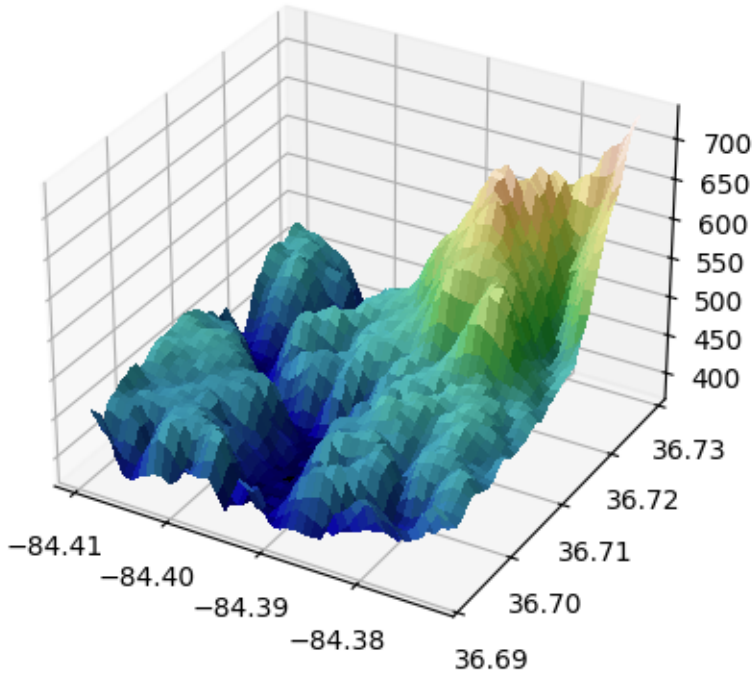
(continues on next page)

(continued from previous page)

```
# Plot projections of the contours for each dimension.  By choosing offsets
# that match the appropriate axes limits, the projected contours will sit on
# the 'walls' of the graph
ax.contourf(X, Y, Z, zdir='z', offset=-100, cmap='coolwarm')
ax.contourf(X, Y, Z, zdir='x', offset=-40, cmap='coolwarm')
ax.contourf(X, Y, Z, zdir='y', offset=40, cmap='coolwarm')

ax.set(xlim=(-40, 40), ylim=(-40, 40), zlim=(-100, 100),
       xlabel='X', ylabel='Y', zlabel='Z')

plt.show()
```

### Custom hillshading in a 3D surface plot

Demonstrates using custom hillshading in a 3D surface plot.




```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib import cbook, cm
```

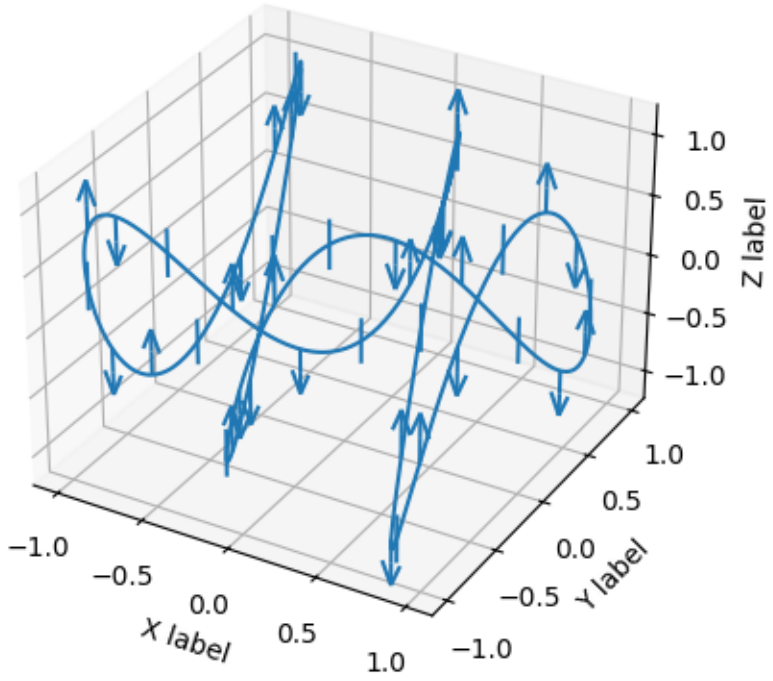
(continues on next page)

(continued from previous page)

```
from matplotlib.colors import LightSource

# Load and format data
dem = cbook.get_sample_data('jacksboro_fault_dem.npz')
z = dem['elevation']
nrows, ncols = z.shape
x = np.linspace(dem['xmin'], dem['xmax'], ncols)
y = np.linspace(dem['ymin'], dem['ymax'], nrows)
x, y = np.meshgrid(x, y)

region = np.s_[5:50, 5:50]
x, y, z = x[region], y[region], z[region]

# Set up plot
fig, ax = plt.subplots(subplot_kw=dict(projection='3d'))

ls = LightSource(270, 45)
# To use a custom hillshading mode, override the built-in shading and pass
# in the rgb colors of the shaded surface calculated from "shade".
rgb = ls.shade(z, cmap=cm.gist_earth, vert_exag=0.1, blend_mode='soft')
surf = ax.plot_surface(x, y, z, rstride=1, cstride=1, facecolors=rgb,
                       linewidth=0, antialiased=False, shade=False)

plt.show()
```

### 3D errorbars

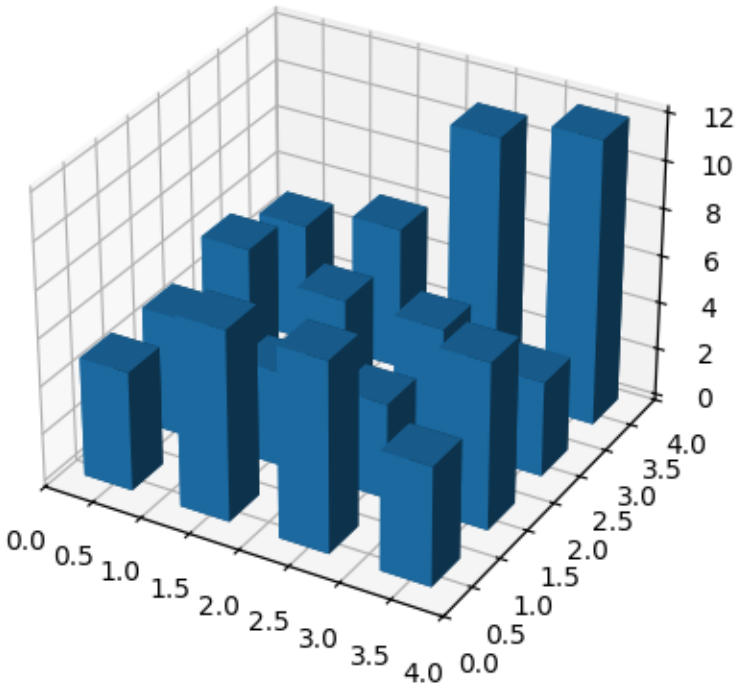
An example of using errorbars with upper and lower limits in mplot3d.




```
import matplotlib.pyplot as plt
import numpy as np

ax = plt.figure().add_subplot(projection='3d')

# setting up a parametric curve
t = np.arange(0, 2*np.pi+.1, 0.01)
x, y, z = np.sin(t), np.cos(3*t), np.sin(5*t)

estep = 15
i = np.arange(t.size)
zuplims = (i % estep == 0) & (i // estep % 3 == 0)
zlolims = (i % estep == 0) & (i // estep % 3 == 2)

ax.errorbar(x, y, z, 0.2, zuplims=zuplims, zlolims=zlolims, errorevery=estep)

ax.set_xlabel("X label")
ax.set_ylabel("Y label")
ax.set_zlabel("Z label")

plt.show()
```

### Create 3D histogram of 2D data

Demo of a histogram for 2D data as a bar graph in 3D.




```
import matplotlib.pyplot as plt
import numpy as np

# Fixing random state for reproducibility
np.random.seed(19680801)


fig = plt.figure()
ax = fig.add_subplot(projection='3d')
x, y = np.random.rand(2, 100) * 4
hist, xedges, yedges = np.histogram2d(x, y, bins=4, range=[[0, 4], [0, 4]])

# Construct arrays for the anchor positions of the 16 bars.
xpos, ypos = np.meshgrid(xedges[:-1] + 0.25, yedges[:-1] + 0.25, indexing="ij
 ↪")
xpos = xpos.ravel()
ypos = ypos.ravel()
zpos = 0
```

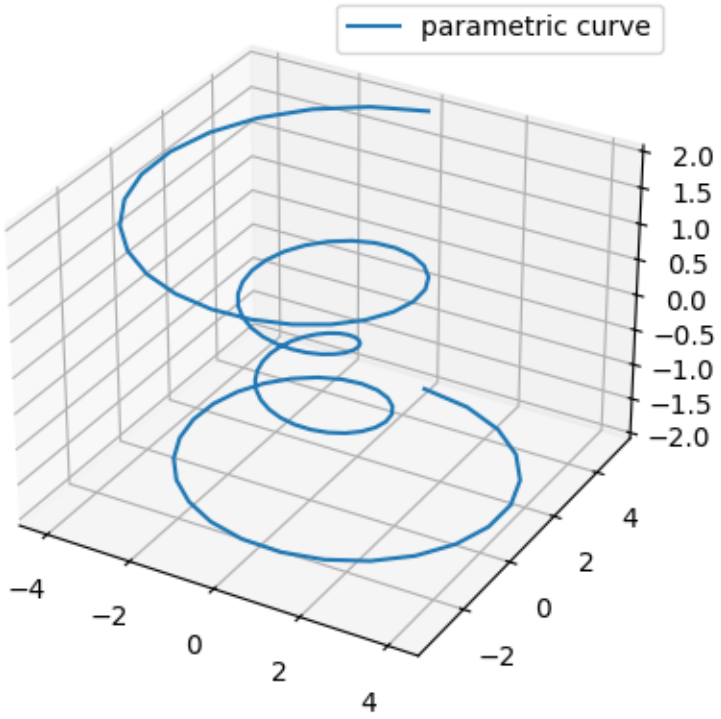
(continues on next page)

(continued from previous page)

```
# Construct arrays with the dimensions for the 16 bars.
dx = dy = 0.5 * np.ones_like(zpos)
dz = hist.ravel()

ax.bar3d(xpos, ypos, zpos, dx, dy, dz, zsort='average')

plt.show()
```

### Parametric curve

This example demonstrates plotting a parametric curve in 3D.




```
import matplotlib.pyplot as plt
import numpy as np

ax = plt.figure().add_subplot(projection='3d')

# Prepare arrays x, y, z
theta = np.linspace(-4 * np.pi, 4 * np.pi, 100)
z = np.linspace(-2, 2, 100)
```

(continues on next page)

(continued from previous page)

```
r = z**2 + 1
x = r * np.sin(theta)
y = r * np.cos(theta)

ax.plot(x, y, z, label='parametric curve')
ax.legend()

plt.show()
```

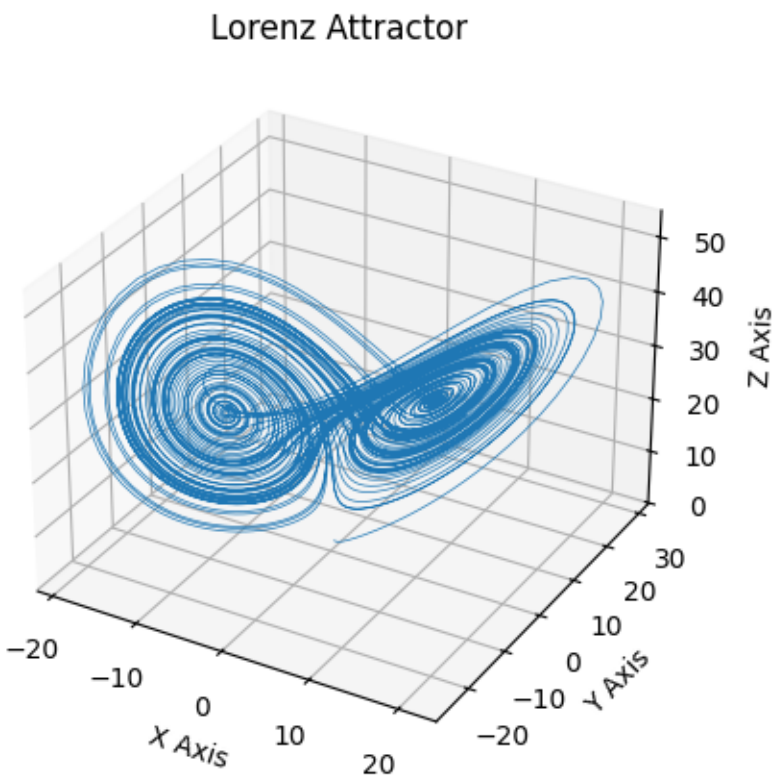
### Lorenz attractor

This is an example of plotting Edward Lorenz's 1963 "Deterministic Nonperiodic Flow" in a 3-dimensional space using mplot3d.

---

**Note:** Because this is a simple non-linear ODE, it would be more easily done using SciPy's ODE solver, but this approach depends only upon NumPy.

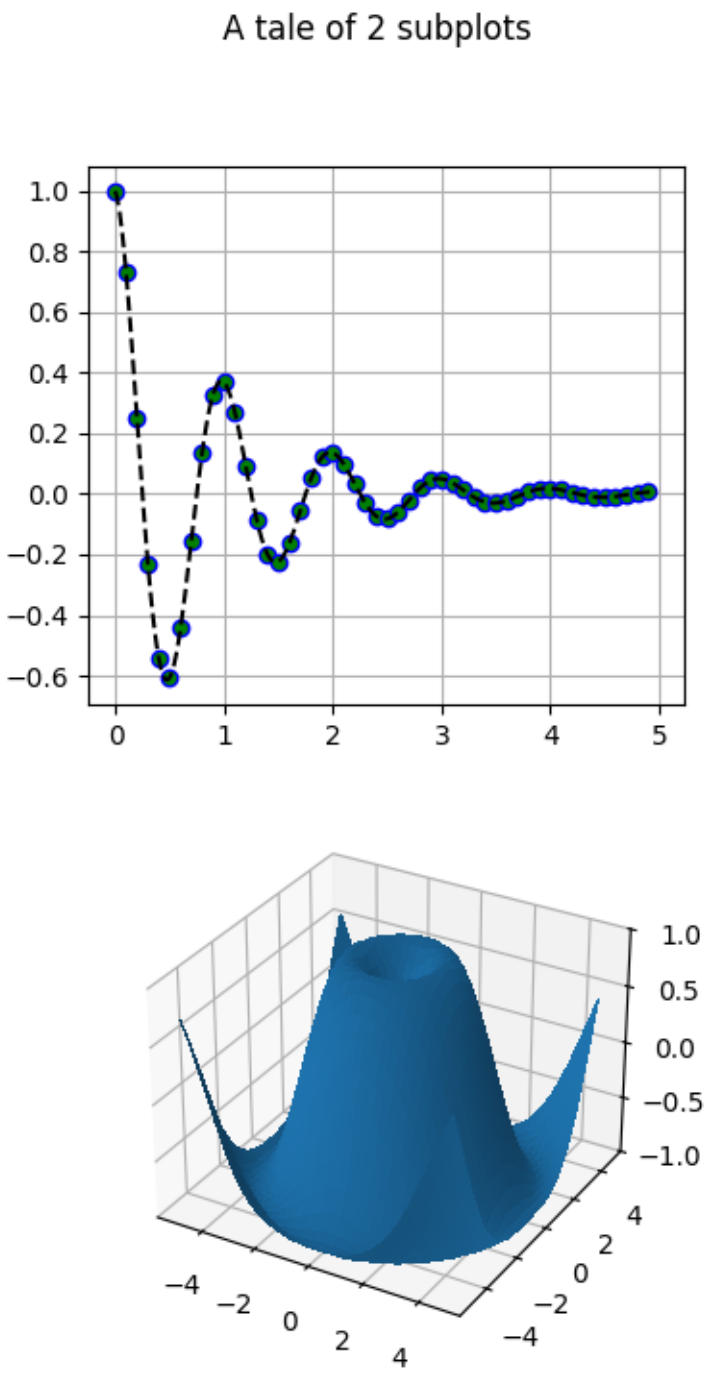
---



```
import matplotlib.pyplot as plt
import numpy as np


def lorenz(xyz, *, s=10, r=28, b=2.667):
    """
    Parameters
    ----------
    xyz : array-like, shape (3,)
       Point of interest in three-dimensional space.
    s, r, b : float
       Parameters defining the Lorenz attractor.

    Returns
    -------
    xyz_dot : array, shape (3,)
       Values of the Lorenz attractor's partial derivatives at *xyz*.
    """
    x, y, z = xyz
    x_dot = s*(y - x)
    y_dot = r*x - y - x*z
    z_dot = x*y - b*z
    return np.array([x_dot, y_dot, z_dot])


dt = 0.01
num_steps = 10000

xyzs = np.empty((num_steps + 1, 3))  # Need one more for the initial values
xyzs[0] = (0., 1., 1.05)  # Set initial values
# Step through "time", calculating the partial derivatives at the current␣
 ↪point
# and using them to estimate the next point
for i in range(num_steps):
    xyzs[i + 1] = xyzs[i] + lorenz(xyzs[i]) * dt

# Plot
ax = plt.figure().add_subplot(projection='3d')

ax.plot(*xyzs.T, lw=0.5)
ax.set_xlabel("X Axis")
ax.set_ylabel("Y Axis")
ax.set_zlabel("Z Axis")
ax.set_title("Lorenz Attractor")

plt.show()
```

### 2D and 3D Axes in same figure

This example shows a how to plot a 2D and a 3D plot on the same figure.

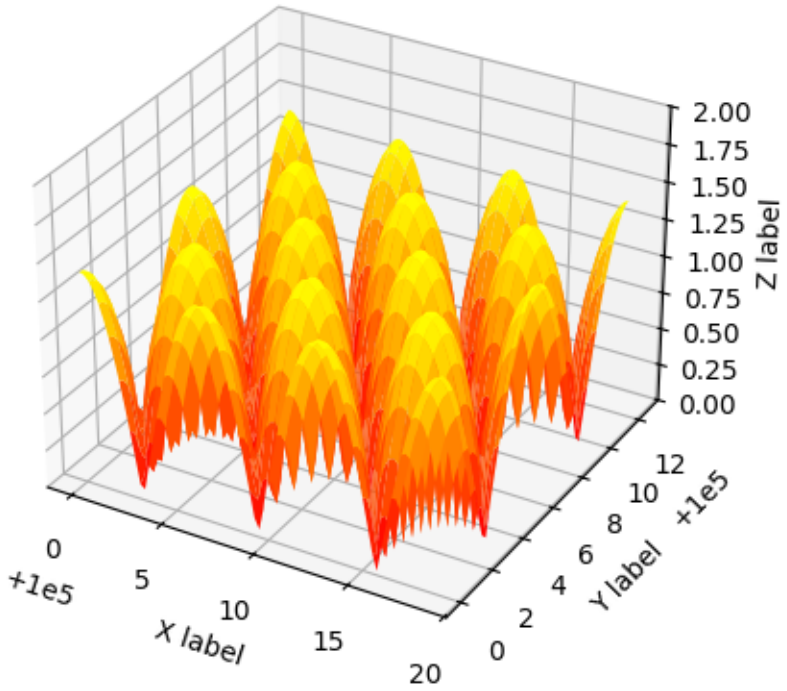


```
import matplotlib.pyplot as plt
import numpy as np


def f(t):
    return np.cos(2*np.pi*t) * np.exp(-t)


# Set up a figure twice as tall as it is wide
fig = plt.figure(figsize=plt.figaspect(2.))
fig.suptitle('A tale of 2 subplots')

# First subplot
ax = fig.add_subplot(2, 1, 1)

t1 = np.arange(0.0, 5.0, 0.1)
t2 = np.arange(0.0, 5.0, 0.02)
t3 = np.arange(0.0, 2.0, 0.01)

ax.plot(t1, f(t1), 'bo',
        t2, f(t2), 'k--', markerfacecolor='green')
ax.grid(True)
ax.set_ylabel('Damped oscillation')

# Second subplot
ax = fig.add_subplot(2, 1, 2, projection='3d')

X = np.arange(-5, 5, 0.25)
Y = np.arange(-5, 5, 0.25)
X, Y = np.meshgrid(X, Y)
R = np.sqrt(X**2 + Y**2)
Z = np.sin(R)

surf = ax.plot_surface(X, Y, Z, rstride=1, cstride=1,
                       linewidth=0, antialiased=False)
ax.set_zlim(-1, 1)

plt.show()
```

### Automatic text offsetting

This example demonstrates mplot3d's offset text display. As one rotates the 3D figure, the offsets should remain oriented the same way as the axis label, and should also be located "away" from the center of the plot.

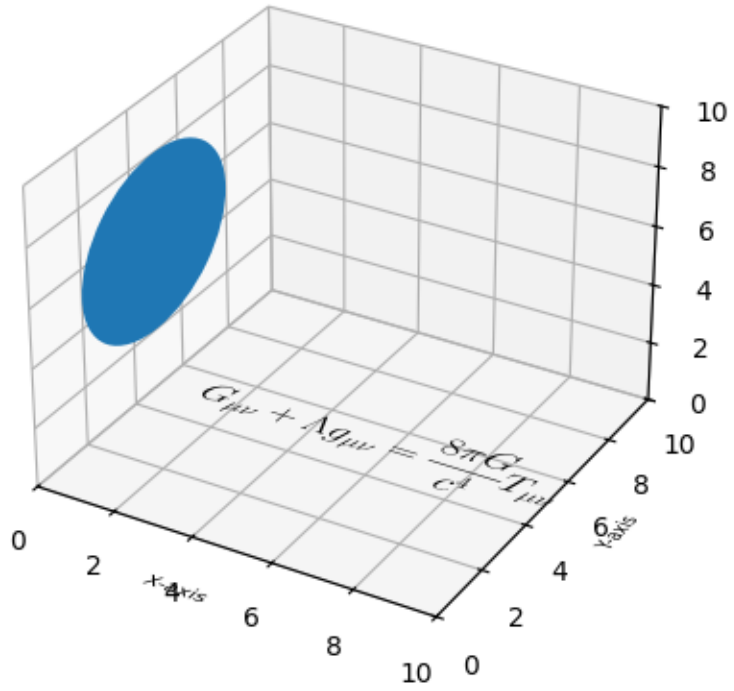
This demo triggers the display of the offset text for the x- and y-axis by adding 1e5 to X and Y. Anything less would not automatically trigger it.




```
import matplotlib.pyplot as plt
import numpy as np

ax = plt.figure().add_subplot(projection='3d')

X, Y = np.mgrid[0:6*np.pi:0.25, 0:4*np.pi:0.25]
Z = np.sqrt(np.abs(np.cos(X) + np.cos(Y)))

ax.plot_surface(X + 1e5, Y + 1e5, Z, cmap='autumn', cstride=2, rstride=2)

ax.set_xlabel("X label")
ax.set_ylabel("Y label")
ax.set_zlabel("Z label")
ax.set_zlim(0, 2)

plt.show()
```

### Draw flat objects in 3D plot

Demonstrate using *pathpatch_2d_to_3d* to 'draw' shapes and text on a 3D plot.




```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.patches import Circle, PathPatch
from matplotlib.text import TextPath
from matplotlib.transforms import Affine2D
import mpl_toolkits.mplot3d.art3d as art3d


def text3d(ax, xyz, s, zdir="z", size=None, angle=0, usetex=False, **kwargs):
    """
    Plots the string *s* on the Axes *ax*, with position *xyz*, size *size*,
    and rotation angle *angle*. *zdir* gives the axis which is to be treated␣
 ↪as
    the third dimension. *usetex* is a boolean indicating whether the string
    should be run through a LaTeX subprocess or not.  Any additional keyword
    arguments are forwarded to `.transform_path`.

    Note: zdir affects the interpretation of xyz.
```

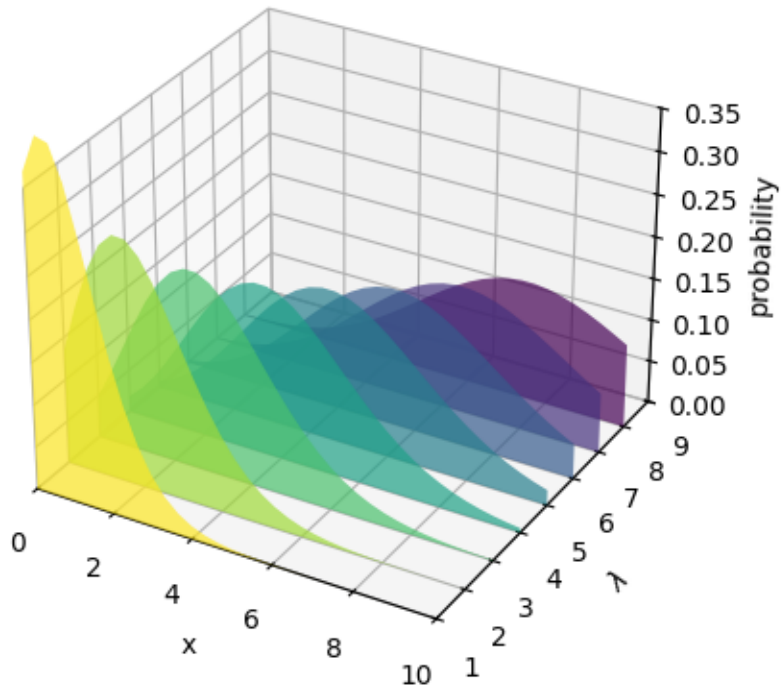
(continues on next page)

(continued from previous page)

```
    """
    x, y, z = xyz
    if zdir == "y":
        xy1, z1 = (x, z), y
    elif zdir == "x":
        xy1, z1 = (y, z), x
    else:
        xy1, z1 = (x, y), z

    text_path = TextPath((0, 0), s, size=size, usetex=usetex)
    trans = Affine2D().rotate(angle).translate(xy1[0], xy1[1])

    p1 = PathPatch(trans.transform_path(text_path), **kwargs)
    ax.add_patch(p1)
    art3d.pathpatch_2d_to_3d(p1, z=z1, zdir=zdir)


fig = plt.figure()
ax = fig.add_subplot(projection='3d')

# Draw a circle on the x=0 'wall'
p = Circle((5, 5), 3)
ax.add_patch(p)
art3d.pathpatch_2d_to_3d(p, z=0, zdir="x")

# Manually label the axes
text3d(ax, (4, -2, 0), "X-axis", zdir="z", size=.5, usetex=False,
       ec="none", fc="k")
text3d(ax, (12, 4, 0), "Y-axis", zdir="z", size=.5, usetex=False,
       angle=np.pi / 2, ec="none", fc="k")
text3d(ax, (12, 10, 4), "Z-axis", zdir="y", size=.5, usetex=False,
       angle=np.pi / 2, ec="none", fc="k")

# Write a Latex formula on the z=0 'floor'
text3d(ax, (1, 5, 0),
       r"$\displaystyle G_{\mu\nu} + \Lambda g_{\mu\nu} = "
       r"\frac{8\pi G}{c^4} T_{\mu\nu}  $",
       zdir="z", size=1, usetex=True,
       ec="none", fc="k")

ax.set_xlim(0, 10)
ax.set_ylim(0, 10)
ax.set_zlim(0, 10)

plt.show()
```

### Generate polygons to fill under 3D line graph

Demonstrate how to create polygons which fill the space under a line graph. In this example polygons are semi-transparent, creating a sort of 'jagged stained glass' effect.




```
import math

import matplotlib.pyplot as plt
import numpy as np

from matplotlib.collections import PolyCollection

# Fixing random state for reproducibility
np.random.seed(19680801)


def polygon_under_graph(x, y):
    """
    Construct the vertex list which defines the polygon filling the space␣
 ↪under
    the (x, y) line graph. This assumes x is in ascending order.
    """
    return [(x[0], 0.), *zip(x, y), (x[-1], 0.)]
```

(continues on next page)

(continued from previous page)

```
ax = plt.figure().add_subplot(projection='3d')

x = np.linspace(0., 10., 31)
lambdas = range(1, 9)

# verts[i] is a list of (x, y) pairs defining polygon i.
gamma = np.vectorize(math.gamma)
verts = [polygon_under_graph(x, l**x * np.exp(-l) / gamma(x + 1))
         for l in lambdas]
facecolors = plt.colormaps['viridis_r'](np.linspace(0, 1, len(verts)))

poly = PolyCollection(verts, facecolors=facecolors, alpha=.7)
ax.add_collection3d(poly, zs=lambdas, zdir='y')

ax.set(xlim=(0, 10), ylim=(1, 9), zlim=(0, 0.35),
       xlabel='x', ylabel=r'$\lambda$', zlabel='probability')

plt.show()
```

### 3D plot projection types

Demonstrates the different camera projections for 3D plots, and the effects of changing the focal length for a perspective projection. Note that Matplotlib corrects for the 'zoom' effect of changing the focal length.

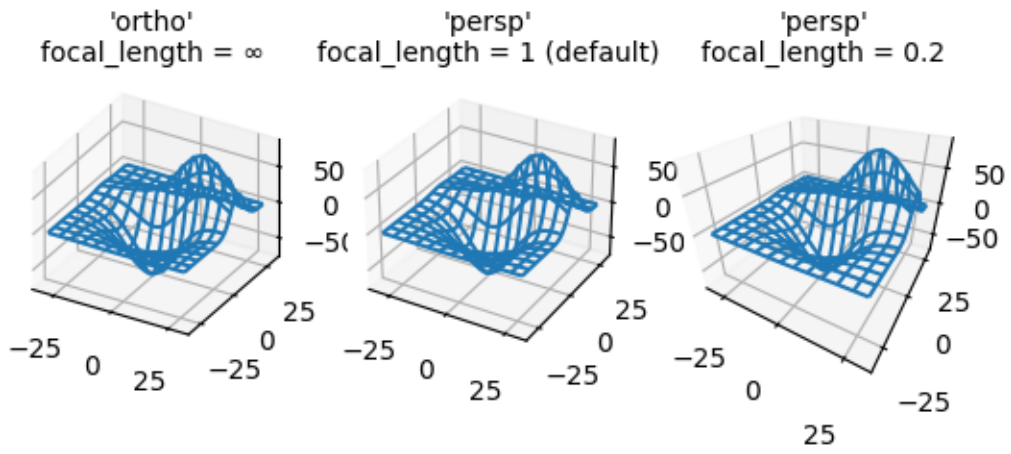
The default focal length of 1 corresponds to a Field of View (FOV) of 90 deg. An increased focal length between 1 and infinity "flattens" the image, while a decreased focal length between 1 and 0 exaggerates the perspective and gives the image more apparent depth. In the limiting case, a focal length of infinity corresponds to an orthographic projection after correction of the zoom effect.

You can calculate focal length from a FOV via the equation:

$$1/\tan(\mathrm{FOV}/2)$$

Or vice versa:

$$\mathrm{FOV} = 2\arctan(1/\mathrm{focallength})$$




```
import matplotlib.pyplot as plt

from mpl_toolkits.mplot3d import axes3d

fig, axs = plt.subplots(1, 3, subplot_kw={'projection': '3d'})

# Get the test data
X, Y, Z = axes3d.get_test_data(0.05)

# Plot the data
for ax in axs:
    ax.plot_wireframe(X, Y, Z, rstride=10, cstride=10)

# Set the orthographic projection.
axs[0].set_proj_type('ortho')  # FOV = 0 deg
axs[0].set_title("'ortho'\nfocal_length = ∞", fontsize=10)

# Set the perspective projections
axs[1].set_proj_type('persp')  # FOV = 90 deg
axs[1].set_title("'persp'\nfocal_length = 1 (default)", fontsize=10)

axs[2].set_proj_type('persp', focal_length=0.2)  # FOV = 157.4 deg
axs[2].set_title("'persp'\nfocal_length = 0.2", fontsize=10)
```

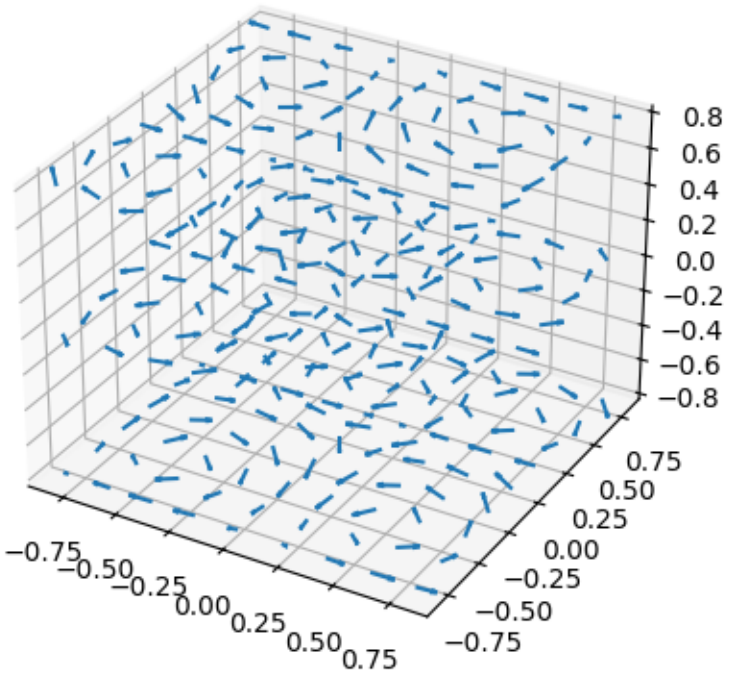
(continues on next page)

(continued from previous page)

```
plt.show()
```

### 3D quiver plot

Demonstrates plotting directional arrows at points on a 3D meshgrid.




```
import matplotlib.pyplot as plt
import numpy as np

ax = plt.figure().add_subplot(projection='3d')

# Make the grid
x, y, z = np.meshgrid(np.arange(-0.8, 1, 0.2),
                      np.arange(-0.8, 1, 0.2),
                      np.arange(-0.8, 1, 0.8))

# Make the direction data for the arrows
u = np.sin(np.pi * x) * np.cos(np.pi * y) * np.cos(np.pi * z)
v = -np.cos(np.pi * x) * np.sin(np.pi * y) * np.cos(np.pi * z)
```

(continues on next page)

(continued from previous page)

```
w = (np.sqrt(2.0 / 3.0) * np.cos(np.pi * x) * np.cos(np.pi * y) *
     np.sin(np.pi * z))

ax.quiver(x, y, z, u, v, w, length=0.1, normalize=True)

plt.show()
```

### Rotating a 3D plot

A very simple animation of a rotating 3D plot about all three axes.

See *Animate a 3D wireframe plot* for another example of animating a 3D plot.

(This example is skipped when building the documentation gallery because it intentionally takes a long time to run)

```
import matplotlib.pyplot as plt

from mpl_toolkits.mplot3d import axes3d

fig = plt.figure()
ax = fig.add_subplot(projection='3d')

# Grab some example data and plot a basic wireframe.
X, Y, Z = axes3d.get_test_data(0.05)
ax.plot_wireframe(X, Y, Z, rstride=10, cstride=10)

# Set the axis labels
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')

# Rotate the axes and update
for angle in range(0, 360*4 + 1):
    # Normalize the angle to the range [-180, 180] for display
    angle_norm = (angle + 180) % 360 - 180

    # Cycle through a full rotation of elevation, then azimuth, roll, and all
    elev = azim = roll = 0
    if angle <= 360:
        elev = angle_norm
    elif angle <= 360*2:
        azim = angle_norm
    elif angle <= 360*3:
        roll = angle_norm
    else:
        elev = azim = roll = angle_norm

    # Update the axis view and title
    ax.view_init(elev, azim, roll)
```

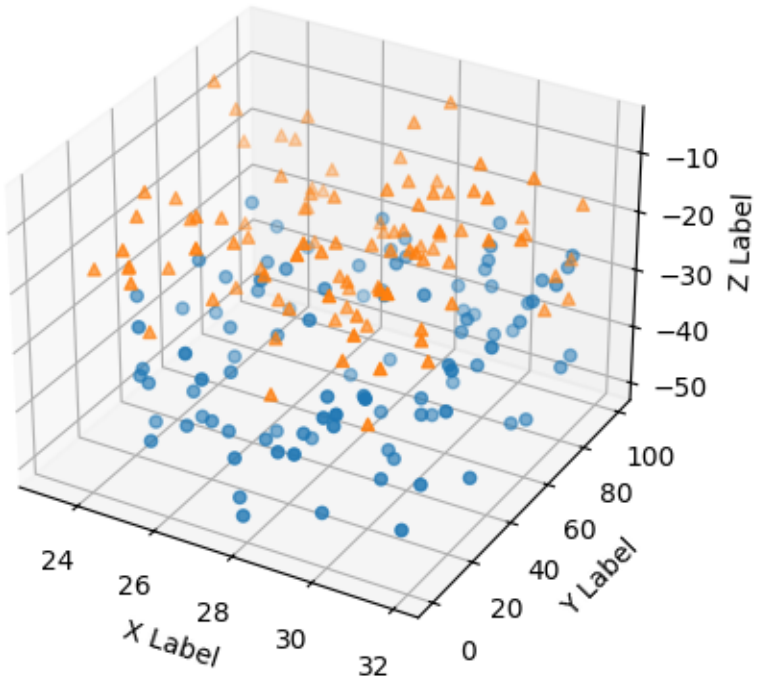
(continues on next page)

(continued from previous page)

```
    plt.title('Elevation: %d°, Azimuth: %d°, Roll: %d°' % (elev, azim, roll))

    plt.draw()
    plt.pause(.001)
```

### 3D scatterplot

Demonstration of a basic scatterplot in 3D.




```
import matplotlib.pyplot as plt
import numpy as np

# Fixing random state for reproducibility
np.random.seed(19680801)


def randrange(n, vmin, vmax):
    """
    Helper function to make an array of random numbers having shape (n, )
    with each number distributed Uniform(vmin, vmax).
```

(continues on next page)

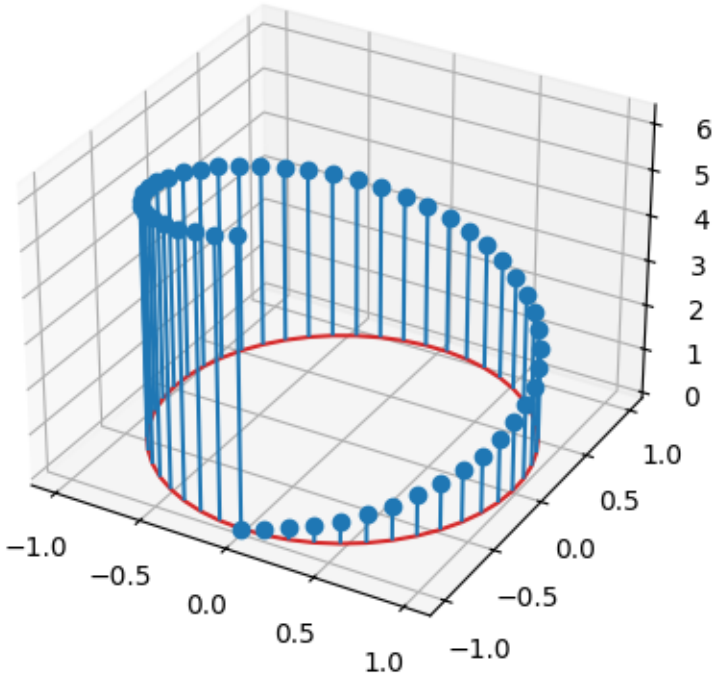
(continued from previous page)

```
    """
    return (vmax - vmin)*np.random.rand(n) + vmin

fig = plt.figure()
ax = fig.add_subplot(projection='3d')

n = 100

# For each set of style and range settings, plot n random points in the box
# defined by x in [23, 32], y in [0, 100], z in [zlow, zhigh].
for m, zlow, zhigh in [('o', -50, -25), ('^', -30, -5)]:
    xs = randrange(n, 23, 32)
    ys = randrange(n, 0, 100)
    zs = randrange(n, zlow, zhigh)
    ax.scatter(xs, ys, zs, marker=m)

ax.set_xlabel('X Label')
ax.set_ylabel('Y Label')
ax.set_zlabel('Z Label')

plt.show()
```

### 3D stem

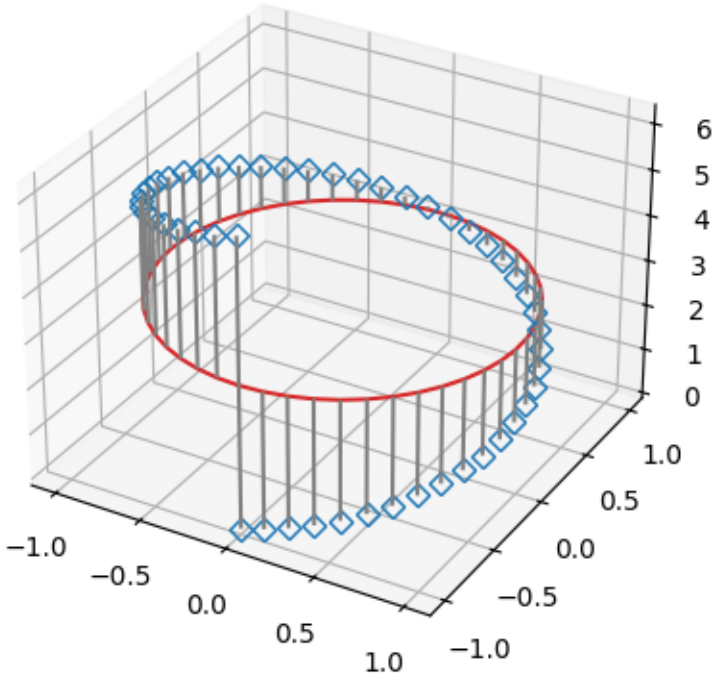
Demonstration of a stem plot in 3D, which plots vertical lines from a baseline to the $z$-coordinate and places a marker at the tip.

```
import matplotlib.pyplot as plt
import numpy as np

theta = np.linspace(0, 2*np.pi)
x = np.cos(theta - np.pi/2)
y = np.sin(theta - np.pi/2)
z = theta

fig, ax = plt.subplots(subplot_kw=dict(projection='3d'))
ax.stem(x, y, z)

plt.show()
```




The position of the baseline can be adapted using *bottom*. The parameters *linefmt*, *markerfmt*, and *basefmt* control basic format properties of the plot. However, in contrast to `plot` not all properties are configurable via keyword arguments. For more advanced control adapt the line objects returned by `stem`.

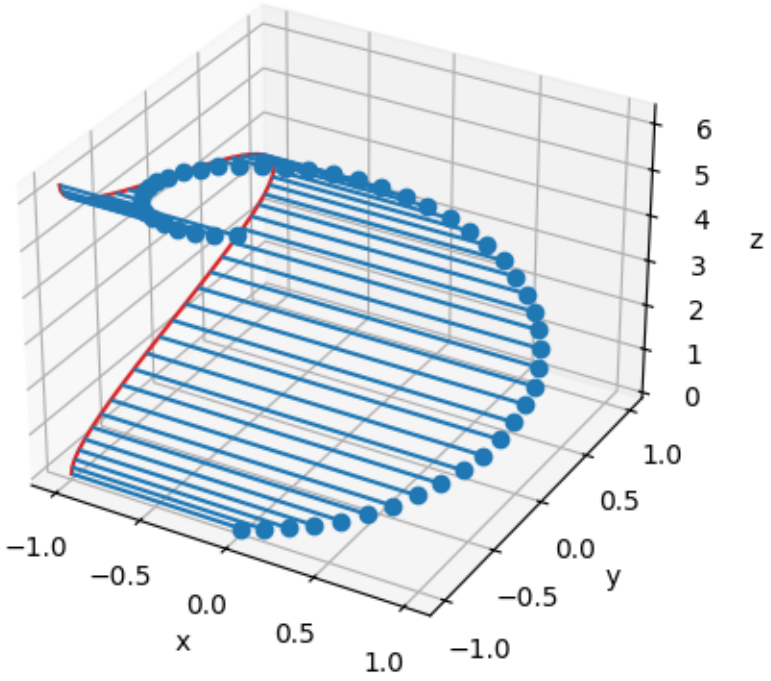
```
fig, ax = plt.subplots(subplot_kw=dict(projection='3d'))
markerline, stemlines, baseline = ax.stem(
    x, y, z, linefmt='grey', markerfmt='D', bottom=np.pi)
markerline.set_markerfacecolor('none')

plt.show()
```




The orientation of the stems and baseline can be changed using *orientation*. This determines in which direction the stems are projected from the head points, towards the *bottom* baseline.

For examples, by setting `orientation='x'`, the stems are projected along the *x*-direction, and the baseline is in the *yz*-plane.

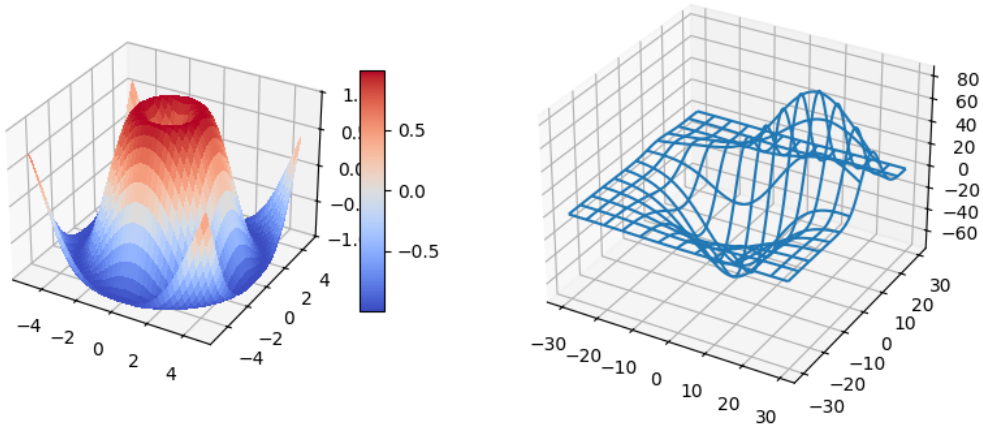
```
fig, ax = plt.subplots(subplot_kw=dict(projection='3d'))
markerline, stemlines, baseline = ax.stem(x, y, z, bottom=-1, orientation='x')
ax.set(xlabel='x', ylabel='y', zlabel='z')

plt.show()
```




**Total running time of the script:** (0 minutes 1.228 seconds)

### 3D plots as subplots

Demonstrate including 3D plots as subplots.




```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib import cm
from mpl_toolkits.mplot3d.axes3d import get_test_data

# set up a figure twice as wide as it is tall
fig = plt.figure(figsize=plt.figaspect(0.5))

# =============
# First subplot
# =============
# set up the Axes for the first plot
ax = fig.add_subplot(1, 2, 1, projection='3d')

# plot a 3D surface like in the example mplot3d/surface3d_demo
X = np.arange(-5, 5, 0.25)
Y = np.arange(-5, 5, 0.25)
X, Y = np.meshgrid(X, Y)
R = np.sqrt(X**2 + Y**2)
Z = np.sin(R)
surf = ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap=cm.coolwarm,
                       linewidth=0, antialiased=False)
ax.set_zlim(-1.01, 1.01)
fig.colorbar(surf, shrink=0.5, aspect=10)

# ==============
# Second subplot
# ==============
# set up the Axes for the second plot
ax = fig.add_subplot(1, 2, 2, projection='3d')
```

(continues on next page)

(continued from previous page)

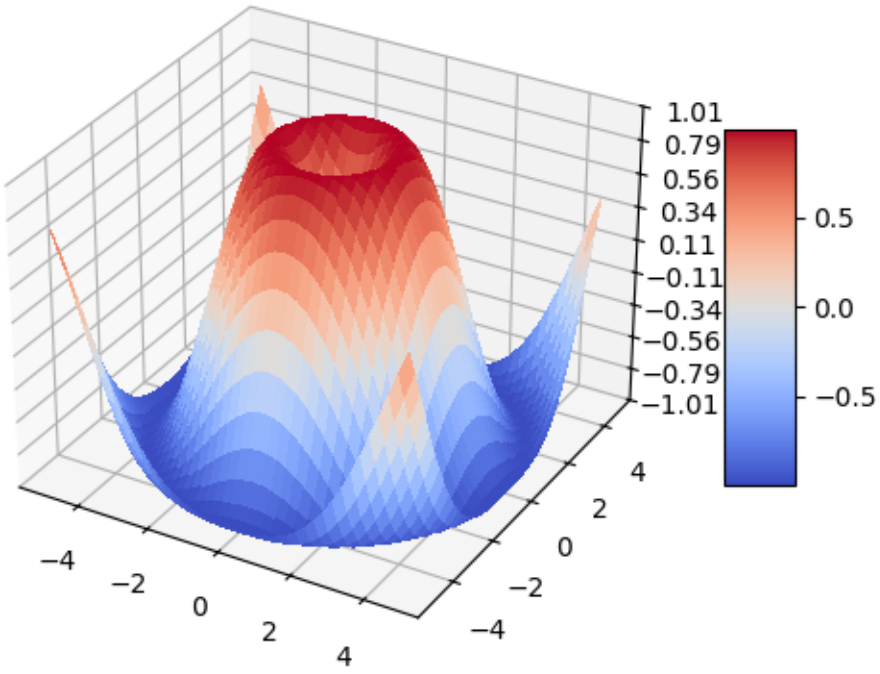
```
# plot a 3D wireframe like in the example mplot3d/wire3d_demo
X, Y, Z = get_test_data(0.05)
ax.plot_wireframe(X, Y, Z, rstride=10, cstride=10)

plt.show()
```

### 3D surface (colormap)

Demonstrates plotting a 3D surface colored with the coolwarm colormap. The surface is made opaque by using `antialiased=False`.

Also demonstrates using the *`LinearLocator`* and custom formatting for the z axis tick labels.

```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib import cm
from matplotlib.ticker import LinearLocator

fig, ax = plt.subplots(subplot_kw={"projection": "3d"})

# Make data.
X = np.arange(-5, 5, 0.25)
Y = np.arange(-5, 5, 0.25)
X, Y = np.meshgrid(X, Y)
R = np.sqrt(X**2 + Y**2)
Z = np.sin(R)

# Plot the surface.
surf = ax.plot_surface(X, Y, Z, cmap=cm.coolwarm,
                       linewidth=0, antialiased=False)

# Customize the z axis.
ax.set_zlim(-1.01, 1.01)
ax.zaxis.set_major_locator(LinearLocator(10))
# A StrMethodFormatter is used automatically
ax.zaxis.set_major_formatter('{x:.02f}')

# Add a color bar which maps values to colors.
fig.colorbar(surf, shrink=0.5, aspect=5)

plt.show()
```



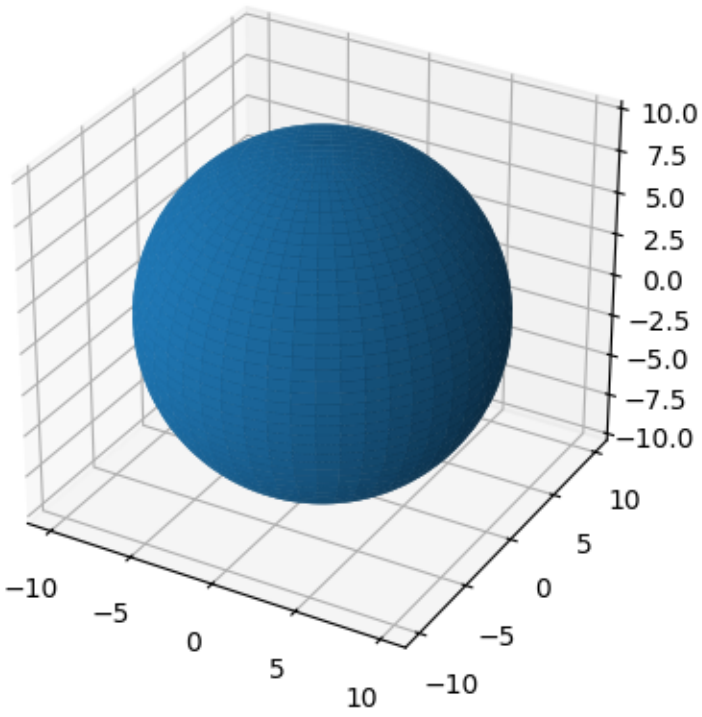

**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.pyplot.subplots*
- *matplotlib.axis.Axis.set_major_formatter*
- *matplotlib.axis.Axis.set_major_locator*
- *matplotlib.ticker.LinearLocator*
- *matplotlib.ticker.StrMethodFormatter*

### 3D surface (solid color)

Demonstrates a very basic plot of a 3D surface using a solid color.




```
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure()
ax = fig.add_subplot(projection='3d')

# Make data
u = np.linspace(0, 2 * np.pi, 100)
v = np.linspace(0, np.pi, 100)
x = 10 * np.outer(np.cos(u), np.sin(v))
y = 10 * np.outer(np.sin(u), np.sin(v))
z = 10 * np.outer(np.ones(np.size(u)), np.cos(v))

# Plot the surface
ax.plot_surface(x, y, z)

# Set an equal aspect ratio
ax.set_aspect('equal')
```

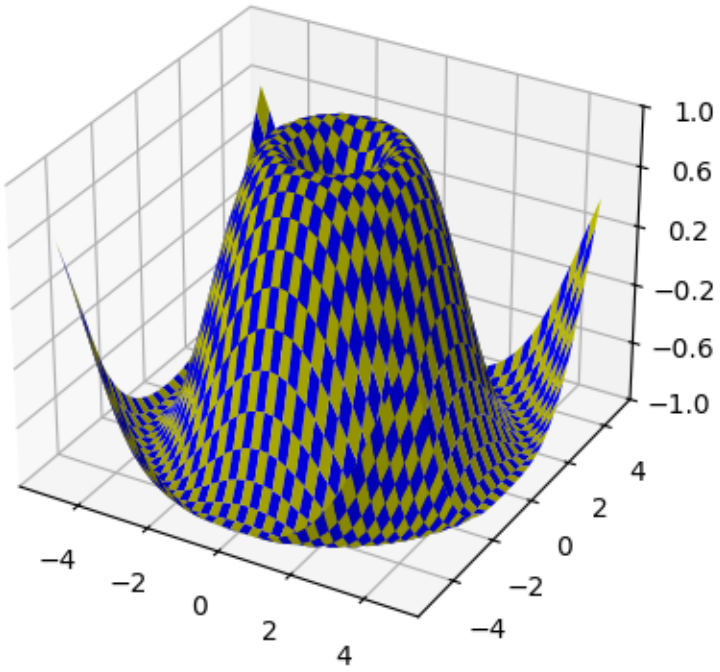
(continues on next page)

(continued from previous page)

```
plt.show()
```

### 3D surface (checkerboard)

Demonstrates plotting a 3D surface colored in a checkerboard pattern.




```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.ticker import LinearLocator

ax = plt.figure().add_subplot(projection='3d')

# Make data.
X = np.arange(-5, 5, 0.25)
xlen = len(X)
Y = np.arange(-5, 5, 0.25)
ylen = len(Y)
X, Y = np.meshgrid(X, Y)
R = np.sqrt(X**2 + Y**2)
```

(continues on next page)

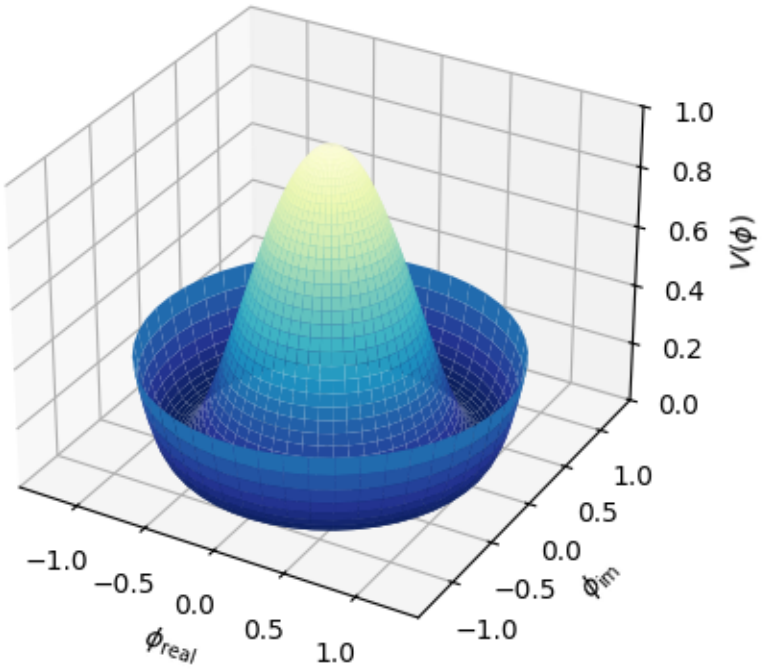
(continued from previous page)

```
Z = np.sin(R)

# Create an empty array of strings with the same shape as the meshgrid, and
# populate it with two colors in a checkerboard pattern.
colortuple = ('y', 'b')
colors = np.empty(X.shape, dtype=str)
for y in range(ylen):
    for x in range(xlen):
        colors[y, x] = colortuple[(x + y) % len(colortuple)]

# Plot the surface with face colors taken from the array we made.
surf = ax.plot_surface(X, Y, Z, facecolors=colors, linewidth=0)

# Customize the z axis.
ax.set_zlim(-1, 1)
ax.zaxis.set_major_locator(LinearLocator(6))

plt.show()
```

### 3D surface with polar coordinates

Demonstrates plotting a surface defined in polar coordinates. Uses the reversed version of the YlGnBu colormap. Also demonstrates writing axis labels with latex math mode.

Example contributed by Armin Moser.




```
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure()
ax = fig.add_subplot(projection='3d')

# Create the mesh in polar coordinates and compute corresponding Z.
r = np.linspace(0, 1.25, 50)
p = np.linspace(0, 2*np.pi, 50)
R, P = np.meshgrid(r, p)
Z = ((R**2 - 1)**2)

# Express the mesh in the cartesian system.
X, Y = R*np.cos(P), R*np.sin(P)

# Plot the surface.
ax.plot_surface(X, Y, Z, cmap=plt.cm.YlGnBu_r)

# Tweak the limits and add latex math labels.
ax.set_zlim(0, 1)
ax.set_xlabel(r'$\phi_\mathrm{real}$')
ax.set_ylabel(r'$\phi_\mathrm{im}$')
ax.set_zlabel(r'$V(\phi)$')
```

(continues on next page)

(continued from previous page)

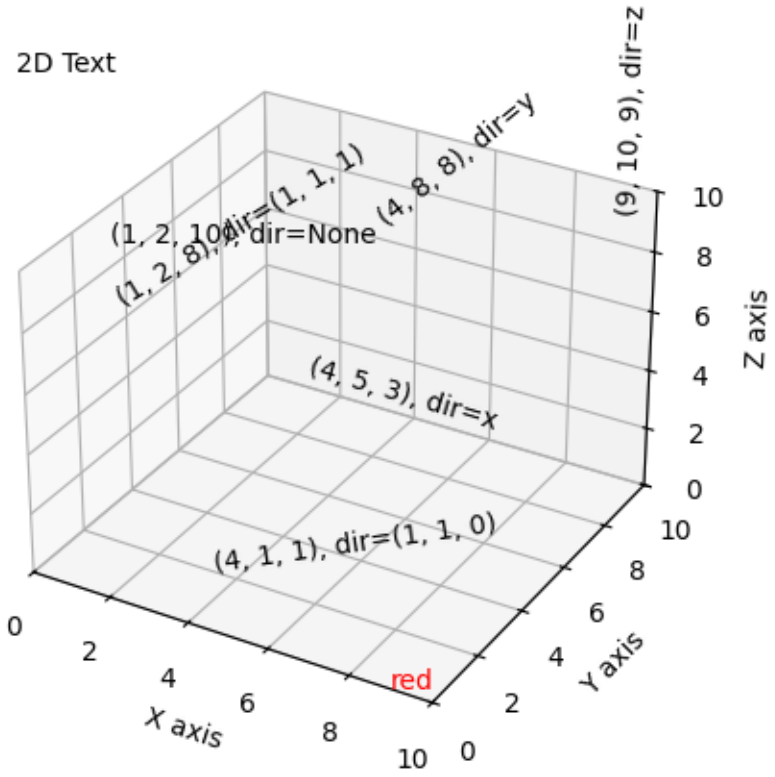
```
plt.show()
```

### Text annotations in 3D

Demonstrates the placement of text annotations on a 3D plot.

Functionality shown:

- Using the `text` function with three types of *zdir* values: None, an axis name (ex. 'x'), or a direction tuple (ex. (1, 1, 0)).
- Using the `text` function with the color keyword.
- Using the `text2D` function to place text on a fixed position on the ax object.




```
import matplotlib.pyplot as plt

ax = plt.figure().add_subplot(projection='3d')

# Demo 1: zdir
```

(continues on next page)

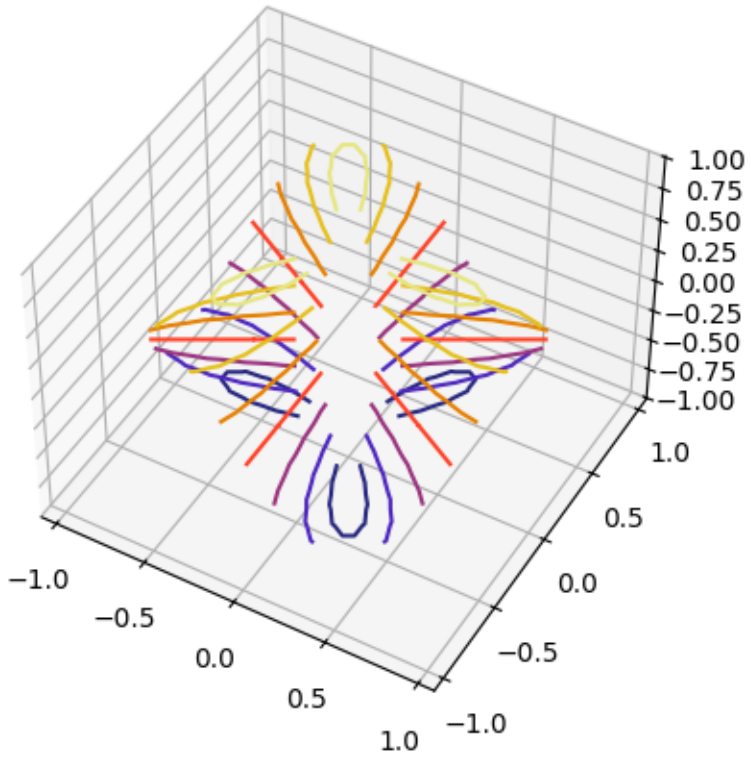
(continued from previous page)

```
zdirs = (None, 'x', 'y', 'z', (1, 1, 0), (1, 1, 1))
xs = (1, 4, 4, 9, 4, 1)
ys = (2, 5, 8, 10, 1, 2)
zs = (10, 3, 8, 9, 1, 8)

for zdir, x, y, z in zip(zdirs, xs, ys, zs):
    label = '(%d, %d, %d), dir=%s' % (x, y, z, zdir)
    ax.text(x, y, z, label, zdir)

# Demo 2: color
ax.text(9, 0, 0, "red", color='red')

# Demo 3: text2D
# Placement 0, 0 would be the bottom left, 1, 1 would be the top right.
ax.text2D(0.05, 0.95, "2D Text", transform=ax.transAxes)

# Tweaking display region and labels
ax.set_xlim(0, 10)
ax.set_ylim(0, 10)
ax.set_zlim(0, 10)
ax.set_xlabel('X axis')
ax.set_ylabel('Y axis')
ax.set_zlabel('Z axis')

plt.show()
```

### Triangular 3D contour plot

Contour plots of unstructured triangular grids.

The data used is the same as in the second plot of *More triangular 3D surfaces*. *Triangular 3D filled contour plot* shows the filled version of this example.




```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib.tri as tri

n_angles = 48
n_radii = 8
min_radius = 0.25

# Create the mesh in polar coordinates and compute x, y, z.
radii = np.linspace(min_radius, 0.95, n_radii)
angles = np.linspace(0, 2*np.pi, n_angles, endpoint=False)
angles = np.repeat(angles[..., np.newaxis], n_radii, axis=1)
angles[:, 1::2] += np.pi/n_angles

x = (radii*np.cos(angles)).flatten()
y = (radii*np.sin(angles)).flatten()
z = (np.cos(radii)*np.cos(3*angles)).flatten()

# Create a custom triangulation.
triang = tri.Triangulation(x, y)

# Mask off unwanted triangles.
```

(continues on next page)

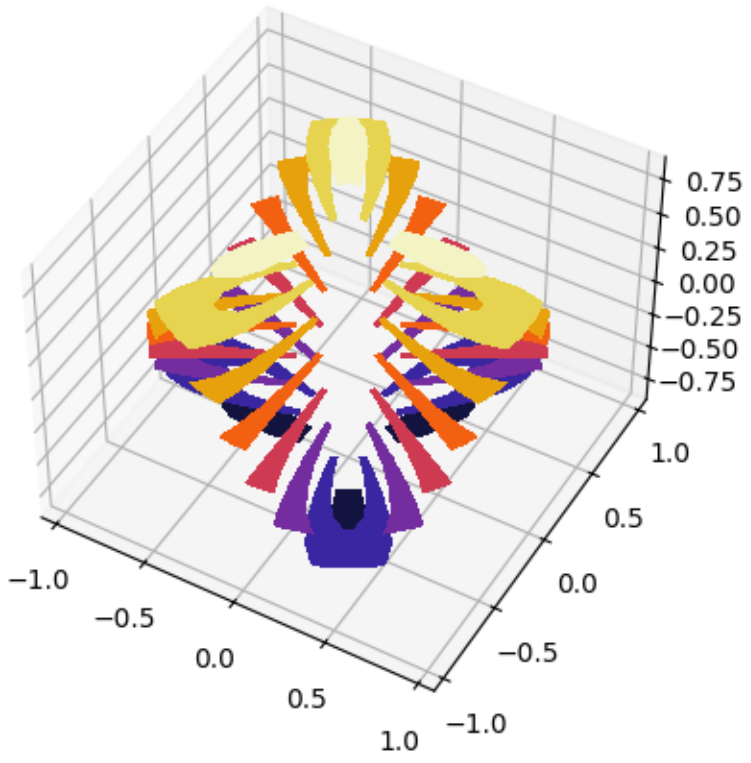
(continued from previous page)

```
triang.set_mask(np.hypot(x[triang.triangles].mean(axis=1),
                         y[triang.triangles].mean(axis=1))
                < min_radius)

ax = plt.figure().add_subplot(projection='3d')
ax.tricontour(triang, z, cmap=plt.cm.CMRmap)

# Customize the view angle so it's easier to understand the plot.
ax.view_init(elev=45.)

plt.show()
```

### Triangular 3D filled contour plot

Filled contour plots of unstructured triangular grids.

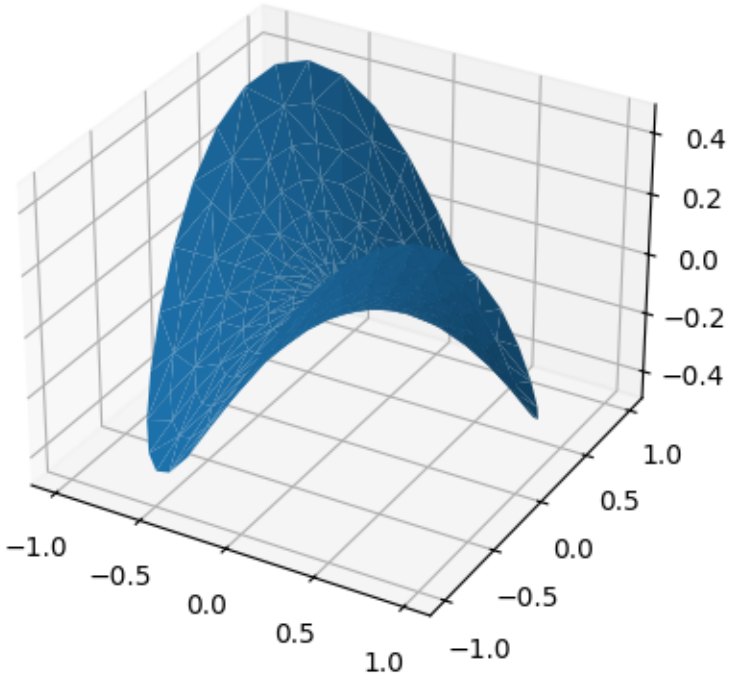
The data used is the same as in the second plot of *More triangular 3D surfaces*. *Triangular 3D contour plot* shows the unfilled version of this example.



```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib.tri as tri

# First create the x, y, z coordinates of the points.
n_angles = 48
n_radii = 8
min_radius = 0.25

# Create the mesh in polar coordinates and compute x, y, z.
radii = np.linspace(min_radius, 0.95, n_radii)
angles = np.linspace(0, 2*np.pi, n_angles, endpoint=False)
angles = np.repeat(angles[..., np.newaxis], n_radii, axis=1)
angles[:, 1::2] += np.pi/n_angles

x = (radii*np.cos(angles)).flatten()
y = (radii*np.sin(angles)).flatten()
z = (np.cos(radii)*np.cos(3*angles)).flatten()

# Create a custom triangulation.
triang = tri.Triangulation(x, y)

# Mask off unwanted triangles.
triang.set_mask(np.hypot(x[triang.triangles].mean(axis=1),
                         y[triang.triangles].mean(axis=1))
                < min_radius)

ax = plt.figure().add_subplot(projection='3d')
ax.tricontourf(triang, z, cmap=plt.cm.CMRmap)

# Customize the view angle so it's easier to understand the plot.
ax.view_init(elev=45.)

plt.show()
```

### Triangular 3D surfaces

Plot a 3D surface with a triangular mesh.




```
import matplotlib.pyplot as plt
import numpy as np

n_radii = 8
n_angles = 36

# Make radii and angles spaces (radius r=0 omitted to eliminate duplication).
radii = np.linspace(0.125, 1.0, n_radii)
angles = np.linspace(0, 2*np.pi, n_angles, endpoint=False)[..., np.newaxis]

# Convert polar (radii, angles) coords to cartesian (x, y) coords.
# (0, 0) is manually added at this stage,  so there will be no duplicate
# points in the (x, y) plane.
x = np.append(0, (radii*np.cos(angles)).flatten())
y = np.append(0, (radii*np.sin(angles)).flatten())

# Compute z to make the pringle surface.
z = np.sin(-x*y)

ax = plt.figure().add_subplot(projection='3d')

ax.plot_trisurf(x, y, z, linewidth=0.2, antialiased=True)
```

(continues on next page)

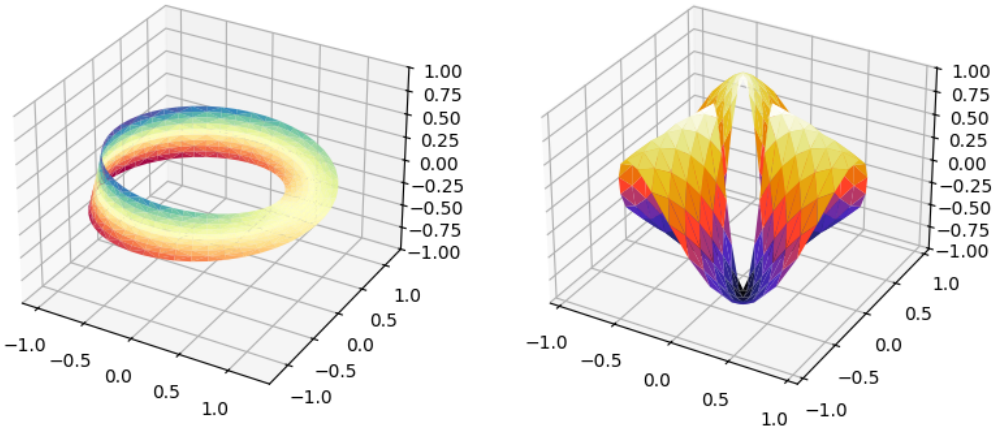
(continued from previous page)

```
plt.show()
```

### More triangular 3D surfaces

Two additional examples of plotting surfaces with triangular mesh.

The first demonstrates use of plot_trisurf's triangles argument, and the second sets a *Triangulation* object's mask and passes the object directly to plot_trisurf.




```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib.tri as mtri

fig = plt.figure(figsize=plt.figaspect(0.5))

# ==========
# First plot
# ==========

# Make a mesh in the space of parameterisation variables u and v
u = np.linspace(0, 2.0 * np.pi, endpoint=True, num=50)
v = np.linspace(-0.5, 0.5, endpoint=True, num=10)
u, v = np.meshgrid(u, v)
u, v = u.flatten(), v.flatten()

# This is the Mobius mapping, taking a u, v pair and returning an x, y, z
# triple
x = (1 + 0.5 * v * np.cos(u / 2.0)) * np.cos(u)
y = (1 + 0.5 * v * np.cos(u / 2.0)) * np.sin(u)
```

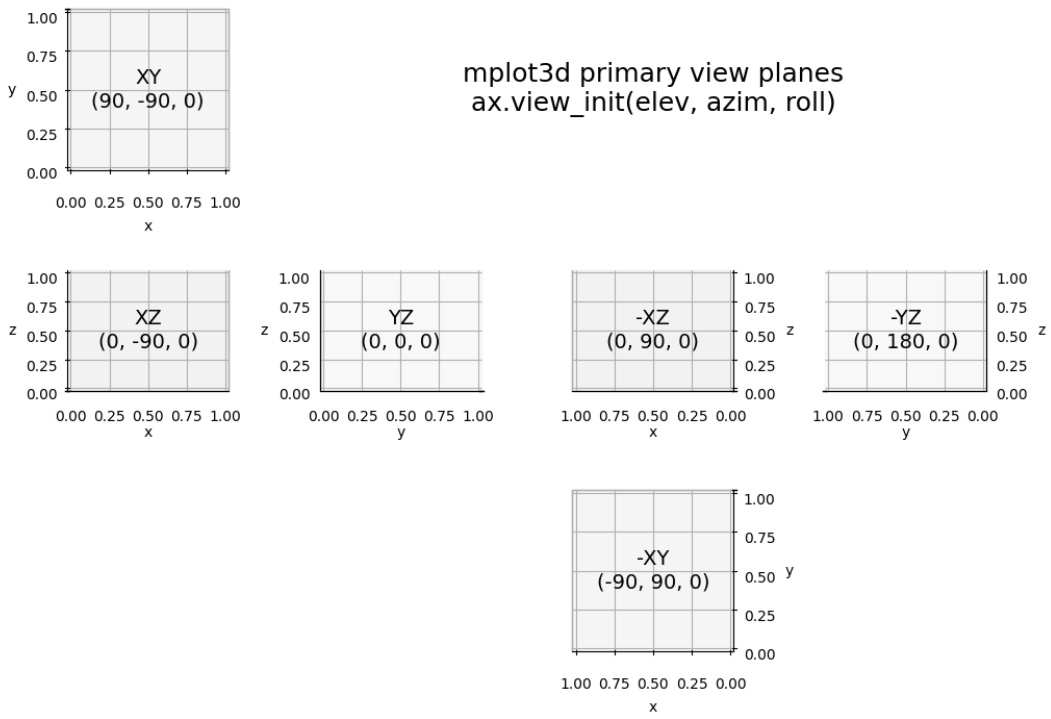
(continues on next page)

(continued from previous page)

```
z = 0.5 * v * np.sin(u / 2.0)

# Triangulate parameter space to determine the triangles
tri = mtri.Triangulation(u, v)

# Plot the surface.  The triangles in parameter space determine which x, y, z
# points are connected by an edge.
ax = fig.add_subplot(1, 2, 1, projection='3d')
ax.plot_trisurf(x, y, z, triangles=tri.triangles, cmap=plt.cm.Spectral)
ax.set_zlim(-1, 1)


# ===========
# Second plot
# ===========

# Make parameter spaces radii and angles.
n_angles = 36
n_radii = 8
min_radius = 0.25
radii = np.linspace(min_radius, 0.95, n_radii)

angles = np.linspace(0, 2*np.pi, n_angles, endpoint=False)
angles = np.repeat(angles[..., np.newaxis], n_radii, axis=1)
angles[:, 1::2] += np.pi/n_angles

# Map radius, angle pairs to x, y, z points.
x = (radii*np.cos(angles)).flatten()
y = (radii*np.sin(angles)).flatten()
z = (np.cos(radii)*np.cos(3*angles)).flatten()

# Create the Triangulation; no triangles so Delaunay triangulation created.
triang = mtri.Triangulation(x, y)

# Mask off unwanted triangles.
xmid = x[triang.triangles].mean(axis=1)
ymid = y[triang.triangles].mean(axis=1)
mask = xmid**2 + ymid**2 < min_radius**2
triang.set_mask(mask)

# Plot the surface.
ax = fig.add_subplot(1, 2, 2, projection='3d')
ax.plot_trisurf(triang, z, cmap=plt.cm.CMRmap)


plt.show()
```

### Primary 3D view planes

This example generates an "unfolded" 3D plot that shows each of the primary 3D view planes. The elevation, azimuth, and roll angles required for each view are labeled. You could print out this image and fold it into a box where each plane forms a side of the box.




```
import matplotlib.pyplot as plt


def annotate_axes(ax, text, fontsize=18):
    ax.text(x=0.5, y=0.5, z=0.5, s=text,
            va="center", ha="center", fontsize=fontsize, color="black")

# (plane, (elev, azim, roll))
views = [('XY',    (90, -90, 0)),
         ('XZ',     (0, -90, 0)),
         ('YZ',     (0,   0, 0)),
         ('-XY', (-90,  90, 0)),
         ('-XZ',    (0,  90, 0)),
         ('-YZ',    (0, 180, 0))]

layout = [['XY',  '.',    'L',    '.'],
          ['XZ', 'YZ', '-XZ', '-YZ'],
          ['.',    '.', '-XY',    '.']]
```

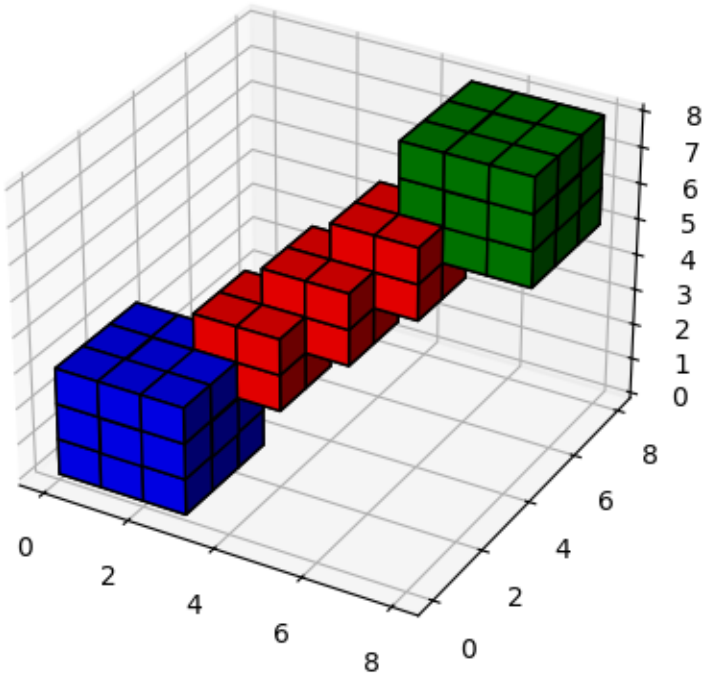
(continues on next page)

(continued from previous page)

```
fig, axd = plt.subplot_mosaic(layout, subplot_kw={'projection': '3d'},
                              figsize=(12, 8.5))
for plane, angles in views:
    axd[plane].set_xlabel('x')
    axd[plane].set_ylabel('y')
    axd[plane].set_zlabel('z')
    axd[plane].set_proj_type('ortho')
    axd[plane].view_init(elev=angles[0], azim=angles[1], roll=angles[2])
    axd[plane].set_box_aspect(None, zoom=1.25)

    label = f'{plane}\n{angles}'
    annotate_axes(axd[plane], label, fontsize=14)

for plane in ('XY', '-XY'):
    axd[plane].set_zticklabels([])
    axd[plane].set_zlabel('')
for plane in ('XZ', '-XZ'):
    axd[plane].set_yticklabels([])
    axd[plane].set_ylabel('')
for plane in ('YZ', '-YZ'):
    axd[plane].set_xticklabels([])
    axd[plane].set_xlabel('')

label = 'mplot3d primary view planes\n' + 'ax.view_init(elev, azim, roll)'
annotate_axes(axd['L'], label, fontsize=18)
axd['L'].set_axis_off()

plt.show()
```

### 3D voxel / volumetric plot

Demonstrates plotting 3D volumetric objects with *Axes3D.voxels*.




```
import matplotlib.pyplot as plt
import numpy as np

# prepare some coordinates
x, y, z = np.indices((8, 8, 8))

# draw cuboids in the top left and bottom right corners, and a link between
# them
cube1 = (x < 3) & (y < 3) & (z < 3)
cube2 = (x >= 5) & (y >= 5) & (z >= 5)
link = abs(x - y) + abs(y - z) + abs(z - x) <= 2

# combine the objects into a single boolean array
voxelarray = cube1 | cube2 | link

# set the colors of each object
colors = np.empty(voxelarray.shape, dtype=object)
colors[link] = 'red'
colors[cube1] = 'blue'
colors[cube2] = 'green'

# and plot everything
ax = plt.figure().add_subplot(projection='3d')
```

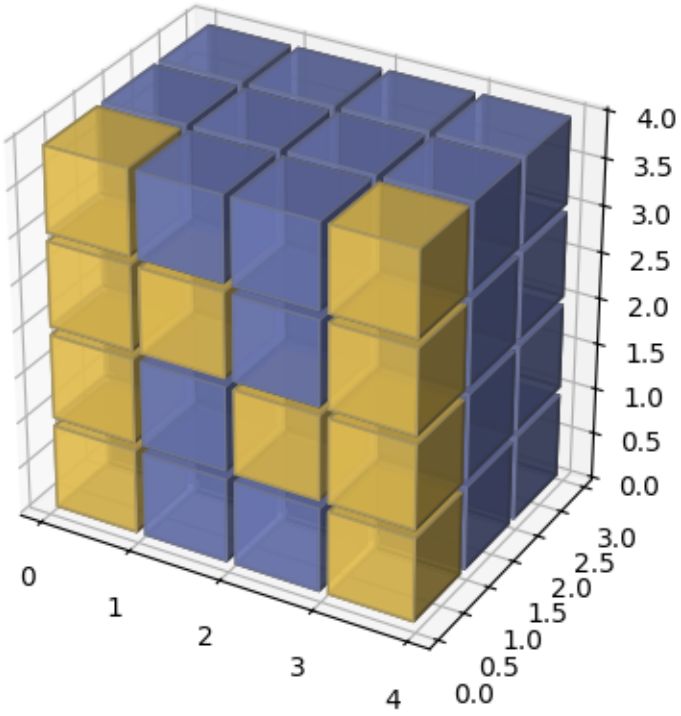
(continues on next page)

(continued from previous page)

```
ax.voxels(voxelarray, facecolors=colors, edgecolor='k')

plt.show()
```

### 3D voxel plot of the NumPy logo

Demonstrates using *Axes3D.voxels* with uneven coordinates.




```
import matplotlib.pyplot as plt
import numpy as np


def explode(data):
    size = np.array(data.shape)*2
    data_e = np.zeros(size - 1, dtype=data.dtype)
    data_e[::2, ::2, ::2] = data
    return data_e

# build up the numpy logo
n_voxels = np.zeros((4, 3, 4), dtype=bool)
```

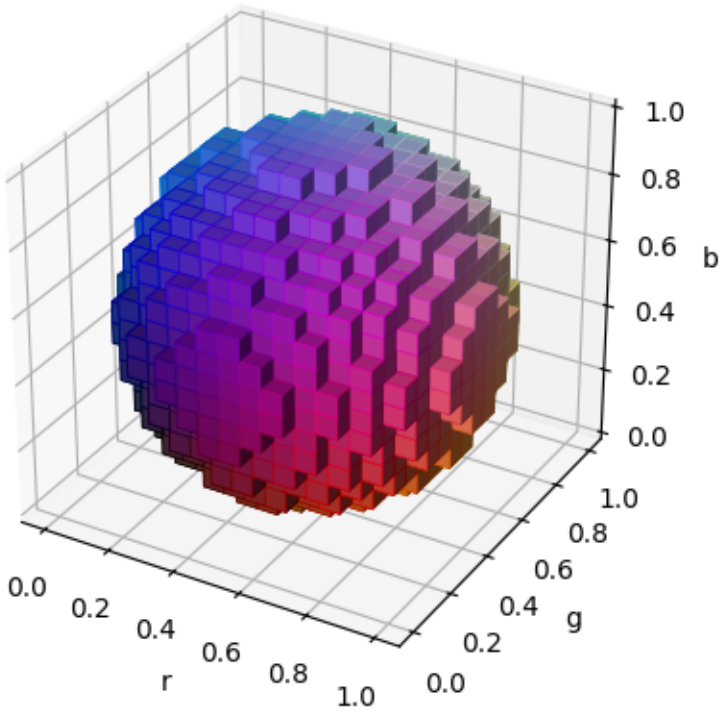
(continues on next page)

(continued from previous page)

```
n_voxels[0, 0, :] = True
n_voxels[-1, 0, :] = True
n_voxels[1, 0, 2] = True
n_voxels[2, 0, 1] = True
facecolors = np.where(n_voxels, '#FFD65DC0', '#7A88CCC0')
edgecolors = np.where(n_voxels, '#BFAB6E', '#7D84A6')
filled = np.ones(n_voxels.shape)

# upscale the above voxel image, leaving gaps
filled_2 = explode(filled)
fcolors_2 = explode(facecolors)
ecolors_2 = explode(edgecolors)

# Shrink the gaps
x, y, z = np.indices(np.array(filled_2.shape) + 1).astype(float) // 2
x[0::2, :, :] += 0.05
y[:, 0::2, :] += 0.05
z[:, :, 0::2] += 0.05
x[1::2, :, :] += 0.95
y[:, 1::2, :] += 0.95
z[:, :, 1::2] += 0.95

ax = plt.figure().add_subplot(projection='3d')
ax.voxels(x, y, z, filled_2, facecolors=fcolors_2, edgecolors=ecolors_2)
ax.set_aspect('equal')

plt.show()
```

### 3D voxel / volumetric plot with RGB colors

Demonstrates using *Axes3D.voxels* to visualize parts of a color space.




```
import matplotlib.pyplot as plt
import numpy as np


def midpoints(x):
    sl = ()
    for _ in range(x.ndim):
        x = (x[sl + np.index_exp[:-1]] + x[sl + np.index_exp[1:]]) / 2.0
        sl += np.index_exp[:]
    return x

# prepare some coordinates, and attach rgb values to each
r, g, b = np.indices((17, 17, 17)) / 16.0
rc = midpoints(r)
gc = midpoints(g)
bc = midpoints(b)

# define a sphere about [0.5, 0.5, 0.5]
sphere = (rc - 0.5)**2 + (gc - 0.5)**2 + (bc - 0.5)**2 < 0.5**2

# combine the color components
colors = np.zeros(sphere.shape + (3,))
colors[..., 0] = rc
```

(continues on next page)

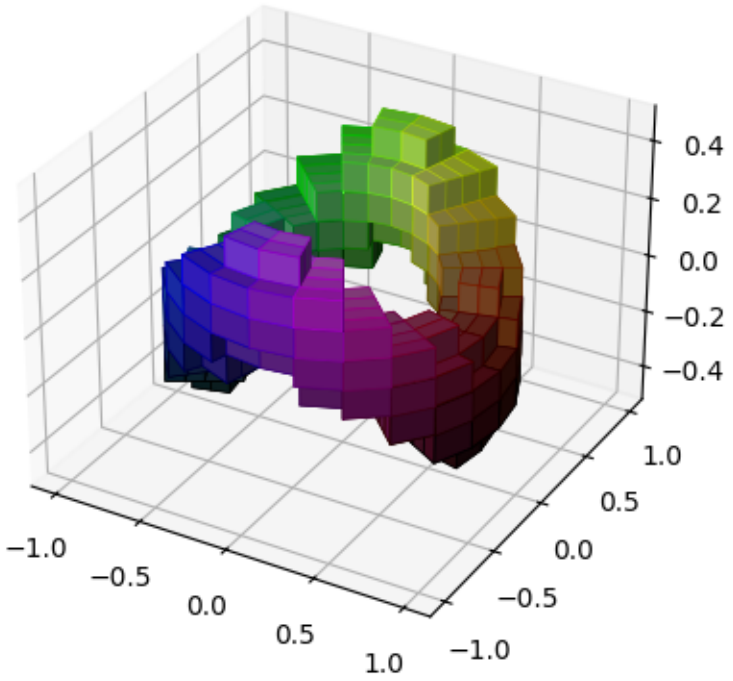
(continued from previous page)

```
colors[..., 1] = gc
colors[..., 2] = bc

# and plot everything
ax = plt.figure().add_subplot(projection='3d')
ax.voxels(r, g, b, sphere,
          facecolors=colors,
          edgecolors=np.clip(2*colors - 0.5, 0, 1),  # brighter
          linewidth=0.5)
ax.set(xlabel='r', ylabel='g', zlabel='b')
ax.set_aspect('equal')

plt.show()
```

**Total running time of the script:** (0 minutes 1.479 seconds)

### 3D voxel / volumetric plot with cylindrical coordinates

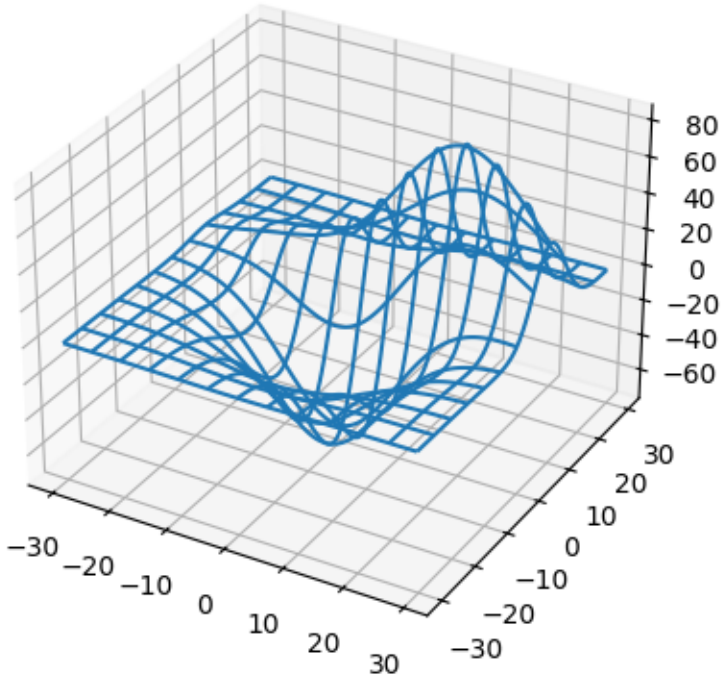
Demonstrates using the *x*, *y*, *z* parameters of *Axes3D.voxels*.



```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib.colors


def midpoints(x):
    sl = ()
    for i in range(x.ndim):
        x = (x[sl + np.index_exp[:-1]] + x[sl + np.index_exp[1:]]) / 2.0
        sl += np.index_exp[:]
    return x

# prepare some coordinates, and attach rgb values to each
r, theta, z = np.mgrid[0:1:11j, 0:np.pi*2:25j, -0.5:0.5:11j]
x = r*np.cos(theta)
y = r*np.sin(theta)

rc, thetac, zc = midpoints(r), midpoints(theta), midpoints(z)

# define a wobbly torus about [0.7, *, 0]
sphere = (rc - 0.7)**2 + (zc + 0.2*np.cos(thetac*2))**2 < 0.2**2

# combine the color components
hsv = np.zeros(sphere.shape + (3,))
hsv[..., 0] = thetac / (np.pi*2)
hsv[..., 1] = rc
hsv[..., 2] = zc + 0.5
colors = matplotlib.colors.hsv_to_rgb(hsv)

# and plot everything
ax = plt.figure().add_subplot(projection='3d')
ax.voxels(x, y, z, sphere,
          facecolors=colors,
          edgecolors=np.clip(2*colors - 0.5, 0, 1),  # brighter
          linewidth=0.5)

plt.show()
```

### 3D wireframe plot

A very basic demonstration of a wireframe plot.




```
import matplotlib.pyplot as plt

from mpl_toolkits.mplot3d import axes3d

fig = plt.figure()
ax = fig.add_subplot(projection='3d')

# Grab some test data.
X, Y, Z = axes3d.get_test_data(0.05)

# Plot a basic wireframe.
ax.plot_wireframe(X, Y, Z, rstride=10, cstride=10)

plt.show()
```

### Animate a 3D wireframe plot

A very simple "animation" of a 3D plot. See also *Rotating a 3D plot*.

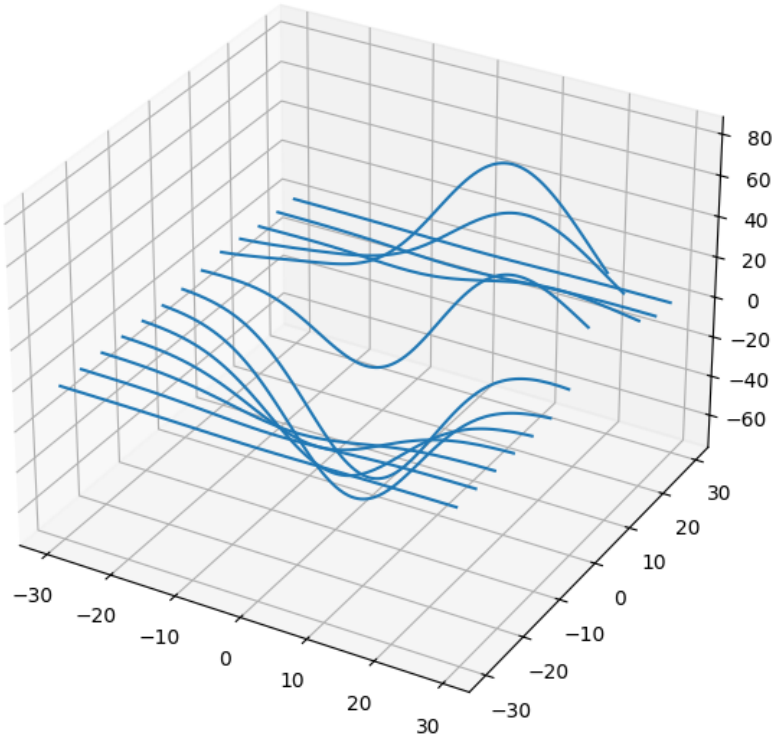
(This example is skipped when building the documentation gallery because it intentionally takes a long time to run.)

```
import time

import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure()
ax = fig.add_subplot(projection='3d')

# Make the X, Y meshgrid.
xs = np.linspace(-1, 1, 50)
ys = np.linspace(-1, 1, 50)
X, Y = np.meshgrid(xs, ys)

# Set the z axis limits, so they aren't recalculated each frame.
ax.set_zlim(-1, 1)

# Begin plotting.
wframe = None
tstart = time.time()
for phi in np.linspace(0, 180. / np.pi, 100):
    # If a line collection is already remove it before drawing.
    if wframe:
        wframe.remove()
    # Generate data.
    Z = np.cos(2 * np.pi * X + phi) * (1 - np.hypot(X, Y))
    # Plot the new wireframe and pause briefly before continuing.
    wframe = ax.plot_wireframe(X, Y, Z, rstride=2, cstride=2)
    plt.pause(.001)

print('Average FPS: %f' % (100 / (time.time() - tstart)))
```

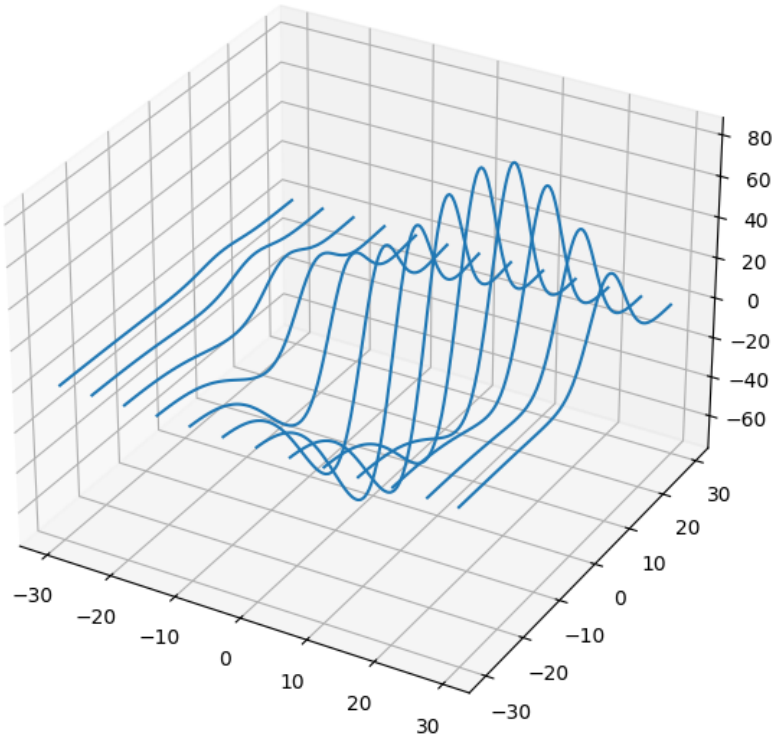
### 3D wireframe plots in one direction

Demonstrates that setting *rstride* or *cstride* to 0 causes wires to not be generated in the corresponding direction.

Column (x) stride set to 0




Row (y) stride set to 0



```
import matplotlib.pyplot as plt

from mpl_toolkits.mplot3d import axes3d

fig, (ax1, ax2) = plt.subplots(
    2, 1, figsize=(8, 12), subplot_kw={'projection': '3d'})

# Get the test data
X, Y, Z = axes3d.get_test_data(0.05)

# Give the first plot only wireframes of the type y = c
ax1.plot_wireframe(X, Y, Z, rstride=10, cstride=0)
ax1.set_title("Column (x) stride set to 0")

# Give the second plot only wireframes of the type x = c
ax2.plot_wireframe(X, Y, Z, rstride=0, cstride=10)
ax2.set_title("Row (y) stride set to 0")

plt.tight_layout()
plt.show()
```

**Total running time of the script:** (0 minutes 1.259 seconds)

### 6.25.18 Scales

These examples cover how different scales are handled in Matplotlib.

#### Asinh Demo

Illustration of the *asinh* axis scaling, which uses the transformation

$$a \to a_0 \sinh^{-1}(a/a_0)$$

For coordinate values close to zero (i.e. much smaller than the "linear width" $a_0$), this leaves values essentially unchanged:

$$a \to a + \mathcal{O}(a^3)$$

but for larger values (i.e. $|a| \gg a_0$, this is asymptotically

$$a \to a_0 \operatorname{sgn}(a) \ln |a| + \mathcal{O}(1)$$

As with the *symlog* scaling, this allows one to plot quantities that cover a very wide dynamic range that includes both positive and negative values. However, `symlog` involves a transformation that has discontinuities in its gradient because it is built from *separate* linear and logarithmic transformations. The `asinh` scaling uses a transformation that is smooth for all (finite) values, which is both mathematically cleaner and reduces visual artifacts associated with an abrupt transition between linear and logarithmic regions of the plot.

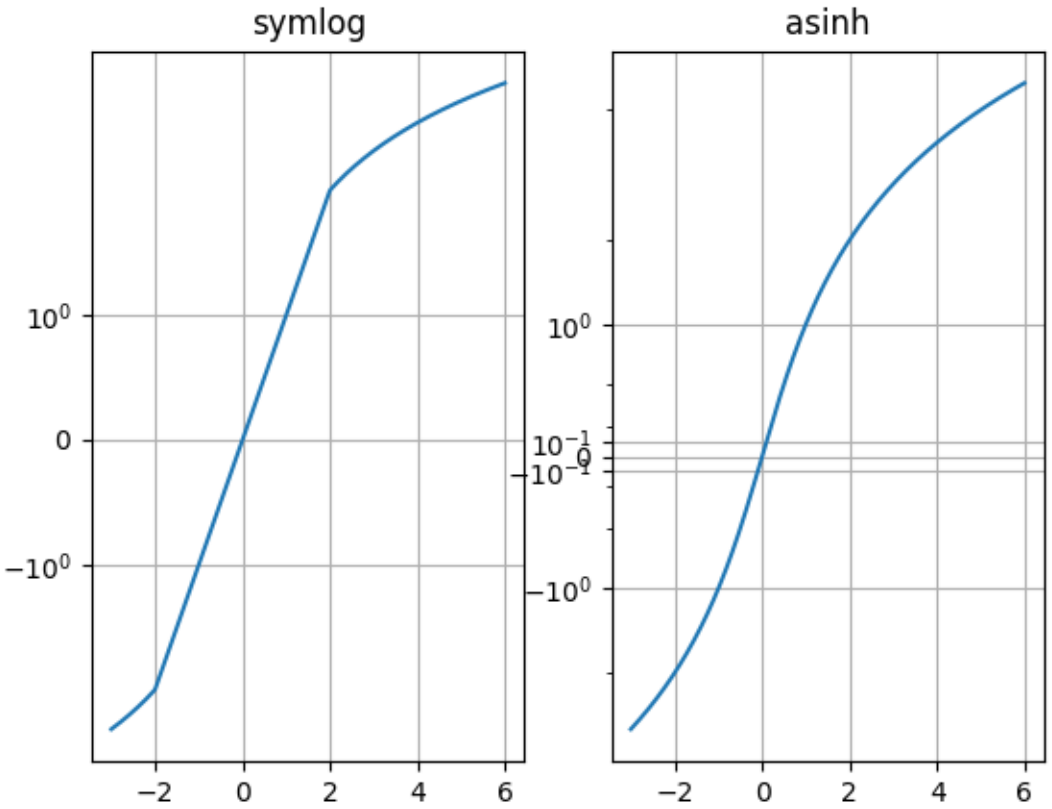
**Note:** *scale.AsinhScale* is experimental, and the API may change.

See *AsinhScale*, *SymmetricalLogScale*.

```
import matplotlib.pyplot as plt
import numpy as np

# Prepare sample values for variations on y=x graph:
x = np.linspace(-3, 6, 500)
```

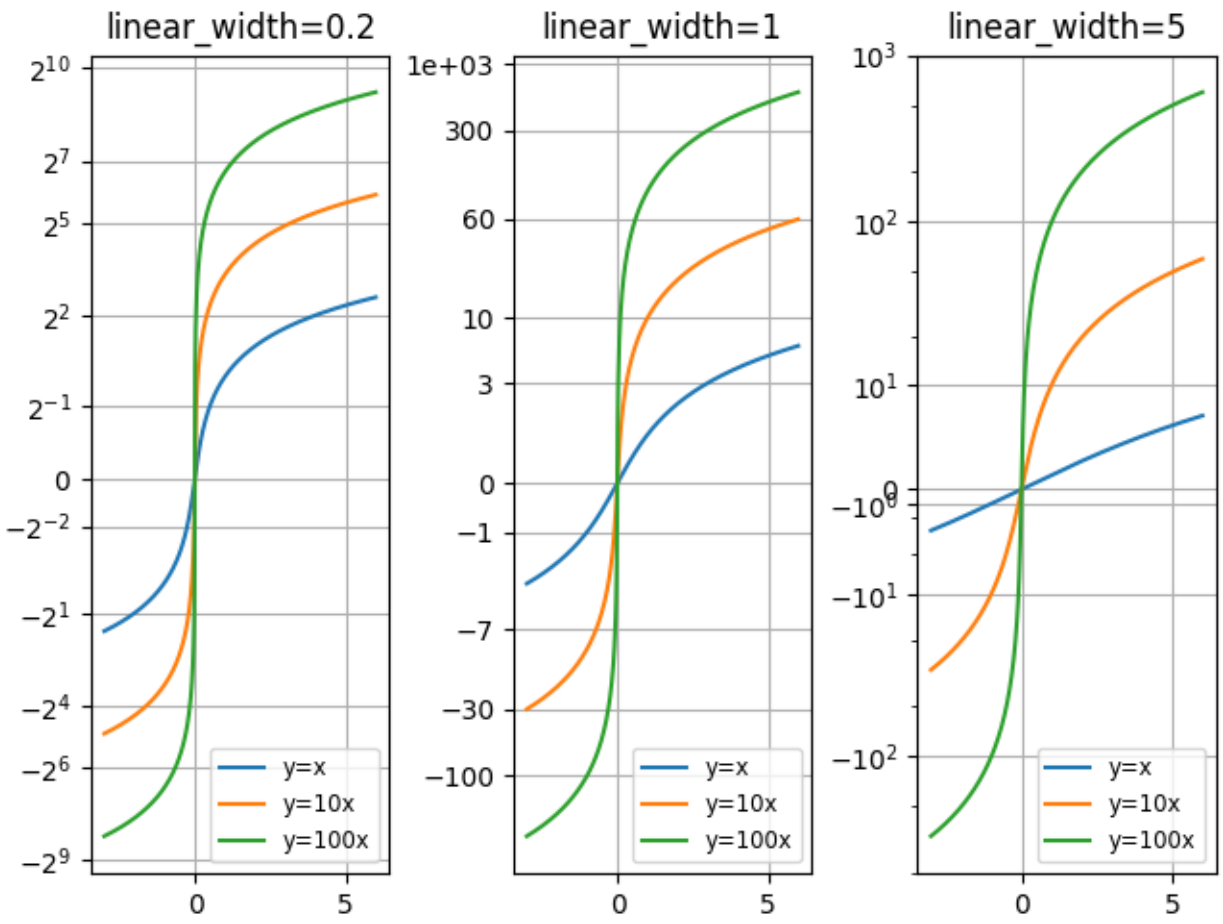
Compare "symlog" and "asinh" behaviour on sample y=x graph, where there is a discontinuous gradient in "symlog" near y=2:

```
fig1 = plt.figure()
ax0, ax1 = fig1.subplots(1, 2, sharex=True)

ax0.plot(x, x)
ax0.set_yscale('symlog')
ax0.grid()
ax0.set_title('symlog')

ax1.plot(x, x)
ax1.set_yscale('asinh')
ax1.grid()
ax1.set_title('asinh')
```



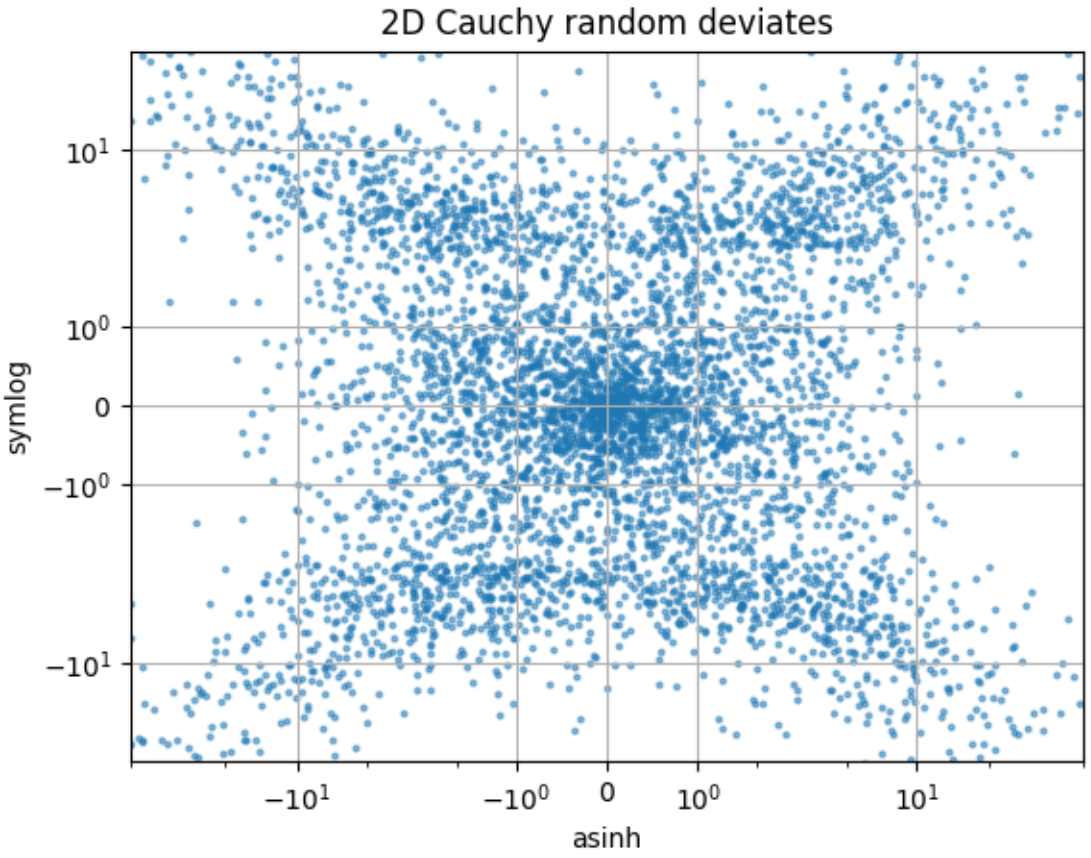

Compare "asinh" graphs with different scale parameter "linear_width":

```
fig2 = plt.figure(layout='constrained')
axs = fig2.subplots(1, 3, sharex=True)
for ax, (a0, base) in zip(axs, ((0.2, 2), (1.0, 0), (5.0, 10))):
    ax.set_title(f'linear_width={a0:.3g}')
    ax.plot(x, x, label='y=x')
    ax.plot(x, 10*x, label='y=10x')
    ax.plot(x, 100*x, label='y=100x')
    ax.set_yscale('asinh', linear_width=a0, base=base)
    ax.grid()
    ax.legend(loc='best', fontsize='small')
```




Compare "symlog" and "asinh" scalings on 2D Cauchy-distributed random numbers, where one may be able to see more subtle artifacts near y=2 due to the gradient-discontinuity in "symlog":

```
fig3 = plt.figure()
ax = fig3.subplots(1, 1)
r = 3 * np.tan(np.random.uniform(-np.pi / 2.02, np.pi / 2.02,
                                 size=(5000,)))
th = np.random.uniform(0, 2*np.pi, size=r.shape)

ax.scatter(r * np.cos(th), r * np.sin(th), s=4, alpha=0.5)
ax.set_xscale('asinh')
ax.set_yscale('symlog')
ax.set_xlabel('asinh')
ax.set_ylabel('symlog')
ax.set_title('2D Cauchy random deviates')
ax.set_xlim(-50, 50)
ax.set_ylim(-50, 50)
ax.grid()

plt.show()
```

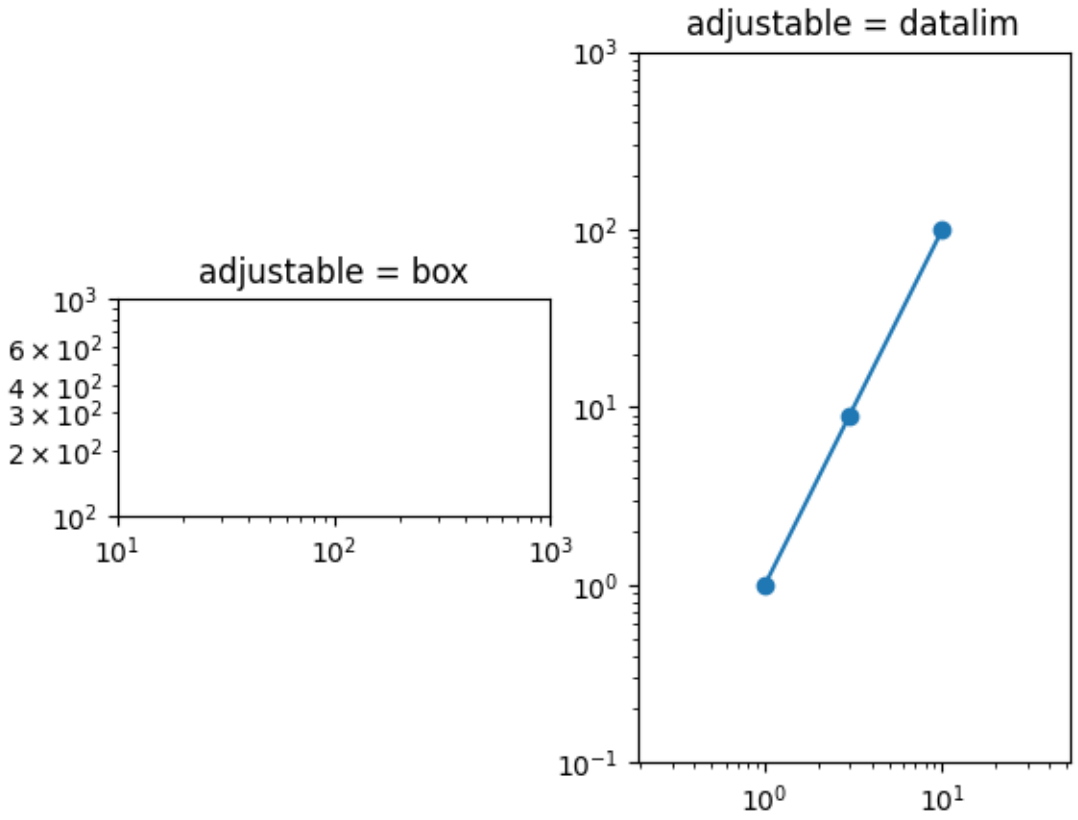



**References**

- *matplotlib.scale.AsinhScale*
- *matplotlib.ticker.AsinhLocator*
- *matplotlib.scale.SymmetricalLogScale*

**Total running time of the script:** (0 minutes 1.671 seconds)

### Loglog Aspect

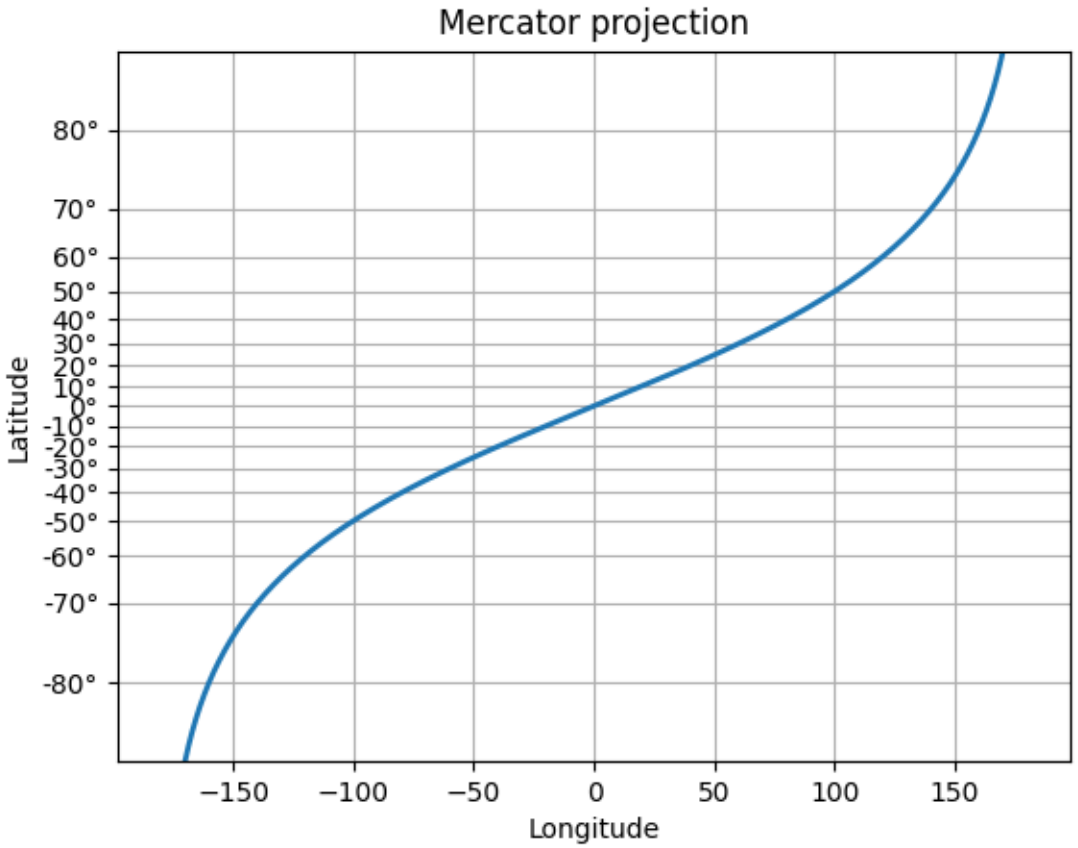



```
import matplotlib.pyplot as plt

fig, (ax1, ax2) = plt.subplots(1, 2)
ax1.set_xscale("log")
ax1.set_yscale("log")
ax1.set_xlim(1e1, 1e3)
ax1.set_ylim(1e2, 1e3)
ax1.set_aspect(1)
ax1.set_title("adjustable = box")

ax2.set_xscale("log")
ax2.set_yscale("log")
ax2.set_adjustable("datalim")
ax2.plot([1, 3, 10], [1, 9, 100], "o-")
ax2.set_xlim(1e-1, 1e2)
ax2.set_ylim(1e-1, 1e3)
ax2.set_aspect(1)
ax2.set_title("adjustable = datalim")

plt.show()
```

### Custom scale

Create a custom scale, by implementing the scaling use for latitude data in a Mercator Projection.

Unless you are making special use of the *Transform* class, you probably don't need to use this verbose method, and instead can use *FuncScale* and the `'function'` option of *set_xscale* and *set_yscale*. See the last example in *Scales*.




```
import numpy as np
from numpy import ma

from matplotlib import scale as mscale
from matplotlib import transforms as mtransforms
from matplotlib.ticker import FixedLocator, FuncFormatter


class MercatorLatitudeScale(mscale.ScaleBase):
    """
    Scales data in range -pi/2 to pi/2 (-90 to 90 degrees) using
    the system used to scale latitudes in a Mercator__ projection.

    The scale function:
      ln(tan(y) + sec(y))
```

(continues on next page)

(continued from previous page)

```
    The inverse scale function:
      atan(sinh(y))

    Since the Mercator scale tends to infinity at +/- 90 degrees,
    there is user-defined threshold, above and below which nothing
    will be plotted.  This defaults to +/- 85 degrees.

    __ https://en.wikipedia.org/wiki/Mercator_projection
    """

    # The scale class must have a member ``name`` that defines the string used
    # to select the scale.  For example, ``ax.set_yscale("mercator")`` would␣
↪be
    # used to select this scale.
    name = 'mercator'

    def __init__(self, axis, *, thresh=np.deg2rad(85), **kwargs):
        """
        Any keyword arguments passed to ``set_xscale`` and ``set_yscale`` will
        be passed along to the scale's constructor.

        thresh: The degree above which to crop the data.
        """
        super().__init__(axis)
        if thresh >= np.pi / 2:
            raise ValueError("thresh must be less than pi/2")
        self.thresh = thresh

    def get_transform(self):
        """
        Override this method to return a new instance that does the
        actual transformation of the data.

        The MercatorLatitudeTransform class is defined below as a
        nested class of this one.
        """
        return self.MercatorLatitudeTransform(self.thresh)

    def set_default_locators_and_formatters(self, axis):
        """
        Override to set up the locators and formatters to use with the
        scale.  This is only required if the scale requires custom
        locators and formatters.  Writing custom locators and
        formatters is rather outside the scope of this example, but
        there are many helpful examples in :mod:`.ticker`.

        In our case, the Mercator example uses a fixed locator from -90 to 90
        degrees and a custom formatter to convert the radians to degrees and
        put a degree symbol after the value.
        """
        fmt = FuncFormatter(
```

(continues on next page)

(continued from previous page)

```
            lambda x, pos=None: f"{np.degrees(x):.0f}\N{DEGREE SIGN}")
        axis.set(major_locator=FixedLocator(np.radians(range(-90, 90, 10))),
                 major_formatter=fmt, minor_formatter=fmt)

    def limit_range_for_scale(self, vmin, vmax, minpos):
        """
        Override to limit the bounds of the axis to the domain of the
        transform.  In the case of Mercator, the bounds should be
        limited to the threshold that was passed in.  Unlike the
        autoscaling provided by the tick locators, this range limiting
        will always be adhered to, whether the axis range is set
        manually, determined automatically or changed through panning
        and zooming.
        """
        return max(vmin, -self.thresh), min(vmax, self.thresh)

    class MercatorLatitudeTransform(mtransforms.Transform):
        # There are two value members that must be defined.
        # ``input_dims`` and ``output_dims`` specify number of input
        # dimensions and output dimensions to the transformation.
        # These are used by the transformation framework to do some
        # error checking and prevent incompatible transformations from
        # being connected together.  When defining transforms for a
        # scale, which are, by definition, separable and have only one
        # dimension, these members should always be set to 1.
        input_dims = output_dims = 1

        def __init__(self, thresh):
            mtransforms.Transform.__init__(self)
            self.thresh = thresh

        def transform_non_affine(self, a):
            """
            This transform takes a numpy array and returns a transformed copy.
            Since the range of the Mercator scale is limited by the
            user-specified threshold, the input array must be masked to
            contain only valid values.  Matplotlib will handle masked arrays
            and remove the out-of-range data from the plot.  However, the
            returned array *must* have the same shape as the input array,␣
↪since
            these values need to remain synchronized with values in the other
            dimension.
            """
            masked = ma.masked_where((a < -self.thresh) | (a > self.thresh),␣
↪a)
            if masked.mask.any():
                return ma.log(np.abs(ma.tan(masked) + 1 / ma.cos(masked)))
            else:
                return np.log(np.abs(np.tan(a) + 1 / np.cos(a)))

        def inverted(self):
            """
```

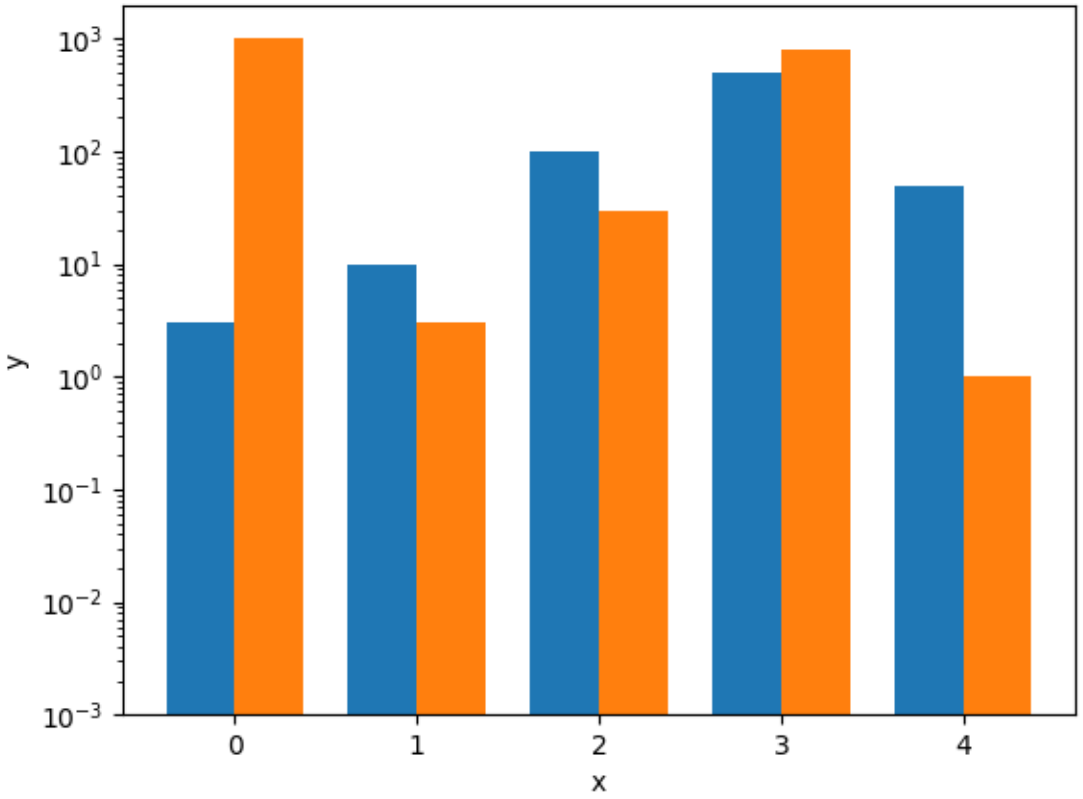
(continues on next page)

(continued from previous page)

```
            Override this method so Matplotlib knows how to get the
            inverse transform for this transform.
            """
            return MercatorLatitudeScale.InvertedMercatorLatitudeTransform(
                self.thresh)

    class InvertedMercatorLatitudeTransform(mtransforms.Transform):
        input_dims = output_dims = 1

        def __init__(self, thresh):
            mtransforms.Transform.__init__(self)
            self.thresh = thresh

        def transform_non_affine(self, a):
            return np.arctan(np.sinh(a))

        def inverted(self):
            return MercatorLatitudeScale.MercatorLatitudeTransform(self.
 ↪thresh)


# Now that the Scale class has been defined, it must be registered so
# that Matplotlib can find it.
mscale.register_scale(MercatorLatitudeScale)


if __name__ == '__main__':
    import matplotlib.pyplot as plt

    t = np.arange(-180.0, 180.0, 0.1)
    s = np.radians(t)/2.

    plt.plot(t, s, '-', lw=2)
    plt.yscale('mercator')

    plt.xlabel('Longitude')
    plt.ylabel('Latitude')
    plt.title('Mercator projection')
    plt.grid(True)

    plt.show()
```

### Log Bar

Plotting a bar chart with a logarithmic y-axis.




```
import matplotlib.pyplot as plt
import numpy as np

data = ((3, 1000), (10, 3), (100, 30), (500, 800), (50, 1))

dim = len(data[0])
w = 0.75
dimw = w / dim

fig, ax = plt.subplots()
x = np.arange(len(data))
for i in range(len(data[0])):
    y = [d[i] for d in data]
    b = ax.bar(x + i * dimw, y, dimw, bottom=0.001)

ax.set_xticks(x + dimw / 2, labels=map(str, x))
ax.set_yscale('log')

ax.set_xlabel('x')
```

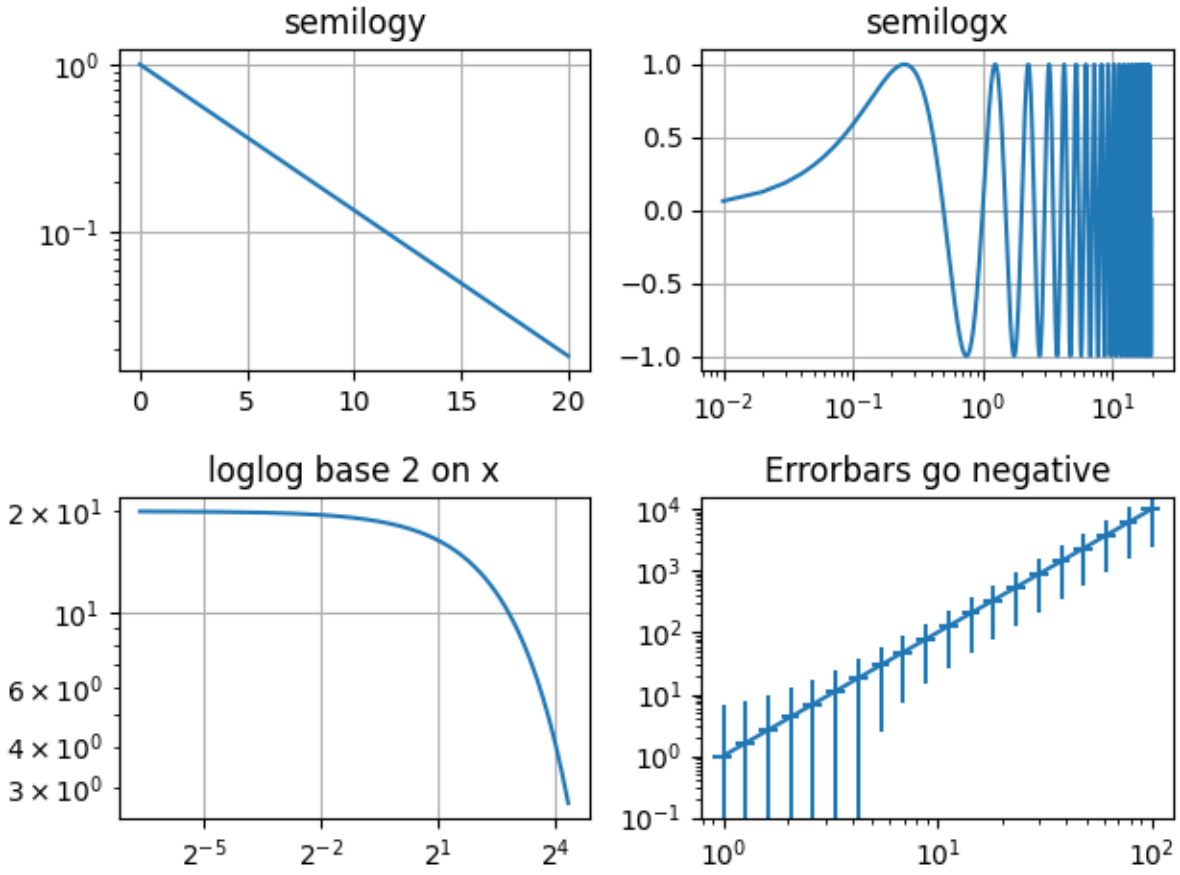
(continues on next page)

(continued from previous page)

```
ax.set_ylabel('y')

plt.show()
```

### Log Demo

Examples of plots with logarithmic axes.




```
import matplotlib.pyplot as plt
import numpy as np

# Data for plotting
t = np.arange(0.01, 20.0, 0.01)

# Create figure
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2)

# log y axis
ax1.semilogy(t, np.exp(-t / 5.0))
ax1.set(title='semilogy')
```

(continues on next page)

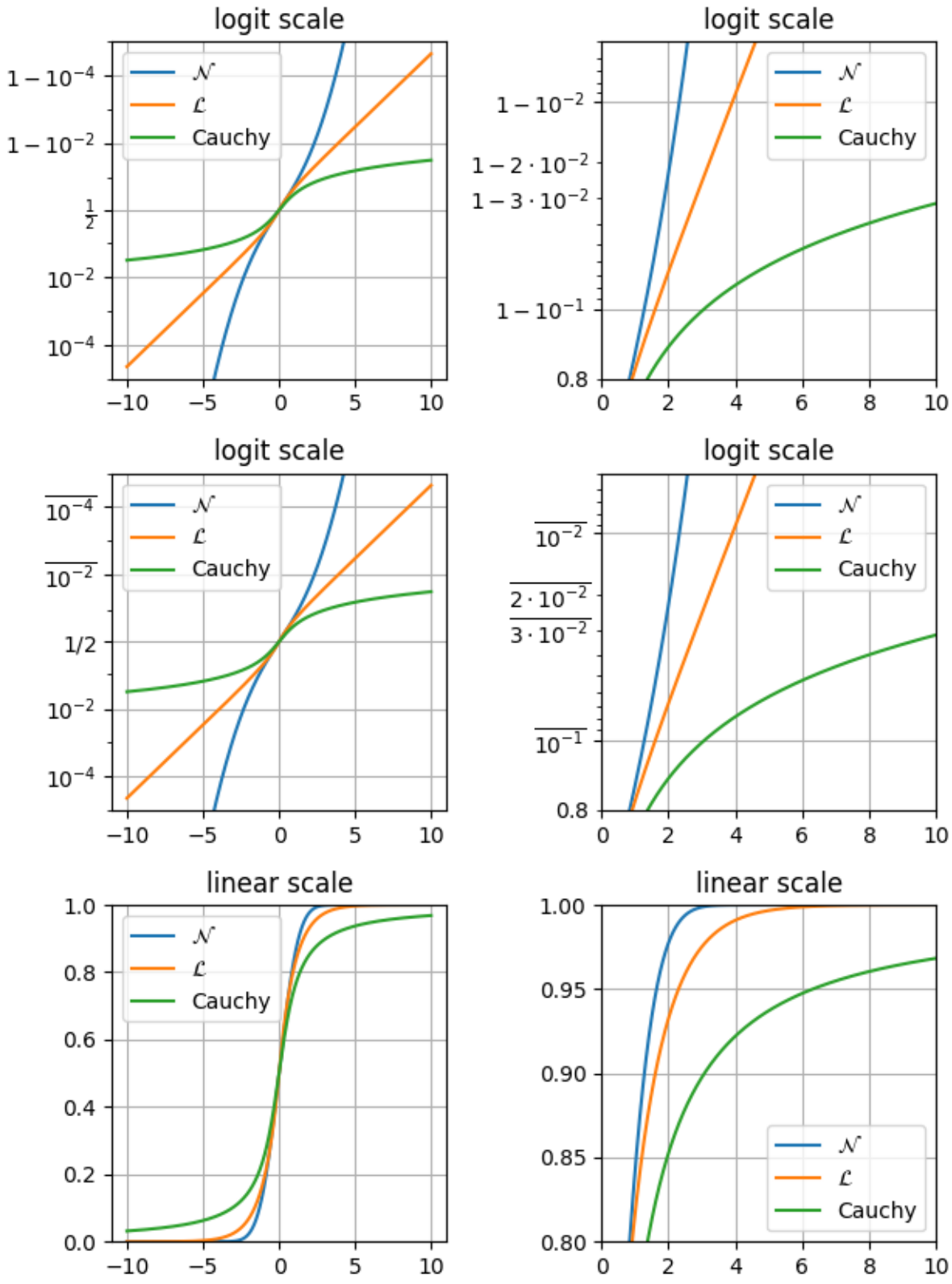
(continued from previous page)

```
ax1.grid()

# log x axis
ax2.semilogx(t, np.sin(2 * np.pi * t))
ax2.set(title='semilogx')
ax2.grid()

# log x and y axis
ax3.loglog(t, 20 * np.exp(-t / 10.0))
ax3.set_xscale('log', base=2)
ax3.set(title='loglog base 2 on x')
ax3.grid()

# With errorbars: clip non-positive values
# Use new data for plotting
x = 10.0**np.linspace(0.0, 2.0, 20)
y = x**2.0

ax4.set_xscale("log", nonpositive='clip')
ax4.set_yscale("log", nonpositive='clip')
ax4.set(title='Errorbars go negative')
ax4.errorbar(x, y, xerr=0.1 * x, yerr=5.0 + 0.75 * y)
# ylim must be set after errorbar to allow errorbar to autoscale limits
ax4.set_ylim(bottom=0.1)

fig.tight_layout()
plt.show()
```

**Total running time of the script:** (0 minutes 1.193 seconds)

### Logit Demo

Examples of plots with logit axes.


logit scale
$1-10^{-4}$
$1-10^{-2}$
$\frac{1}{2}$
$10^{-2}$
$10^{-4}$
−10
−5
0
5
10
$\mathcal{N}$
$\mathcal{L}$
Cauchy
logit scale
$1-10^{-2}$
$1-2\cdot10^{-2}$
$1-3\cdot10^{-2}$
$1-10^{-1}$
0.8
0
2
4
6
8
10
$\mathcal{N}$
$\mathcal{L}$
Cauchy
logit scale
$10^{-4}$
$10^{-2}$
1/2
$10^{-2}$
$10^{-4}$
−10
−5
0
5
10
$\mathcal{N}$
$\mathcal{L}$
Cauchy
logit scale
$10^{-2}$
$2\cdot10^{-2}$
$3\cdot10^{-2}$
$10^{-1}$
0.8
0
2
4
6
8
10
$\mathcal{N}$
$\mathcal{L}$
Cauchy
linear scale
1.0
0.8
0.6
0.4
0.2
0.0
−10
−5
0
5
10
$\mathcal{N}$
$\mathcal{L}$
Cauchy
linear scale
1.00
0.95
0.90
0.85
0.80
0
2
4
6
8
10
$\mathcal{N}$
$\mathcal{L}$
Cauchy

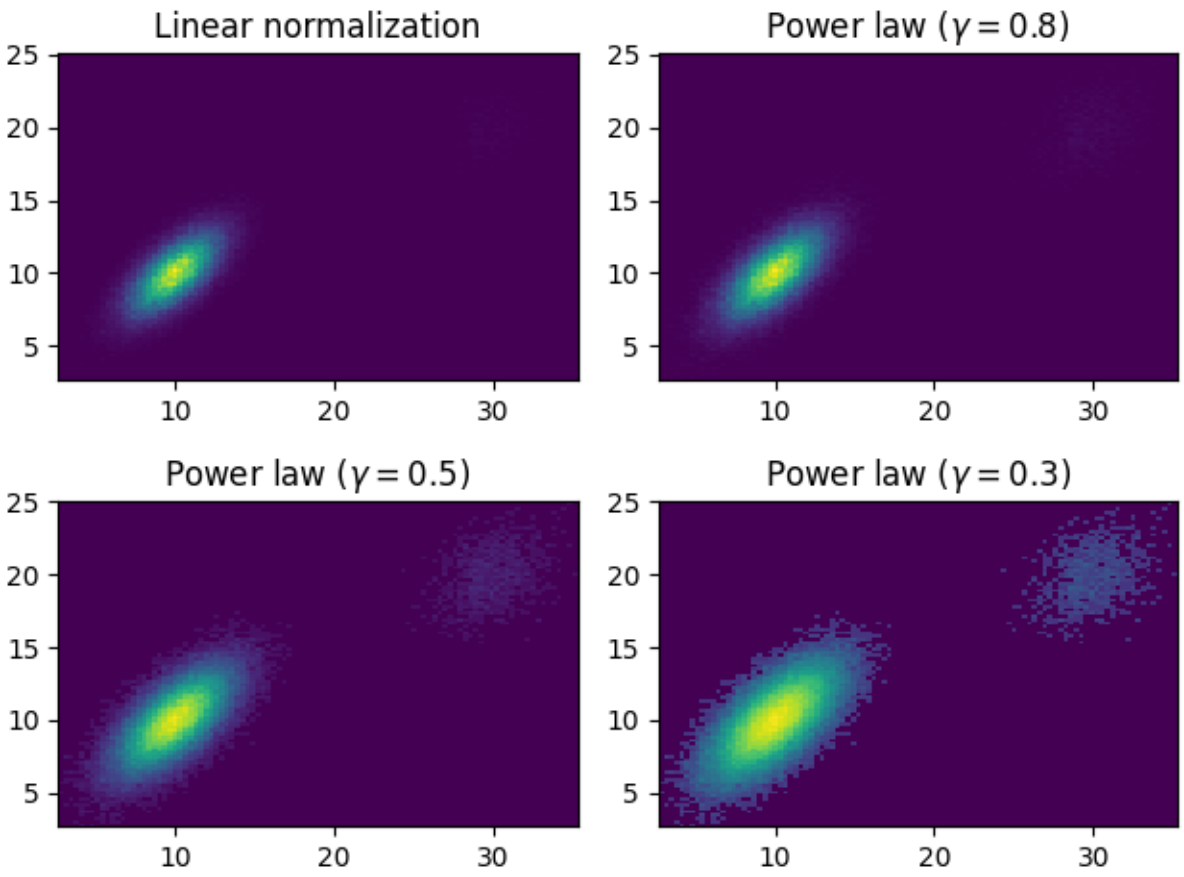
```
import math

import matplotlib.pyplot as plt
import numpy as np

xmax = 10
x = np.linspace(-xmax, xmax, 10000)
cdf_norm = [math.erf(w / np.sqrt(2)) / 2 + 1 / 2 for w in x]
cdf_laplacian = np.where(x < 0, 1 / 2 * np.exp(x), 1 - 1 / 2 * np.exp(-x))
cdf_cauchy = np.arctan(x) / np.pi + 1 / 2

fig, axs = plt.subplots(nrows=3, ncols=2, figsize=(6.4, 8.5))

# Common part, for the example, we will do the same plots on all graphs
for i in range(3):
    for j in range(2):
        axs[i, j].plot(x, cdf_norm, label=r"$\mathcal{N}$")
        axs[i, j].plot(x, cdf_laplacian, label=r"$\mathcal{L}$")
        axs[i, j].plot(x, cdf_cauchy, label="Cauchy")
        axs[i, j].legend()
        axs[i, j].grid()

# First line, logitscale, with standard notation
axs[0, 0].set(title="logit scale")
axs[0, 0].set_yscale("logit")
axs[0, 0].set_ylim(1e-5, 1 - 1e-5)

axs[0, 1].set(title="logit scale")
axs[0, 1].set_yscale("logit")
axs[0, 1].set_xlim(0, xmax)
axs[0, 1].set_ylim(0.8, 1 - 5e-3)

# Second line, logitscale, with survival notation (with `use_overline`), and
# other format display 1/2
axs[1, 0].set(title="logit scale")
axs[1, 0].set_yscale("logit", one_half="1/2", use_overline=True)
axs[1, 0].set_ylim(1e-5, 1 - 1e-5)

axs[1, 1].set(title="logit scale")
axs[1, 1].set_yscale("logit", one_half="1/2", use_overline=True)
axs[1, 1].set_xlim(0, xmax)
axs[1, 1].set_ylim(0.8, 1 - 5e-3)

# Third line, linear scale
axs[2, 0].set(title="linear scale")
axs[2, 0].set_ylim(0, 1)

axs[2, 1].set(title="linear scale")
axs[2, 1].set_xlim(0, xmax)
axs[2, 1].set_ylim(0.8, 1)

fig.tight_layout()
plt.show()
```

**Total running time of the script:** (0 minutes 1.570 seconds)

### Exploring normalizations

Various normalization on a multivariate normal distribution.

```
import matplotlib.pyplot as plt
import numpy as np
from numpy.random import multivariate_normal

import matplotlib.colors as mcolors

# Fixing random state for reproducibility.
np.random.seed(19680801)

data = np.vstack([
    multivariate_normal([10, 10], [[3, 2], [2, 3]], size=100000),
    multivariate_normal([30, 20], [[3, 1], [1, 3]], size=1000)
])

gammas = [0.8, 0.5, 0.3]

fig, axs = plt.subplots(nrows=2, ncols=2)

axs[0, 0].set_title('Linear normalization')
axs[0, 0].hist2d(data[:, 0], data[:, 1], bins=100)

for ax, gamma in zip(axs.flat[1:], gammas):
    ax.set_title(r'Power law $(\gamma=%1.1f)$' % gamma)
    ax.hist2d(data[:, 0], data[:, 1], bins=100, norm=mcolors.PowerNorm(gamma))

fig.tight_layout()

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.colors*
- *matplotlib.colors.PowerNorm*
- *matplotlib.axes.Axes.hist2d*
- *matplotlib.pyplot.hist2d*

## Scales

Illustrate the scale transformations applied to axes, e.g. log, symlog, logit.

The last two examples are examples of using the `'function'` scale by supplying forward and inverse functions for the scale transformation.

```
import matplotlib.pyplot as plt
import numpy as np
```

(continues on next page)

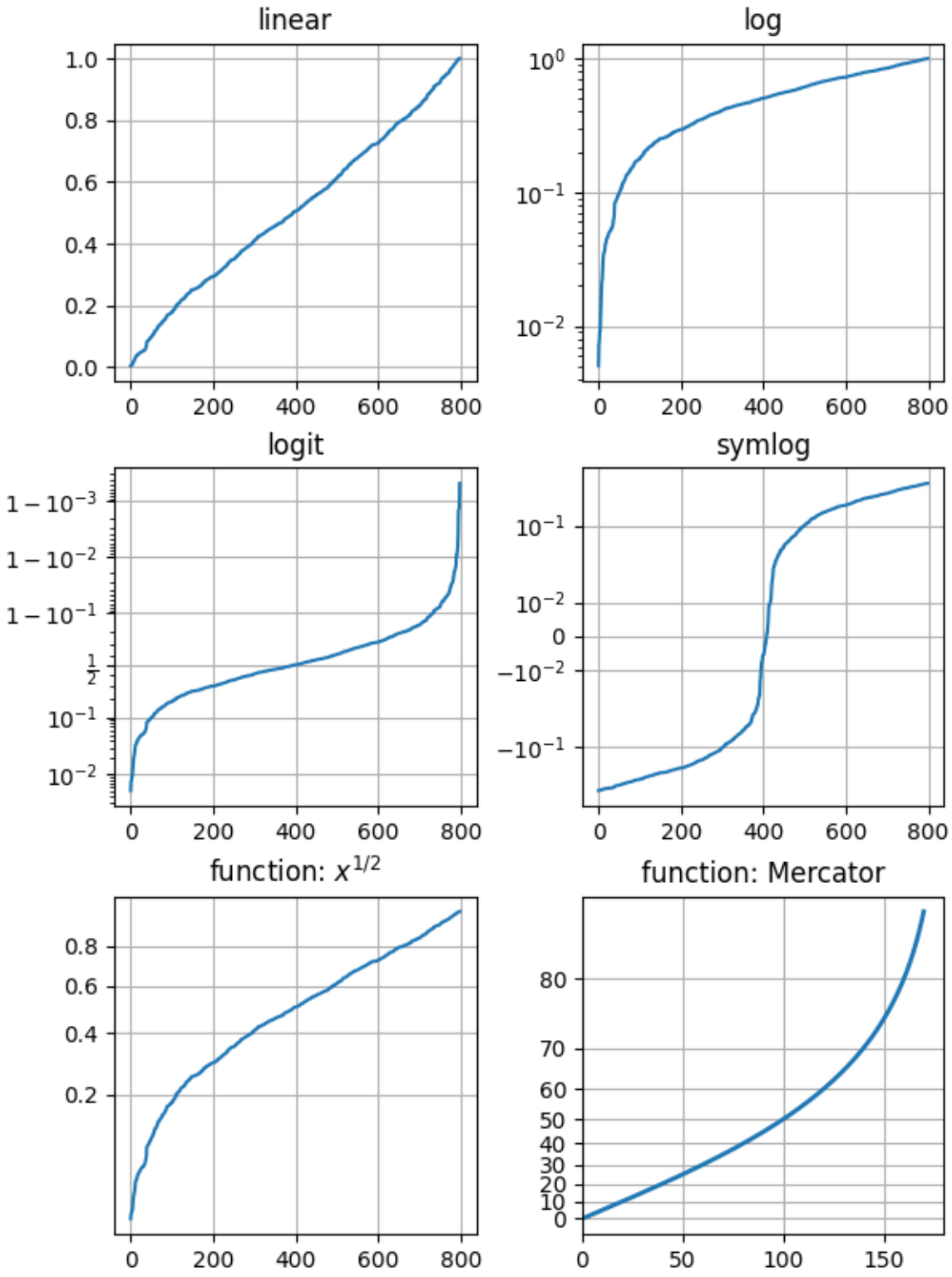
(continued from previous page)

```
from matplotlib.ticker import FixedLocator, NullFormatter

# Fixing random state for reproducibility
np.random.seed(19680801)

# make up some data in the interval ]0, 1[
y = np.random.normal(loc=0.5, scale=0.4, size=1000)
y = y[(y > 0) & (y < 1)]
y.sort()
x = np.arange(len(y))

# plot with various axes scales
fig, axs = plt.subplots(3, 2, figsize=(6, 8), layout='constrained')

# linear
ax = axs[0, 0]
ax.plot(x, y)
ax.set_yscale('linear')
ax.set_title('linear')
ax.grid(True)


# log
ax = axs[0, 1]
ax.plot(x, y)
ax.set_yscale('log')
ax.set_title('log')
ax.grid(True)


# symmetric log
ax = axs[1, 1]
ax.plot(x, y - y.mean())
ax.set_yscale('symlog', linthresh=0.02)
ax.set_title('symlog')
ax.grid(True)

# logit
ax = axs[1, 0]
ax.plot(x, y)
ax.set_yscale('logit')
ax.set_title('logit')
ax.grid(True)


# Function x**(1/2)
def forward(x):
    return x**(1/2)


def inverse(x):
```

(continues on next page)

(continued from previous page)

```
    return x**2


ax = axs[2, 0]
ax.plot(x, y)
ax.set_yscale('function', functions=(forward, inverse))
ax.set_title('function: $x^{1/2}$')
ax.grid(True)
ax.yaxis.set_major_locator(FixedLocator(np.arange(0, 1, 0.2)**2))
ax.yaxis.set_major_locator(FixedLocator(np.arange(0, 1, 0.2)))


# Function Mercator transform
def forward(a):
    a = np.deg2rad(a)
    return np.rad2deg(np.log(np.abs(np.tan(a) + 1.0 / np.cos(a))))


def inverse(a):
    a = np.deg2rad(a)
    return np.rad2deg(np.arctan(np.sinh(a)))

ax = axs[2, 1]

t = np.arange(0, 170.0, 0.1)
s = t / 2.

ax.plot(t, s, '-', lw=2)

ax.set_yscale('function', functions=(forward, inverse))
ax.set_title('function: Mercator')
ax.grid(True)
ax.set_xlim([0, 180])
ax.yaxis.set_minor_formatter(NullFormatter())
ax.yaxis.set_major_locator(FixedLocator(np.arange(0, 90, 10)))

plt.show()
```




**References**

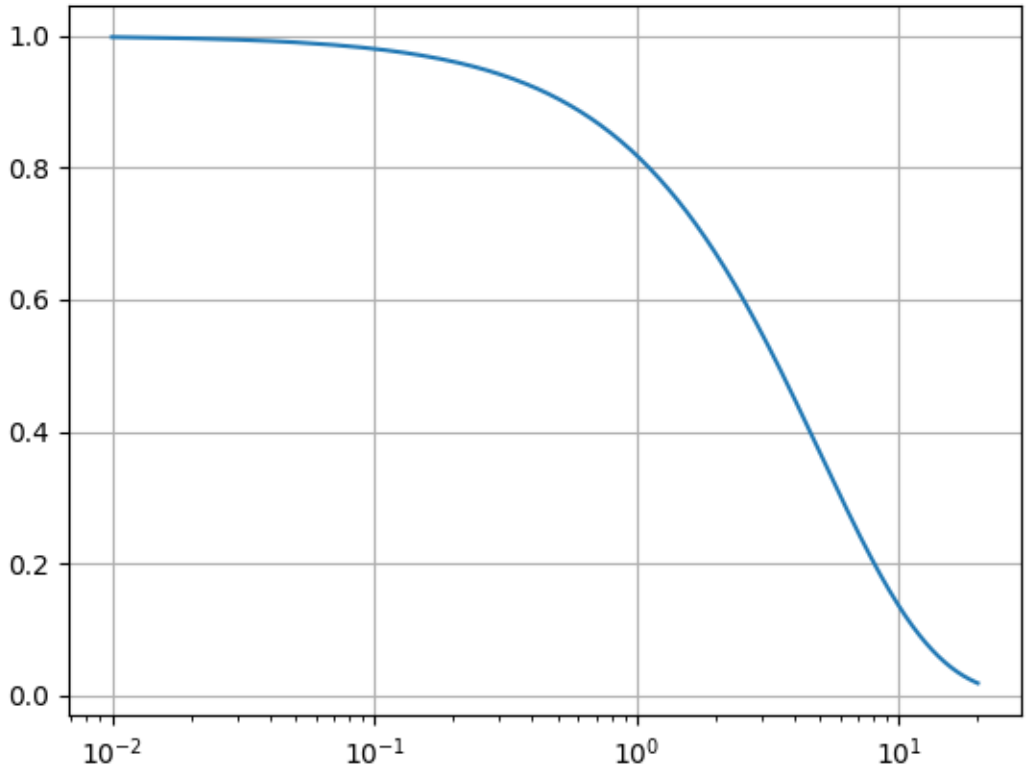
The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.set_xscale*
- *matplotlib.axes.Axes.set_yscale*
- *matplotlib.axis.Axis.set_major_locator*
- *matplotlib.scale.LinearScale*
- *matplotlib.scale.LogScale*
- *matplotlib.scale.SymmetricalLogScale*
- *matplotlib.scale.LogitScale*
- *matplotlib.scale.FuncScale*

**Total running time of the script:** (0 minutes 1.335 seconds)

### Log Axis

This is an example of assigning a log-scale for the x-axis using *semilogx*.

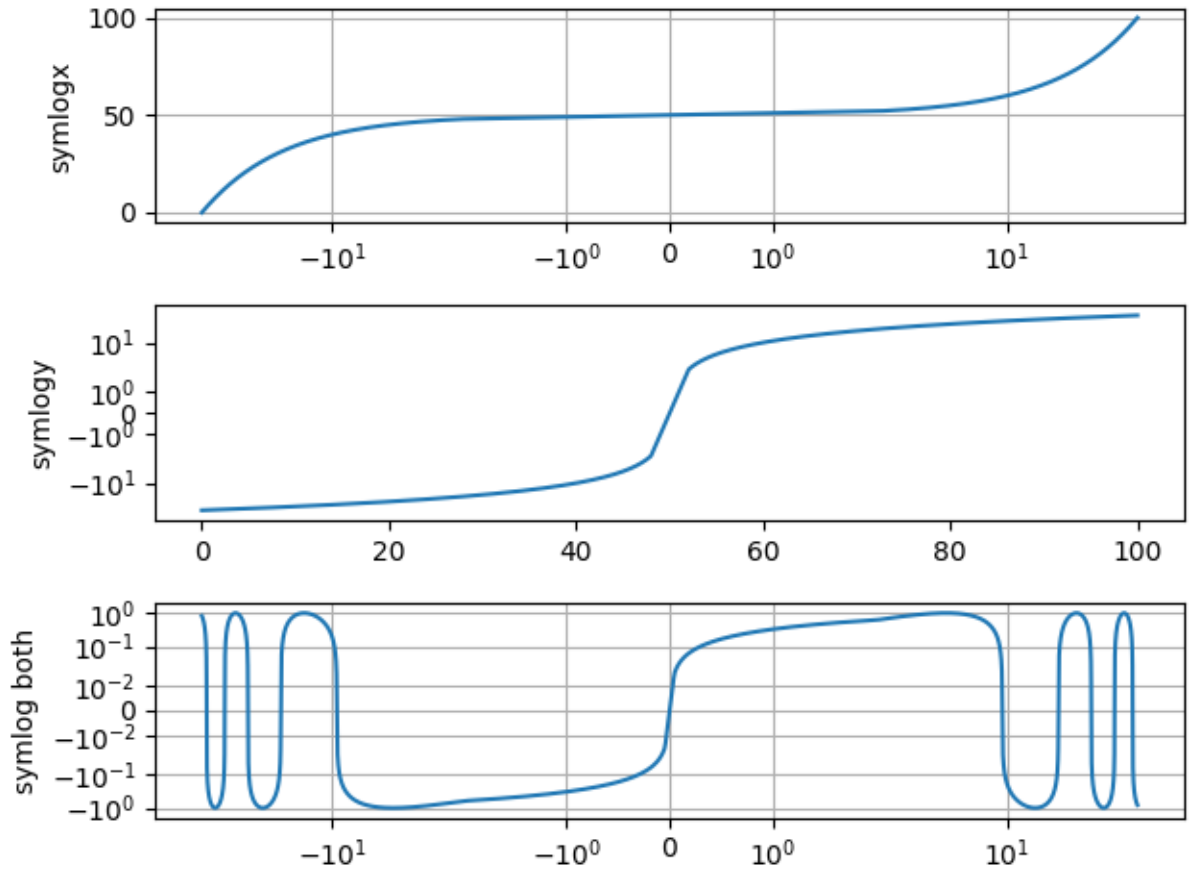


```
import matplotlib.pyplot as plt
import numpy as np

fig, ax = plt.subplots()

dt = 0.01
t = np.arange(dt, 20.0, dt)

ax.semilogx(t, np.exp(-t / 5.0))
ax.grid()

plt.show()
```

### Symlog Demo

Example use of symlog (symmetric log) axis scaling.

```
import matplotlib.pyplot as plt
import numpy as np

dt = 0.01
x = np.arange(-50.0, 50.0, dt)
y = np.arange(0, 100.0, dt)

fig, (ax0, ax1, ax2) = plt.subplots(nrows=3)

ax0.plot(x, y)
ax0.set_xscale('symlog')
ax0.set_ylabel('symlogx')
ax0.grid()
ax0.xaxis.grid(which='minor')  # minor grid on too

ax1.plot(y, x)
ax1.set_yscale('symlog')
ax1.set_ylabel('symlogy')

ax2.plot(x, np.sin(x / 3.0))
ax2.set_xscale('symlog')
ax2.set_yscale('symlog', linthresh=0.015)
ax2.grid()
ax2.set_ylabel('symlog both')

fig.tight_layout()
plt.show()
```




It should be noted that the coordinate transform used by `symlog` has a discontinuous gradient at the transition between its linear and logarithmic regions. The `asinh` axis scale is an alternative technique that may avoid visual artifacts caused by these discontinuities.

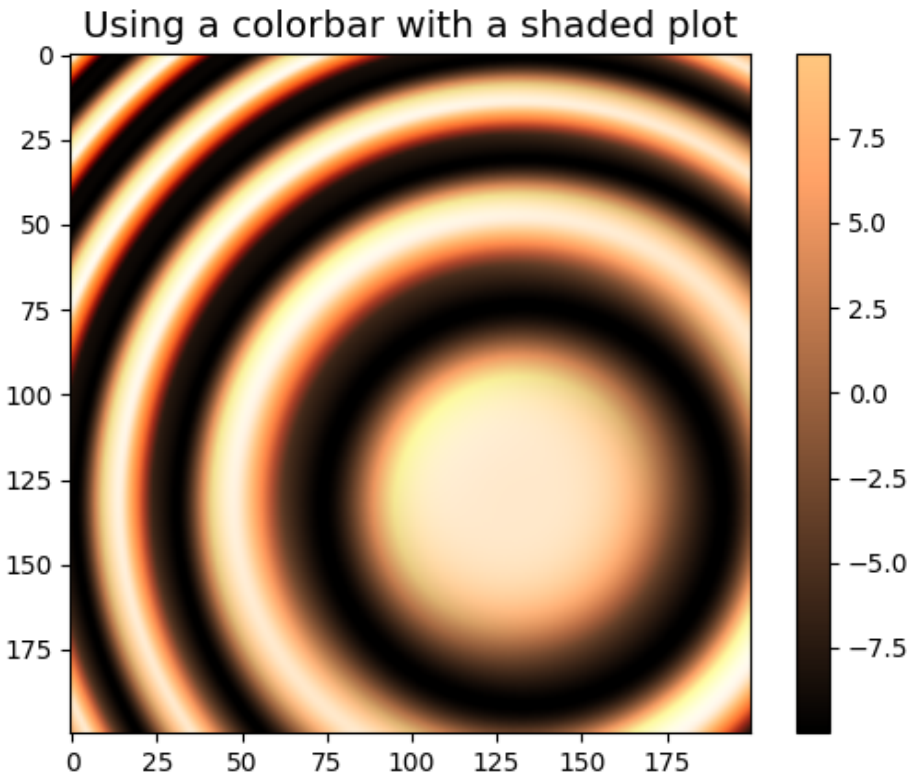
**References**

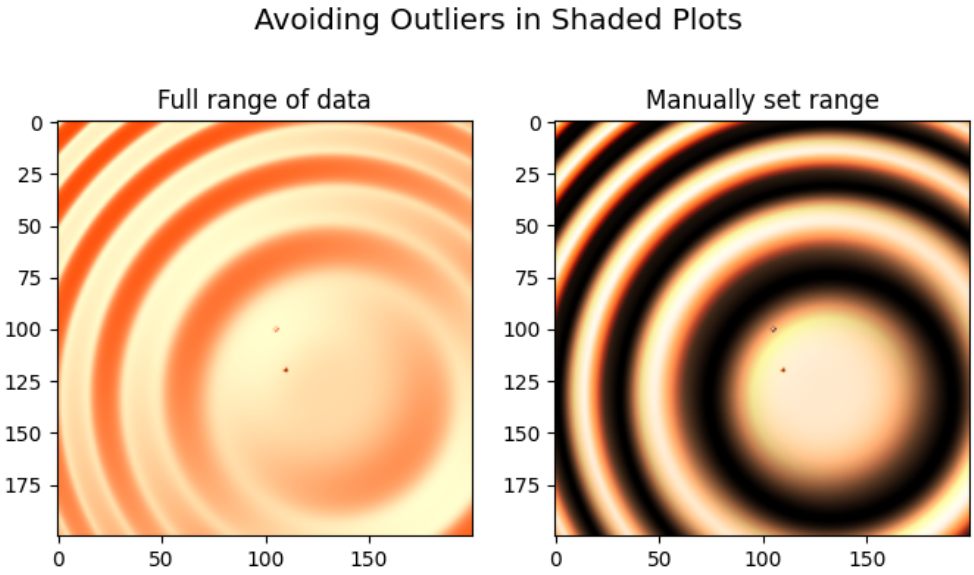
- *matplotlib.scale.SymmetricalLogScale*
- *matplotlib.ticker.SymmetricalLogLocator*
- *matplotlib.scale.AsinhScale*

### 6.25.19 Specialty plots

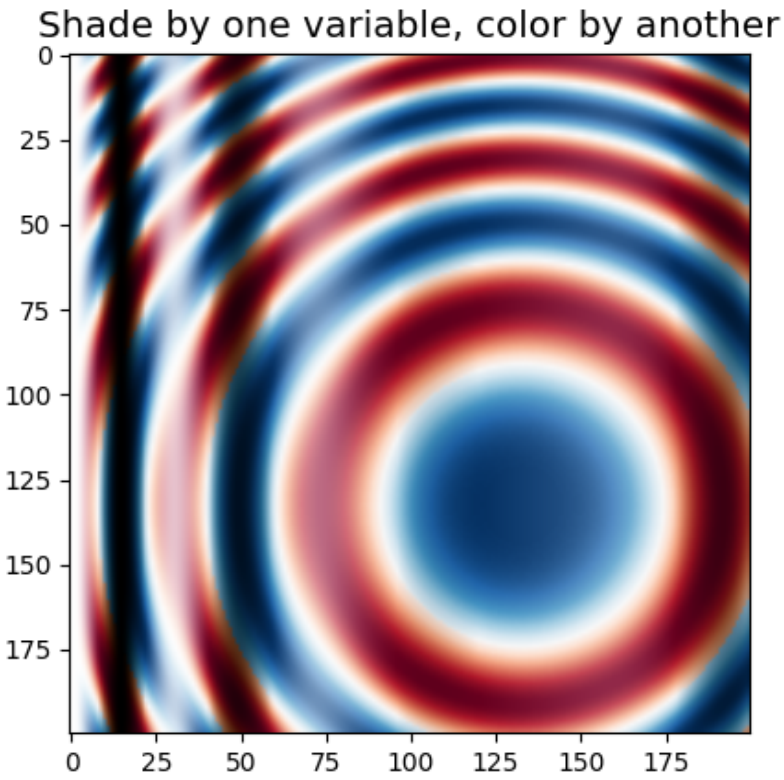
#### Hillshading

Demonstrates a few common tricks with shaded plots.


Using a colorbar with a shaded plot
0
25
50
75
100
125
150
175
0
25
50
75
100
125
150
175
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5

- 
Avoiding Outliers in Shaded Plots
Full range of data
Manually set range
0
25
50
75
100
125
150
175
0
50
100
150

-




-

```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.colors import LightSource, Normalize


def display_colorbar():
    """Display a correct numeric colorbar for a shaded plot."""
    y, x = np.mgrid[-4:2:200j, -4:2:200j]
    z = 10 * np.cos(x**2 + y**2)

    cmap = plt.cm.copper
    ls = LightSource(315, 45)
    rgb = ls.shade(z, cmap)

    fig, ax = plt.subplots()
    ax.imshow(rgb, interpolation='bilinear')

    # Use a proxy artist for the colorbar...
    im = ax.imshow(z, cmap=cmap)
    im.remove()
    fig.colorbar(im, ax=ax)

    ax.set_title('Using a colorbar with a shaded plot', size='x-large')
```

(continues on next page)

(continued from previous page)

```
def avoid_outliers():
    """Use a custom norm to control the displayed z-range of a shaded plot."""
    y, x = np.mgrid[-4:2:200j, -4:2:200j]
    z = 10 * np.cos(x**2 + y**2)

    # Add some outliers...
    z[100, 105] = 2000
    z[120, 110] = -9000

    ls = LightSource(315, 45)
    fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(8, 4.5))

    rgb = ls.shade(z, plt.cm.copper)
    ax1.imshow(rgb, interpolation='bilinear')
    ax1.set_title('Full range of data')

    rgb = ls.shade(z, plt.cm.copper, vmin=-10, vmax=10)
    ax2.imshow(rgb, interpolation='bilinear')
    ax2.set_title('Manually set range')

    fig.suptitle('Avoiding Outliers in Shaded Plots', size='x-large')


def shade_other_data():
    """Demonstrates displaying different variables through shade and color."""
    y, x = np.mgrid[-4:2:200j, -4:2:200j]
    z1 = np.sin(x**2)  # Data to hillshade
    z2 = np.cos(x**2 + y**2)  # Data to color

    norm = Normalize(z2.min(), z2.max())
    cmap = plt.cm.RdBu

    ls = LightSource(315, 45)
    rgb = ls.shade_rgb(cmap(norm(z2)), z1)

    fig, ax = plt.subplots()
    ax.imshow(rgb, interpolation='bilinear')
    ax.set_title('Shade by one variable, color by another', size='x-large')

display_colorbar()
avoid_outliers()
shade_other_data()
plt.show()
```

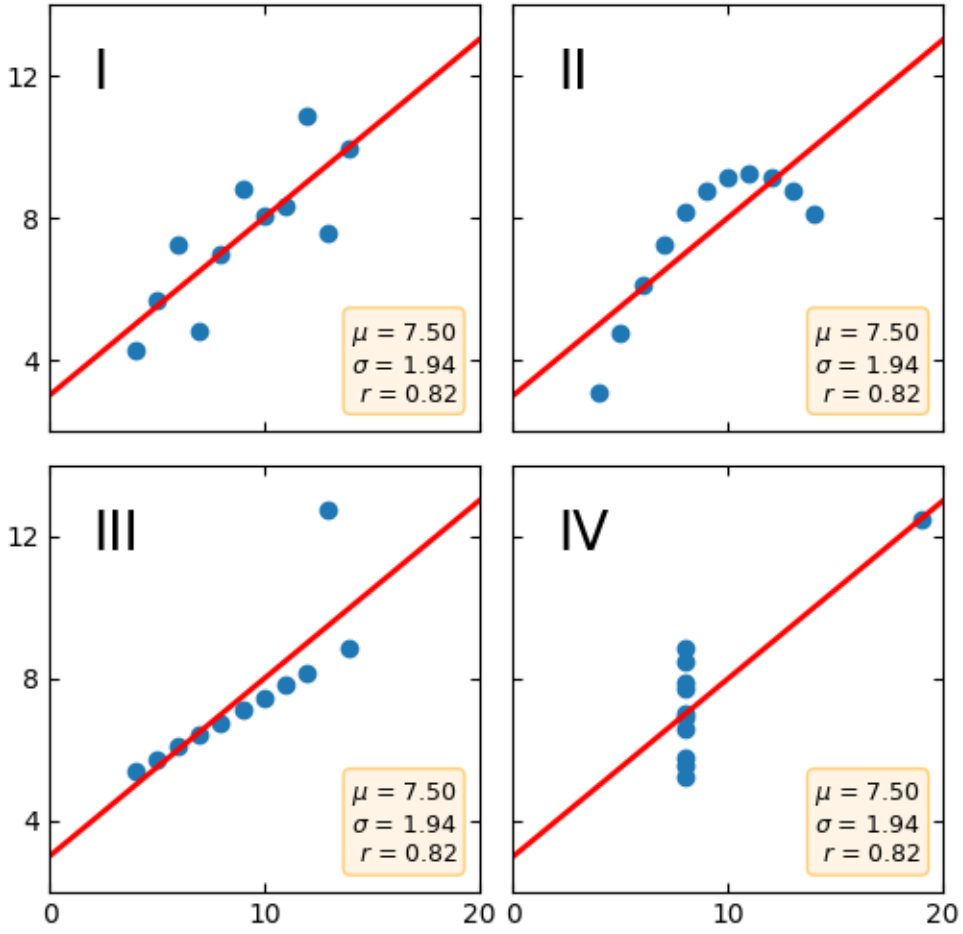
**Total running time of the script:** (0 minutes 2.346 seconds)

### Anscombe's quartet

Anscombe's quartet is a group of datasets (x, y) that have the same mean, standard deviation, and regression line, but which are qualitatively different.

It is often used to illustrate the importance of looking at a set of data graphically and not only relying on basic statistic properties.

```
import matplotlib.pyplot as plt
import numpy as np

x = [10, 8, 13, 9, 11, 14, 6, 4, 12, 7, 5]
y1 = [8.04, 6.95, 7.58, 8.81, 8.33, 9.96, 7.24, 4.26, 10.84, 4.82, 5.68]
y2 = [9.14, 8.14, 8.74, 8.77, 9.26, 8.10, 6.13, 3.10, 9.13, 7.26, 4.74]
y3 = [7.46, 6.77, 12.74, 7.11, 7.81, 8.84, 6.08, 5.39, 8.15, 6.42, 5.73]
x4 = [8, 8, 8, 8, 8, 8, 8, 19, 8, 8, 8]
y4 = [6.58, 5.76, 7.71, 8.84, 8.47, 7.04, 5.25, 12.50, 5.56, 7.91, 6.89]

datasets = {
    'I': (x, y1),
    'II': (x, y2),
    'III': (x, y3),
    'IV': (x4, y4)
}

fig, axs = plt.subplots(2, 2, sharex=True, sharey=True, figsize=(6, 6),
                        gridspec_kw={'wspace': 0.08, 'hspace': 0.08})
axs[0, 0].set(xlim=(0, 20), ylim=(2, 14))
axs[0, 0].set(xticks=(0, 10, 20), yticks=(4, 8, 12))

for ax, (label, (x, y)) in zip(axs.flat, datasets.items()):
    ax.text(0.1, 0.9, label, fontsize=20, transform=ax.transAxes, va='top')
    ax.tick_params(direction='in', top=True, right=True)
    ax.plot(x, y, 'o')

    # linear regression
    p1, p0 = np.polyfit(x, y, deg=1)  # slope, intercept
    ax.axline(xy1=(0, p0), slope=p1, color='r', lw=2)

    # add text box for the statistics
    stats = (f'$\\mu$ = {np.mean(y):.2f}\n'
             f'$\\sigma$ = {np.std(y):.2f}\n'
             f'$r$ = {np.corrcoef(x, y)[0][1]:.2f}')
    bbox = dict(boxstyle='round', fc='blanchedalmond', ec='orange', alpha=0.5)
    ax.text(0.95, 0.07, stats, fontsize=9, bbox=bbox,
            transform=ax.transAxes, horizontalalignment='right')

plt.show()
```




**References**

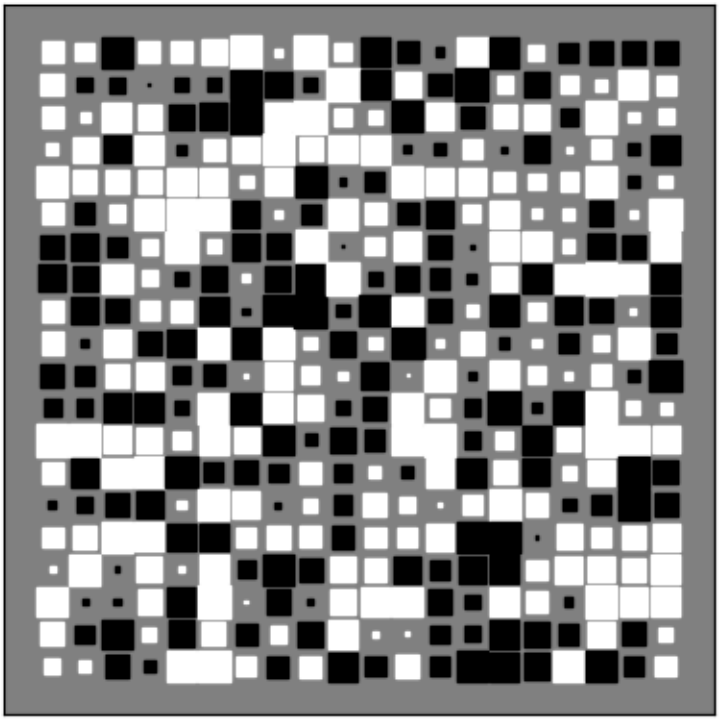
The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.axline* / *matplotlib.pyplot.axline*
- *matplotlib.axes.Axes.text* / *matplotlib.pyplot.text*
- *matplotlib.axes.Axes.tick_params* / matplotlib.pyplot.tick_params`

### Hinton diagrams

Hinton diagrams are useful for visualizing the values of a 2D array (e.g. a weight matrix): Positive and negative values are represented by white and black squares, respectively, and the size of each square represents the magnitude of each value.

Initial idea from David Warde-Farley on the SciPy Cookbook



```
import matplotlib.pyplot as plt
import numpy as np


def hinton(matrix, max_weight=None, ax=None):
    """Draw Hinton diagram for visualizing a weight matrix."""
    ax = ax if ax is not None else plt.gca()

    if not max_weight:
        max_weight = 2 ** np.ceil(np.log2(np.abs(matrix).max()))

    ax.patch.set_facecolor('gray')
    ax.set_aspect('equal', 'box')
    ax.xaxis.set_major_locator(plt.NullLocator())
    ax.yaxis.set_major_locator(plt.NullLocator())
```

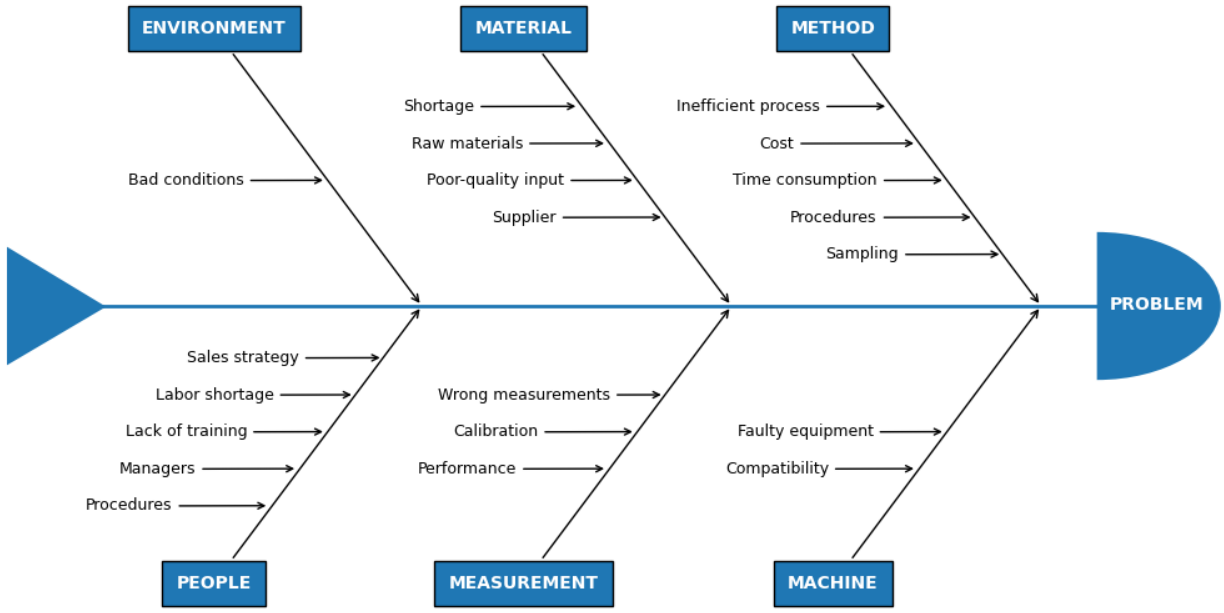
(continues on next page)

(continued from previous page)

```
    for (x, y), w in np.ndenumerate(matrix):
        color = 'white' if w > 0 else 'black'
        size = np.sqrt(abs(w) / max_weight)
        rect = plt.Rectangle([x - size / 2, y - size / 2], size, size,
                             facecolor=color, edgecolor=color)
        ax.add_patch(rect)

    ax.autoscale_view()
    ax.invert_yaxis()


if __name__ == '__main__':
    # Fixing random state for reproducibility
    np.random.seed(19680801)

    hinton(np.random.rand(20, 20) - 0.5)
    plt.show()
```

## Ishikawa Diagram

Ishikawa Diagrams, fishbone diagrams, herringbone diagrams, or cause-and-effect diagrams are used to identify problems in a system by showing how causes and effects are linked. Source: https://en.wikipedia.org/wiki/Ishikawa_diagram



```
import math

import matplotlib.pyplot as plt

from matplotlib.patches import Polygon, Wedge

fig, ax = plt.subplots(figsize=(10, 6), layout='constrained')
ax.set_xlim(-5, 5)
ax.set_ylim(-5, 5)
ax.axis('off')


def problems(data: str,
             problem_x: float, problem_y: float,
             angle_x: float, angle_y: float):
    """
    Draw each problem section of the Ishikawa plot.

    Parameters
    ----------
    data : str
        The name of the problem category.
    problem_x, problem_y : float, optional
        The `X` and `Y` positions of the problem arrows (`Y` defaults to␣
 ↪zero).
    angle_x, angle_y : float, optional
        The angle of the problem annotations. They are always angled towards
        the tail of the plot.

    Returns
    -------
    None.

    """
    ax.annotate(str.upper(data), xy=(problem_x, problem_y),
                xytext=(angle_x, angle_y),
                fontsize=10,
                color='white',
                weight='bold',
                xycoords='data',
                verticalalignment='center',
                horizontalalignment='center',
                textcoords='offset fontsize',
                arrowprops=dict(arrowstyle="->", facecolor='black'),
                bbox=dict(boxstyle='square',
                          facecolor='tab:blue',
                          pad=0.8))


def causes(data: list,
           cause_x: float, cause_y: float,
           cause_xytext=(-9, -0.3), top: bool = True):
```

(continues on next page)

(continued from previous page)

```
    """
    Place each cause to a position relative to the problems
    annotations.

    Parameters
    ----------
    data : indexable object
        The input data. IndexError is
        raised if more than six arguments are passed.
    cause_x, cause_y : float
        The `X` and `Y` position of the cause annotations.
    cause_xytext : tuple, optional
        Adjust to set the distance of the cause text from the problem
        arrow in fontsize units.
    top : bool, default: True
        Determines whether the next cause annotation will be
        plotted above or below the previous one.

    Returns
    -------
    None.

    """
    for index, cause in enumerate(data):
        # [<x pos>, <y pos>]
        coords = [[0.02, 0],
                  [0.23, 0.5],
                  [-0.46, -1],
                  [0.69, 1.5],
                  [-0.92, -2],
                  [1.15, 2.5]]

        # First 'cause' annotation is placed in the middle of the 'problems'␣
↪arrow
        # and each subsequent cause is plotted above or below it in␣
↪succession.
        cause_x -= coords[index][0]
        cause_y += coords[index][1] if top else -coords[index][1]

        ax.annotate(cause, xy=(cause_x, cause_y),
                    horizontalalignment='center',
                    xytext=cause_xytext,
                    fontsize=9,
                    xycoords='data',
                    textcoords='offset fontsize',
                    arrowprops=dict(arrowstyle="->",
                                    facecolor='black'))


def draw_body(data: dict):
    """
    Place each problem section in its correct place by changing
```

(continues on next page)

(continued from previous page)

```
    the coordinates on each loop.

    Parameters
    ----------
    data : dict
        The input data (can be a dict of lists or tuples). ValueError
        is raised if more than six arguments are passed.

    Returns
    -------
    None.

    """
    # Set the length of the spine according to the number of 'problem'␣
↪categories.
    length = (math.ceil(len(data) / 2)) - 1
    draw_spine(-2 - length, 2 + length)

    # Change the coordinates of the 'problem' annotations after each one is␣
↪rendered.
    offset = 0
    prob_section = [1.55, 0.8]
    for index, problem in enumerate(data.values()):
        plot_above = index % 2 == 0
        cause_arrow_y = 1.7 if plot_above else -1.7
        y_prob_angle = 16 if plot_above else -16

        # Plot each section in pairs along the main spine.
        prob_arrow_x = prob_section[0] + length + offset
        cause_arrow_x = prob_section[1] + length + offset
        if not plot_above:
            offset -= 2.5
        if index > 5:
            raise ValueError(f'Maximum number of problems is 6, you have␣
↪entered '
                             f'{len(data)}')

        problems(list(data.keys())[index], prob_arrow_x, 0, -12, y_prob_angle)
        causes(problem, cause_arrow_x, cause_arrow_y, top=plot_above)


def draw_spine(xmin: int, xmax: int):
    """
    Draw main spine, head and tail.

    Parameters
    ----------
    xmin : int
        The default position of the head of the spine's
        x-coordinate.
    xmax : int
        The default position of the tail of the spine's
```

(continues on next page)

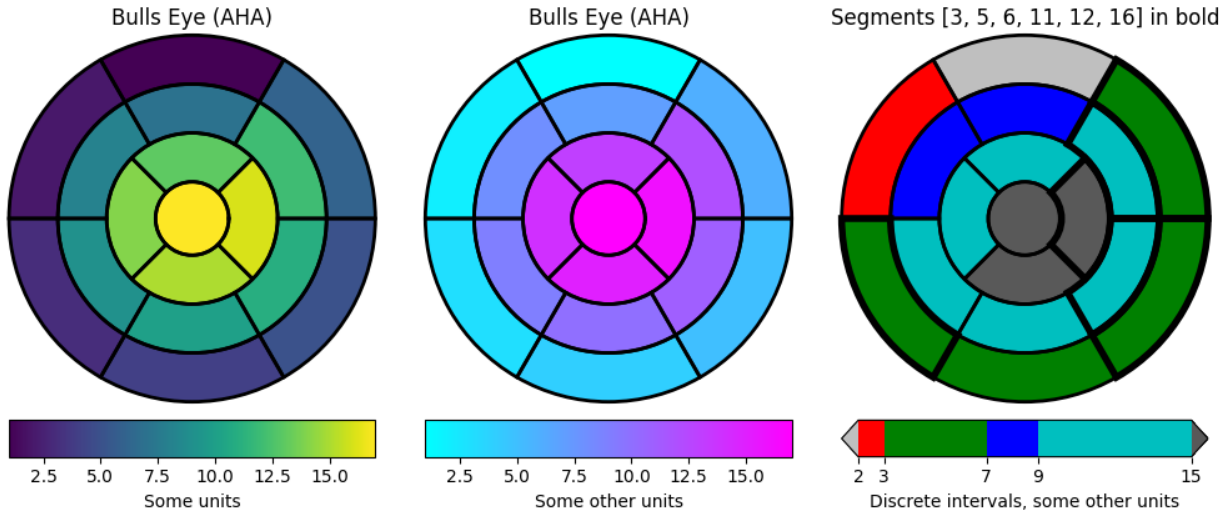
(continued from previous page)

```
        x-coordinate.

    Returns
    -------
    None.

    """
    # draw main spine
    ax.plot([xmin - 0.1, xmax], [0, 0], color='tab:blue', linewidth=2)
    # draw fish head
    ax.text(xmax + 0.1, - 0.05, 'PROBLEM', fontsize=10,
            weight='bold', color='white')
    semicircle = Wedge((xmax, 0), 1, 270, 90, fc='tab:blue')
    ax.add_patch(semicircle)
    # draw fish tail
    tail_pos = [[xmin - 0.8, 0.8], [xmin - 0.8, -0.8], [xmin, -0.01]]
    triangle = Polygon(tail_pos, fc='tab:blue')
    ax.add_patch(triangle)


# Input data
categories = {
    'Method': ['Time consumption', 'Cost', 'Procedures', 'Inefficient process
 ↪',
               'Sampling'],
    'Machine': ['Faulty equipment', 'Compatibility'],
    'Material': ['Poor-quality input', 'Raw materials', 'Supplier',
                 'Shortage'],
    'Measurement': ['Calibration', 'Performance', 'Wrong measurements'],
    'Environment': ['Bad conditions'],
    'People': ['Lack of training', 'Managers', 'Labor shortage',
               'Procedures', 'Sales strategy']
}

draw_body(categories)
plt.show()
```

### Left ventricle bullseye

This example demonstrates how to create the 17 segment model for the left ventricle recommended by the American Heart Association (AHA).

See also the *Nested pie charts* example.




```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib as mpl


def bullseye_plot(ax, data, seg_bold=None, cmap="viridis", norm=None):
    """
    Bullseye representation for the left ventricle.

    Parameters
    ----------
    ax : Axes
    data : list[float]
        The intensity values for each of the 17 segments.
    seg_bold : list[int], optional
        A list with the segments to highlight.
    cmap : colormap, default: "viridis"
        Colormap for the data.
    norm : Normalize or None, optional
        Normalizer for the data.

    Notes
    -----
    This function creates the 17 segment model for the left ventricle␣
↪according
    to the American Heart Association (AHA) [1]_

    References
    ----------
    .. [1] M. D. Cerqueira, N. J. Weissman, V. Dilsizian, A. K. Jacobs,
        S. Kaul, W. K. Laskey, D. J. Pennell, J. A. Rumberger, T. Ryan,
```

(continues on next page)

(continued from previous page)

```
        and M. S. Verani, "Standardized myocardial segmentation and
        nomenclature for tomographic imaging of the heart",
        Circulation, vol. 105, no. 4, pp. 539-542, 2002.
    """

    data = np.ravel(data)
    if seg_bold is None:
        seg_bold = []
    if norm is None:
        norm = mpl.colors.Normalize(vmin=data.min(), vmax=data.max())

    r = np.linspace(0.2, 1, 4)

    ax.set(ylim=[0, 1], xticklabels=[], yticklabels=[])
    ax.grid(False)  # Remove grid

    # Fill segments 1-6, 7-12, 13-16.
    for start, stop, r_in, r_out in [
            (0, 6, r[2], r[3]),
            (6, 12, r[1], r[2]),
            (12, 16, r[0], r[1]),
            (16, 17, 0, r[0]),
    ]:
        n = stop - start
        dtheta = 2*np.pi / n
        ax.bar(np.arange(n) * dtheta + np.pi/2, r_out - r_in, dtheta, r_in,
               color=cmap(norm(data[start:stop])))

    # Now, draw the segment borders.  In order for the outer bold borders not
    # to be covered by inner segments, the borders are all drawn separately
    # after the segments have all been filled.  We also disable clipping,␣
↪which
    # would otherwise affect the outermost segment edges.
    # Draw edges of segments 1-6, 7-12, 13-16.
    for start, stop, r_in, r_out in [
            (0, 6, r[2], r[3]),
            (6, 12, r[1], r[2]),
            (12, 16, r[0], r[1]),
    ]:
        n = stop - start
        dtheta = 2*np.pi / n
        ax.bar(np.arange(n) * dtheta + np.pi/2, r_out - r_in, dtheta, r_in,
               clip_on=False, color="none", edgecolor="k", linewidth=[
                   4 if i + 1 in seg_bold else 2 for i in range(start, stop)])
    # Draw edge of segment 17 -- here; the edge needs to be drawn differently,
    # using plot().
    ax.plot(np.linspace(0, 2*np.pi), np.linspace(r[0], r[0]), "k",
            linewidth=(4 if 17 in seg_bold else 2))


# Create the fake data
data = np.arange(17) + 1
```

(continues on next page)

(continued from previous page)

```
# Make a figure and Axes with dimensions as desired.
fig = plt.figure(figsize=(10, 5), layout="constrained")
fig.get_layout_engine().set(wspace=.1, w_pad=.2)
axs = fig.subplots(1, 3, subplot_kw=dict(projection='polar'))
fig.canvas.manager.set_window_title('Left Ventricle Bulls Eyes (AHA)')


# Set the colormap and norm to correspond to the data for which
# the colorbar will be used.
cmap = mpl.cm.viridis
norm = mpl.colors.Normalize(vmin=1, vmax=17)
# Create an empty ScalarMappable to set the colorbar's colormap and norm.
# The following gives a basic continuous colorbar with ticks and labels.
fig.colorbar(mpl.cm.ScalarMappable(cmap=cmap, norm=norm),
             cax=axs[0].inset_axes([0, -.15, 1, .1]),
             orientation='horizontal', label='Some units')


# And again for the second colorbar.
cmap2 = mpl.cm.cool
norm2 = mpl.colors.Normalize(vmin=1, vmax=17)
fig.colorbar(mpl.cm.ScalarMappable(cmap=cmap2, norm=norm2),
             cax=axs[1].inset_axes([0, -.15, 1, .1]),
             orientation='horizontal', label='Some other units')


# The second example illustrates the use of a ListedColormap, a
# BoundaryNorm, and extended ends to show the "over" and "under"
# value colors.
cmap3 = (mpl.colors.ListedColormap(['r', 'g', 'b', 'c'])
         .with_extremes(over='0.35', under='0.75'))
# If a ListedColormap is used, the length of the bounds array must be
# one greater than the length of the color list.  The bounds must be
# monotonically increasing.
bounds = [2, 3, 7, 9, 15]
norm3 = mpl.colors.BoundaryNorm(bounds, cmap3.N)
fig.colorbar(mpl.cm.ScalarMappable(cmap=cmap3, norm=norm3),
             cax=axs[2].inset_axes([0, -.15, 1, .1]),
             extend='both',
             ticks=bounds,  # optional
             spacing='proportional',
             orientation='horizontal',
             label='Discrete intervals, some other units')


# Create the 17 segment model
bullseye_plot(axs[0], data, cmap=cmap, norm=norm)
axs[0].set_title('Bulls Eye (AHA)')

bullseye_plot(axs[1], data, cmap=cmap2, norm=norm2)
```

(continues on next page)

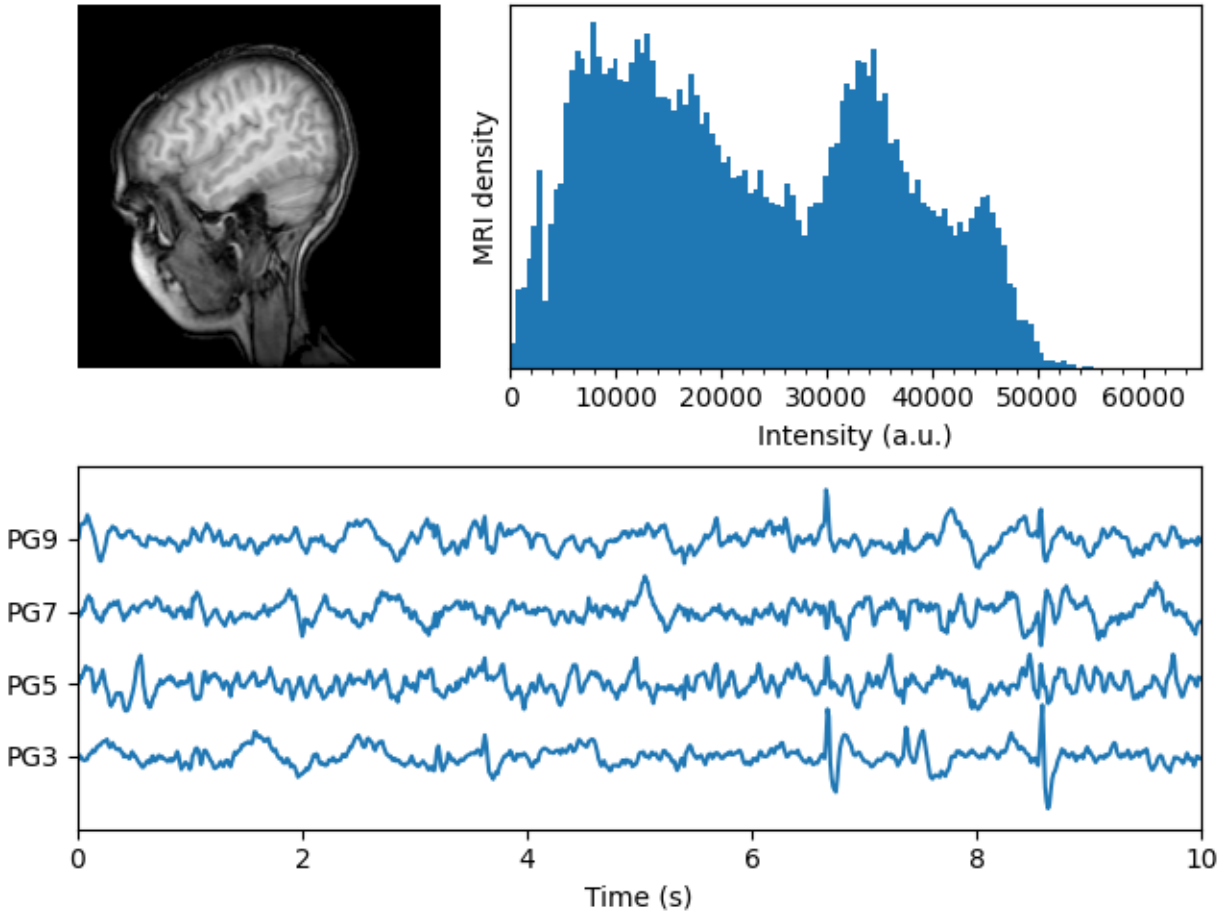
(continued from previous page)

```
axs[1].set_title('Bulls Eye (AHA)')

bullseye_plot(axs[2], data, seg_bold=[3, 5, 6, 11, 12, 16],
              cmap=cmap3, norm=norm3)
axs[2].set_title('Segments [3, 5, 6, 11, 12, 16] in bold')

plt.show()
```

**Total running time of the script:** (0 minutes 1.592 seconds)

### MRI with EEG

Displays a set of subplots with an MRI image, its intensity histogram and some EEG traces.




```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib.cbook as cbook

fig, axd = plt.subplot_mosaic(
```

(continues on next page)

(continued from previous page)

```
    [["image", "density"],
     ["EEG", "EEG"]],
    layout="constrained",
    # "image" will contain a square image. We fine-tune the width so that
    # there is no excess horizontal or vertical margin around the image.
    width_ratios=[1.05, 2],
)

# Load the MRI data (256x256 16-bit integers)
with cbook.get_sample_data('s1045.ima.gz') as dfile:
    im = np.frombuffer(dfile.read(), np.uint16).reshape((256, 256))

# Plot the MRI image
axd["image"].imshow(im, cmap="gray")
axd["image"].axis('off')

# Plot the histogram of MRI intensity
im = im[im.nonzero()]  # Ignore the background
axd["density"].hist(im, bins=np.arange(0, 2**16+1, 512))
axd["density"].set(xlabel='Intensity (a.u.)', xlim=(0, 2**16),
                   ylabel='MRI density', yticks=[])
axd["density"].minorticks_on()

# Load the EEG data
n_samples, n_rows = 800, 4
with cbook.get_sample_data('eeg.dat') as eegfile:
    data = np.fromfile(eegfile, dtype=float).reshape((n_samples, n_rows))
t = 10 * np.arange(n_samples) / n_samples

# Plot the EEG
axd["EEG"].set_xlabel('Time (s)')
axd["EEG"].set_xlim(0, 10)
dy = (data.min() - data.max()) * 0.7  # Crowd them a bit.
axd["EEG"].set_ylim(-dy, n_rows * dy)
axd["EEG"].set_yticks([0, dy, 2*dy, 3*dy], labels=['PG3', 'PG5', 'PG7', 'PG9
 ↪'])

for i, data_col in enumerate(data.T):
    axd["EEG"].plot(t, data_col + i*dy, color="C0")

plt.show()
```

**Total running time of the script:** (0 minutes 1.448 seconds)

### Radar chart (aka spider or star chart)

This example creates a radar chart, also known as a spider or star chart[1].

Although this example allows a frame of either 'circle' or 'polygon', polygon frames don't have proper gridlines (the lines are circles instead of polygons). It's possible to get a polygon grid by setting GRID_LINE_INTERPOLATION_STEPS in *matplotlib.axis* to the desired number of vertices, but the orientation of the polygon is not aligned with the radial axis.

```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.patches import Circle, RegularPolygon
from matplotlib.path import Path
from matplotlib.projections import register_projection
from matplotlib.projections.polar import PolarAxes
from matplotlib.spines import Spine
from matplotlib.transforms import Affine2D


def radar_factory(num_vars, frame='circle'):
    """
    Create a radar chart with `num_vars` Axes.

    This function creates a RadarAxes projection and registers it.

    Parameters
    ----------
    num_vars : int
        Number of variables for radar chart.
    frame : {'circle', 'polygon'}
        Shape of frame surrounding Axes.

    """
    # calculate evenly-spaced axis angles
    theta = np.linspace(0, 2*np.pi, num_vars, endpoint=False)

    class RadarTransform(PolarAxes.PolarTransform):

        def transform_path_non_affine(self, path):
            # Paths with non-unit interpolation steps correspond to gridlines,
            # in which case we force interpolation (to defeat PolarTransform's
            # autoconversion to circular arcs).
            if path._interpolation_steps > 1:
                path = path.interpolated(num_vars)
            return Path(self.transform(path.vertices), path.codes)

    class RadarAxes(PolarAxes):

        name = 'radar'
        PolarTransform = RadarTransform
```

(continues on next page)

[1] https://en.wikipedia.org/wiki/Radar_chart

(continued from previous page)

```
        def __init__(self, *args, **kwargs):
            super().__init__(*args, **kwargs)
            # rotate plot such that the first axis is at the top
            self.set_theta_zero_location('N')

        def fill(self, *args, closed=True, **kwargs):
            """Override fill so that line is closed by default"""
            return super().fill(closed=closed, *args, **kwargs)

        def plot(self, *args, **kwargs):
            """Override plot so that line is closed by default"""
            lines = super().plot(*args, **kwargs)
            for line in lines:
                self._close_line(line)

        def _close_line(self, line):
            x, y = line.get_data()
            # FIXME: markers at x[0], y[0] get doubled-up
            if x[0] != x[-1]:
                x = np.append(x, x[0])
                y = np.append(y, y[0])
                line.set_data(x, y)

        def set_varlabels(self, labels):
            self.set_thetagrids(np.degrees(theta), labels)

        def _gen_axes_patch(self):
            # The Axes patch must be centered at (0.5, 0.5) and of radius 0.5
            # in axes coordinates.
            if frame == 'circle':
                return Circle((0.5, 0.5), 0.5)
            elif frame == 'polygon':
                return RegularPolygon((0.5, 0.5), num_vars,
                                      radius=.5, edgecolor="k")
            else:
                raise ValueError("Unknown value for 'frame': %s" % frame)

        def _gen_axes_spines(self):
            if frame == 'circle':
                return super()._gen_axes_spines()
            elif frame == 'polygon':
                # spine_type must be 'left'/'right'/'top'/'bottom'/'circle'.
                spine = Spine(axes=self,
                              spine_type='circle',
                              path=Path.unit_regular_polygon(num_vars))
                # unit_regular_polygon gives a polygon of radius 1 centered at
                # (0, 0) but we want a polygon of radius 0.5 centered at (0.5,
                # 0.5) in axes coordinates.
                spine.set_transform(Affine2D().scale(.5).translate(.5, .5)
                                    + self.transAxes)
                return {'polar': spine}
```

(continues on next page)

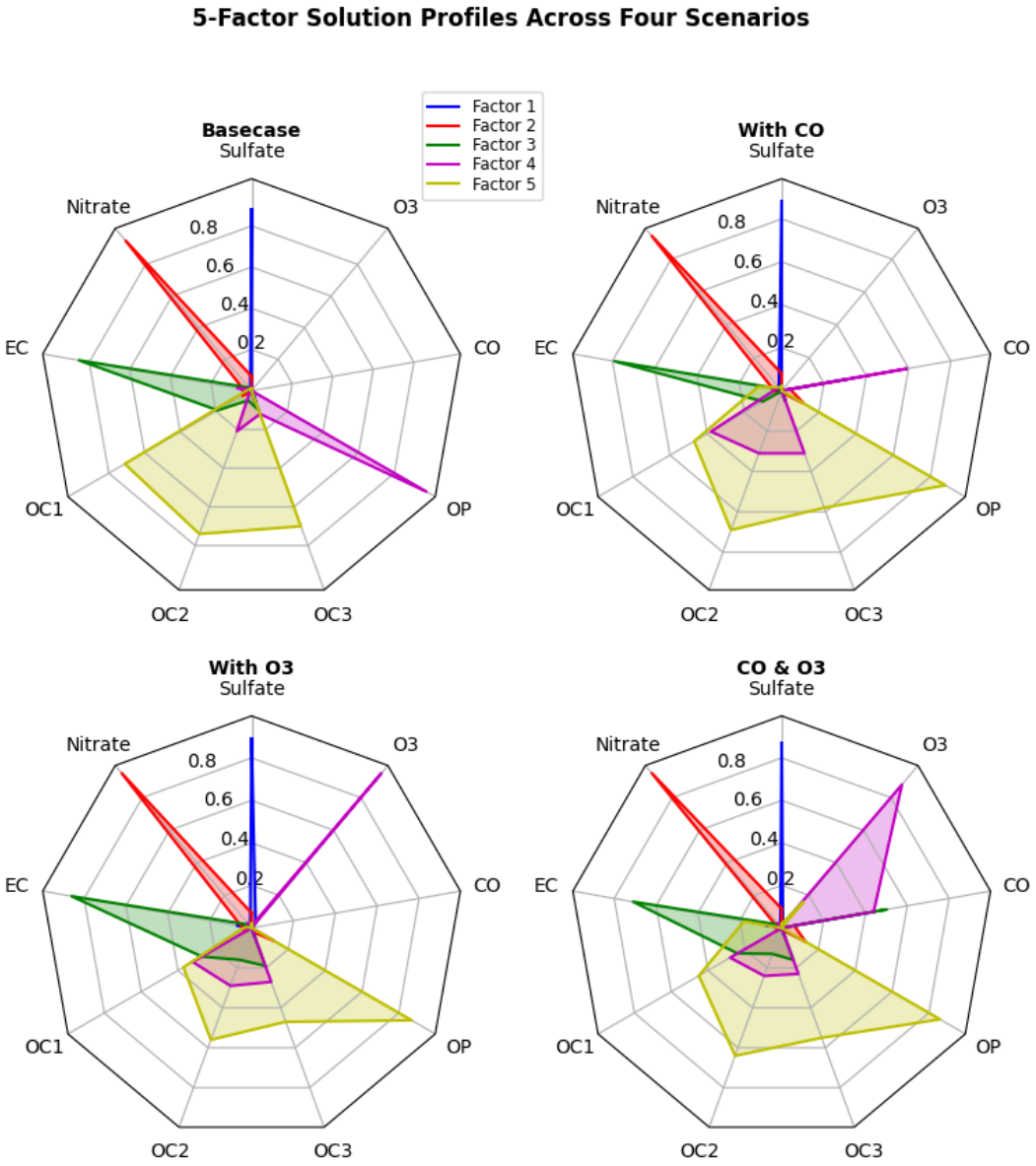
(continued from previous page)

```
            else:
                raise ValueError("Unknown value for 'frame': %s" % frame)

    register_projection(RadarAxes)
    return theta


def example_data():
    # The following data is from the Denver Aerosol Sources and Health study.
    # See doi:10.1016/j.atmosenv.2008.12.017
    #
    # The data are pollution source profile estimates for five modeled
    # pollution sources (e.g., cars, wood-burning, etc) that emit 7-9 chemical
    # species. The radar charts are experimented with here to see if we can
    # nicely visualize how the modeled source profiles change across four
    # scenarios:
    #  1) No gas-phase species present, just seven particulate counts on
    #     Sulfate
    #     Nitrate
    #     Elemental Carbon (EC)
    #     Organic Carbon fraction 1 (OC)
    #     Organic Carbon fraction 2 (OC2)
    #     Organic Carbon fraction 3 (OC3)
    #     Pyrolyzed Organic Carbon (OP)
    #  2)Inclusion of gas-phase specie carbon monoxide (CO)
    #  3)Inclusion of gas-phase specie ozone (O3).
    #  4)Inclusion of both gas-phase species is present...
    data = [
        ['Sulfate', 'Nitrate', 'EC', 'OC1', 'OC2', 'OC3', 'OP', 'CO', 'O3'],
        ('Basecase', [
            [0.88, 0.01, 0.03, 0.03, 0.00, 0.06, 0.01, 0.00, 0.00],
            [0.07, 0.95, 0.04, 0.05, 0.00, 0.02, 0.01, 0.00, 0.00],
            [0.01, 0.02, 0.85, 0.19, 0.05, 0.10, 0.00, 0.00, 0.00],
            [0.02, 0.01, 0.07, 0.01, 0.21, 0.12, 0.98, 0.00, 0.00],
            [0.01, 0.01, 0.02, 0.71, 0.74, 0.70, 0.00, 0.00, 0.00]]),
        ('With CO', [
            [0.88, 0.02, 0.02, 0.02, 0.00, 0.05, 0.00, 0.05, 0.00],
            [0.08, 0.94, 0.04, 0.02, 0.00, 0.01, 0.12, 0.04, 0.00],
            [0.01, 0.01, 0.79, 0.10, 0.00, 0.05, 0.00, 0.31, 0.00],
            [0.00, 0.02, 0.03, 0.38, 0.31, 0.31, 0.00, 0.59, 0.00],
            [0.02, 0.02, 0.11, 0.47, 0.69, 0.58, 0.88, 0.00, 0.00]]),
        ('With O3', [
            [0.89, 0.01, 0.07, 0.00, 0.00, 0.05, 0.00, 0.00, 0.03],
            [0.07, 0.95, 0.05, 0.04, 0.00, 0.02, 0.12, 0.00, 0.00],
            [0.01, 0.02, 0.86, 0.27, 0.16, 0.19, 0.00, 0.00, 0.00],
            [0.01, 0.03, 0.00, 0.32, 0.29, 0.27, 0.00, 0.00, 0.95],
            [0.02, 0.00, 0.03, 0.37, 0.56, 0.47, 0.87, 0.00, 0.00]]),
        ('CO & O3', [
            [0.87, 0.01, 0.08, 0.00, 0.00, 0.04, 0.00, 0.00, 0.01],
            [0.09, 0.95, 0.02, 0.03, 0.00, 0.01, 0.13, 0.06, 0.00],
            [0.01, 0.02, 0.71, 0.24, 0.13, 0.16, 0.00, 0.50, 0.00],
            [0.01, 0.03, 0.00, 0.28, 0.24, 0.23, 0.00, 0.44, 0.88],
```

(continues on next page)

(continued from previous page)

```
             [0.02, 0.00, 0.18, 0.45, 0.64, 0.55, 0.86, 0.00, 0.16]])
    ]
    return data


if __name__ == '__main__':
    N = 9
    theta = radar_factory(N, frame='polygon')

    data = example_data()
    spoke_labels = data.pop(0)

    fig, axs = plt.subplots(figsize=(9, 9), nrows=2, ncols=2,
                            subplot_kw=dict(projection='radar'))
    fig.subplots_adjust(wspace=0.25, hspace=0.20, top=0.85, bottom=0.05)

    colors = ['b', 'r', 'g', 'm', 'y']
    # Plot the four cases from the example data on separate Axes
    for ax, (title, case_data) in zip(axs.flat, data):
        ax.set_rgrids([0.2, 0.4, 0.6, 0.8])
        ax.set_title(title, weight='bold', size='medium', position=(0.5, 1.1),
                     horizontalalignment='center', verticalalignment='center')
        for d, color in zip(case_data, colors):
            ax.plot(theta, d, color=color)
            ax.fill(theta, d, facecolor=color, alpha=0.25, label='_nolegend_')
        ax.set_varlabels(spoke_labels)

    # add legend relative to top-left plot
    labels = ('Factor 1', 'Factor 2', 'Factor 3', 'Factor 4', 'Factor 5')
    legend = axs[0, 0].legend(labels, loc=(0.9, .95),
                              labelspacing=0.1, fontsize='small')

    fig.text(0.5, 0.965, '5-Factor Solution Profiles Across Four Scenarios',
             horizontalalignment='center', color='black', weight='bold',
             size='large')

    plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.path*
- *matplotlib.path.Path*
- *matplotlib.spines*
- *matplotlib.spines.Spine*
- *matplotlib.projections*
- *matplotlib.projections.polar*

- *matplotlib.projections.polar.PolarAxes*
- *matplotlib.projections.register_projection*

**Total running time of the script:** (0 minutes 1.223 seconds)

### The Sankey class

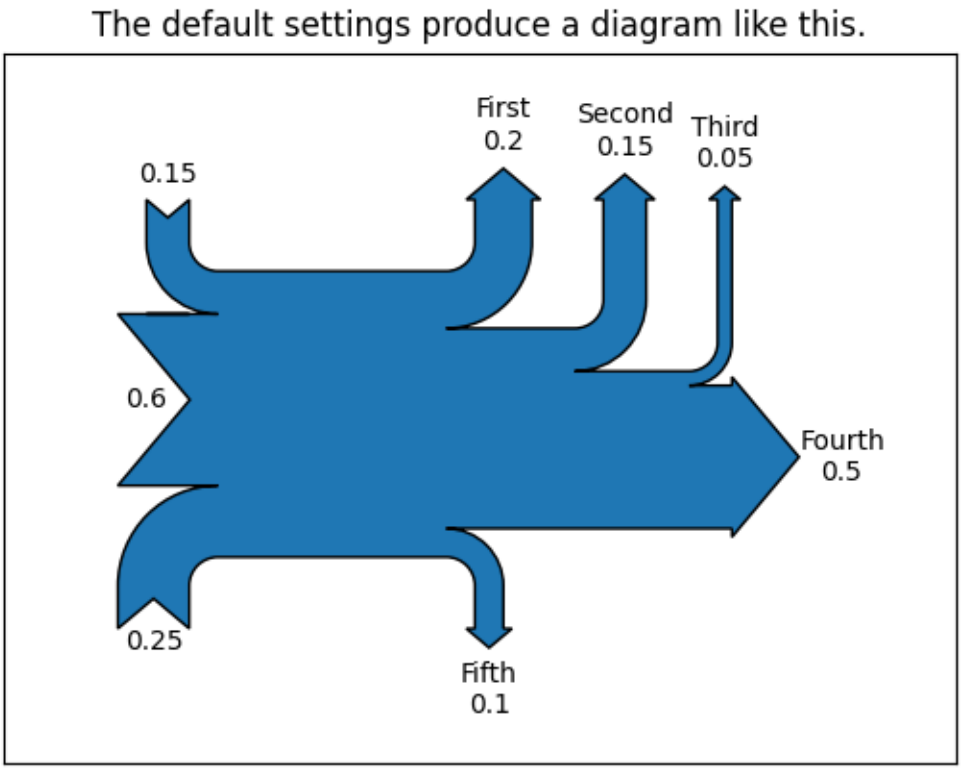
Demonstrate the Sankey class by producing three basic diagrams.

```
import matplotlib.pyplot as plt

from matplotlib.sankey import Sankey
```

Example 1 -- Mostly defaults

This demonstrates how to create a simple diagram by implicitly calling the Sankey.add() method and by appending finish() to the call to the class.

```
Sankey(flows=[0.25, 0.15, 0.60, -0.20, -0.15, -0.05, -0.50, -0.10],
       labels=['', '', '', 'First', 'Second', 'Third', 'Fourth', 'Fifth'],
       orientations=[-1, 1, 0, 1, 1, 1, 0, -1]).finish()
plt.title("The default settings produce a diagram like this.")
```




Notice:

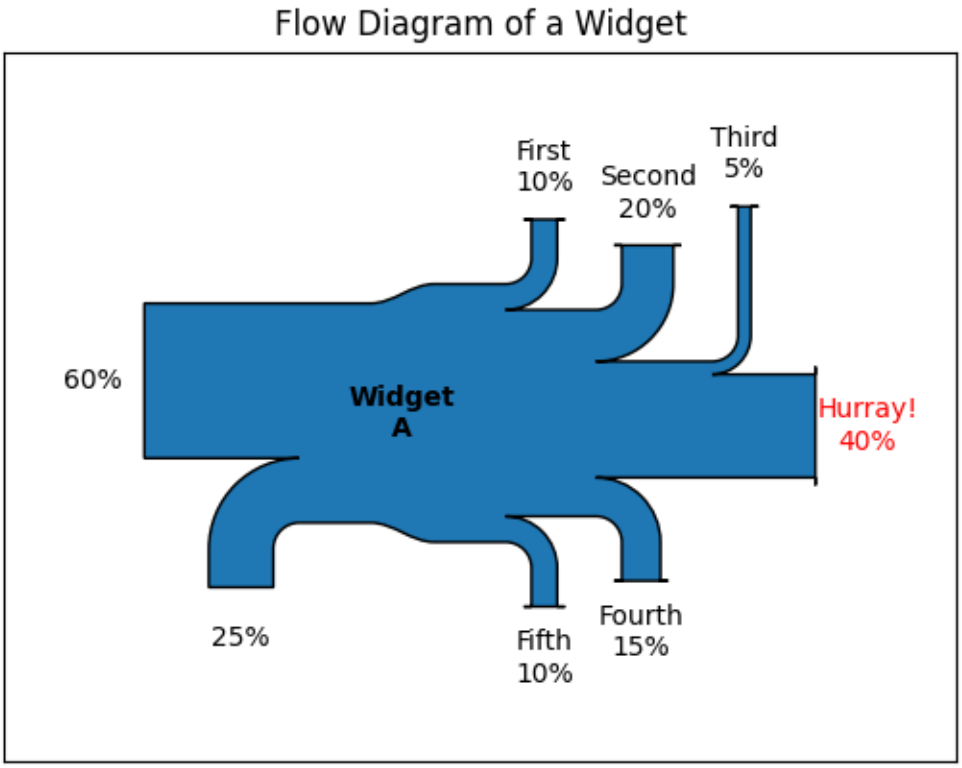
1. Axes weren't provided when Sankey() was instantiated, so they were created automatically.
2. The scale argument wasn't necessary since the data was already normalized.
3. By default, the lengths of the paths are justified.

Example 2

This demonstrates:

1. Setting one path longer than the others
2. Placing a label in the middle of the diagram
3. Using the scale argument to normalize the flows
4. Implicitly passing keyword arguments to PathPatch()
5. Changing the angle of the arrow heads
6. Changing the offset between the tips of the paths and their labels
7. Formatting the numbers in the path labels and the associated unit
8. Changing the appearance of the patch and the labels after the figure is created

```
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1, xticks=[], yticks=[],
                     title="Flow Diagram of a Widget")
sankey = Sankey(ax=ax, scale=0.01, offset=0.2, head_angle=180,
                format='%.0f', unit='%')
sankey.add(flows=[25, 0, 60, -10, -20, -5, -15, -10, -40],
           labels=['', '', '', 'First', 'Second', 'Third', 'Fourth',
                   'Fifth', 'Hurray!'],
           orientations=[-1, 1, 0, 1, 1, 1, -1, -1, 0],
           pathlengths=[0.25, 0.25, 0.25, 0.25, 0.25, 0.6, 0.25, 0.25,
                        0.25],
           patchlabel="Widget\nA")  # Arguments to matplotlib.patches.
↪PathPatch
diagrams = sankey.finish()
diagrams[0].texts[-1].set_color('r')
diagrams[0].text.set_fontweight('bold')
```




Notice:

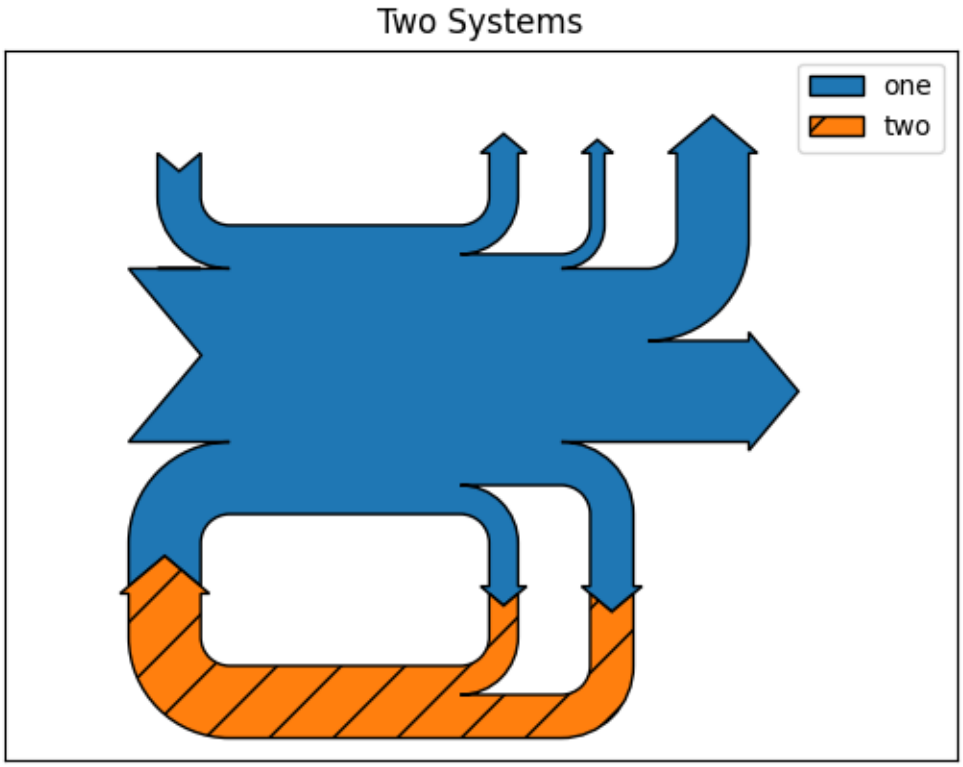
1. Since the sum of the flows is nonzero, the width of the trunk isn't uniform. The matplotlib logging system logs this at the DEBUG level.
2. The second flow doesn't appear because its value is zero. Again, this is logged at the DEBUG level.

Example 3

This demonstrates:

1. Connecting two systems
2. Turning off the labels of the quantities
3. Adding a legend

```
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1, xticks=[], yticks=[], title="Two Systems")
flows = [0.25, 0.15, 0.60, -0.10, -0.05, -0.25, -0.15, -0.10, -0.35]
sankey = Sankey(ax=ax, unit=None)
sankey.add(flows=flows, label='one',
           orientations=[-1, 1, 0, 1, 1, 1, -1, -1, 0])
sankey.add(flows=[-0.25, 0.15, 0.1], label='two',
           orientations=[-1, -1, -1], prior=0, connect=(0, 0))
diagrams = sankey.finish()
diagrams[-1].patch.set_hatch('/')
plt.legend()
```




Notice that only one connection is specified, but the systems form a circuit since: (1) the lengths of the paths are justified and (2) the orientation and ordering of the flows is mirrored.

```
plt.show()
```

**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.sankey*
- *matplotlib.sankey.Sankey*
- *matplotlib.sankey.Sankey.add*
- *matplotlib.sankey.Sankey.finish*

## Long chain of connections using Sankey

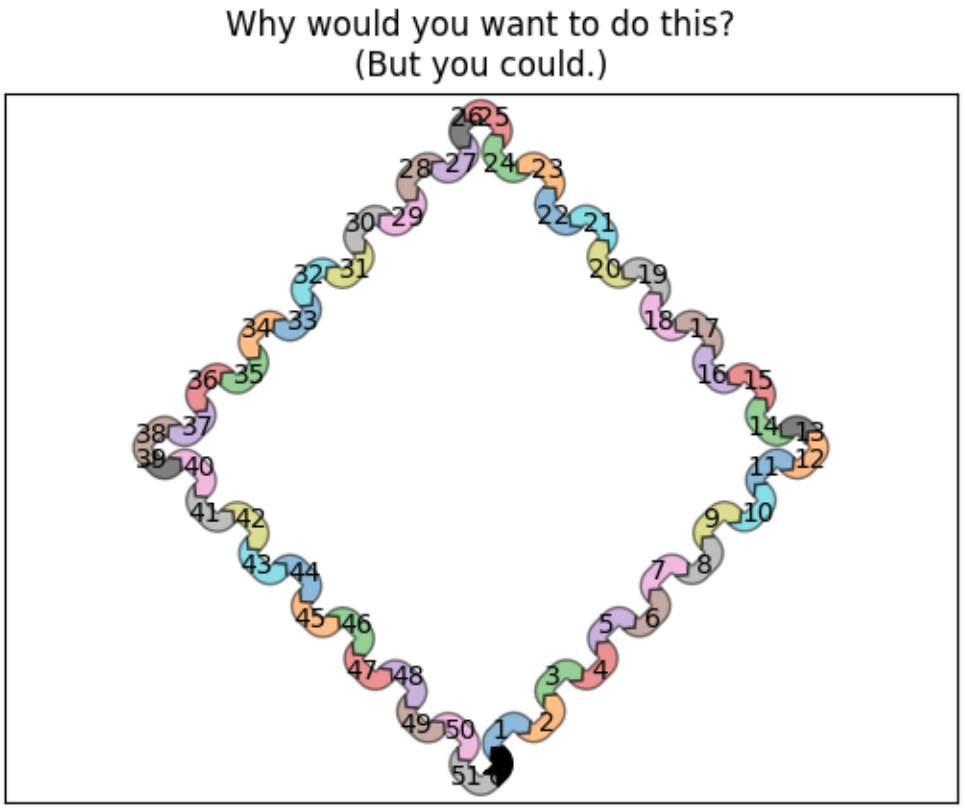
Demonstrate/test the Sankey class by producing a long chain of connections.

```
import matplotlib.pyplot as plt

from matplotlib.sankey import Sankey

links_per_side = 6


def side(sankey, n=1):
    """Generate a side chain."""
    prior = len(sankey.diagrams)
    for i in range(0, 2*n, 2):
        sankey.add(flows=[1, -1], orientations=[-1, -1],
                   patchlabel=str(prior + i),
                   prior=prior + i - 1, connect=(1, 0), alpha=0.5)
        sankey.add(flows=[1, -1], orientations=[1, 1],
                   patchlabel=str(prior + i + 1),
                   prior=prior + i, connect=(1, 0), alpha=0.5)


def corner(sankey):
    """Generate a corner link."""
    prior = len(sankey.diagrams)
    sankey.add(flows=[1, -1], orientations=[0, 1],
               patchlabel=str(prior), facecolor='k',
               prior=prior - 1, connect=(1, 0), alpha=0.5)


fig = plt.figure()
ax = fig.add_subplot(1, 1, 1, xticks=[], yticks=[],
                     title="Why would you want to do this?\n(But you could.)")
sankey = Sankey(ax=ax, unit=None)
sankey.add(flows=[1, -1], orientations=[0, 1],
```

(continues on next page)

(continued from previous page)

```
            patchlabel="0", facecolor='k',
            rotation=45)
side(sankey, n=links_per_side)
corner(sankey)
side(sankey, n=links_per_side)
corner(sankey)
side(sankey, n=links_per_side)
corner(sankey)
side(sankey, n=links_per_side)
sankey.finish()
# Notice:
# 1. The alignment doesn't drift significantly (if at all; with 16007
#    subdiagrams there is still closure).
# 2. The first diagram is rotated 45 deg, so all other diagrams are rotated
#    accordingly.

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.sankey*
- *matplotlib.sankey.Sankey*
- *matplotlib.sankey.Sankey.add*
- *matplotlib.sankey.Sankey.finish*

### Rankine power cycle

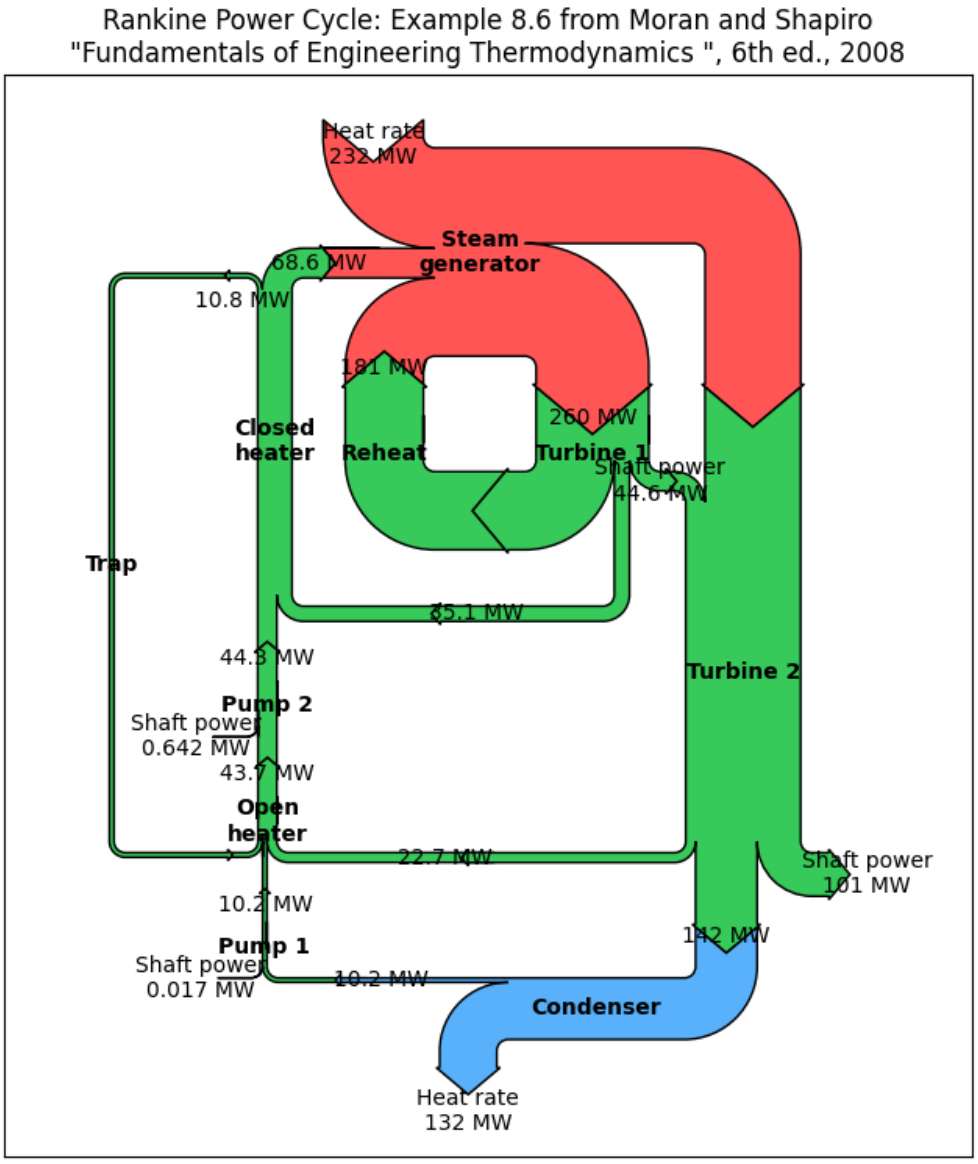
Demonstrate the Sankey class with a practical example of a Rankine power cycle.

```
import matplotlib.pyplot as plt

from matplotlib.sankey import Sankey

fig = plt.figure(figsize=(8, 9))
ax = fig.add_subplot(1, 1, 1, xticks=[], yticks=[],
                     title="Rankine Power Cycle: Example 8.6 from Moran and "
                     "Shapiro\n\x22Fundamentals of Engineering Thermodynamics
 ↪"
                     "\x22, 6th ed., 2008")
Hdot = [260.431, 35.078, 180.794, 221.115, 22.700,
        142.361, 10.193, 10.210, 43.670, 44.312,
        68.631, 10.758, 10.758, 0.017, 0.642,
        232.121, 44.559, 100.613, 132.168]  # MW
sankey = Sankey(ax=ax, format='%.3G', unit=' MW', gap=0.5, scale=1.0/Hdot[0])
sankey.add(patchlabel='\n\nPump 1', rotation=90, facecolor='#37c959',
           flows=[Hdot[13], Hdot[6], -Hdot[7]],
           labels=['Shaft power', '', None],
           pathlengths=[0.4, 0.883, 0.25],
           orientations=[1, -1, 0])
sankey.add(patchlabel='\n\nOpen\nheater', facecolor='#37c959',
           flows=[Hdot[11], Hdot[7], Hdot[4], -Hdot[8]],
           labels=[None, '', None, None],
           pathlengths=[0.25, 0.25, 1.93, 0.25],
           orientations=[1, 0, -1, 0], prior=0, connect=(2, 1))
sankey.add(patchlabel='\n\nPump 2', facecolor='#37c959',
           flows=[Hdot[14], Hdot[8], -Hdot[9]],
           labels=['Shaft power', '', None],
           pathlengths=[0.4, 0.25, 0.25],
           orientations=[1, 0, 0], prior=1, connect=(3, 1))
sankey.add(patchlabel='Closed\nheater', trunklength=2.914, fc='#37c959',
           flows=[Hdot[9], Hdot[1], -Hdot[11], -Hdot[10]],
           pathlengths=[0.25, 1.543, 0.25, 0.25],
           labels=['', '', None, None],
           orientations=[0, -1, 1, -1], prior=2, connect=(2, 0))
sankey.add(patchlabel='Trap', facecolor='#37c959', trunklength=5.102,
           flows=[Hdot[11], -Hdot[12]],
           labels=['\n', None],
           pathlengths=[1.0, 1.01],
```

(continues on next page)

(continued from previous page)

```
            orientations=[1, 1], prior=3, connect=(2, 0))
sankey.add(patchlabel='Steam\ngenerator', facecolor='#ff5555',
            flows=[Hdot[15], Hdot[10], Hdot[2], -Hdot[3], -Hdot[0]],
            labels=['Heat rate', '', '', None, None],
            pathlengths=0.25,
            orientations=[1, 0, -1, -1, -1], prior=3, connect=(3, 1))
sankey.add(patchlabel='\n\n\nTurbine 1', facecolor='#37c959',
            flows=[Hdot[0], -Hdot[16], -Hdot[1], -Hdot[2]],
            labels=['', None, None, None],
            pathlengths=[0.25, 0.153, 1.543, 0.25],
            orientations=[0, 1, -1, -1], prior=5, connect=(4, 0))
sankey.add(patchlabel='\n\n\nReheat', facecolor='#37c959',
            flows=[Hdot[2], -Hdot[2]],
            labels=[None, None],
            pathlengths=[0.725, 0.25],
            orientations=[-1, 0], prior=6, connect=(3, 0))
sankey.add(patchlabel='Turbine 2', trunklength=3.212, facecolor='#37c959',
            flows=[Hdot[3], Hdot[16], -Hdot[5], -Hdot[4], -Hdot[17]],
            labels=[None, 'Shaft power', None, '', 'Shaft power'],
            pathlengths=[0.751, 0.15, 0.25, 1.93, 0.25],
            orientations=[0, -1, 0, -1, 1], prior=6, connect=(1, 1))
sankey.add(patchlabel='Condenser', facecolor='#58b1fa', trunklength=1.764,
            flows=[Hdot[5], -Hdot[18], -Hdot[6]],
            labels=['', 'Heat rate', None],
            pathlengths=[0.45, 0.25, 0.883],
            orientations=[-1, 1, 0], prior=8, connect=(2, 0))
diagrams = sankey.finish()
for diagram in diagrams:
    diagram.text.set_fontweight('bold')
    diagram.text.set_fontsize('10')
    for text in diagram.texts:
        text.set_fontsize('10')
# Notice that the explicit connections are handled automatically, but the
# implicit ones currently are not.  The lengths of the paths and the trunks
# must be adjusted manually, and that is a bit tricky.

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.sankey*
- *matplotlib.sankey.Sankey*
- *matplotlib.sankey.Sankey.add*

- *matplotlib.sankey.Sankey.finish*

### SkewT-logP diagram: using transforms and custom projections

This serves as an intensive exercise of Matplotlib's transforms and custom projection API. This example produces a so-called SkewT-logP diagram, which is a common plot in meteorology for displaying vertical profiles of temperature. As far as Matplotlib is concerned, the complexity comes from having X and Y axes that are not orthogonal. This is handled by including a skew component to the basic Axes transforms. Additional complexity comes in handling the fact that the upper and lower X-axes have different data ranges, which necessitates a bunch of custom classes for ticks, spines, and axis to handle this.

```
from contextlib import ExitStack

from matplotlib.axes import Axes
import matplotlib.axis as maxis
from matplotlib.projections import register_projection
import matplotlib.spines as mspines
import matplotlib.transforms as transforms


# The sole purpose of this class is to look at the upper, lower, or total
# interval as appropriate and see what parts of the tick to draw, if any.
class SkewXTick(maxis.XTick):
    def draw(self, renderer):
        # When adding the callbacks with `stack.callback`, we fetch the␣
↪current
        # visibility state of the artist with `get_visible`; the ExitStack␣
↪will
        # restore these states (`set_visible`) at the end of the block (after
        # the draw).
        with ExitStack() as stack:
            for artist in [self.gridline, self.tick1line, self.tick2line,
                           self.label1, self.label2]:
                stack.callback(artist.set_visible, artist.get_visible())
            needs_lower = transforms.interval_contains(
                self.axes.lower_xlim, self.get_loc())
            needs_upper = transforms.interval_contains(
                self.axes.upper_xlim, self.get_loc())
            self.tick1line.set_visible(
                self.tick1line.get_visible() and needs_lower)
            self.label1.set_visible(
                self.label1.get_visible() and needs_lower)
            self.tick2line.set_visible(
                self.tick2line.get_visible() and needs_upper)
            self.label2.set_visible(
                self.label2.get_visible() and needs_upper)
            super().draw(renderer)

    def get_view_interval(self):
        return self.axes.xaxis.get_view_interval()
```

(continues on next page)

(continued from previous page)

```
# This class exists to provide two separate sets of intervals to the tick,
# as well as create instances of the custom tick
class SkewXAxis(maxis.XAxis):
    def _get_tick(self, major):
        return SkewXTick(self.axes, None, major=major)

    def get_view_interval(self):
        return self.axes.upper_xlim[0], self.axes.lower_xlim[1]


# This class exists to calculate the separate data range of the
# upper X-axis and draw the spine there. It also provides this range
# to the X-axis artist for ticking and gridlines
class SkewSpine(mspines.Spine):
    def _adjust_location(self):
        pts = self._path.vertices
        if self.spine_type == 'top':
            pts[:, 0] = self.axes.upper_xlim
        else:
            pts[:, 0] = self.axes.lower_xlim


# This class handles registration of the skew-xaxes as a projection as well
# as setting up the appropriate transformations. It also overrides standard
# spines and axes instances as appropriate.
class SkewXAxes(Axes):
    # The projection must specify a name.  This will be used be the
    # user to select the projection, i.e. ``subplot(projection='skewx')``.
    name = 'skewx'

    def _init_axis(self):
        # Taken from Axes and modified to use our modified X-axis
        self.xaxis = SkewXAxis(self)
        self.spines.top.register_axis(self.xaxis)
        self.spines.bottom.register_axis(self.xaxis)
        self.yaxis = maxis.YAxis(self)
        self.spines.left.register_axis(self.yaxis)
        self.spines.right.register_axis(self.yaxis)

    def _gen_axes_spines(self):
        spines = {'top': SkewSpine.linear_spine(self, 'top'),
                  'bottom': mspines.Spine.linear_spine(self, 'bottom'),
                  'left': mspines.Spine.linear_spine(self, 'left'),
                  'right': mspines.Spine.linear_spine(self, 'right')}
        return spines

    def _set_lim_and_transforms(self):
        """
        This is called once when the plot is created to set up all the
        transforms for the data, text and grids.
```

(continues on next page)

(continued from previous page)

```
        """
        rot = 30

        # Get the standard transform setup from the Axes base class
        super()._set_lim_and_transforms()

        # Need to put the skew in the middle, after the scale and limits,
        # but before the transAxes. This way, the skew is done in Axes
        # coordinates thus performing the transform around the proper origin
        # We keep the pre-transAxes transform around for other users, like the
        # spines for finding bounds
        self.transDataToAxes = (
            self.transScale
            + self.transLimits
            + transforms.Affine2D().skew_deg(rot, 0)
        )
        # Create the full transform from Data to Pixels
        self.transData = self.transDataToAxes + self.transAxes

        # Blended transforms like this need to have the skewing applied using
        # both axes, in axes coords like before.
        self._xaxis_transform = (
            transforms.blended_transform_factory(
                self.transScale + self.transLimits,
                transforms.IdentityTransform())
            + transforms.Affine2D().skew_deg(rot, 0)
            + self.transAxes
        )

    @property
    def lower_xlim(self):
        return self.axes.viewLim.intervalx

    @property
    def upper_xlim(self):
        pts = [[0., 1.], [1., 1.]]
        return self.transDataToAxes.inverted().transform(pts)[:, 0]


# Now register the projection with matplotlib so the user can select it.
register_projection(SkewXAxes)

if __name__ == '__main__':
    # Now make a simple example using the custom projection.
    from io import StringIO

    import matplotlib.pyplot as plt
    import numpy as np

    from matplotlib.ticker import (MultipleLocator, NullFormatter,
                                   ScalarFormatter)
```

(continues on next page)

(continued from previous page)

```
    # Some example data.
    data_txt = '''
        978.0    345    7.8    0.8
        971.0    404    7.2    0.2
        946.7    610    5.2   -1.8
        944.0    634    5.0   -2.0
        925.0    798    3.4   -2.6
        911.8    914    2.4   -2.7
        906.0    966    2.0   -2.7
        877.9   1219    0.4   -3.2
        850.0   1478   -1.3   -3.7
        841.0   1563   -1.9   -3.8
        823.0   1736    1.4   -0.7
        813.6   1829    4.5    1.2
        809.0   1875    6.0    2.2
        798.0   1988    7.4   -0.6
        791.0   2061    7.6   -1.4
        783.9   2134    7.0   -1.7
        755.1   2438    4.8   -3.1
        727.3   2743    2.5   -4.4
        700.5   3048    0.2   -5.8
        700.0   3054    0.2   -5.8
        698.0   3077    0.0   -6.0
        687.0   3204   -0.1   -7.1
        648.9   3658   -3.2  -10.9
        631.0   3881   -4.7  -12.7
        600.7   4267   -6.4  -16.7
        592.0   4381   -6.9  -17.9
        577.6   4572   -8.1  -19.6
        555.3   4877  -10.0  -22.3
        536.0   5151  -11.7  -24.7
        533.8   5182  -11.9  -25.0
        500.0   5680  -15.9  -29.9
        472.3   6096  -19.7  -33.4
        453.0   6401  -22.4  -36.0
        400.0   7310  -30.7  -43.7
        399.7   7315  -30.8  -43.8
        387.0   7543  -33.1  -46.1
        382.7   7620  -33.8  -46.8
        342.0   8398  -40.5  -53.5
        320.4   8839  -43.7  -56.7
        318.0   8890  -44.1  -57.1
        310.0   9060  -44.7  -58.7
        306.1   9144  -43.9  -57.9
        305.0   9169  -43.7  -57.7
        300.0   9280  -43.5  -57.5
        292.0   9462  -43.7  -58.7
        276.0   9838  -47.1  -62.1
        264.0  10132  -47.5  -62.5
        251.0  10464  -49.7  -64.7
        250.0  10490  -49.7  -64.7
        247.0  10569  -48.7  -63.7
```

(continues on next page)

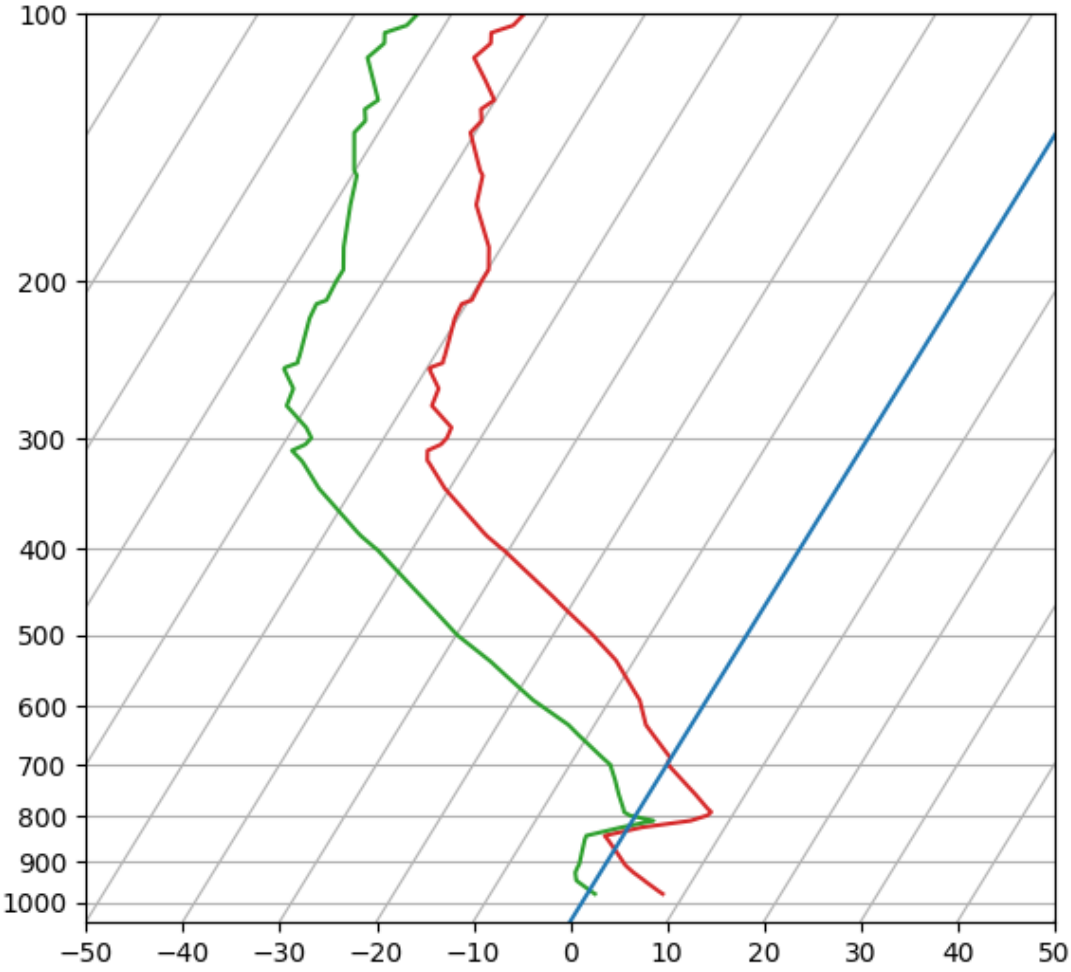
(continued from previous page)

```
    244.0  10649  -48.9  -63.9
    243.3  10668  -48.9  -63.9
    220.0  11327  -50.3  -65.3
    212.0  11569  -50.5  -65.5
    210.0  11631  -49.7  -64.7
    200.0  11950  -49.9  -64.9
    194.0  12149  -49.9  -64.9
    183.0  12529  -51.3  -66.3
    164.0  13233  -55.3  -68.3
    152.0  13716  -56.5  -69.5
    150.0  13800  -57.1  -70.1
    136.0  14414  -60.5  -72.5
    132.0  14600  -60.1  -72.1
    131.4  14630  -60.2  -72.2
    128.0  14792  -60.9  -72.9
    125.0  14939  -60.1  -72.1
    119.0  15240  -62.2  -73.8
    112.0  15616  -64.9  -75.9
    108.0  15838  -64.1  -75.1
    107.8  15850  -64.1  -75.1
    105.0  16010  -64.7  -75.7
    103.0  16128  -62.9  -73.9
    100.0  16310  -62.5  -73.5
'''

# Parse the data
sound_data = StringIO(data_txt)
p, h, T, Td = np.loadtxt(sound_data, unpack=True)

# Create a new figure. The dimensions here give a good aspect ratio
fig = plt.figure(figsize=(6.5875, 6.2125))
ax = fig.add_subplot(projection='skewx')

plt.grid(True)

# Plot the data using normal plotting functions, in this case using
# log scaling in Y, as dictated by the typical meteorological plot
ax.semilogy(T, p, color='C3')
ax.semilogy(Td, p, color='C2')

# An example of a slanted line at constant X
l = ax.axvline(0, color='C0')

# Disables the log-formatting that comes with semilogy
ax.yaxis.set_major_formatter(ScalarFormatter())
ax.yaxis.set_minor_formatter(NullFormatter())
ax.set_yticks(np.linspace(100, 1000, 10))
ax.set_ylim(1050, 100)

ax.xaxis.set_major_locator(MultipleLocator(10))
ax.set_xlim(-50, 50)
```

(continues on next page)

(continued from previous page)

```
    plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.transforms*
- *matplotlib.spines*
- *matplotlib.spines.Spine*
- *matplotlib.spines.Spine.register_axis*
- *matplotlib.projections*

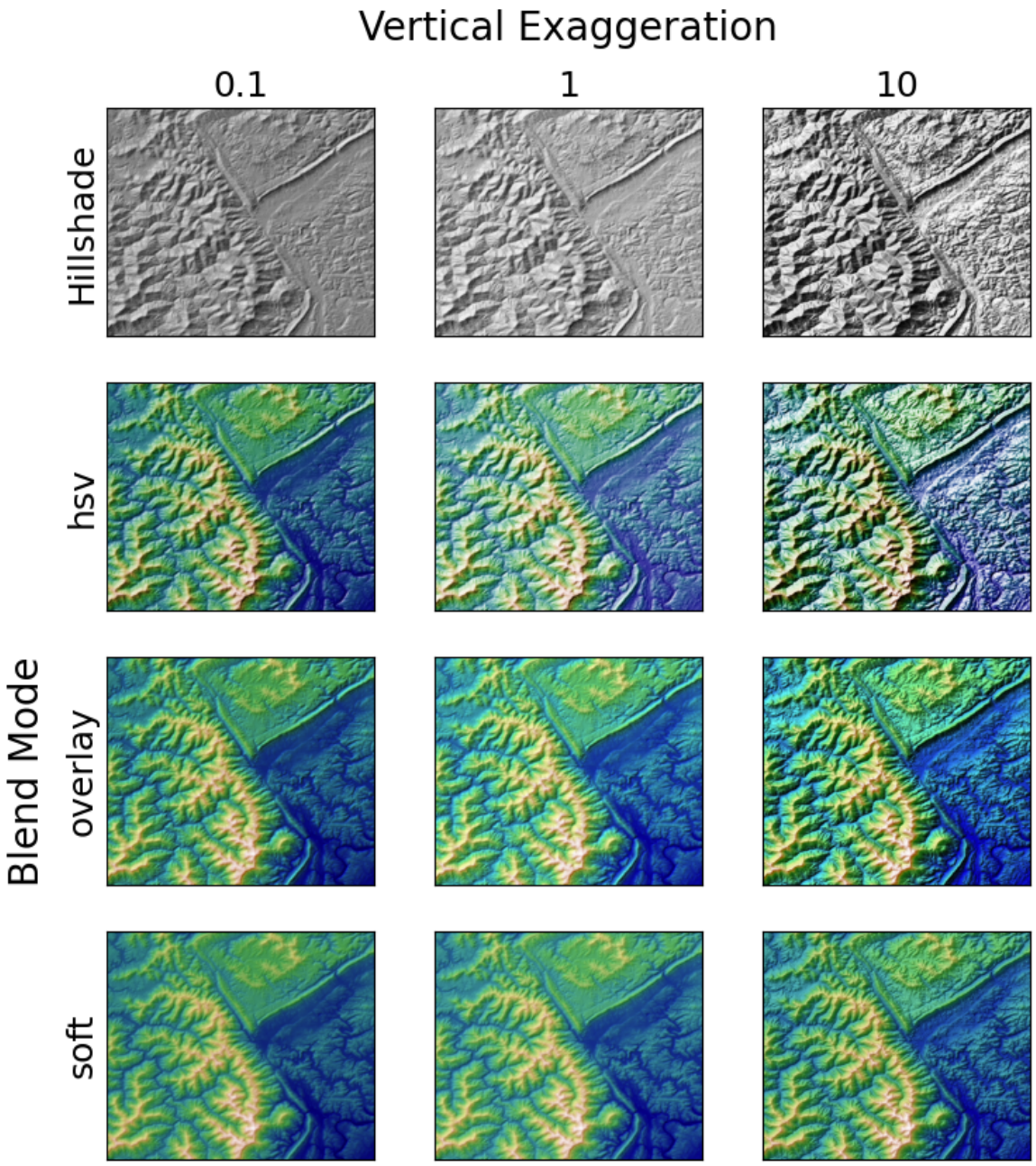
- *matplotlib.projections.register_projection*

### Topographic hillshading

Demonstrates the visual effect of varying blend mode and vertical exaggeration on "hillshaded" plots.

Note that the "overlay" and "soft" blend modes work well for complex surfaces such as this example, while the default "hsv" blend mode works best for smooth surfaces such as many mathematical functions.

In most cases, hillshading is used purely for visual purposes, and *dx*/*dy* can be safely ignored. In that case, you can tweak *vert_exag* (vertical exaggeration) by trial and error to give the desired visual effect. However, this example demonstrates how to use the *dx* and *dy* keyword arguments to ensure that the *vert_exag* parameter is the true vertical exaggeration.




```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.cbook import get_sample_data
from matplotlib.colors import LightSource

dem = get_sample_data('jacksboro_fault_dem.npz')
z = dem['elevation']
```

(continues on next page)

(continued from previous page)

```
# -- Optional dx and dy for accurate vertical exaggeration ----------------
# If you need topographically accurate vertical exaggeration, or you don't want
# to guess at what *vert_exag* should be, you'll need to specify the cellsize
# of the grid (i.e. the *dx* and *dy* parameters).  Otherwise, any *vert_exag*
# value you specify will be relative to the grid spacing of your input data
# (in other words, *dx* and *dy* default to 1.0, and *vert_exag* is calculated
# relative to those parameters).  Similarly, *dx* and *dy* are assumed to be in
# the same units as your input z-values.  Therefore, we'll need to convert the
# given dx and dy from decimal degrees to meters.
dx, dy = dem['dx'], dem['dy']
dy = 111200 * dy
dx = 111200 * dx * np.cos(np.radians(dem['ymin']))
# ------------------------------------------------------------------------

# Shade from the northwest, with the sun 45 degrees from horizontal
ls = LightSource(azdeg=315, altdeg=45)
cmap = plt.cm.gist_earth

fig, axs = plt.subplots(nrows=4, ncols=3, figsize=(8, 9))
plt.setp(axs.flat, xticks=[], yticks=[])

# Vary vertical exaggeration and blend mode and plot all combinations
for col, ve in zip(axs.T, [0.1, 1, 10]):
    # Show the hillshade intensity image in the first row
    col[0].imshow(ls.hillshade(z, vert_exag=ve, dx=dx, dy=dy), cmap='gray')

    # Place hillshaded plots with different blend modes in the rest of the rows
    for ax, mode in zip(col[1:], ['hsv', 'overlay', 'soft']):
        rgb = ls.shade(z, cmap=cmap, blend_mode=mode,
                       vert_exag=ve, dx=dx, dy=dy)
        ax.imshow(rgb)

# Label rows and columns
for ax, ve in zip(axs[0], [0.1, 1, 10]):
    ax.set_title(f'{ve}', size=18)
for ax, mode in zip(axs[:, 0], ['Hillshade', 'hsv', 'overlay', 'soft']):
    ax.set_ylabel(mode, size=18)

# Group labels...
axs[0, 1].annotate('Vertical Exaggeration', (0.5, 1), xytext=(0, 30),
                   textcoords='offset points', xycoords='axes fraction',
                   ha='center', va='bottom', size=20)
axs[2, 0].annotate('Blend Mode', (0, 0.5), xytext=(-30, 0),
                   textcoords='offset points', xycoords='axes fraction',
                   ha='right', va='center', size=20, rotation=90)
fig.subplots_adjust(bottom=0.05, right=0.95)
```

(continues on next page)

(continued from previous page)

```
plt.show()
```

**Total running time of the script:** (0 minutes 1.507 seconds)

### 6.25.20 Spines

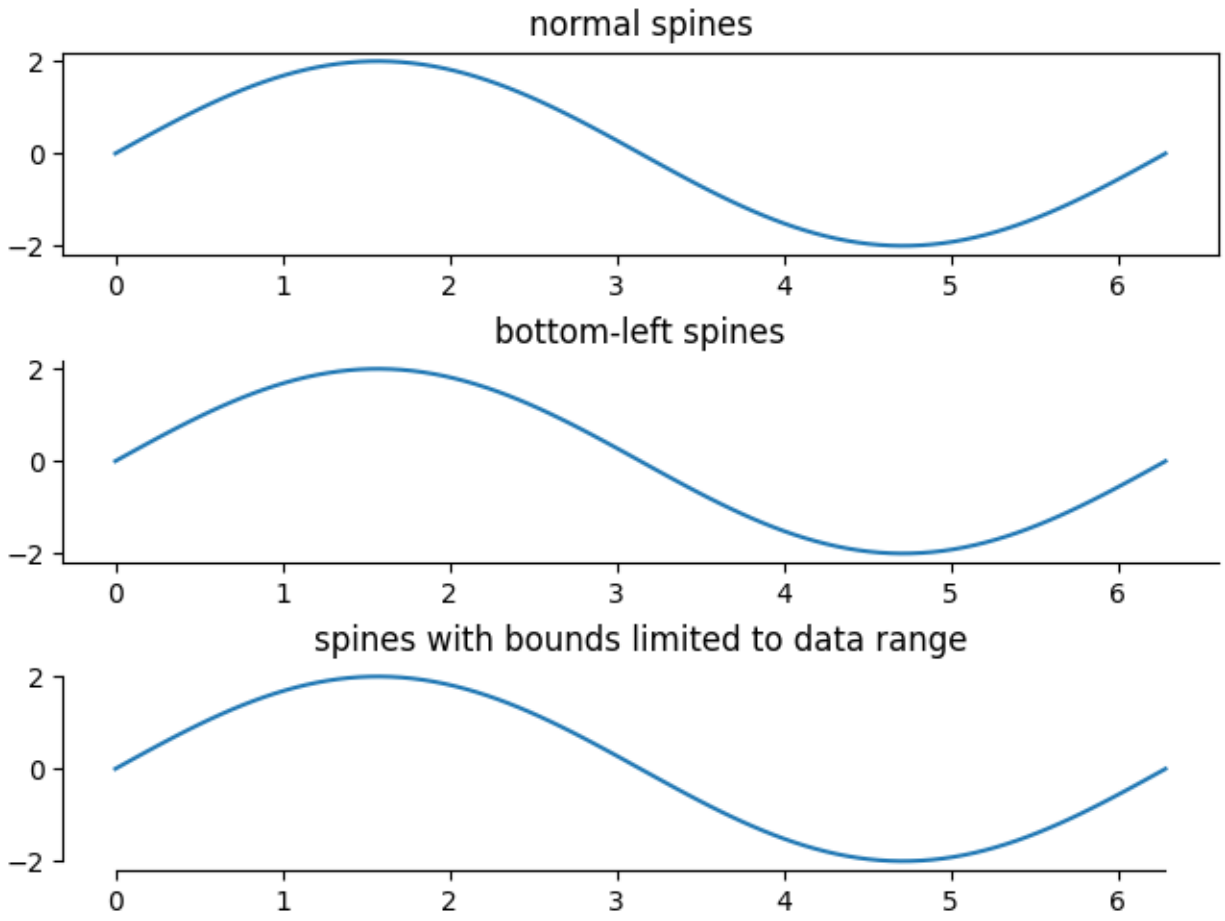
#### Spines

This demo compares:

- normal Axes, with spines on all four sides;
- an Axes with spines only on the left and bottom;
- an Axes using custom bounds to limit the extent of the spine.

Each *axes.Axes* has a list of *Spine* objects, accessible via the container `ax.spines`.

```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 2 * np.pi, 100)
y = 2 * np.sin(x)

# Constrained layout makes sure the labels don't overlap the Axes.
fig, (ax0, ax1, ax2) = plt.subplots(nrows=3, layout='constrained')

ax0.plot(x, y)
ax0.set_title('normal spines')

ax1.plot(x, y)
ax1.set_title('bottom-left spines')

# Hide the right and top spines
ax1.spines.right.set_visible(False)
ax1.spines.top.set_visible(False)

ax2.plot(x, y)
ax2.set_title('spines with bounds limited to data range')

# Only draw spines for the data range, not in the margins
ax2.spines.bottom.set_bounds(x.min(), x.max())
ax2.spines.left.set_bounds(y.min(), y.max())
# Hide the right and top spines
ax2.spines.right.set_visible(False)
ax2.spines.top.set_visible(False)

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.artist.Artist.set_visible*
- *matplotlib.spines.Spine.set_bounds*

## Spine placement

The position of the axis spines can be influenced using *set_position*.

Note: If you want to obtain arrow heads at the ends of the axes, also check out the *Centered spines with arrows* example.

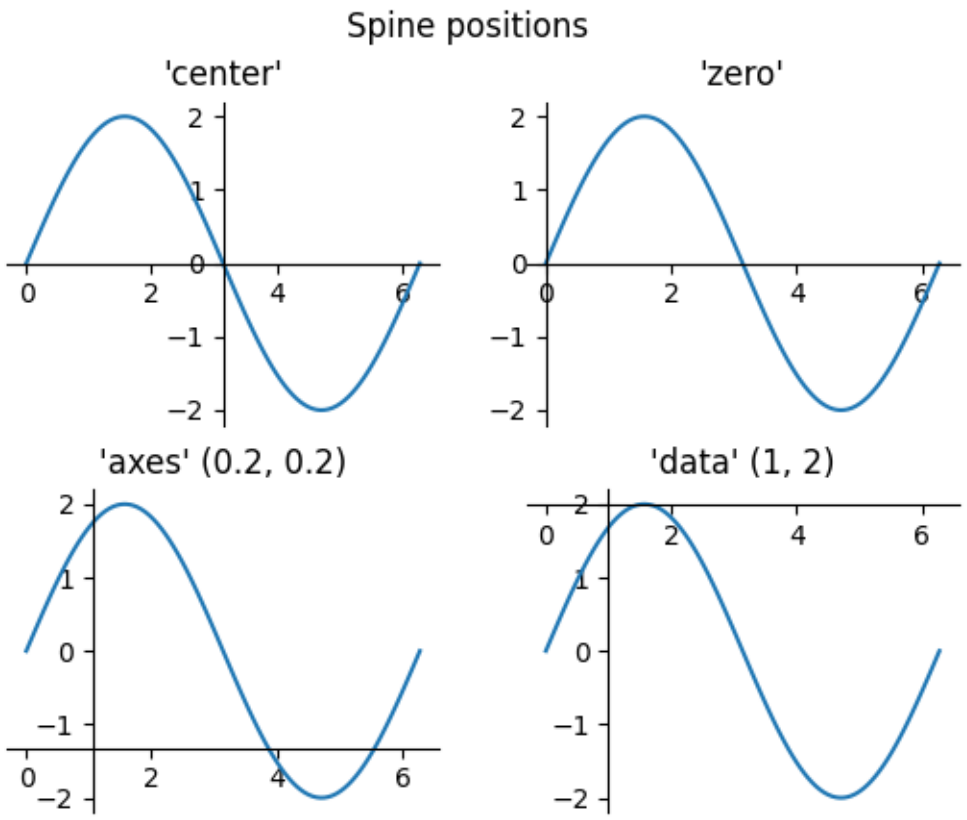
```
import matplotlib.pyplot as plt
import numpy as np
```

```
x = np.linspace(0, 2*np.pi, 100)
y = 2 * np.sin(x)
```

(continues on next page)

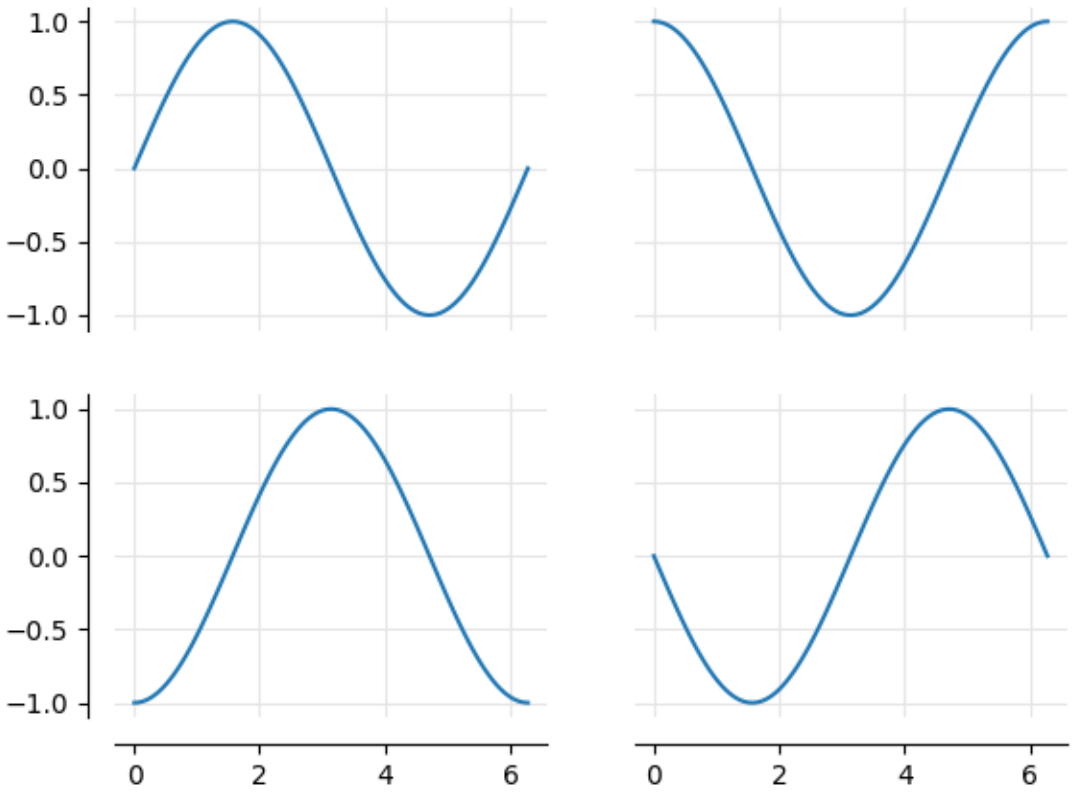
(continued from previous page)

```
fig, ax_dict = plt.subplot_mosaic(
    [['center', 'zero'],
     ['axes', 'data']]
)
fig.suptitle('Spine positions')


ax = ax_dict['center']
ax.set_title("'center'")
ax.plot(x, y)
ax.spines[['left', 'bottom']].set_position('center')
ax.spines[['top', 'right']].set_visible(False)

ax = ax_dict['zero']
ax.set_title("'zero'")
ax.plot(x, y)
ax.spines[['left', 'bottom']].set_position('zero')
ax.spines[['top', 'right']].set_visible(False)

ax = ax_dict['axes']
ax.set_title("'axes' (0.2, 0.2)")
ax.plot(x, y)
ax.spines.left.set_position(('axes', 0.2))
ax.spines.bottom.set_position(('axes', 0.2))
ax.spines[['top', 'right']].set_visible(False)

ax = ax_dict['data']
ax.set_title("'data' (1, 2)")
ax.plot(x, y)
ax.spines.left.set_position(('data', 1))
ax.spines.bottom.set_position(('data', 2))
ax.spines[['top', 'right']].set_visible(False)

plt.show()
```




### Dropped spines

Demo of spines offset from the axes (a.k.a. "dropped spines").




```
import matplotlib.pyplot as plt
import numpy as np


def adjust_spines(ax, visible_spines):
    ax.label_outer(remove_inner_ticks=True)
    ax.grid(color='0.9')

    for loc, spine in ax.spines.items():
        if loc in visible_spines:
            spine.set_position(('outward', 10))  # outward by 10 points
        else:
            spine.set_visible(False)


x = np.linspace(0, 2 * np.pi, 100)

fig, axs = plt.subplots(2, 2)

axs[0, 0].plot(x, np.sin(x))
axs[0, 1].plot(x, np.cos(x))
axs[1, 0].plot(x, -np.cos(x))
axs[1, 1].plot(x, -np.sin(x))
```

(continues on next page)

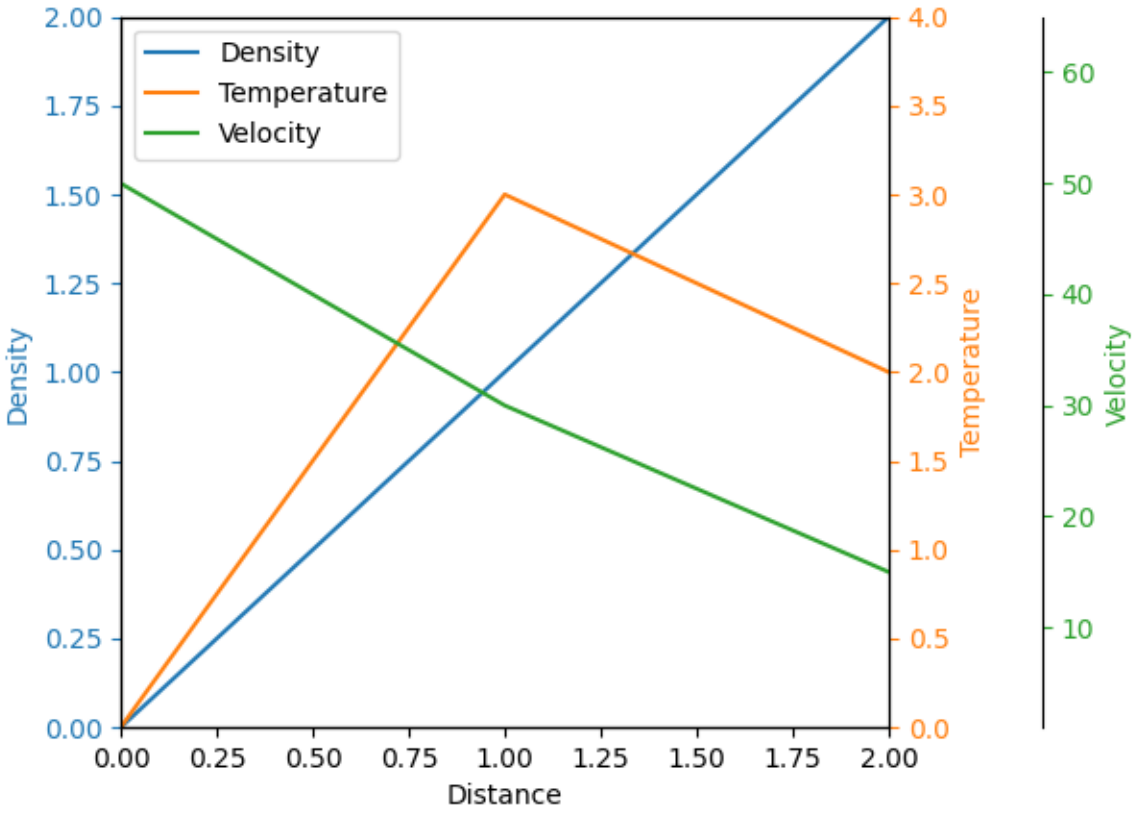
(continued from previous page)

```
adjust_spines(axs[0, 0], ['left'])
adjust_spines(axs[0, 1], [])
adjust_spines(axs[1, 0], ['left', 'bottom'])
adjust_spines(axs[1, 1], ['bottom'])

plt.show()
```

### Multiple y-axis with Spines

Create multiple y axes with a shared x-axis. This is done by creating a *twinx* Axes, turning all spines but the right one invisible and offset its position using *set_position*.

Note that this approach uses *matplotlib.axes.Axes* and their *Spine*s. Alternative approaches using non-standard Axes are shown in the *Parasite Axes demo* and *Parasite axis demo* examples.




```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
fig.subplots_adjust(right=0.75)
```

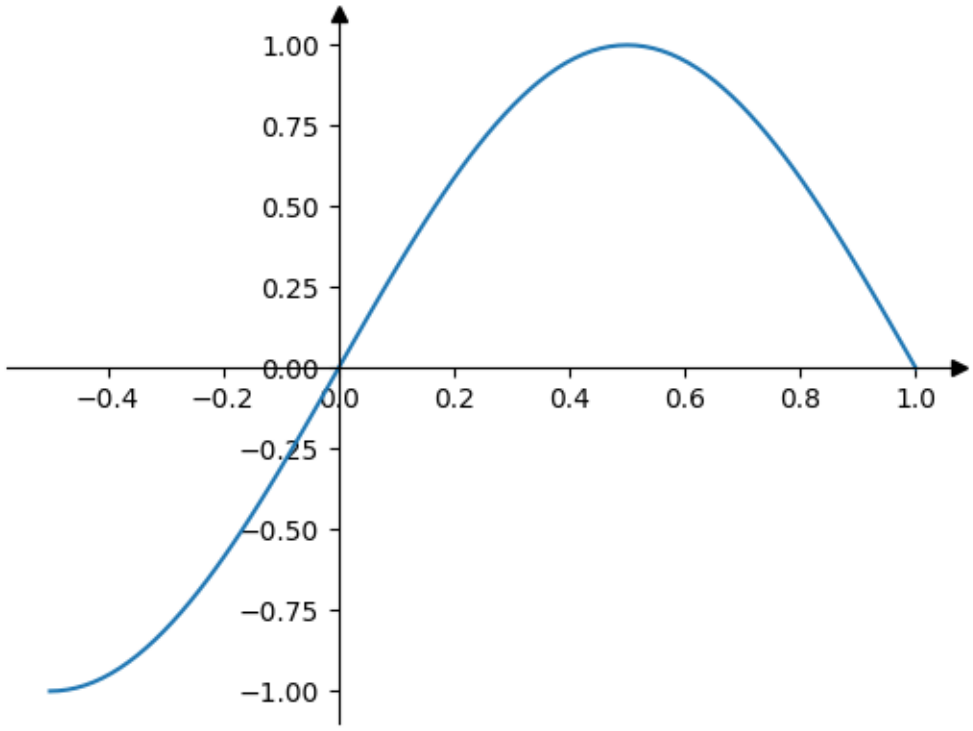
(continues on next page)

(continued from previous page)

```
twin1 = ax.twinx()
twin2 = ax.twinx()

# Offset the right spine of twin2.  The ticks and label have already been
# placed on the right by twinx above.
twin2.spines.right.set_position(("axes", 1.2))

p1, = ax.plot([0, 1, 2], [0, 1, 2], "C0", label="Density")
p2, = twin1.plot([0, 1, 2], [0, 3, 2], "C1", label="Temperature")
p3, = twin2.plot([0, 1, 2], [50, 30, 15], "C2", label="Velocity")

ax.set(xlim=(0, 2), ylim=(0, 2), xlabel="Distance", ylabel="Density")
twin1.set(ylim=(0, 4), ylabel="Temperature")
twin2.set(ylim=(1, 65), ylabel="Velocity")

ax.yaxis.label.set_color(p1.get_color())
twin1.yaxis.label.set_color(p2.get_color())
twin2.yaxis.label.set_color(p3.get_color())

ax.tick_params(axis='y', colors=p1.get_color())
twin1.tick_params(axis='y', colors=p2.get_color())
twin2.tick_params(axis='y', colors=p3.get_color())

ax.legend(handles=[p1, p2, p3])

plt.show()
```

### Centered spines with arrows

This example shows a way to draw a "math textbook" style plot, where the spines ("axes lines") are drawn at `x = 0` and `y = 0`, and have arrows at their ends.




```
import matplotlib.pyplot as plt
import numpy as np

fig, ax = plt.subplots()
# Move the left and bottom spines to x = 0 and y = 0, respectively.
ax.spines[["left", "bottom"]].set_position(("data", 0))
# Hide the top and right spines.
ax.spines[["top", "right"]].set_visible(False)

# Draw arrows (as black triangles: ">k"/"^k") at the end of the axes.  In each
# case, one of the coordinates (0) is a data coordinate (i.e., y = 0 or x = 0,
# respectively) and the other one (1) is an axes coordinate (i.e., at the very
# right/top of the axes).  Also, disable clipping (clip_on=False) as the␣
 ↪marker
# actually spills out of the Axes.
ax.plot(1, 0, ">k", transform=ax.get_yaxis_transform(), clip_on=False)
ax.plot(0, 1, "^k", transform=ax.get_xaxis_transform(), clip_on=False)

# Some sample data.
x = np.linspace(-0.5, 1., 100)
ax.plot(x, np.sin(x*np.pi))

plt.show()
```

### 6.25.21 Ticks

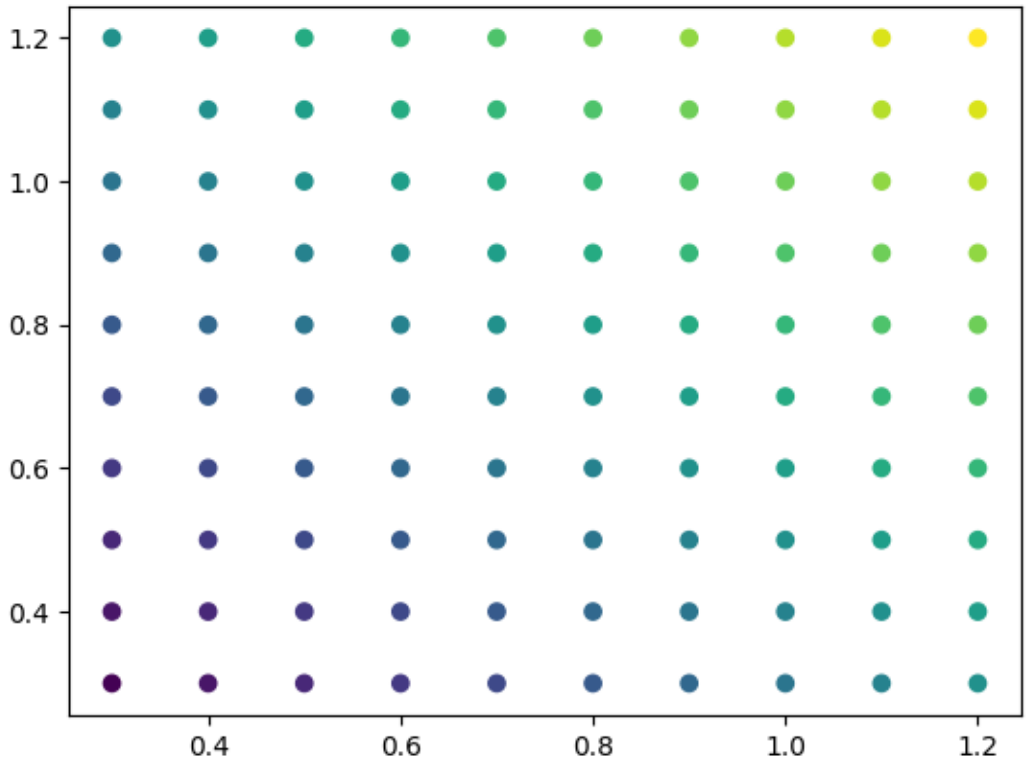
#### Automatically setting tick positions

Setting the behavior of tick auto-placement.

By default, Matplotlib will choose the number of ticks and tick positions so that there is a reasonable number of ticks on the axis and they are located at "round" numbers.

As a result, there may be no ticks on the edges of the plot.

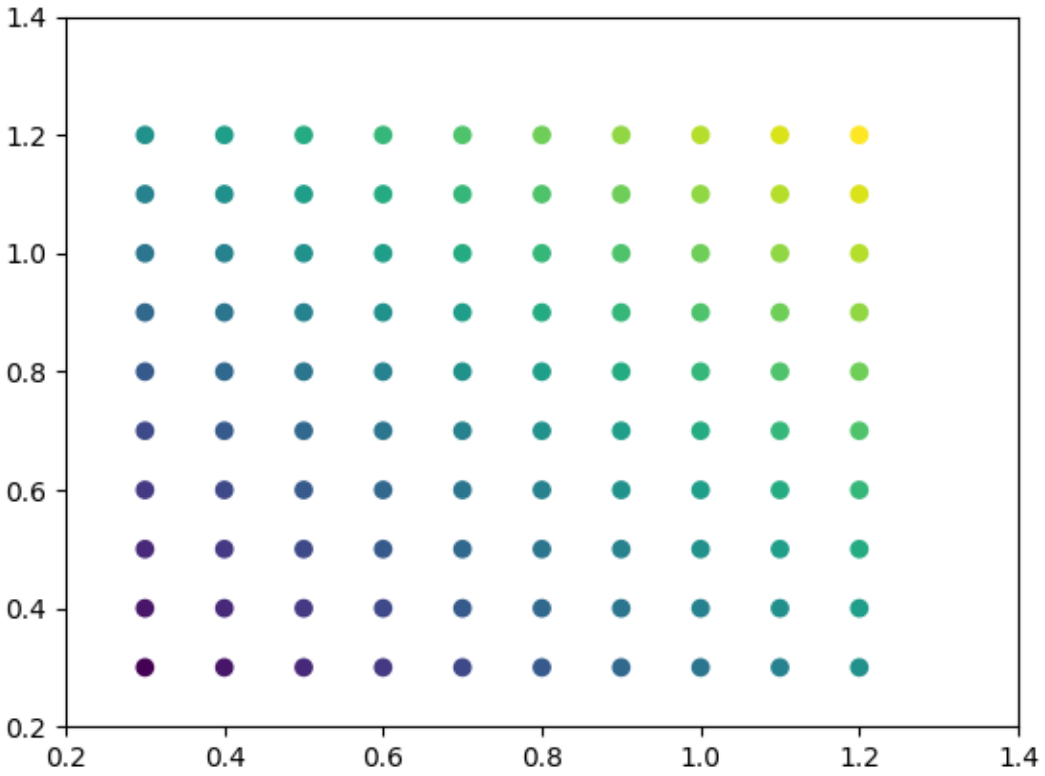
```
import matplotlib.pyplot as plt
import numpy as np

np.random.seed(19680801)

fig, ax = plt.subplots()
dots = np.linspace(0.3, 1.2, 10)
X, Y = np.meshgrid(dots, dots)
x, y = X.ravel(), Y.ravel()
ax.scatter(x, y, c=x+y)
plt.show()
```




If you want to keep ticks at round numbers, and also have ticks at the edges you can switch

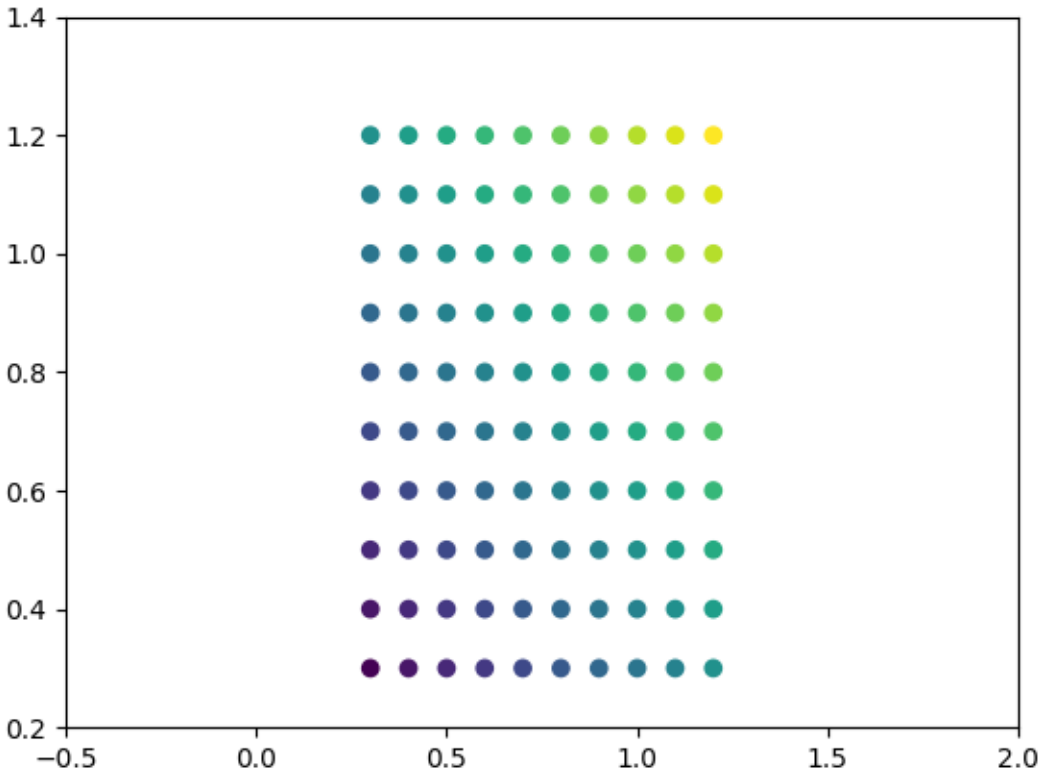
*rcParams["axes.autolimit_mode"]* (default: 'data') to 'round_numbers'. This expands the axis limits to the next round number.

```
plt.rcParams['axes.autolimit_mode'] = 'round_numbers'

# Note: The limits are calculated at draw-time. Therefore, when using
# :rc:`axes.autolimit_mode` in a context manager, it is important that
# the ``show()`` command is within the context.

fig, ax = plt.subplots()
ax.scatter(x, y, c=x+y)
plt.show()
```




The round numbers autolimit_mode is still respected if you set an additional margin around the data using *Axes.set_xmargin* / *Axes.set_ymargin*:

```
fig, ax = plt.subplots()
ax.scatter(x, y, c=x+y)
ax.set_xmargin(0.8)
plt.show()
```




### Centering labels between ticks

Ticklabels are aligned relative to their associated tick. The alignment 'center', 'left', or 'right' can be controlled using the horizontal alignment property:

```
for label in ax.get_xticklabels():
    label.set_horizontalalignment('right')
```

However, there is no direct way to center the labels between ticks. To fake this behavior, one can place a label on the minor ticks in between the major ticks, and hide the major tick labels and minor ticks.

Here is an example that labels the months, centered between the ticks.

```
import matplotlib.pyplot as plt

import matplotlib.cbook as cbook
import matplotlib.dates as dates
import matplotlib.ticker as ticker

# Load some financial data; Google's stock price
r = cbook.get_sample_data('goog.npz')['price_data']
r = r[-250:]  # get the last 250 days

fig, ax = plt.subplots()
ax.plot(r["date"], r["adj_close"])

ax.xaxis.set_major_locator(dates.MonthLocator())
# 16 is a slight approximation since months differ in number of days.
ax.xaxis.set_minor_locator(dates.MonthLocator(bymonthday=16))

ax.xaxis.set_major_formatter(ticker.NullFormatter())
ax.xaxis.set_minor_formatter(dates.DateFormatter('%b'))

# Remove the tick lines
ax.tick_params(axis='x', which='minor', tick1On=False, tick2On=False)
```

(continues on next page)

(continued from previous page)

```
# Align the minor tick label
for label in ax.get_xticklabels(minor=True):
    label.set_horizontalalignment('center')
imid = len(r) // 2
ax.set_xlabel(str(r["date"][imid].item().year))
plt.show()
```

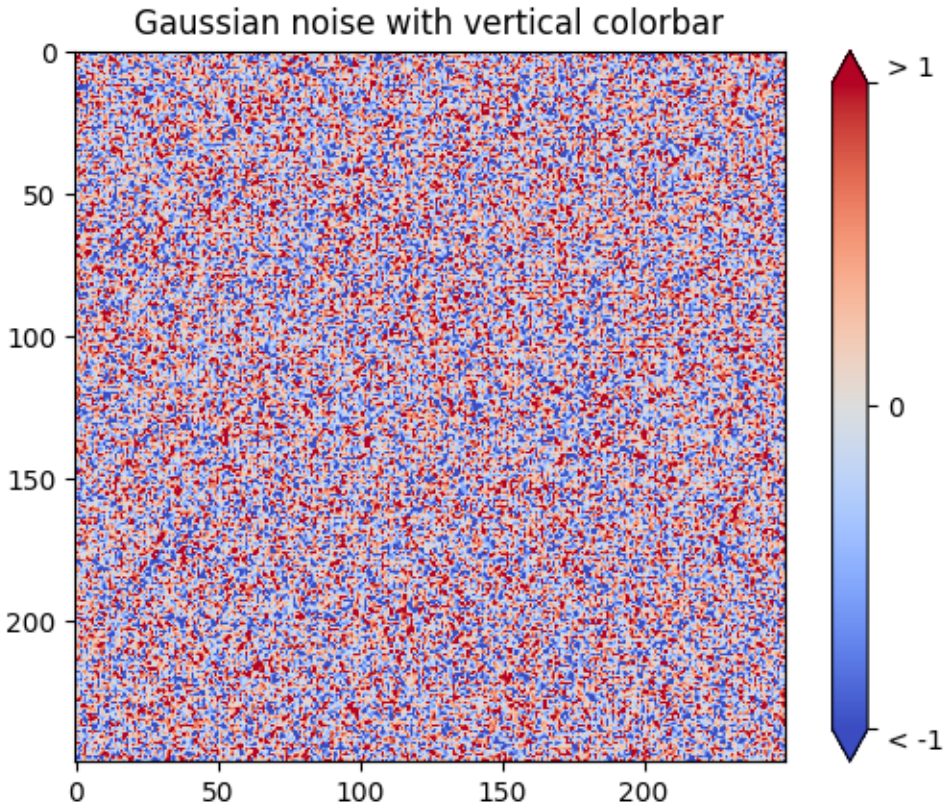
### Colorbar Tick Labelling

Vertical colorbars have ticks, tick labels, and labels visible on the *y* axis, horizontal colorbars on the *x* axis. The `ticks` parameter can be used to set the ticks and the `format` parameter can be used to format the tick labels of the visible colorbar Axes. For further adjustments, the `yaxis` or `xaxis` Axes of the colorbar can be retrieved using its `ax` property.

```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib.ticker as mticker

# Fixing random state for reproducibility
rng = np.random.default_rng(seed=19680801)
```

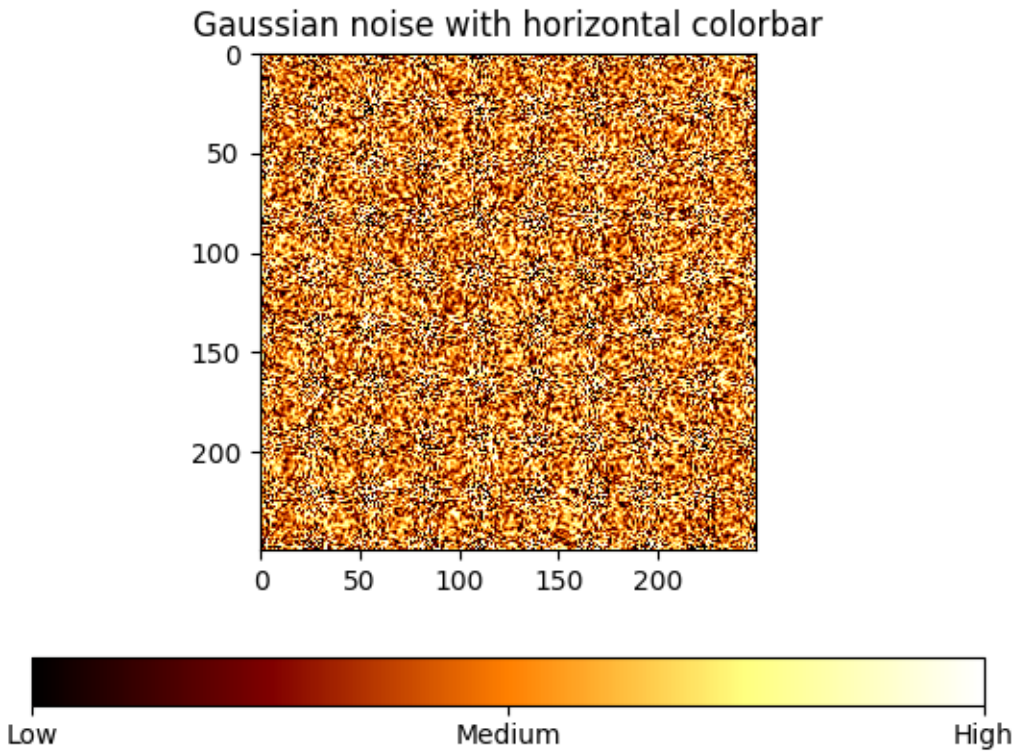
Make plot with vertical (default) colorbar

```
fig, ax = plt.subplots()

data = rng.standard_normal((250, 250))

cax = ax.imshow(data, vmin=-1, vmax=1, cmap='coolwarm')
ax.set_title('Gaussian noise with vertical colorbar')

# Add colorbar, make sure to specify tick locations to match desired␣
↪ticklabels
cbar = fig.colorbar(cax,
                    ticks=[-1, 0, 1],
                    format=mticker.FixedFormatter(['< -1', '0', '> 1']),
                    extend='both'
                    )
labels = cbar.ax.get_yticklabels()
labels[0].set_verticalalignment('top')
labels[-1].set_verticalalignment('bottom')
```




Make plot with horizontal colorbar

```
fig, ax = plt.subplots()

data = np.clip(data, -1, 1)

cax = ax.imshow(data, cmap='afmhot')
ax.set_title('Gaussian noise with horizontal colorbar')

# Add colorbar and adjust ticks afterwards
cbar = fig.colorbar(cax, orientation='horizontal')
cbar.set_ticks(ticks=[-1, 0, 1], labels=['Low', 'Medium', 'High'])

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.colorbar.Colorbar.set_ticks*
- *matplotlib.figure.Figure.colorbar* / *matplotlib.pyplot.colorbar*

## Custom Ticker

The *matplotlib.ticker* module defines many preset tickers, but was primarily designed for extensibility, i.e., to support user customized ticking.

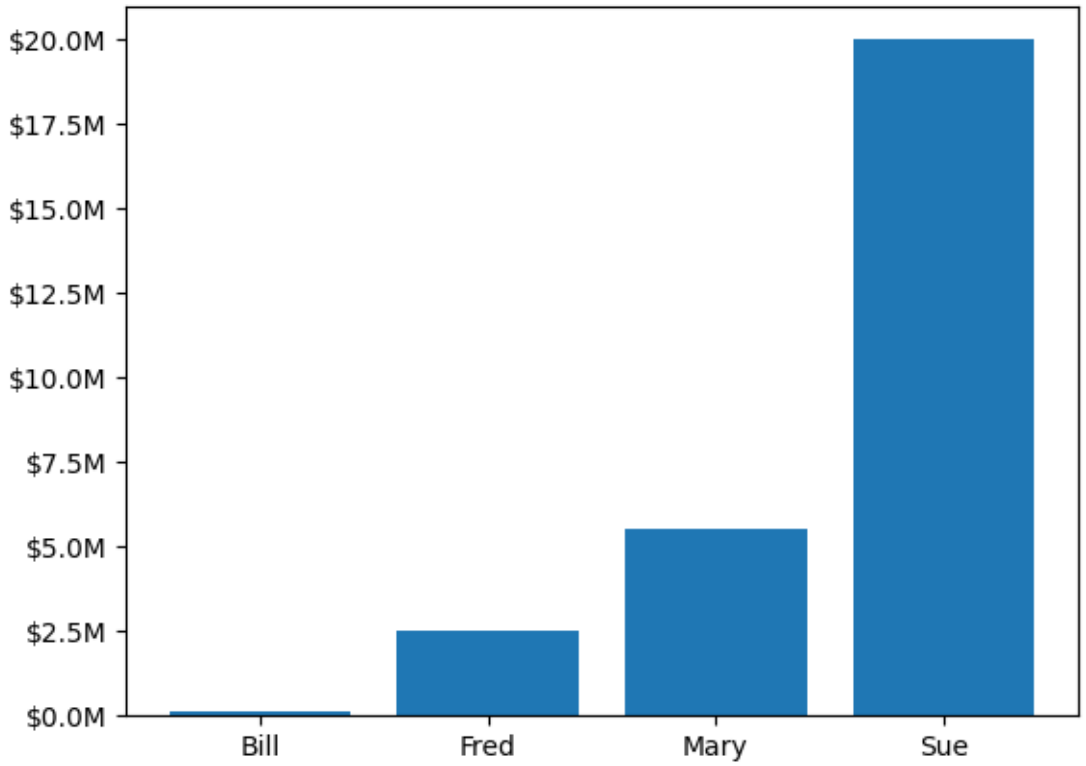
In this example, a user defined function is used to format the ticks in millions of dollars on the y-axis.

```
import matplotlib.pyplot as plt


def millions(x, pos):
    """The two arguments are the value and tick position."""
    return f'${x*1e-6:1.1f}M'
```

(continues on next page)

(continued from previous page)

```
fig, ax = plt.subplots()
# set_major_formatter internally creates a FuncFormatter from the callable.
ax.yaxis.set_major_formatter(millions)
money = [1.5e5, 2.5e6, 5.5e6, 2.0e7]
ax.bar(['Bill', 'Fred', 'Mary', 'Sue'], money)
plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axis.Axis.set_major_formatter*

### Formatting date ticks using ConciseDateFormatter

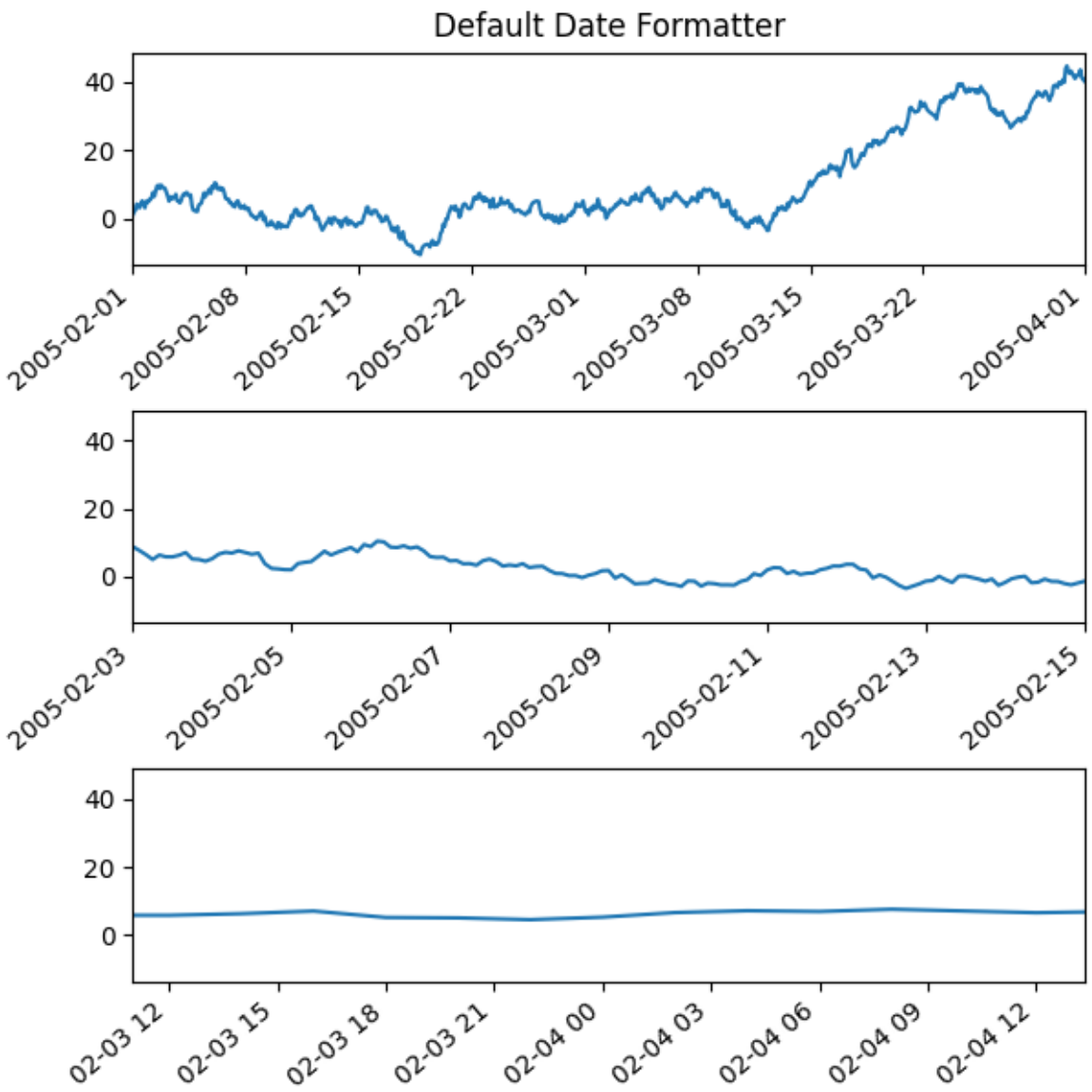
Finding good tick values and formatting the ticks for an axis that has date data is often a challenge. *ConciseDateFormatter* is meant to improve the strings chosen for the ticklabels, and to minimize the strings used in those tick labels as much as possible.

**Note:** This formatter is a candidate to become the default date tick formatter in future versions of Matplotlib. Please report any issues or suggestions for improvement to the GitHub repository or mailing list.

```
import datetime

import matplotlib.pyplot as plt
import numpy as np

import matplotlib.dates as mdates
```

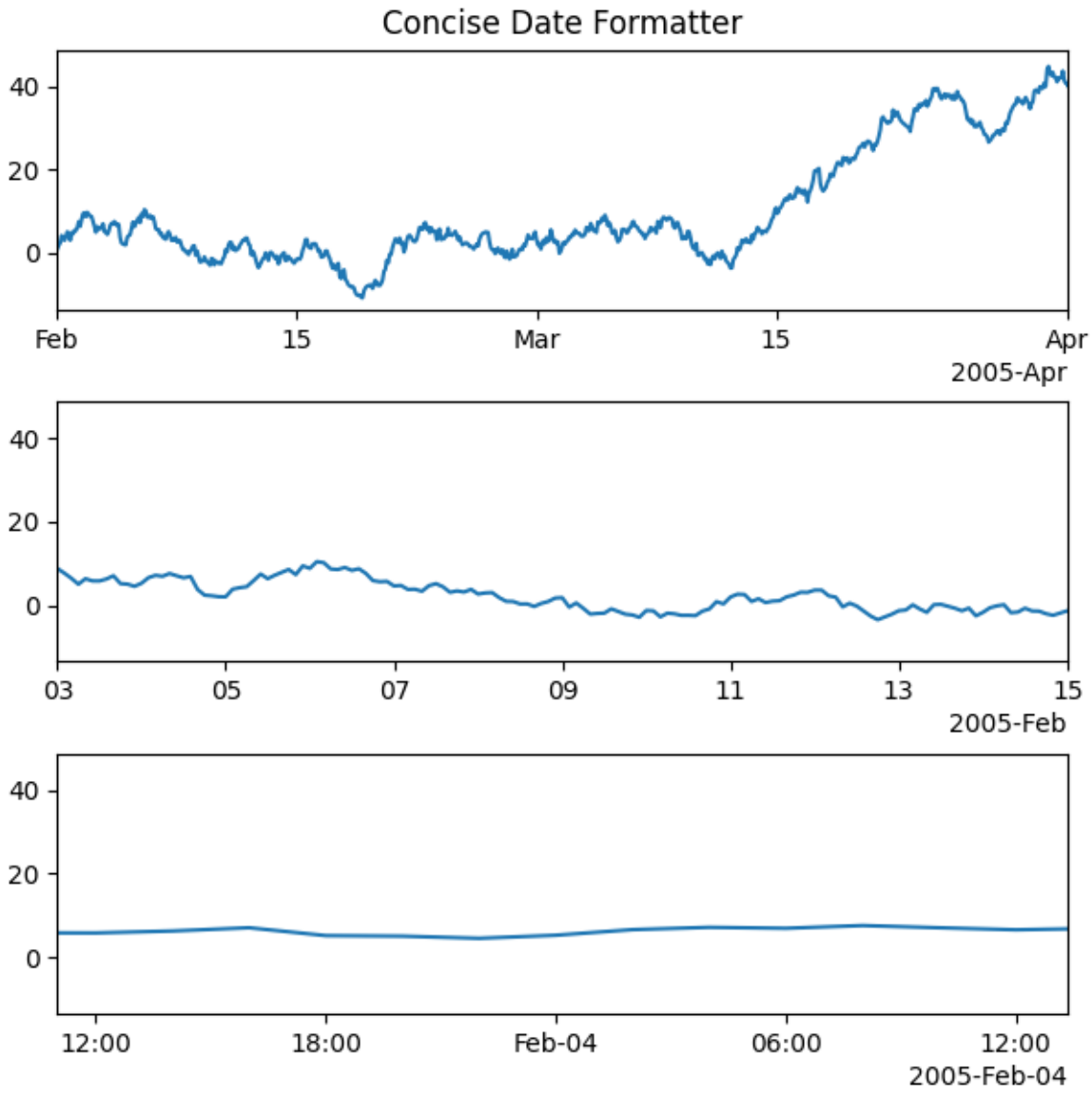
First, the default formatter.

```
base = datetime.datetime(2005, 2, 1)
dates = [base + datetime.timedelta(hours=(2 * i)) for i in range(732)]
N = len(dates)
np.random.seed(19680801)
y = np.cumsum(np.random.randn(N))

fig, axs = plt.subplots(3, 1, layout='constrained', figsize=(6, 6))
lims = [(np.datetime64('2005-02'), np.datetime64('2005-04')),
        (np.datetime64('2005-02-03'), np.datetime64('2005-02-15')),
        (np.datetime64('2005-02-03 11:00'), np.datetime64('2005-02-04 13:20
 ↪'))]
for nn, ax in enumerate(axs):
    ax.plot(dates, y)
    ax.set_xlim(lims[nn])
    # rotate_labels...
    for label in ax.get_xticklabels():
        label.set_rotation(40)
        label.set_horizontalalignment('right')
axs[0].set_title('Default Date Formatter')
plt.show()
```




The default date formatter is quite verbose, so we have the option of using *ConciseDateFormatter*, as shown below. Note that for this example the labels do not need to be rotated as they do for the default formatter because the labels are as small as possible.

```
fig, axs = plt.subplots(3, 1, layout='constrained', figsize=(6, 6))
for nn, ax in enumerate(axs):
    locator = mdates.AutoDateLocator(minticks=3, maxticks=7)
    formatter = mdates.ConciseDateFormatter(locator)
    ax.xaxis.set_major_locator(locator)
    ax.xaxis.set_major_formatter(formatter)

    ax.plot(dates, y)
    ax.set_xlim(lims[nn])
axs[0].set_title('Concise Date Formatter')

plt.show()
```



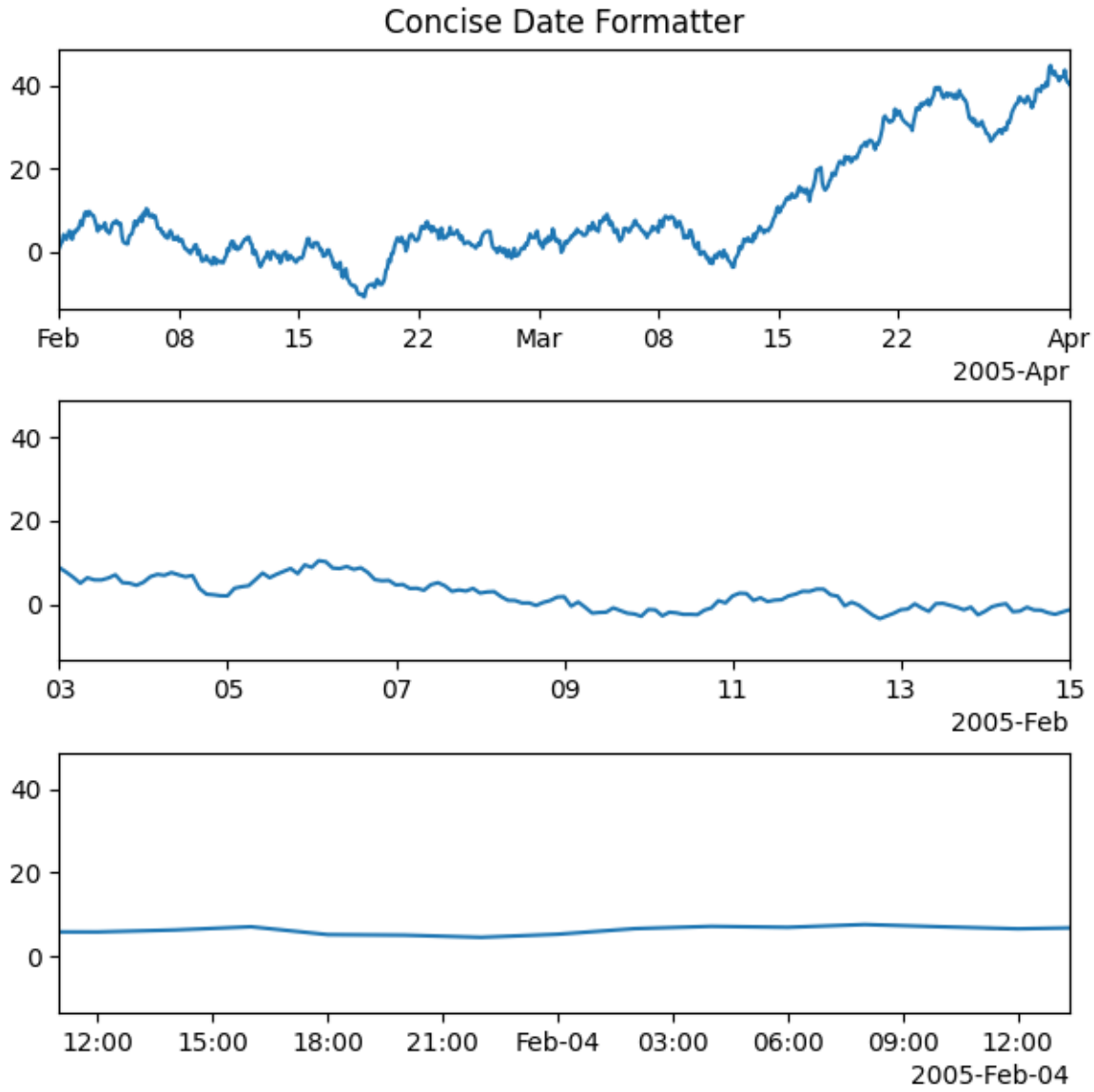

If all calls to axes that have dates are to be made using this converter, it is probably most convenient to use the units registry where you do imports:

```
import matplotlib.units as munits

converter = mdates.ConciseDateConverter()
munits.registry[np.datetime64] = converter
munits.registry[datetime.date] = converter
munits.registry[datetime.datetime] = converter

fig, axs = plt.subplots(3, 1, figsize=(6, 6), layout='constrained')
for nn, ax in enumerate(axs):
    ax.plot(dates, y)
    ax.set_xlim(lims[nn])
axs[0].set_title('Concise Date Formatter')
```

(continues on next page)

(continued from previous page)

```
plt.show()
```




### Localization of date formats

Dates formats can be localized if the default formats are not desirable by manipulating one of three lists of strings.

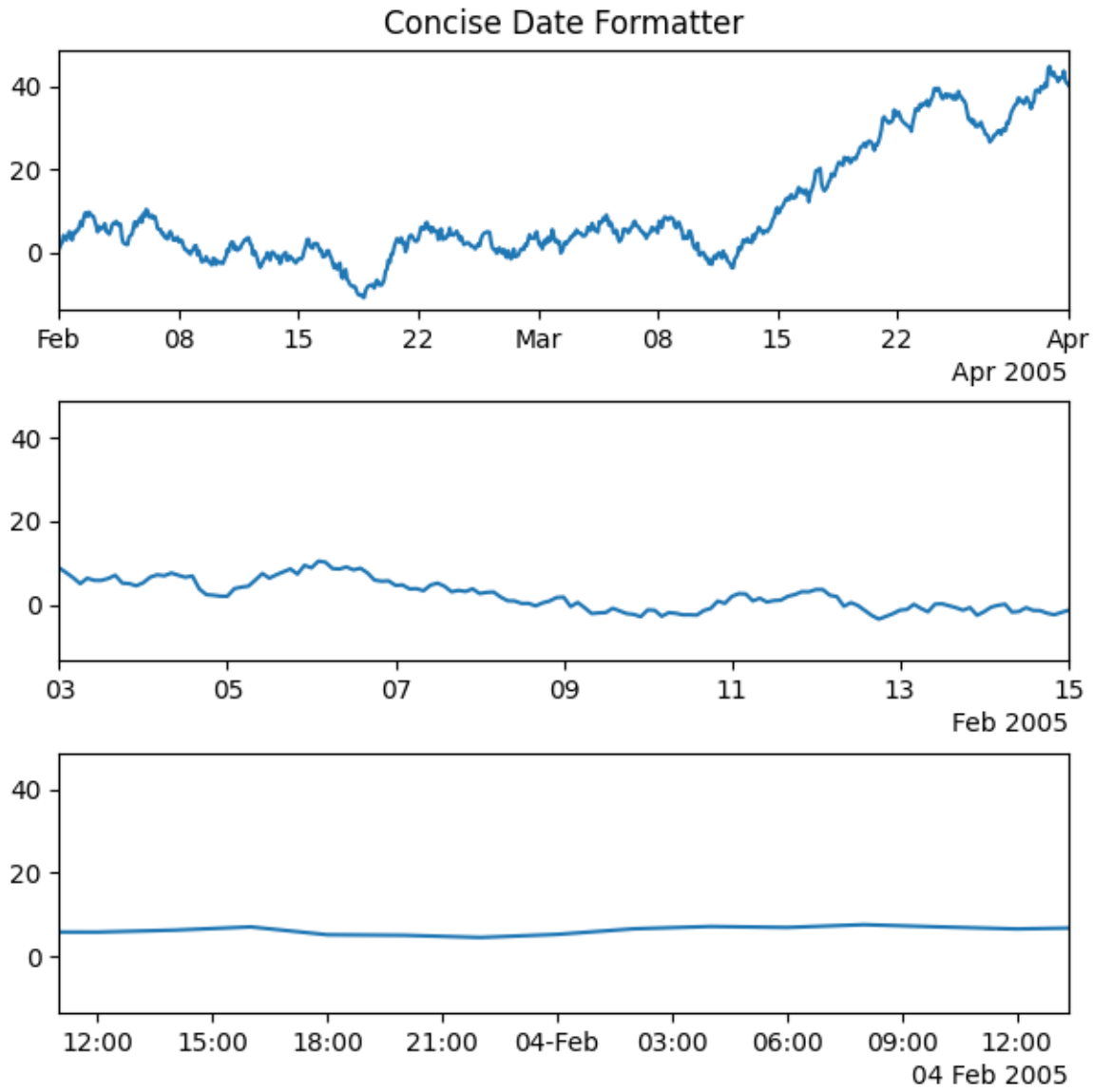
The `formatter.formats` list of formats is for the normal tick labels, There are six levels: years, months, days, hours, minutes, seconds. The `formatter.offset_formats` is how the "offset" string on the right of the axis is formatted. This is usually much more verbose than the tick labels. Finally, the `formatter.zero_formats` are the formats of the ticks that are "zeros". These are tick values that are either the first of the year, month, or day of month, or the zeroth hour, minute, or second. These are usually the same as the format of the ticks a level above. For example if the axis limits mean the ticks are mostly days, then we

label 1 Mar 2005 simply with a "Mar". If the axis limits are mostly hours, we label Feb 4 00:00 as simply "Feb-4".

Note that these format lists can also be passed to *ConciseDateFormatter* as optional keyword arguments.

Here we modify the labels to be "day month year", instead of the ISO "year month day":

```
fig, axs = plt.subplots(3, 1, layout='constrained', figsize=(6, 6))

for nn, ax in enumerate(axs):
    locator = mdates.AutoDateLocator()
    formatter = mdates.ConciseDateFormatter(locator)
    formatter.formats = ['%y',  # ticks are mostly years
                         '%b',       # ticks are mostly months
                         '%d',       # ticks are mostly days
                         '%H:%M',    # hrs
                         '%H:%M',    # min
                         '%S.%f', ]  # secs
    # these are mostly just the level above...
    formatter.zero_formats = [''] + formatter.formats[:-1]
    # ...except for ticks that are mostly hours, then it is nice to have
    # month-day:
    formatter.zero_formats[3] = '%d-%b'

    formatter.offset_formats = ['',
                                '%Y',
                                '%b %Y',
                                '%d %b %Y',
                                '%d %b %Y',
                                '%d %b %Y %H:%M', ]
    ax.xaxis.set_major_locator(locator)
    ax.xaxis.set_major_formatter(formatter)

    ax.plot(dates, y)
    ax.set_xlim(lims[nn])
axs[0].set_title('Concise Date Formatter')

plt.show()
```




### Registering a converter with localization

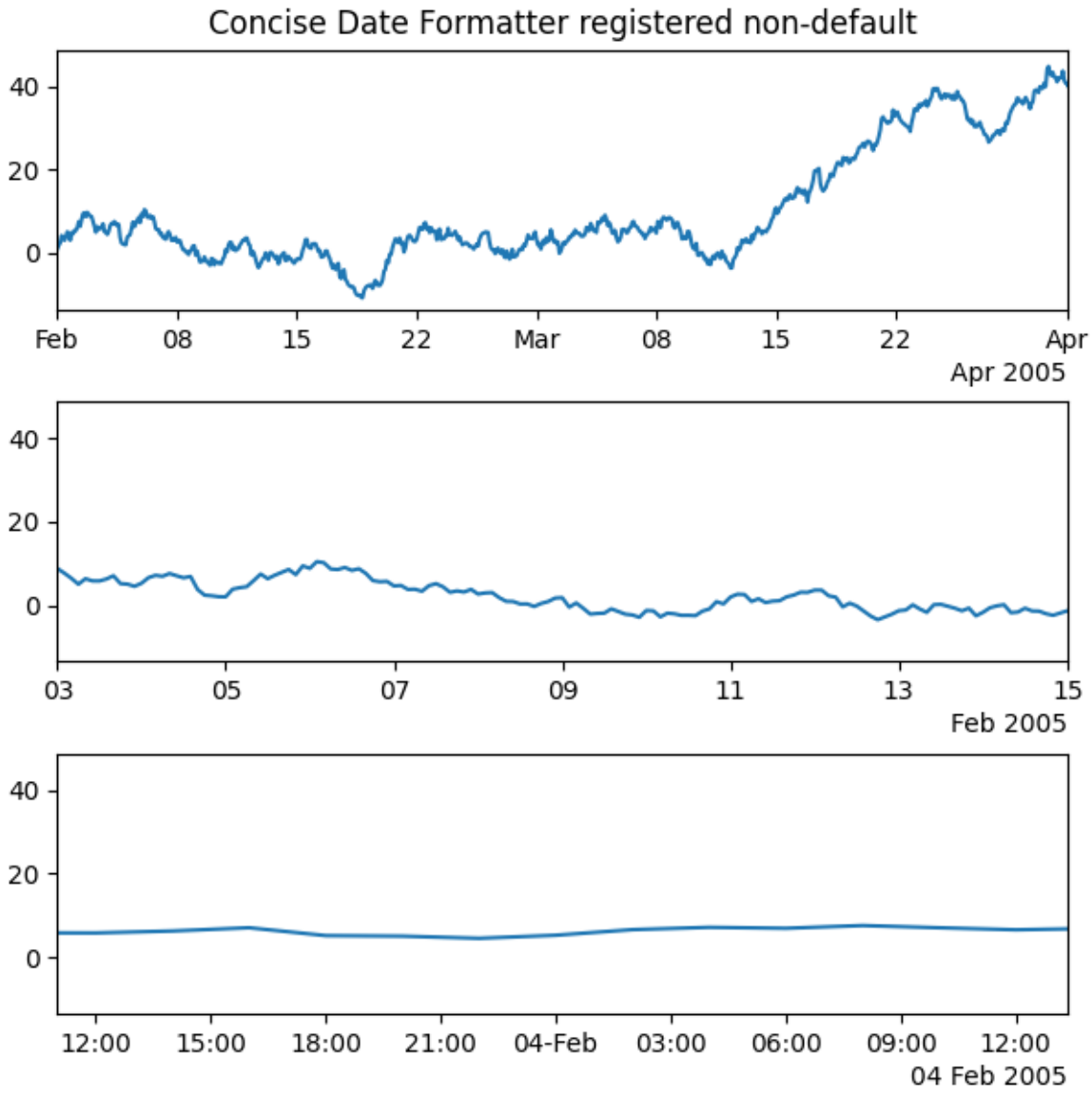
`ConciseDateFormatter` doesn't have rcParams entries, but localization can be accomplished by passing keyword arguments to `ConciseDateConverter` and registering the datatypes you will use with the units registry:

```
import datetime

formats = ['%y',          # ticks are mostly years
           '%b',      # ticks are mostly months
           '%d',      # ticks are mostly days
           '%H:%M',   # hrs
           '%H:%M',   # min
           '%S.%f', ]  # secs
```

(continues on next page)

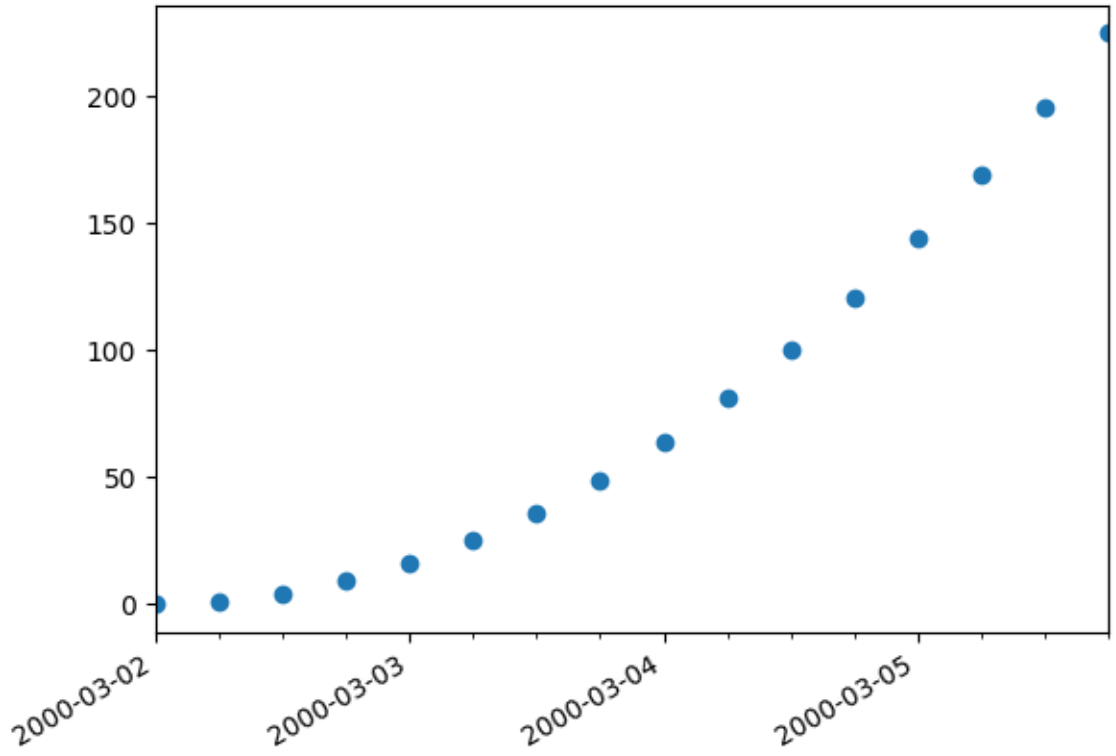
(continued from previous page)

```
# these can be the same, except offset by one level....
zero_formats = [''] + formats[:-1]
# ...except for ticks that are mostly hours, then it's nice to have month-day
zero_formats[3] = '%d-%b'
offset_formats = ['',
                  '%Y',
                  '%b %Y',
                  '%d %b %Y',
                  '%d %b %Y',
                  '%d %b %Y %H:%M', ]

converter = mdates.ConciseDateConverter(
    formats=formats, zero_formats=zero_formats, offset_formats=offset_formats)

munits.registry[np.datetime64] = converter
munits.registry[datetime.date] = converter
munits.registry[datetime.datetime] = converter

fig, axs = plt.subplots(3, 1, layout='constrained', figsize=(6, 6))
for nn, ax in enumerate(axs):
    ax.plot(dates, y)
    ax.set_xlim(lims[nn])
axs[0].set_title('Concise Date Formatter registered non-default')

plt.show()
```




**Total running time of the script:** (0 minutes 3.133 seconds)

### Date Demo Convert




```
import datetime

import matplotlib.pyplot as plt
import numpy as np

from matplotlib.dates import DateFormatter, DayLocator, HourLocator, drange

date1 = datetime.datetime(2000, 3, 2)
date2 = datetime.datetime(2000, 3, 6)
delta = datetime.timedelta(hours=6)
dates = drange(date1, date2, delta)

y = np.arange(len(dates))

fig, ax = plt.subplots()
ax.plot(dates, y**2, 'o')

# this is superfluous, since the autoscaler should get it right, but
# use date2num and num2date to convert between dates and floats if
# you want; both date2num and num2date convert an instance or sequence
ax.set_xlim(dates[0], dates[-1])
```

(continues on next page)

(continued from previous page)

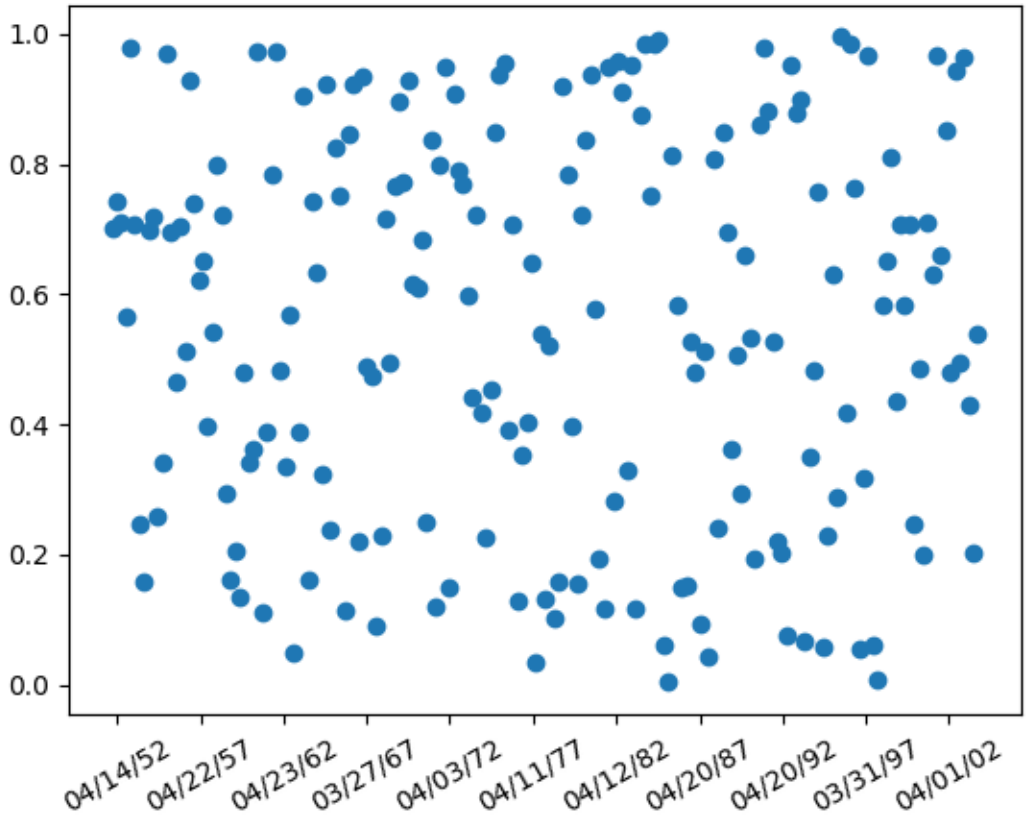
```
# The hour locator takes the hour or sequence of hours you want to
# tick, not the base multiple

ax.xaxis.set_major_locator(DayLocator())
ax.xaxis.set_minor_locator(HourLocator(range(0, 25, 6)))
ax.xaxis.set_major_formatter(DateFormatter('%Y-%m-%d'))

ax.fmt_xdata = DateFormatter('%Y-%m-%d %H:%M:%S')
fig.autofmt_xdate()

plt.show()
```

### Placing date ticks using recurrence rules

The iCalender RFC specifies *recurrence rules* (rrules), that define date sequences. You can use rrules in Matplotlib to place date ticks.

This example sets custom date ticks on every 5th easter.

See https://dateutil.readthedocs.io/en/stable/rrule.html for help with rrules.



```
import datetime

import matplotlib.pyplot as plt
import numpy as np

from matplotlib.dates import (YEARLY, DateFormatter, RRuleLocator, drange,
                              rrulewrapper)

# Fixing random state for reproducibility
np.random.seed(19680801)


# tick every 5th easter
rule = rrulewrapper(YEARLY, byeaster=1, interval=5)
loc = RRuleLocator(rule)
formatter = DateFormatter('%m/%d/%y')
date1 = datetime.date(1952, 1, 1)
date2 = datetime.date(2004, 4, 12)
delta = datetime.timedelta(days=100)

dates = drange(date1, date2, delta)
s = np.random.rand(len(dates))  # make up some random y values


fig, ax = plt.subplots()
plt.plot(dates, s, 'o')
ax.xaxis.set_major_locator(loc)
ax.xaxis.set_major_formatter(formatter)
ax.xaxis.set_tick_params(rotation=30, labelsize=10)

plt.show()
```

### Date tick locators and formatters

This example illustrates the usage and effect of the various date locators and formatters.

```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.dates import (FR, MO, MONTHLY, SA, SU, TH, TU, WE,
                              AutoDateFormatter, AutoDateLocator,
                              ConciseDateFormatter, DateFormatter, DayLocator,
                              HourLocator, MicrosecondLocator, MinuteLocator,
                              MonthLocator, RRuleLocator, SecondLocator,
                              WeekdayLocator, YearLocator, rrulewrapper)
import matplotlib.ticker as ticker


def plot_axis(ax, locator=None, xmax='2002-02-01', fmt=None, formatter=None):
    """Set up common parameters for the Axes in the example."""
    ax.spines[['left', 'right', 'top']].set_visible(False)
```

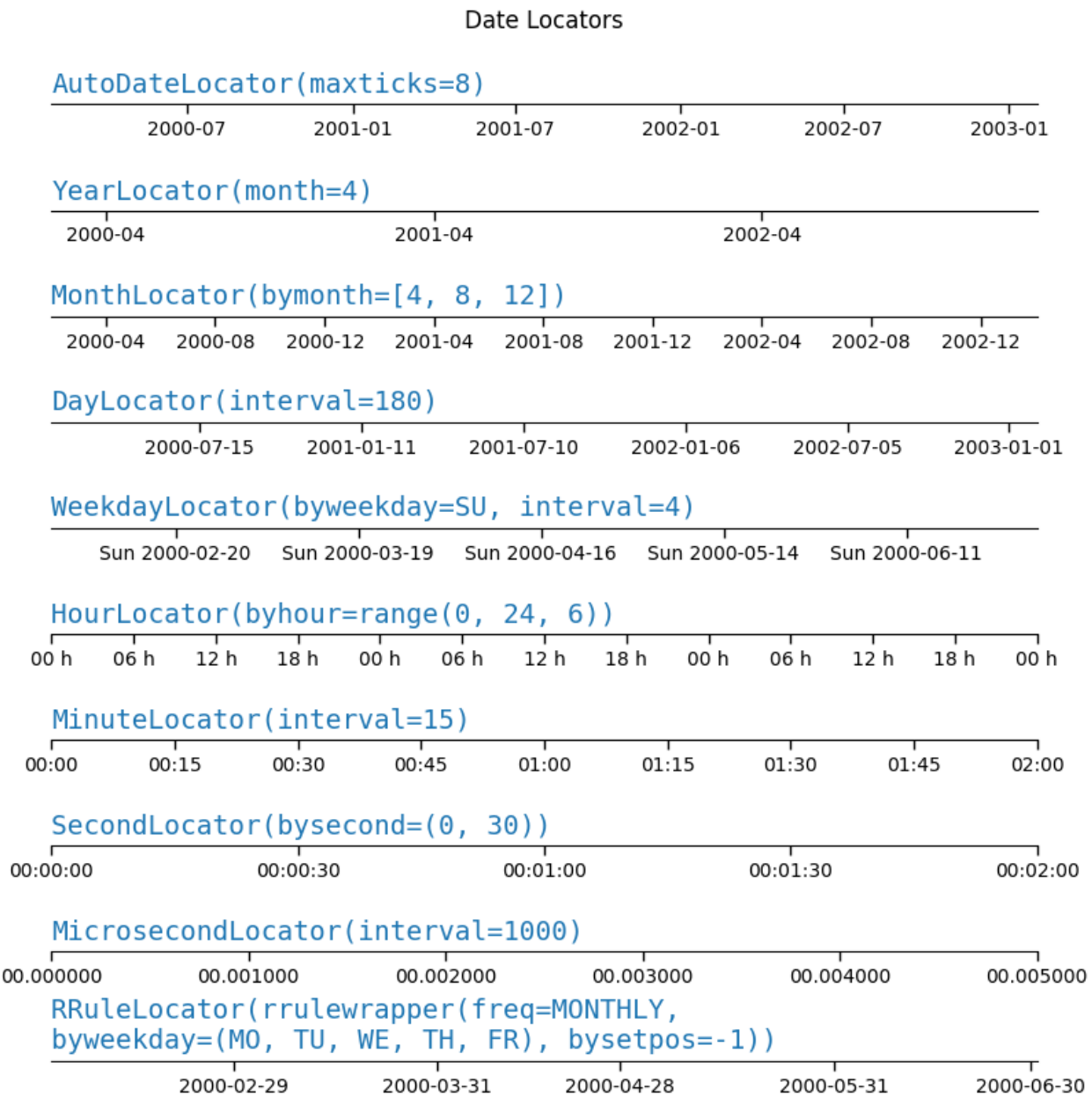
(continues on next page)

(continued from previous page)

```
    ax.yaxis.set_major_locator(ticker.NullLocator())
    ax.tick_params(which='major', width=1.00, length=5)
    ax.tick_params(which='minor', width=0.75, length=2.5)
    ax.set_xlim(np.datetime64('2000-02-01'), np.datetime64(xmax))
    if locator:
        ax.xaxis.set_major_locator(eval(locator))
        ax.xaxis.set_major_formatter(DateFormatter(fmt))
    else:
        ax.xaxis.set_major_formatter(eval(formatter))
    ax.text(0.0, 0.2, locator or formatter, transform=ax.transAxes,
            fontsize=14, fontname='Monospace', color='tab:blue')
```

### Date tick locators

```
locators = [
    # locator as str, xmax, fmt
    ('AutoDateLocator(maxticks=8)', '2003-02-01', '%Y-%m'),
    ('YearLocator(month=4)', '2003-02-01', '%Y-%m'),
    ('MonthLocator(bymonth=[4, 8, 12])', '2003-02-01', '%Y-%m'),
    ('DayLocator(interval=180)', '2003-02-01', '%Y-%m-%d'),
    ('WeekdayLocator(byweekday=SU, interval=4)', '2000-07-01', '%a %Y-%m-%d'),
    ('HourLocator(byhour=range(0, 24, 6))', '2000-02-04', '%H h'),
    ('MinuteLocator(interval=15)', '2000-02-01 02:00', '%H:%M'),
    ('SecondLocator(bysecond=(0, 30))', '2000-02-01 00:02', '%H:%M:%S'),
    ('MicrosecondLocator(interval=1000)', '2000-02-01 00:00:00.005', '%S.%f'),
    ('RRuleLocator(rrulewrapper(freq=MONTHLY, \nbyweekday=(MO, TU, WE, TH,␣
 ↪FR), '
     'bysetpos=-1))', '2000-07-01', '%Y-%m-%d'),
]

fig, axs = plt.subplots(len(locators), 1, figsize=(8, len(locators) * .8),
                        layout='constrained')
fig.suptitle('Date Locators')
for ax, (locator, xmax, fmt) in zip(axs, locators):
    plot_axis(ax, locator, xmax, fmt)
```



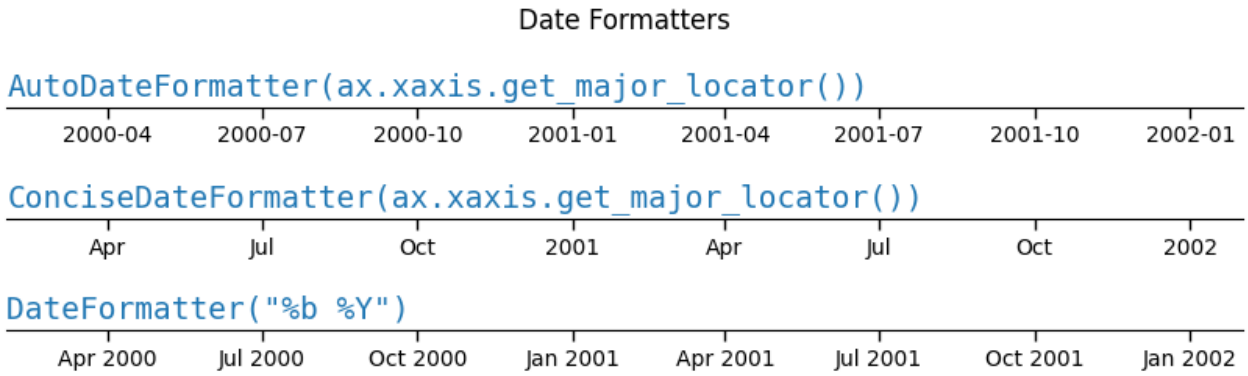

## Date formatters

```
formatters = [
    'AutoDateFormatter(ax.xaxis.get_major_locator())',
    'ConciseDateFormatter(ax.xaxis.get_major_locator())',
    'DateFormatter("%b %Y")',
]

fig, axs = plt.subplots(len(formatters), 1, figsize=(8, len(formatters) * .8),
                        layout='constrained')
fig.suptitle('Date Formatters')
```

(continues on next page)

(continued from previous page)

```
for ax, fmt in zip(axs, formatters):
    plot_axis(ax, formatter=fmt)
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.dates.AutoDateLocator*
- *matplotlib.dates.YearLocator*
- *matplotlib.dates.MonthLocator*
- *matplotlib.dates.DayLocator*
- *matplotlib.dates.WeekdayLocator*
- *matplotlib.dates.HourLocator*
- *matplotlib.dates.MinuteLocator*
- *matplotlib.dates.SecondLocator*
- *matplotlib.dates.MicrosecondLocator*
- *matplotlib.dates.RRuleLocator*
- *matplotlib.dates.rrulewrapper*
- *matplotlib.dates.DateFormatter*
- *matplotlib.dates.AutoDateFormatter*
- *matplotlib.dates.ConciseDateFormatter*

**Total running time of the script:** (0 minutes 1.718 seconds)

### Custom tick formatter for time series

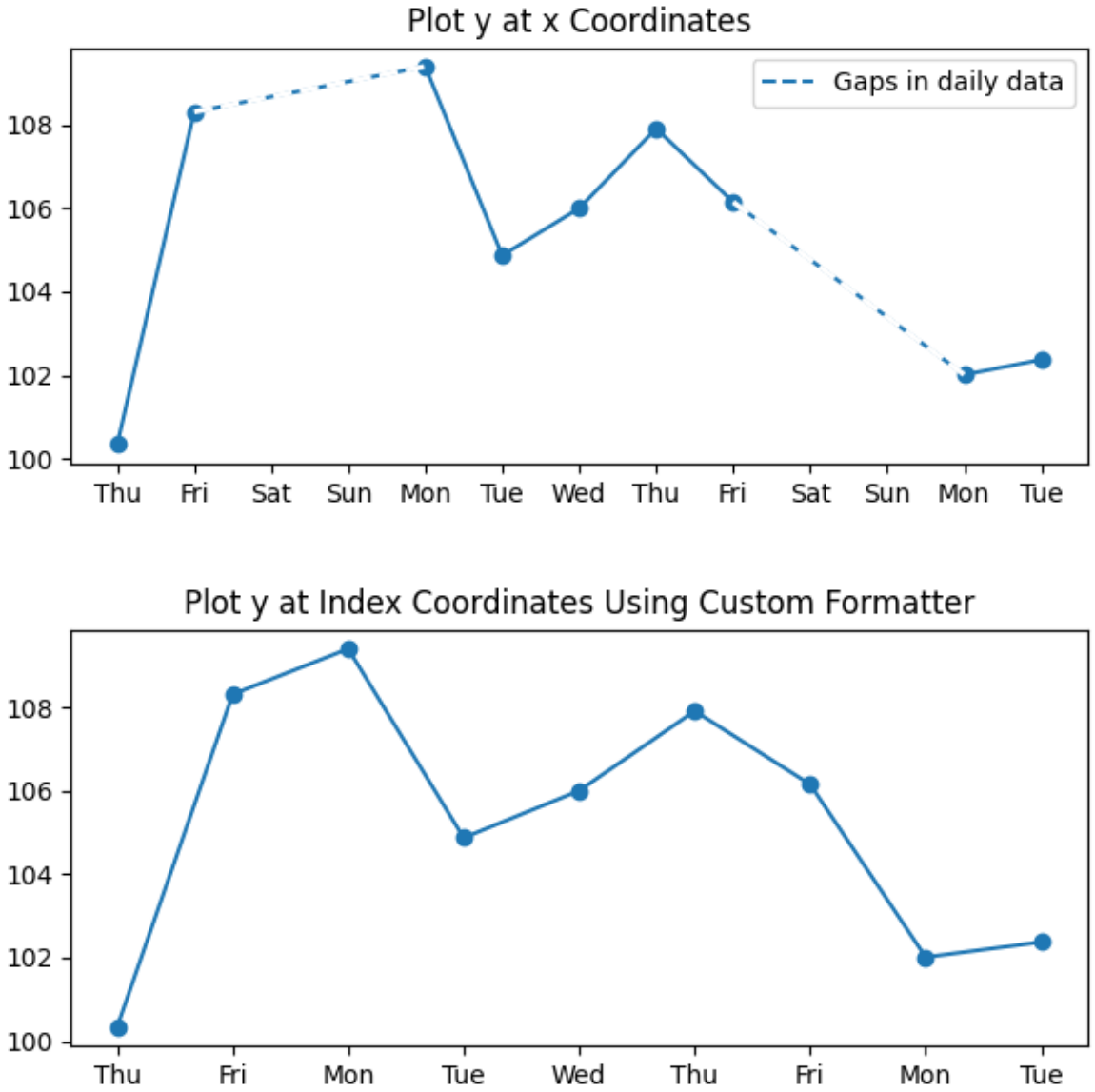
When plotting daily data, e.g., financial time series, one often wants to leave out days on which there is no data, for instance weekends, so that the data are plotted at regular intervals without extra spaces for the days with no data. The example shows how to use an 'index formatter' to achieve the desired plot.

```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib.cbook as cbook
from matplotlib.dates import DateFormatter, DayLocator
import matplotlib.lines as ml
from matplotlib.ticker import Formatter

# Load a structured numpy array from yahoo csv data with fields date, open,␣
 ↪high,
# low, close, volume, adj_close from the mpl-data/sample_data directory. The
# record array stores the date as an np.datetime64 with a day unit ('D') in
# the date column (``r['date']``).
r = cbook.get_sample_data('goog.npz')['price_data']
r = r[:9]  # get the first 9 days

fig, (ax1, ax2) = plt.subplots(nrows=2, figsize=(6, 6), layout='constrained')
fig.get_layout_engine().set(hspace=0.15)

# First we'll do it the default way, with gaps on weekends
ax1.plot(r["date"], r["adj_close"], 'o-')

# Highlight gaps in daily data
gaps = np.flatnonzero(np.diff(r["date"]) > np.timedelta64(1, 'D'))
for gap in r[['date', 'adj_close']][np.stack((gaps, gaps + 1)).T]:
    ax1.plot(gap['date'], gap['adj_close'], 'w--', lw=2)
ax1.legend(handles=[ml.Line2D([], [], ls='--', label='Gaps in daily data')])

ax1.set_title("Plot y at x Coordinates")
ax1.xaxis.set_major_locator(DayLocator())
ax1.xaxis.set_major_formatter(DateFormatter('%a'))


# Next we'll write a custom index formatter. Below we will plot
# the data against an index that goes from 0, 1,  ... len(data).  Instead of
# formatting the tick marks as integers, we format as times.
def format_date(x, _):
    try:
        # convert datetime64 to datetime, and use datetime's strftime:
        return r["date"][round(x)].item().strftime('%a')
    except IndexError:
        pass

# Create an index plot (x defaults to range(len(y)) if omitted)
ax2.plot(r["adj_close"], 'o-')
```

(continues on next page)

(continued from previous page)

```
ax2.set_title("Plot y at Index Coordinates Using Custom Formatter")
ax2.xaxis.set_major_formatter(format_date)  # internally creates FuncFormatter
```




Instead of passing a function into *Axis.set_major_formatter* you can use any other callable, e.g. an instance of a class that implements __call__:

```
class MyFormatter(Formatter):
    def __init__(self, dates, fmt='%a'):
        self.dates = dates
        self.fmt = fmt

    def __call__(self, x, pos=0):
        """Return the label for time x at position pos."""
        try:
            return self.dates[round(x)].item().strftime(self.fmt)
```

(continues on next page)

(continued from previous page)

```
        except IndexError:
            pass


ax2.xaxis.set_major_formatter(MyFormatter(r["date"], '%a'))

plt.show()
```

### Date Precision and Epochs

Matplotlib can handle `datetime` objects and `numpy.datetime64` objects using a unit converter that recognizes these dates and converts them to floating point numbers.

Before Matplotlib 3.3, the default for this conversion returns a float that was days since "0000-12-31T00:00:00". As of Matplotlib 3.3, the default is days from "1970-01-01T00:00:00". This allows more resolution for modern dates. "2020-01-01" with the old epoch converted to 730120, and a 64-bit floating point number has a resolution of 2^{-52}, or approximately 14 microseconds, so microsecond precision was lost. With the new default epoch "2020-01-01" is 10957.0, so the achievable resolution is 0.21 microseconds.

```
import datetime

import matplotlib.pyplot as plt
import numpy as np

import matplotlib.dates as mdates


def _reset_epoch_for_tutorial():
    """
    Users (and downstream libraries) should not use the private method of
    resetting the epoch.
    """
    mdates._reset_epoch_test_example()
```

### Datetime

Python `datetime` objects have microsecond resolution, so with the old default matplotlib dates could not round-trip full-resolution datetime objects.

```
old_epoch = '0000-12-31T00:00:00'
new_epoch = '1970-01-01T00:00:00'

_reset_epoch_for_tutorial()  # Don't do this.  Just for this tutorial.
mdates.set_epoch(old_epoch)  # old epoch (pre MPL 3.3)

date1 = datetime.datetime(2000, 1, 1, 0, 10, 0, 12,
                          tzinfo=datetime.timezone.utc)
```

(continues on next page)

(continued from previous page)

```
mdate1 = mdates.date2num(date1)
print('Before Roundtrip: ', date1, 'Matplotlib date:', mdate1)
date2 = mdates.num2date(mdate1)
print('After Roundtrip:  ', date2)
```

```
Before Roundtrip:  2000-01-01 00:10:00.000012+00:00 Matplotlib date: 730120.
↪0069444446
After Roundtrip:   2000-01-01 00:10:00.000020+00:00
```

Note this is only a round-off error, and there is no problem for dates closer to the old epoch:

```
date1 = datetime.datetime(10, 1, 1, 0, 10, 0, 12,
                          tzinfo=datetime.timezone.utc)
mdate1 = mdates.date2num(date1)
print('Before Roundtrip: ', date1, 'Matplotlib date:', mdate1)
date2 = mdates.num2date(mdate1)
print('After Roundtrip:  ', date2)
```

```
Before Roundtrip:  0010-01-01 00:10:00.000012+00:00 Matplotlib date: 3288.
↪006944444583
After Roundtrip:   0010-01-01 00:10:00.000012+00:00
```

If a user wants to use modern dates at microsecond precision, they can change the epoch using *set_epoch*. However, the epoch has to be set before any date operations to prevent confusion between different epochs. Trying to change the epoch later will raise a RuntimeError.

```
try:
    mdates.set_epoch(new_epoch)  # this is the new MPL 3.3 default.
except RuntimeError as e:
    print('RuntimeError:', str(e))
```

```
RuntimeError: set_epoch must be called before dates plotted.
```

For this tutorial, we reset the sentinel using a private method, but users should just set the epoch once, if at all.

```
_reset_epoch_for_tutorial()  # Just being done for this tutorial.
mdates.set_epoch(new_epoch)

date1 = datetime.datetime(2020, 1, 1, 0, 10, 0, 12,
                          tzinfo=datetime.timezone.utc)
mdate1 = mdates.date2num(date1)
print('Before Roundtrip: ', date1, 'Matplotlib date:', mdate1)
date2 = mdates.num2date(mdate1)
print('After Roundtrip:  ', date2)
```

```
Before Roundtrip:  2020-01-01 00:10:00.000012+00:00 Matplotlib date: 18262.
↪006944444583
After Roundtrip:   2020-01-01 00:10:00.000012+00:00
```

### datetime64

`numpy.datetime64` objects have microsecond precision for a much larger timespace than `datetime` objects. However, currently Matplotlib time is only converted back to datetime objects, which have microsecond resolution, and years that only span 0000 to 9999.

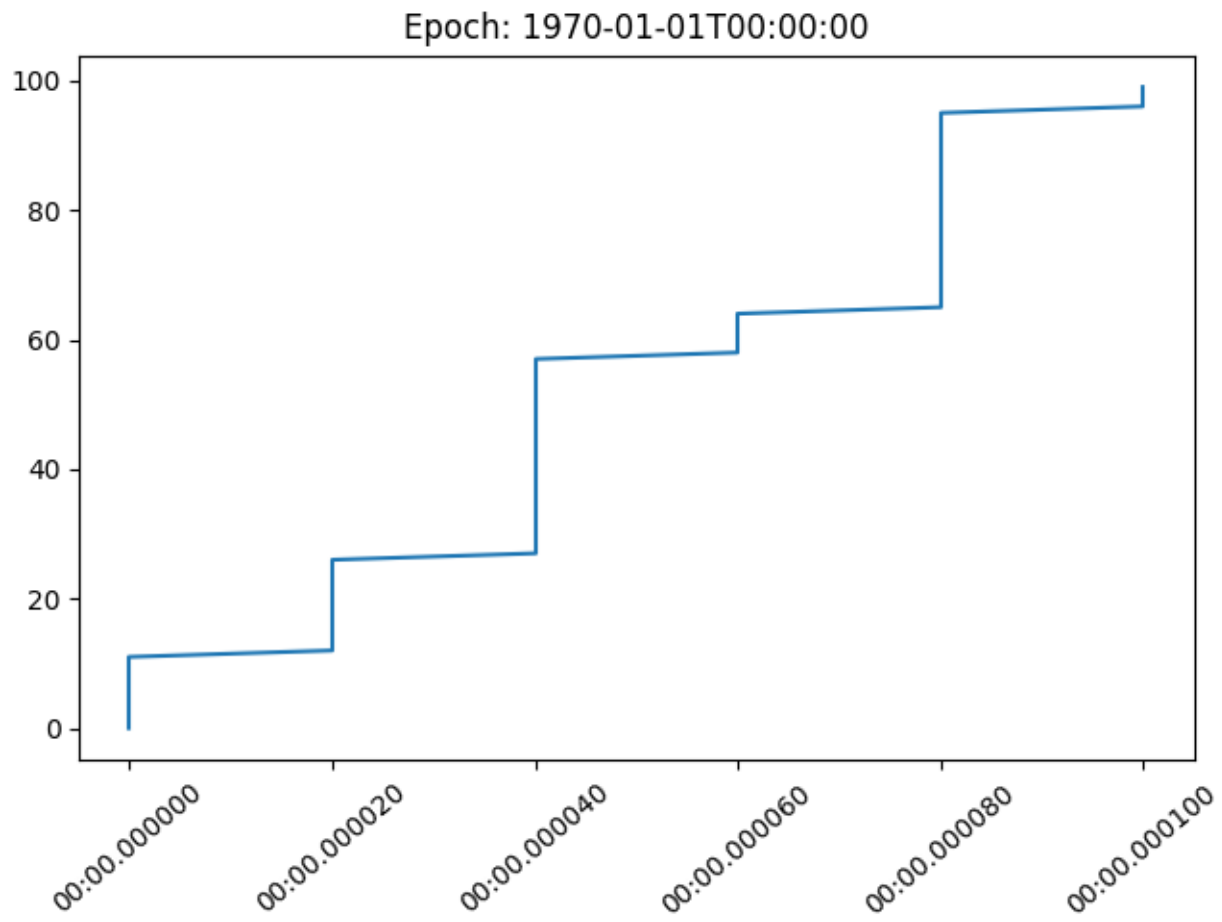
```
_reset_epoch_for_tutorial()  # Don't do this.  Just for this tutorial.
mdates.set_epoch(new_epoch)

date1 = np.datetime64('2000-01-01T00:10:00.000012')
mdate1 = mdates.date2num(date1)
print('Before Roundtrip: ', date1, 'Matplotlib date:', mdate1)
date2 = mdates.num2date(mdate1)
print('After Roundtrip:  ', date2)
```

```
Before Roundtrip:  2000-01-01T00:10:00.000012 Matplotlib date: 10957.
↪006944444583
After Roundtrip:   2000-01-01 00:10:00.000012+00:00
```

### Plotting

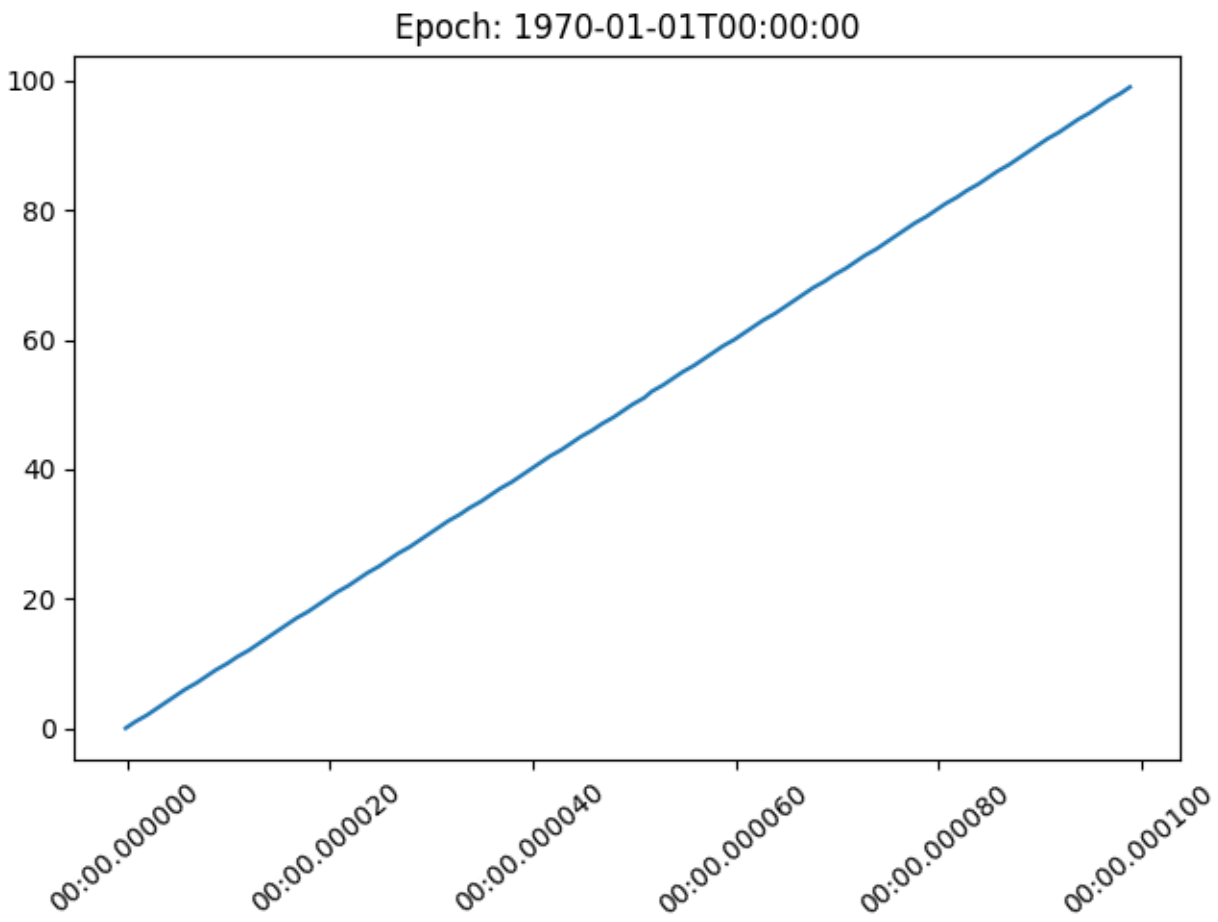
This all of course has an effect on plotting. With the old default epoch the times were rounded during the internal `date2num` conversion, leading to jumps in the data:

```
_reset_epoch_for_tutorial()  # Don't do this.  Just for this tutorial.
mdates.set_epoch(old_epoch)

x = np.arange('2000-01-01T00:00:00.0', '2000-01-01T00:00:00.000100',
              dtype='datetime64[us]')
# simulate the plot being made using the old epoch
xold = np.array([mdates.num2date(mdates.date2num(d)) for d in x])
y = np.arange(0, len(x))

# resetting the Epoch so plots are comparable
_reset_epoch_for_tutorial()  # Don't do this.  Just for this tutorial.
mdates.set_epoch(new_epoch)

fig, ax = plt.subplots(layout='constrained')
ax.plot(xold, y)
ax.set_title('Epoch: ' + mdates.get_epoch())
ax.xaxis.set_tick_params(rotation=40)
plt.show()
```




For dates plotted using the more recent epoch, the plot is smooth:

```
fig, ax = plt.subplots(layout='constrained')
ax.plot(x, y)
ax.set_title('Epoch: ' + mdates.get_epoch())
ax.xaxis.set_tick_params(rotation=40)
plt.show()

_reset_epoch_for_tutorial()  # Don't do this.  Just for this tutorial.
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.dates.num2date*
- *matplotlib.dates.date2num*
- *matplotlib.dates.set_epoch*

## Dollar ticks

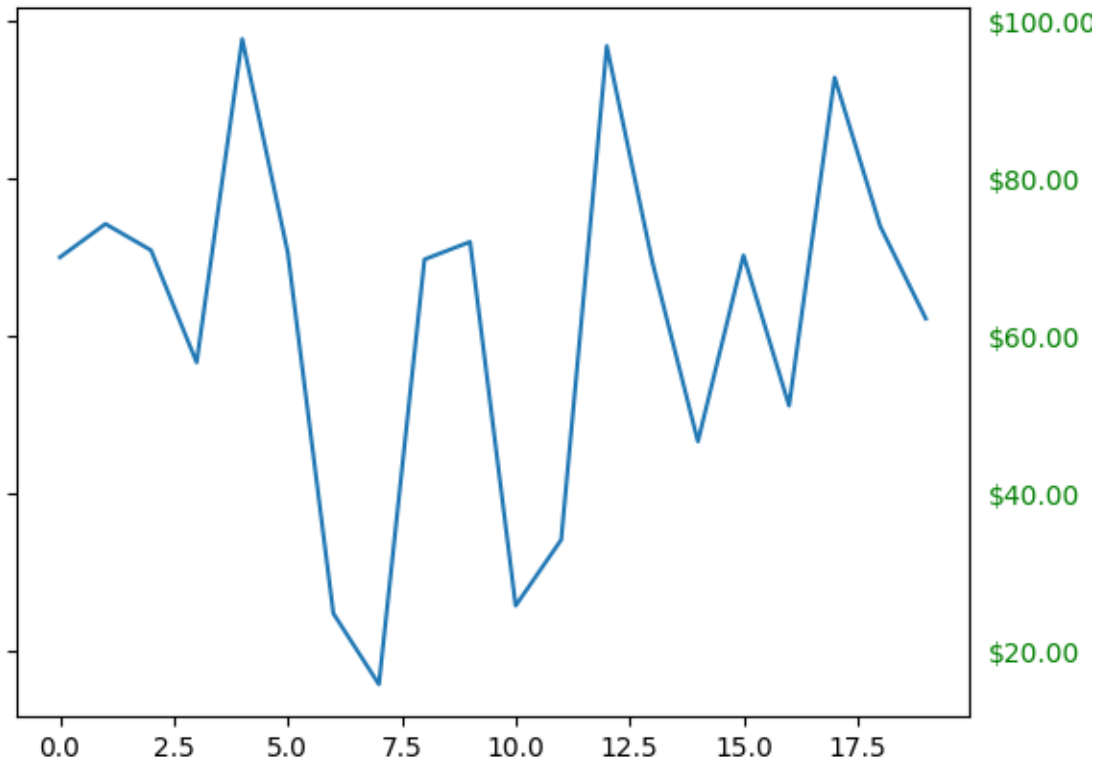
Use a format string to prepend dollar signs on y-axis labels.

```
import matplotlib.pyplot as plt
import numpy as np

# Fixing random state for reproducibility
np.random.seed(19680801)

fig, ax = plt.subplots()
```

(continues on next page)

(continued from previous page)

```
ax.plot(100*np.random.rand(20))

# Use automatic StrMethodFormatter
ax.yaxis.set_major_formatter('${x:1.2f}')

ax.yaxis.set_tick_params(which='major', labelcolor='green',
                         labelleft=False, labelright=True)

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.pyplot.subplots*
- *matplotlib.axis.Axis.set_major_formatter*
- *matplotlib.axis.Axis.set_tick_params*
- *matplotlib.axis.Tick*
- *matplotlib.ticker.StrMethodFormatter*

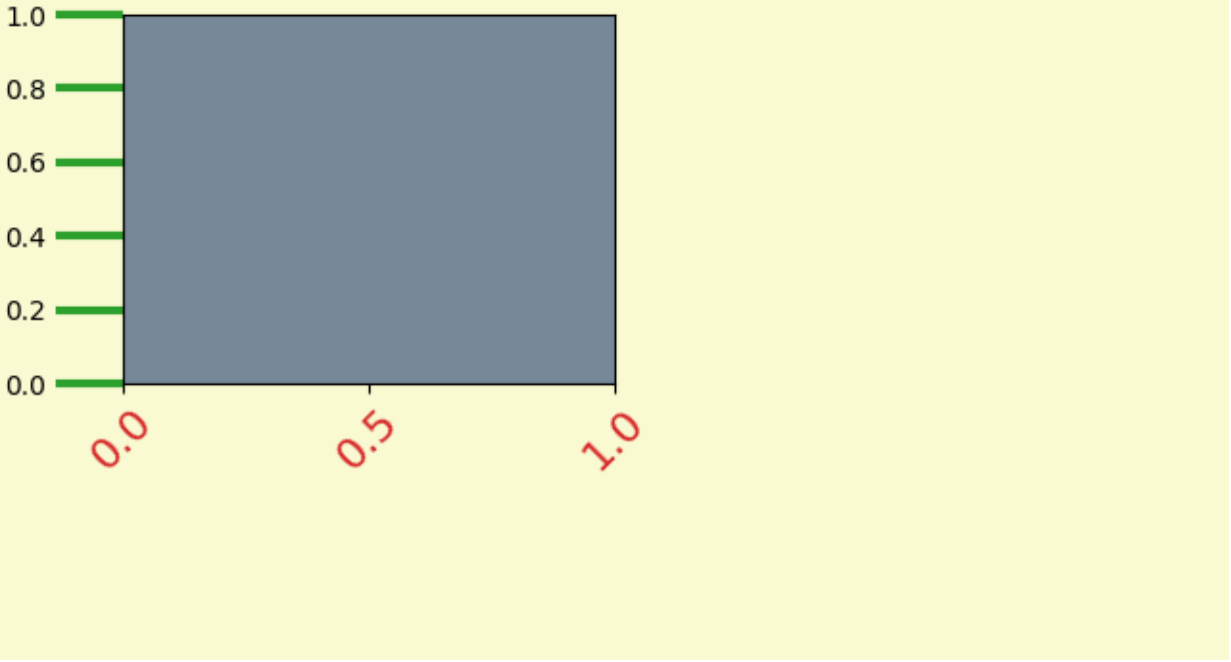
### Fig Axes Customize Simple

Customize the background, labels and ticks of a simple plot.

```
import matplotlib.pyplot as plt
```

*pyplot.figure* creates a *matplotlib.figure.Figure* instance.

```
fig = plt.figure()
rect = fig.patch  # a rectangle instance
rect.set_facecolor('lightgoldenrodyellow')

ax1 = fig.add_axes([0.1, 0.3, 0.4, 0.4])
rect = ax1.patch
rect.set_facecolor('lightslategray')

ax1.tick_params(axis='x', labelcolor='tab:red', labelrotation=45,␣
 ↪labelsize=16)
ax1.tick_params(axis='y', color='tab:green', size=25, width=3)

plt.show()
```



**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axis.Axis.get_ticklabels*
- *matplotlib.axis.Axis.get_ticklines*
- *matplotlib.text.Text.set_rotation*
- *matplotlib.text.Text.set_fontsize*
- *matplotlib.text.Text.set_color*
- *matplotlib.lines.Line2D*
- *matplotlib.lines.Line2D.set_markeredgecolor*
- *matplotlib.lines.Line2D.set_markersize*
- *matplotlib.lines.Line2D.set_markeredgewidth*
- *matplotlib.patches.Patch.set_facecolor*

## Major and minor ticks

Demonstrate how to use major and minor tickers.

The two relevant classes are *Locator*s and *Formatter*s. Locators determine where the ticks are, and formatters control the formatting of tick labels.

Minor ticks are off by default (using *NullLocator* and *NullFormatter*). Minor ticks can be turned on without labels by setting the minor locator. Minor tick labels can be turned on by setting the minor formatter.

*MultipleLocator* places ticks on multiples of some base. *StrMethodFormatter* uses a format string (e.g., `'{x:d}'` or `'{x:1.2f}'` or `'{x:1.1f} cm'`) to format the tick labels (the variable in the format string must be `'x'`). For a *StrMethodFormatter*, the string can be passed directly to *Axis.set_major_formatter* or *Axis.set_minor_formatter*. An appropriate *StrMethodFormatter* will be created and used automatically.

*pyplot.grid* changes the grid settings of the major ticks of the x- and y-axis together. If you want to control the grid of the minor ticks for a given axis, use for example

```
ax.xaxis.grid(True, which='minor')
```

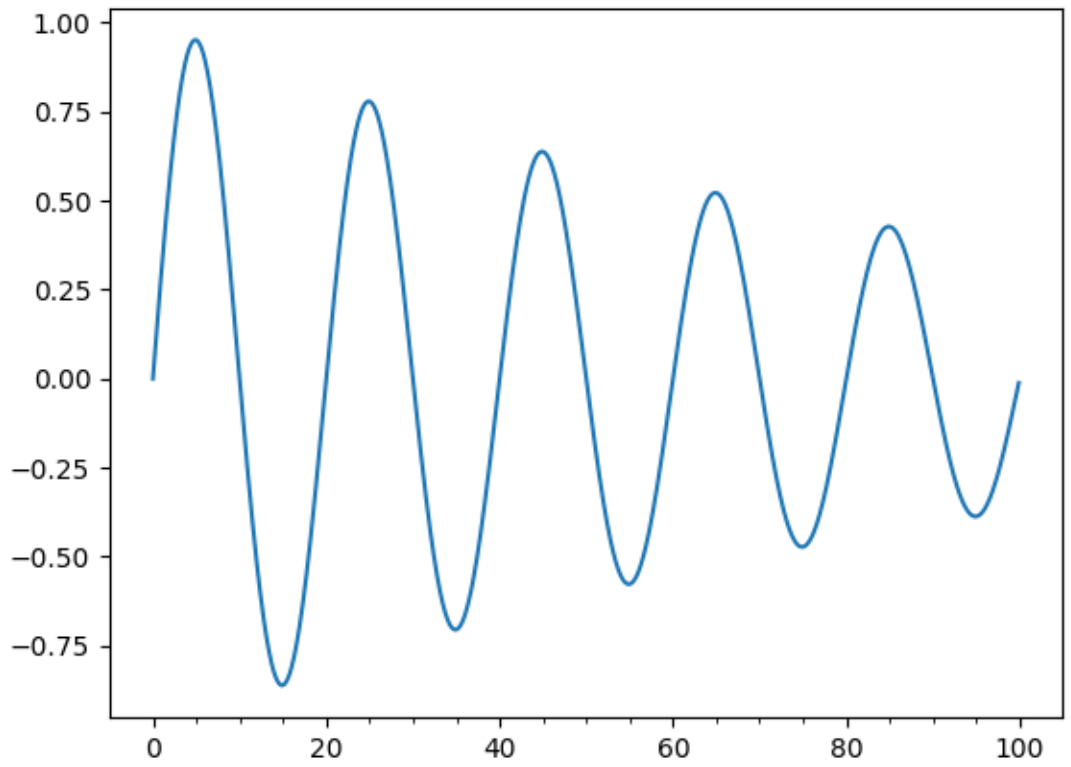
Note that a given locator or formatter instance can only be used on a single axis (because the locator stores references to the axis data and view limits).

```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.ticker import AutoMinorLocator, MultipleLocator

```

(continues on next page)

(continued from previous page)

```
t = np.arange(0.0, 100.0, 0.1)
s = np.sin(0.1 * np.pi * t) * np.exp(-t * 0.01)

fig, ax = plt.subplots()
ax.plot(t, s)

# Make a plot with major ticks that are multiples of 20 and minor ticks that
# are multiples of 5.  Label major ticks with '.0f' formatting but don't label
# minor ticks.  The string is used directly, the `StrMethodFormatter` is
# created automatically.
ax.xaxis.set_major_locator(MultipleLocator(20))
ax.xaxis.set_major_formatter('{x:.0f}')

# For the minor ticks, use no labels; default NullFormatter.
ax.xaxis.set_minor_locator(MultipleLocator(5))

plt.show()
```

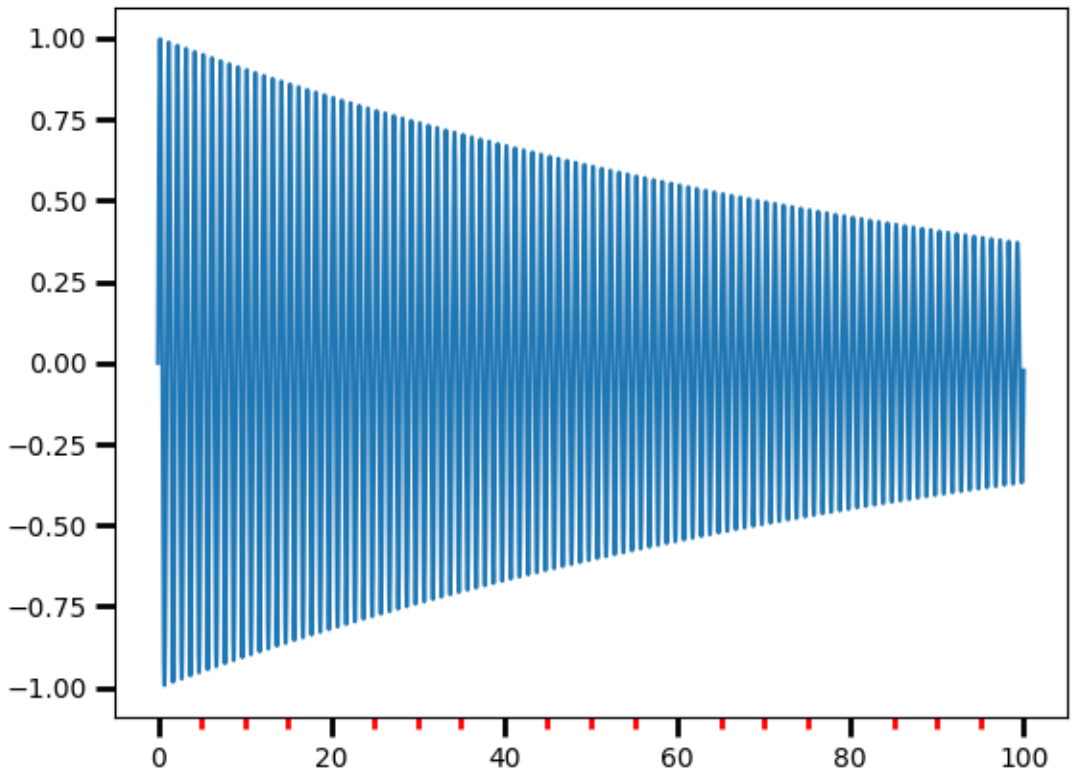



Automatic tick selection for major and minor ticks.

Use interactive pan and zoom to see how the tick intervals change. There will be either 4 or 5 minor tick intervals per major interval, depending on the major interval.

One can supply an argument to *AutoMinorLocator* to specify a fixed number of minor intervals per major interval, e.g. `AutoMinorLocator(2)` would lead to a single minor tick between major ticks.

```
t = np.arange(0.0, 100.0, 0.01)
s = np.sin(2 * np.pi * t) * np.exp(-t * 0.01)

fig, ax = plt.subplots()
ax.plot(t, s)

ax.xaxis.set_minor_locator(AutoMinorLocator())

ax.tick_params(which='both', width=2)
ax.tick_params(which='major', length=7)
ax.tick_params(which='minor', length=4, color='r')

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.pyplot.subplots*
- *matplotlib.axis.Axis.set_major_formatter*

- *matplotlib.axis.Axis.set_major_locator*
- *matplotlib.axis.Axis.set_minor_locator*
- *matplotlib.ticker.AutoMinorLocator*
- *matplotlib.ticker.MultipleLocator*
- *matplotlib.ticker.StrMethodFormatter*

**Total running time of the script:** (0 minutes 1.162 seconds)

### Multilevel (nested) ticks

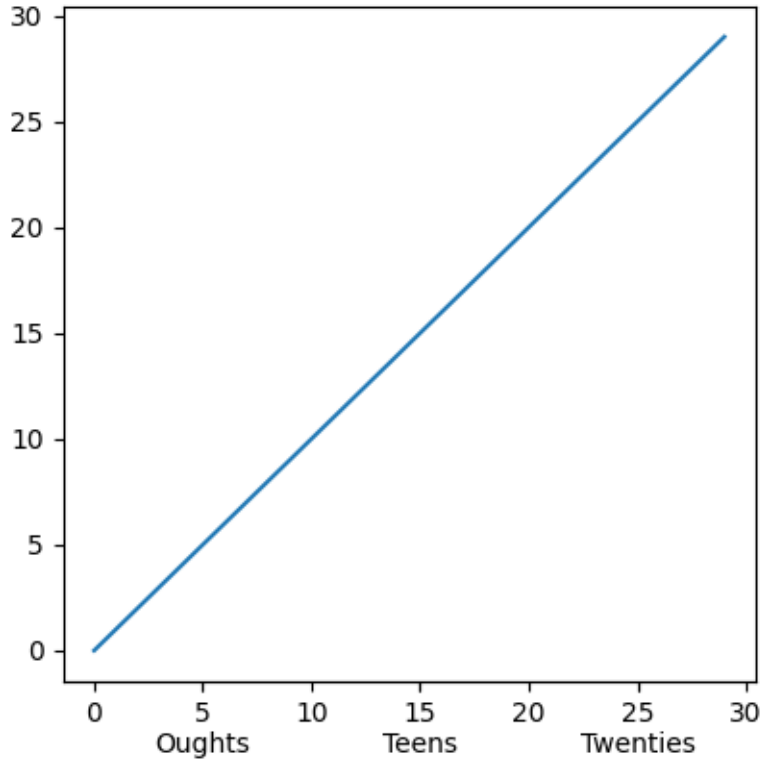
Sometimes we want another level of tick labels on an axis, perhaps to indicate a grouping of the ticks.

Matplotlib does not provide an automated way to do this, but it is relatively straightforward to annotate below the main axis.

These examples use *Axes.secondary_xaxis*, which is one approach. It has the advantage that we can use Matplotlib Locators and Formatters on the axis that does the grouping if we want.

This first example creates a secondary xaxis and manually adds the ticks and labels using *Axes.set_xticks*. Note that the tick labels have a newline (e.g. `"\nOughts"`) at the beginning of them to put the second-level tick labels below the main tick labels.

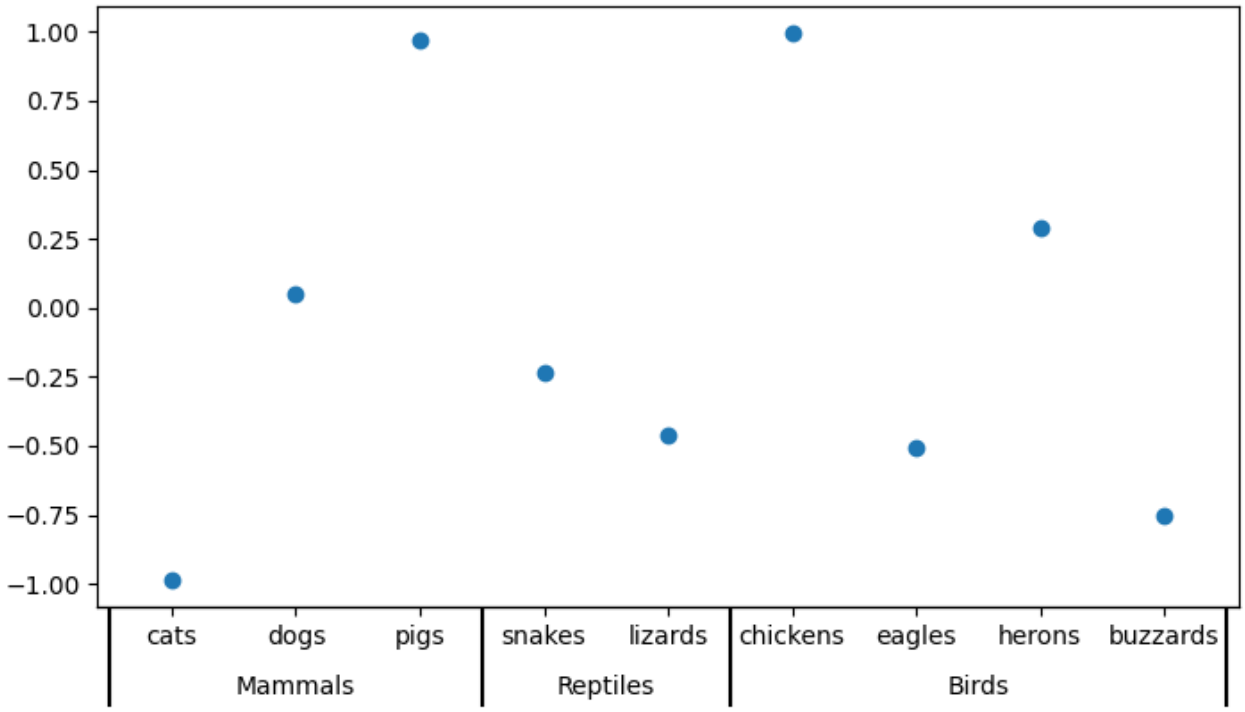
```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib.dates as mdates

rng = np.random.default_rng(19680801)

fig, ax = plt.subplots(layout='constrained', figsize=(4, 4))

ax.plot(np.arange(30))

sec = ax.secondary_xaxis(location=0)
sec.set_xticks([5, 15, 25], labels=['\nOughts', '\nTeens', '\nTwenties'])
```




This second example adds a second level of annotation to a categorical axis. Here we need to note that each animal (category) is assigned an integer, so `cats` is at x=0, `dogs` at x=1 etc. Then we place the ticks on the second level on an x that is at the middle of the animal class we are trying to delineate.

This example also adds tick marks between the classes by adding a second secondary xaxis, and placing long, wide ticks at the boundaries between the animal classes.

```
fig, ax = plt.subplots(layout='constrained', figsize=(7, 4))

ax.plot(['cats', 'dogs', 'pigs', 'snakes', 'lizards', 'chickens',
         'eagles', 'herons', 'buzzards'],
        rng.normal(size=9), 'o')

# label the classes:
sec = ax.secondary_xaxis(location=0)
sec.set_xticks([1, 3.5, 6.5], labels=['\n\nMammals', '\n\nReptiles', '\n\
 ↪nBirds'])
sec.tick_params('x', length=0)

# lines between the classes:
sec2 = ax.secondary_xaxis(location=0)
sec2.set_xticks([-0.5, 2.5, 4.5, 8.5], labels=[])
sec2.tick_params('x', length=40, width=1.5)
ax.set_xlim(-0.6, 8.6)
```




Dates are another common place where we may want to have a second level of tick labels. In this last example, we take advantage of the ability to add an automatic locator and formatter to the secondary xaxis, which means we do not need to set the ticks manually.

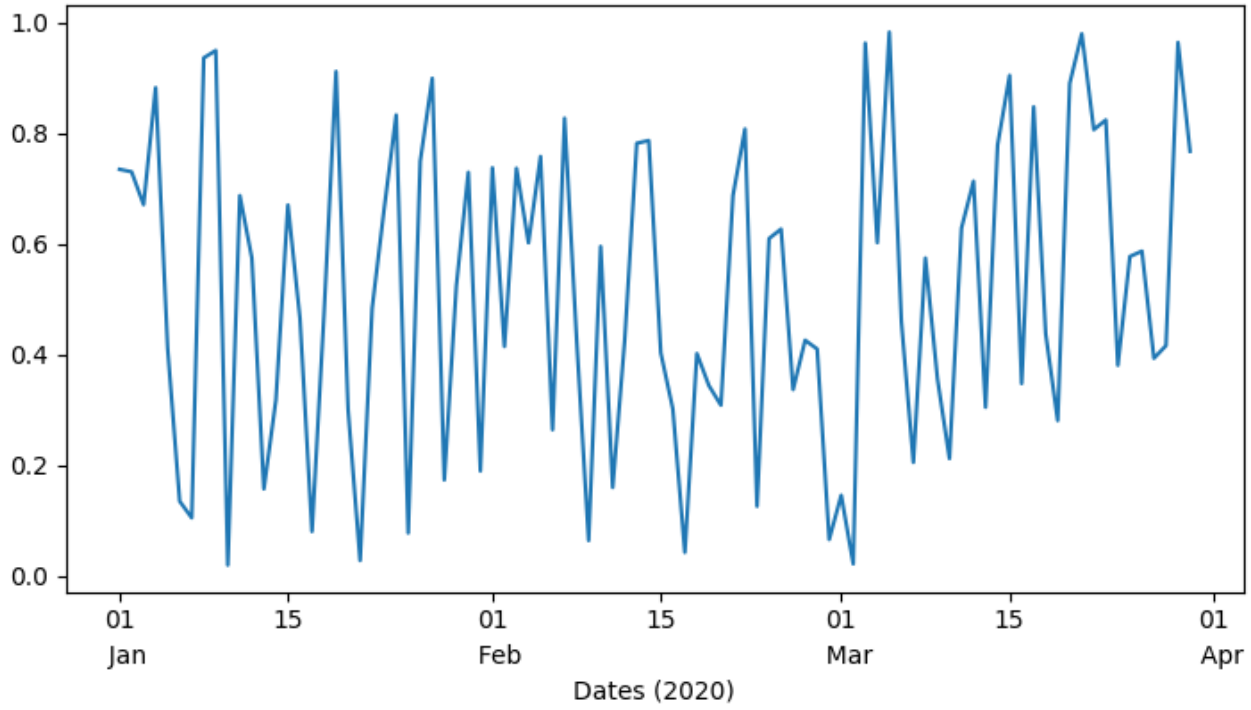
This example also differs from the above, in that we placed it at a location below the main axes `location=-0.075` and then we hide the spine by setting the line width to zero. That means that our formatter no longer needs the carriage returns of the previous two examples.

```
fig, ax = plt.subplots(layout='constrained', figsize=(7, 4))

time = np.arange(np.datetime64('2020-01-01'), np.datetime64('2020-03-31'),
                 np.timedelta64(1, 'D'))

ax.plot(time, rng.random(size=len(time)))

# just format the days:
ax.xaxis.set_major_formatter(mdates.DateFormatter('%d'))

# label the months:
sec = ax.secondary_xaxis(location=-0.075)
sec.xaxis.set_major_locator(mdates.MonthLocator(bymonthday=1))

# note the extra spaces in the label to align the month label inside the␣
 ↪month.
# Note that this could have been done by changing ``bymonthday`` above as␣
 ↪well:
sec.xaxis.set_major_formatter(mdates.DateFormatter('  %b'))
sec.tick_params('x', length=0)
sec.spines['bottom'].set_linewidth(0)
```

(continues on next page)

(continued from previous page)

```
# label the xaxis, but note for this to look good, it needs to be on the
# secondary xaxis.
sec.set_xlabel('Dates (2020)')

plt.show()
```

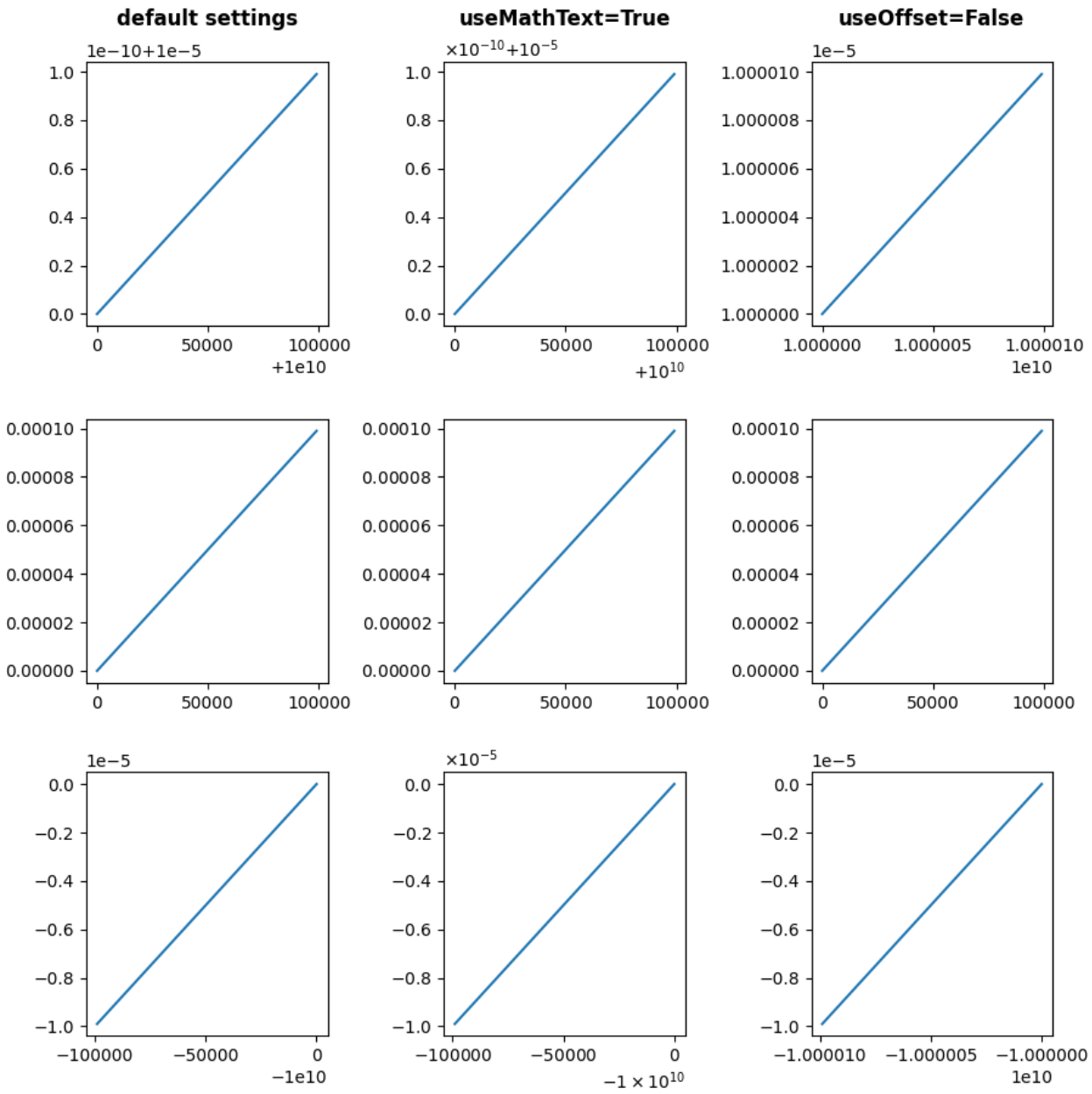



**Total running time of the script:** (0 minutes 1.220 seconds)

### The default tick formatter

By default, tick labels are formatted using a *ScalarFormatter*, which can be configured via *ticklabel_format*. This example illustrates some possible configurations:

- Default.
- `useMathText=True`: Fancy formatting of mathematical expressions.
- `useOffset=False`: Do not use offset notation; see *ScalarFormatter.set_useOffset*.




```
import matplotlib.pyplot as plt
import numpy as np

x = np.arange(0, 1, .01)
fig, axs = plt.subplots(
    3, 3, figsize=(9, 9), layout="constrained", gridspec_kw={"hspace": 0.1})

for col in axs.T:
    col[0].plot(x * 1e5 + 1e10, x * 1e-10 + 1e-5)
    col[1].plot(x * 1e5, x * 1e-4)
    col[2].plot(-x * 1e5 - 1e10, -x * 1e-5 - 1e-10)

for ax in axs[:, 1]:
```

(continues on next page)

(continued from previous page)

```
    ax.ticklabel_format(useMathText=True)
for ax in axs[:, 2]:
    ax.ticklabel_format(useOffset=False)

plt.rcParams.update({"axes.titleweight": "bold", "axes.titley": 1.1})
axs[0, 0].set_title("default settings")
axs[0, 1].set_title("useMathText=True")
axs[0, 2].set_title("useOffset=False")

plt.show()
```

**Total running time of the script:** (0 minutes 1.357 seconds)

### Tick formatters

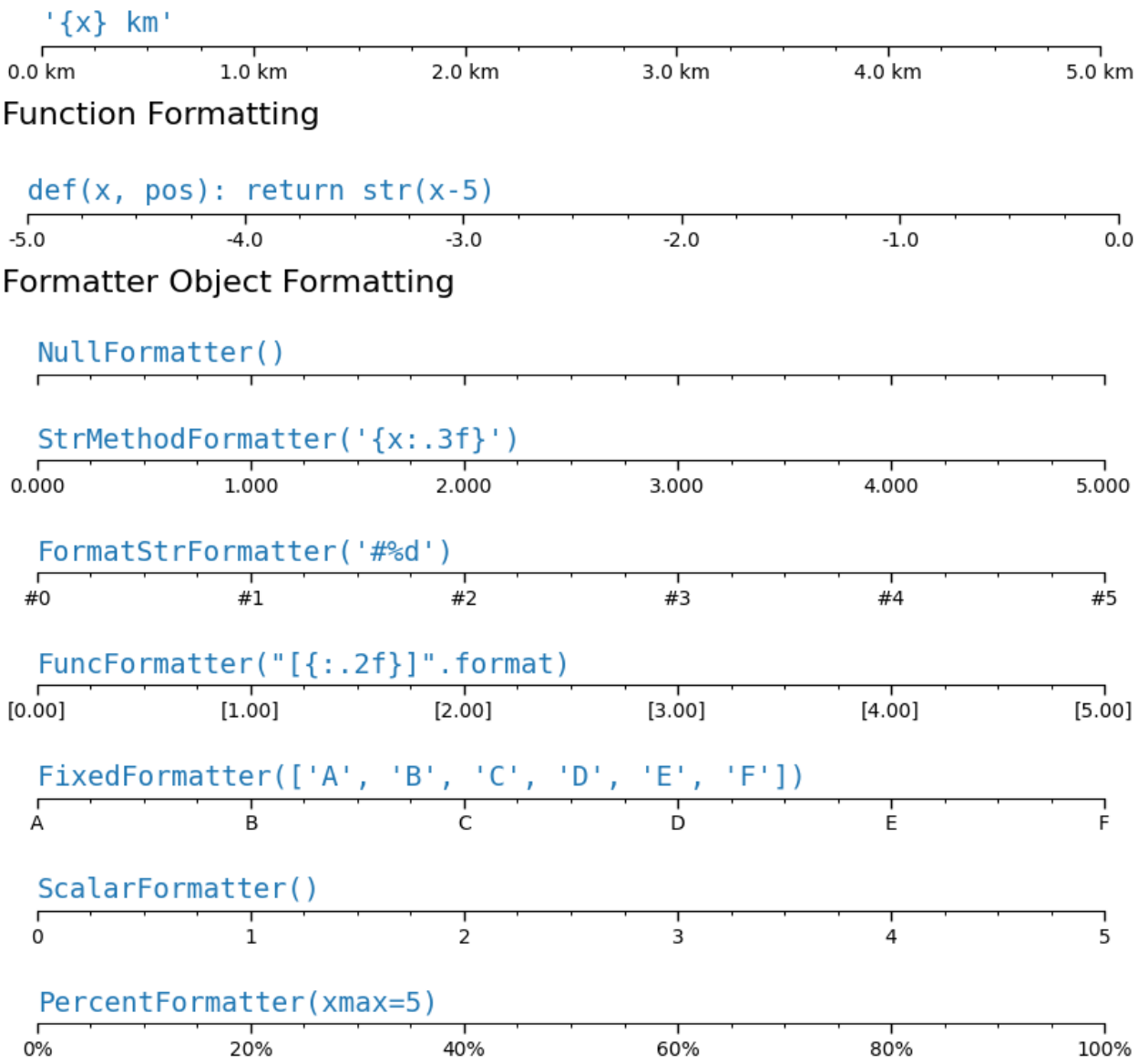
Tick formatters define how the numeric value associated with a tick on an axis is formatted as a string.

This example illustrates the usage and effect of the most common formatters.

The tick format is configured via the function *set_major_formatter* or *set_minor_formatter*. It accepts:

- a format string, which implicitly creates a *StrMethodFormatter*.
- a function, implicitly creates a *FuncFormatter*.
- an instance of a *Formatter* subclass. The most common are
  - *NullFormatter*: No labels on the ticks.
  - *StrMethodFormatter*: Use string `str.format` method.
  - *FormatStrFormatter*: Use %-style formatting.
  - *FuncFormatter*: Define labels through a function.
  - *FixedFormatter*: Set the label strings explicitly.
  - *ScalarFormatter*: Default formatter for scalars: auto-pick the format string.
  - *PercentFormatter*: Format labels as a percentage.

  See *Tick formatting* for a complete list.




```
import matplotlib.pyplot as plt

from matplotlib import ticker


def setup(ax, title):
    """Set up common parameters for the Axes in the example."""
    # only show the bottom spine
    ax.yaxis.set_major_locator(ticker.NullLocator())
    ax.spines[['left', 'right', 'top']].set_visible(False)

    # define tick positions
    ax.xaxis.set_major_locator(ticker.MultipleLocator(1.00))
```

(continues on next page)

(continued from previous page)

```
    ax.xaxis.set_minor_locator(ticker.MultipleLocator(0.25))

    ax.xaxis.set_ticks_position('bottom')
    ax.tick_params(which='major', width=1.00, length=5)
    ax.tick_params(which='minor', width=0.75, length=2.5, labelsize=10)
    ax.set_xlim(0, 5)
    ax.set_ylim(0, 1)
    ax.text(0.0, 0.2, title, transform=ax.transAxes,
            fontsize=14, fontname='Monospace', color='tab:blue')


fig = plt.figure(figsize=(8, 8), layout='constrained')
fig0, fig1, fig2 = fig.subfigures(3, height_ratios=[1.5, 1.5, 7.5])

fig0.suptitle('String Formatting', fontsize=16, x=0, ha='left')
ax0 = fig0.subplots()

setup(ax0, title="'{x} km'")
ax0.xaxis.set_major_formatter('{x} km')


fig1.suptitle('Function Formatting', fontsize=16, x=0, ha='left')
ax1 = fig1.subplots()

setup(ax1, title="def(x, pos): return str(x-5)")
ax1.xaxis.set_major_formatter(lambda x, pos: str(x-5))


fig2.suptitle('Formatter Object Formatting', fontsize=16, x=0, ha='left')
axs2 = fig2.subplots(7, 1)

setup(axs2[0], title="NullFormatter()")
axs2[0].xaxis.set_major_formatter(ticker.NullFormatter())

setup(axs2[1], title="StrMethodFormatter('{x:.3f}')")
axs2[1].xaxis.set_major_formatter(ticker.StrMethodFormatter("{x:.3f}"))

setup(axs2[2], title="FormatStrFormatter('#%d')")
axs2[2].xaxis.set_major_formatter(ticker.FormatStrFormatter("#%d"))


def fmt_two_digits(x, pos):
    return f'[{x:.2f}]'


setup(axs2[3], title='FuncFormatter("[{:.2f}]".format)')
axs2[3].xaxis.set_major_formatter(ticker.FuncFormatter(fmt_two_digits))

setup(axs2[4], title="FixedFormatter(['A', 'B', 'C', 'D', 'E', 'F'])")
# FixedFormatter should only be used together with FixedLocator.
# Otherwise, one cannot be sure where the labels will end up.
positions = [0, 1, 2, 3, 4, 5]
```

(continues on next page)

(continued from previous page)

```
labels = ['A', 'B', 'C', 'D', 'E', 'F']
axs2[4].xaxis.set_major_locator(ticker.FixedLocator(positions))
axs2[4].xaxis.set_major_formatter(ticker.FixedFormatter(labels))

setup(axs2[5], title="ScalarFormatter()")
axs2[5].xaxis.set_major_formatter(ticker.ScalarFormatter(useMathText=True))

setup(axs2[6], title="PercentFormatter(xmax=5)")
axs2[6].xaxis.set_major_formatter(ticker.PercentFormatter(xmax=5))

plt.show()
```

**Total running time of the script:** (0 minutes 1.152 seconds)

### Tick locators

Tick locators define the position of the ticks.

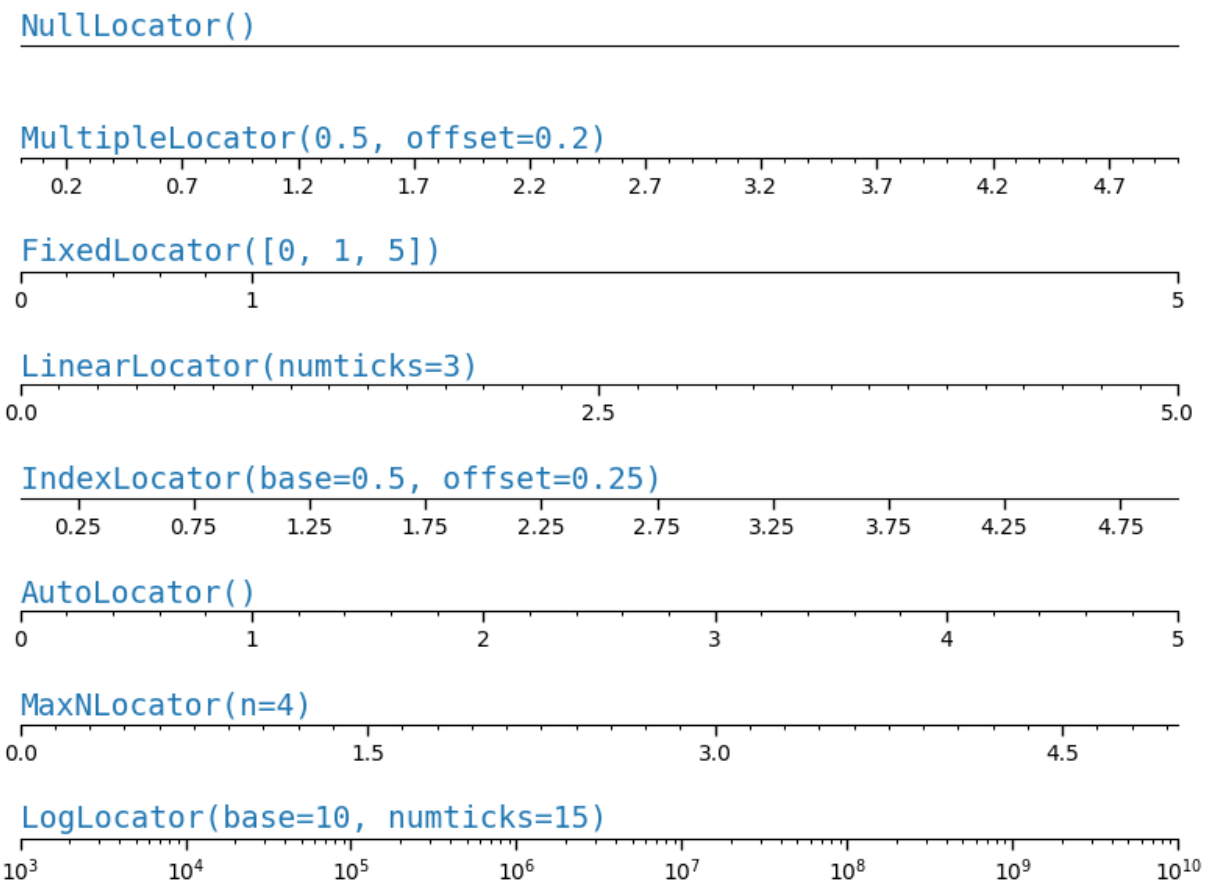
This example illustrates the usage and effect of the most common locators.

```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib.ticker as ticker


def setup(ax, title):
    """Set up common parameters for the Axes in the example."""
    # only show the bottom spine
    ax.yaxis.set_major_locator(ticker.NullLocator())
    ax.spines[['left', 'right', 'top']].set_visible(False)

    ax.xaxis.set_ticks_position('bottom')
    ax.tick_params(which='major', width=1.00, length=5)
    ax.tick_params(which='minor', width=0.75, length=2.5)
    ax.set_xlim(0, 5)
    ax.set_ylim(0, 1)
    ax.text(0.0, 0.2, title, transform=ax.transAxes,
            fontsize=14, fontname='Monospace', color='tab:blue')


fig, axs = plt.subplots(8, 1, figsize=(8, 6))

# Null Locator
setup(axs[0], title="NullLocator()")
axs[0].xaxis.set_major_locator(ticker.NullLocator())
axs[0].xaxis.set_minor_locator(ticker.NullLocator())

# Multiple Locator
setup(axs[1], title="MultipleLocator(0.5, offset=0.2)")
axs[1].xaxis.set_major_locator(ticker.MultipleLocator(0.5, offset=0.2))
```

(continues on next page)

(continued from previous page)

```
axs[1].xaxis.set_minor_locator(ticker.MultipleLocator(0.1))

# Fixed Locator
setup(axs[2], title="FixedLocator([0, 1, 5])")
axs[2].xaxis.set_major_locator(ticker.FixedLocator([0, 1, 5]))
axs[2].xaxis.set_minor_locator(ticker.FixedLocator(np.linspace(0.2, 0.8, 4)))

# Linear Locator
setup(axs[3], title="LinearLocator(numticks=3)")
axs[3].xaxis.set_major_locator(ticker.LinearLocator(3))
axs[3].xaxis.set_minor_locator(ticker.LinearLocator(31))

# Index Locator
setup(axs[4], title="IndexLocator(base=0.5, offset=0.25)")
axs[4].plot([0]*5, color='white')
axs[4].xaxis.set_major_locator(ticker.IndexLocator(base=0.5, offset=0.25))

# Auto Locator
setup(axs[5], title="AutoLocator()")
axs[5].xaxis.set_major_locator(ticker.AutoLocator())
axs[5].xaxis.set_minor_locator(ticker.AutoMinorLocator())

# MaxN Locator
setup(axs[6], title="MaxNLocator(n=4)")
axs[6].xaxis.set_major_locator(ticker.MaxNLocator(4))
axs[6].xaxis.set_minor_locator(ticker.MaxNLocator(40))

# Log Locator
setup(axs[7], title="LogLocator(base=10, numticks=15)")
axs[7].set_xlim(10**3, 10**10)
axs[7].set_xscale('log')
axs[7].xaxis.set_major_locator(ticker.LogLocator(base=10, numticks=15))

plt.tight_layout()
plt.show()
```




**References**

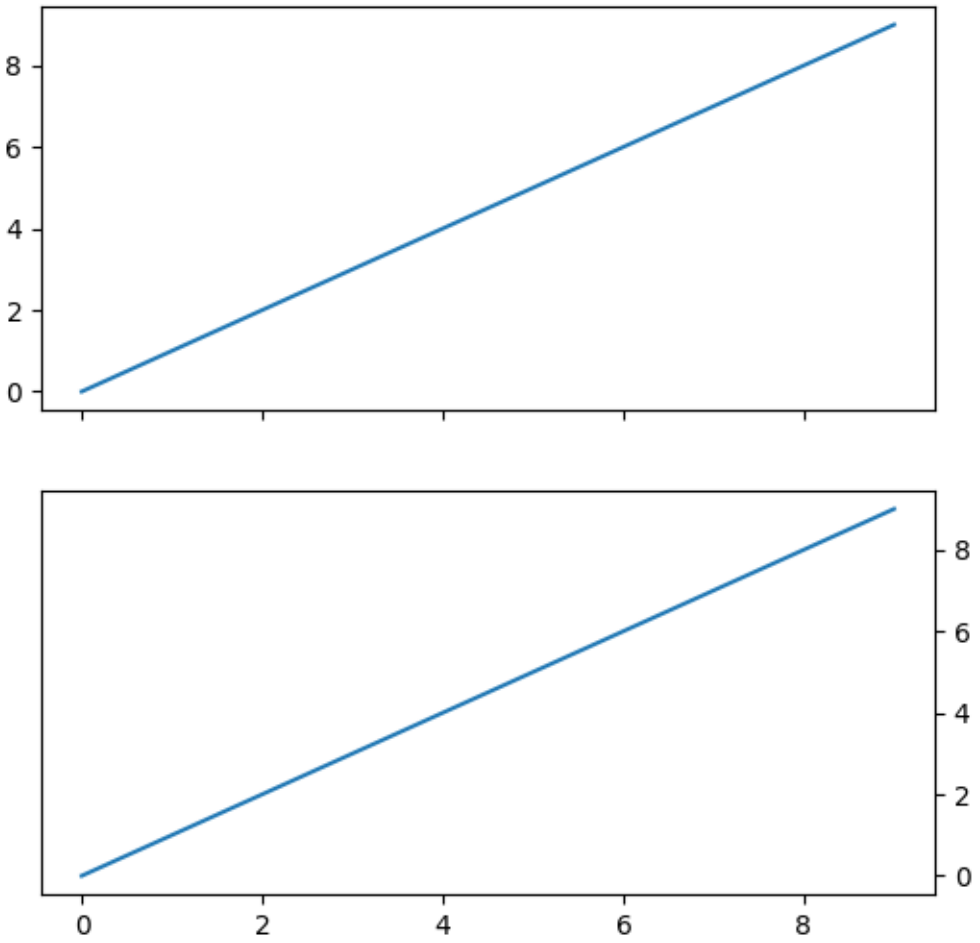
The following functions, methods, classes and modules are used in this example:

- *matplotlib.axis.Axis.set_major_locator*
- *matplotlib.axis.Axis.set_minor_locator*
- *matplotlib.ticker.NullLocator*
- *matplotlib.ticker.MultipleLocator*
- *matplotlib.ticker.FixedLocator*
- *matplotlib.ticker.LinearLocator*
- *matplotlib.ticker.IndexLocator*
- *matplotlib.ticker.AutoLocator*
- *matplotlib.ticker.MaxNLocator*
- *matplotlib.ticker.LogLocator*

**Total running time of the script:** (0 minutes 1.011 seconds)

### Set default y-axis tick labels on the right

We can use *rcParams["ytick.labelright"]* (default: `False`), *rcParams["ytick.right"]* (default: `False`), *rcParams["ytick.labelleft"]* (default: `True`), and *rcParams["ytick.left"]* (default: `True`) to control where on the axes ticks and their labels appear. These properties can also be set in `.matplotlib/matplotlibrc`.




```
import matplotlib.pyplot as plt
import numpy as np

plt.rcParams['ytick.right'] = plt.rcParams['ytick.labelright'] = True
plt.rcParams['ytick.left'] = plt.rcParams['ytick.labelleft'] = False

x = np.arange(10)
```

(continues on next page)

(continued from previous page)

```
fig, (ax0, ax1) = plt.subplots(2, 1, sharex=True, figsize=(6, 6))

ax0.plot(x)
ax0.yaxis.tick_left()

# use default parameter in rcParams, not calling tick_right()
ax1.plot(x)

plt.show()
```

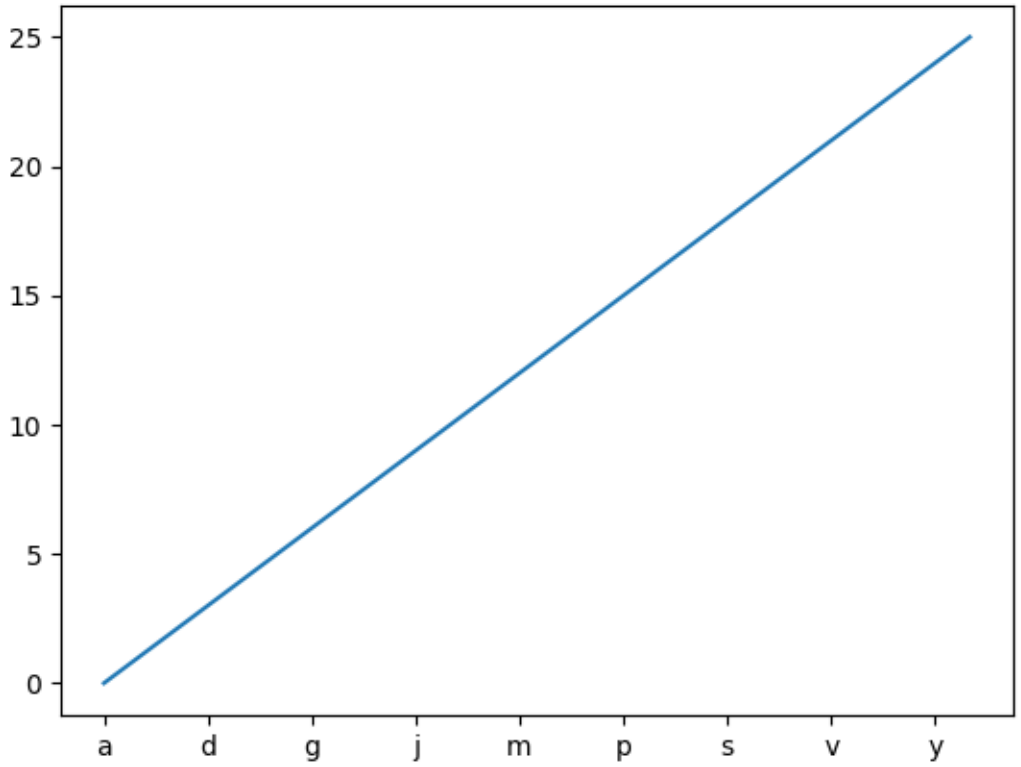
### Setting tick labels from a list of values

Using *Axes.set_xticks* causes the tick labels to be set on the currently chosen ticks. However, you may want to allow matplotlib to dynamically choose the number of ticks and their spacing.

In this case it may be better to determine the tick label from the value at the tick. The following example shows how to do this.

NB: The *ticker.MaxNLocator* is used here to ensure that the tick values take integer values.

```
import matplotlib.pyplot as plt

from matplotlib.ticker import MaxNLocator

fig, ax = plt.subplots()
xs = range(26)
ys = range(26)
labels = list('abcdefghijklmnopqrstuvwxyz')


def format_fn(tick_val, tick_pos):
    if int(tick_val) in xs:
        return labels[int(tick_val)]
    else:
        return ''


# A FuncFormatter is created automatically.
ax.xaxis.set_major_formatter(format_fn)
ax.xaxis.set_major_locator(MaxNLocator(integer=True))
ax.plot(xs, ys)
plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.pyplot.subplots*
- *matplotlib.axis.Axis.set_major_formatter*
- *matplotlib.axis.Axis.set_major_locator*
- *matplotlib.ticker.FuncFormatter*
- *matplotlib.ticker.MaxNLocator*

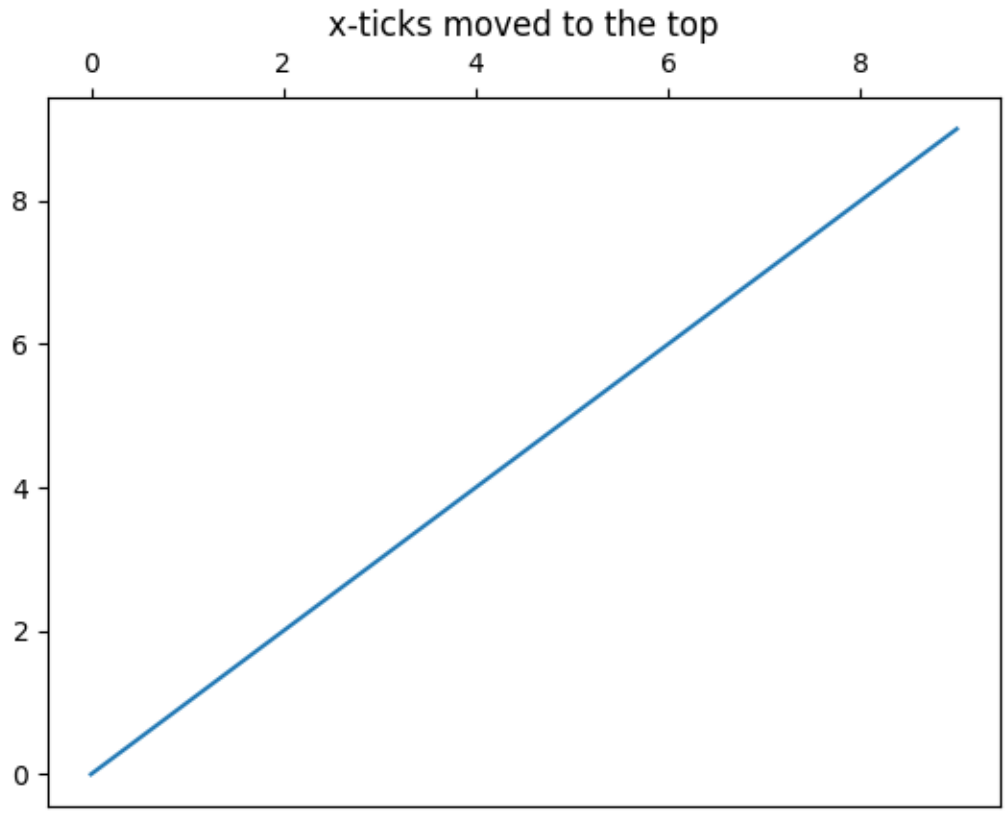
### Move x-axis tick labels to the top

*tick_params* can be used to configure the ticks. *top* and *labeltop* control the visibility tick lines and labels at the top x-axis. To move x-axis ticks from bottom to top, we have to activate the top ticks and deactivate the bottom ticks:

```
ax.tick_params(top=True, labeltop=True, bottom=False, labelbottom=False)
```

**Note:** If the change should be made for all future plots and not only the current Axes, you can adapt the respective config parameters

- *rcParams["xtick.top"]* (default: `False`)
- *rcParams["xtick.labeltop"]* (default: `False`)
- *rcParams["xtick.bottom"]* (default: `True`)
- *rcParams["xtick.labelbottom"]* (default: `True`)




```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
```

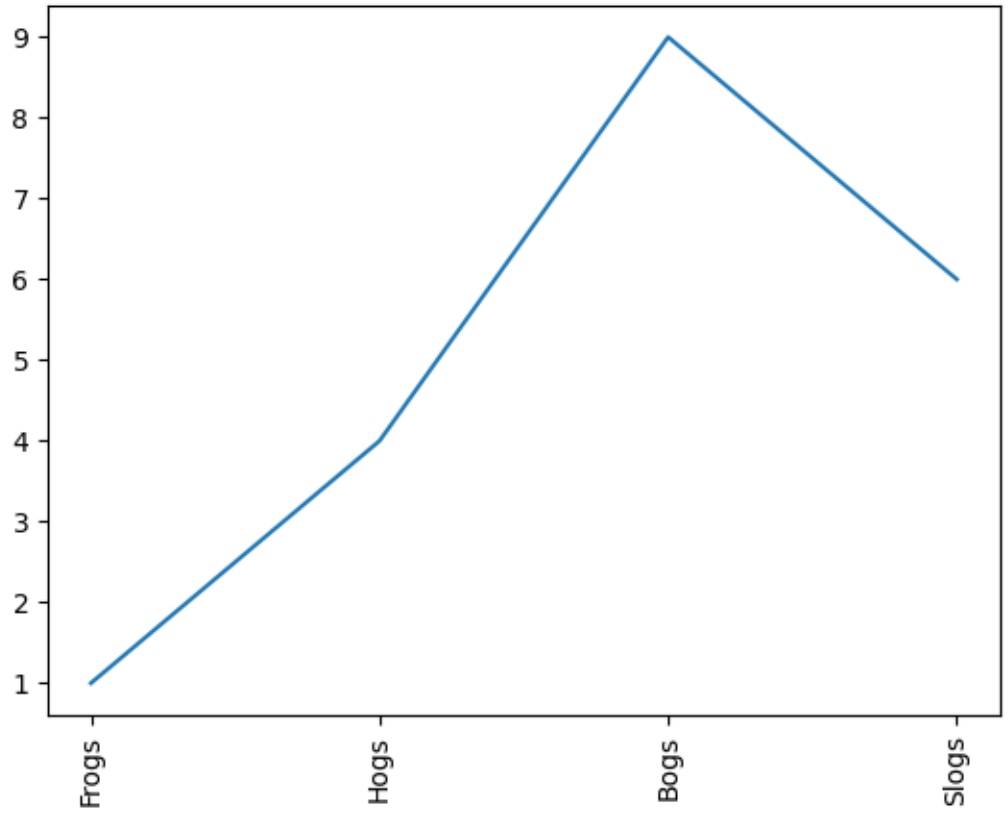
(continues on next page)

(continued from previous page)

```
ax.plot(range(10))
ax.tick_params(top=True, labeltop=True, bottom=False, labelbottom=False)
ax.set_title('x-ticks moved to the top')

plt.show()
```

### Rotating custom tick labels

Demo of custom tick-labels with user-defined rotation.




```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4]
y = [1, 4, 9, 6]
labels = ['Frogs', 'Hogs', 'Bogs', 'Slogs']

fig, ax = plt.subplots()
ax.plot(x, y)
# You can specify a rotation for the tick labels in degrees or with keywords.
ax.set_xticks(x, labels, rotation='vertical')
```

(continues on next page)

(continued from previous page)

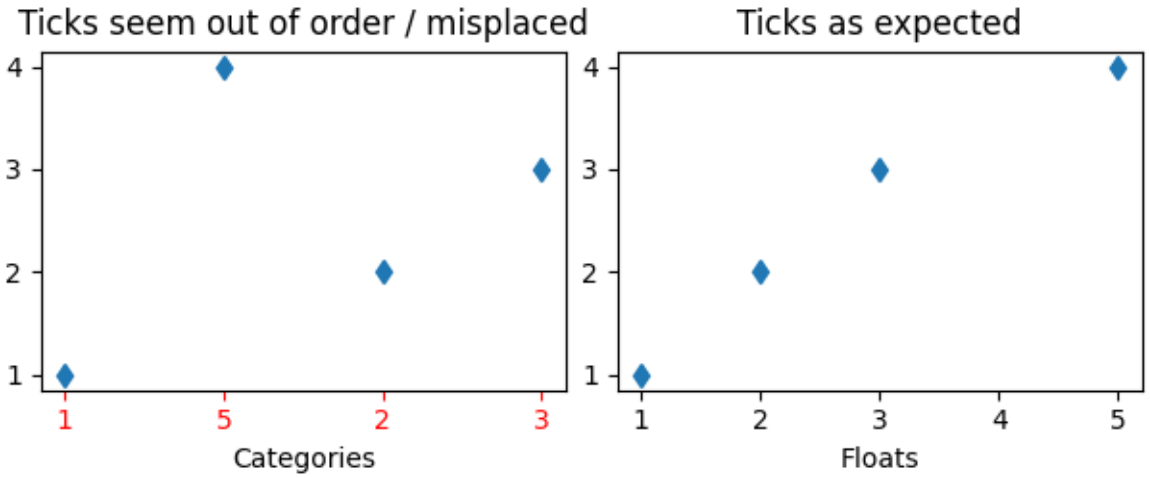
```
plt.show()
```

### Fixing too many ticks

One common cause for unexpected tick behavior is passing a list of strings instead of numbers or datetime objects. This can easily happen without notice when reading in a comma-delimited text file. Matplotlib treats lists of strings as *categorical* variables (*Plotting categorical variables*), and by default puts one tick per category, and plots them in the order in which they are supplied. If this is not desired, the solution is to convert the strings to a numeric type as in the following examples.

### Example 1: Strings can lead to an unexpected order of number ticks

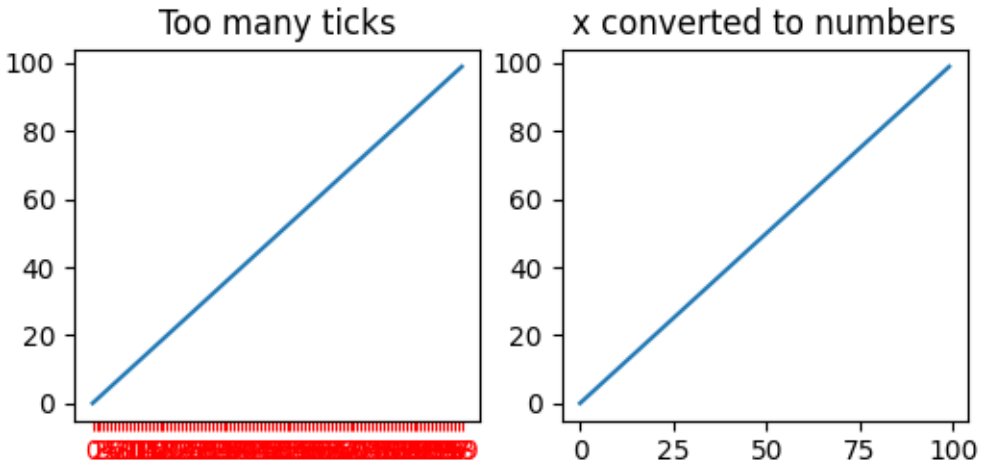
```
import matplotlib.pyplot as plt
import numpy as np

fig, ax = plt.subplots(1, 2, layout='constrained', figsize=(6, 2.5))
x = ['1', '5', '2', '3']
y = [1, 4, 2, 3]
ax[0].plot(x, y, 'd')
ax[0].tick_params(axis='x', color='r', labelcolor='r')
ax[0].set_xlabel('Categories')
ax[0].set_title('Ticks seem out of order / misplaced')

# convert to numbers:
x = np.asarray(x, dtype='float')
ax[1].plot(x, y, 'd')
ax[1].set_xlabel('Floats')
ax[1].set_title('Ticks as expected')
```



### Example 2: Strings can lead to very many ticks

If *x* has 100 elements, all strings, then we would have 100 (unreadable) ticks, and again the solution is to convert the strings to floats:

```
fig, ax = plt.subplots(1, 2, figsize=(6, 2.5))
x = [f'{xx}' for xx in np.arange(100)]
y = np.arange(100)
ax[0].plot(x, y)
ax[0].tick_params(axis='x', color='r', labelcolor='r')
ax[0].set_title('Too many ticks')
ax[0].set_xlabel('Categories')

ax[1].plot(np.asarray(x, float), y)
ax[1].set_title('x converted to numbers')
ax[1].set_xlabel('Floats')
```




### Example 3: Strings can lead to an unexpected order of datetime ticks

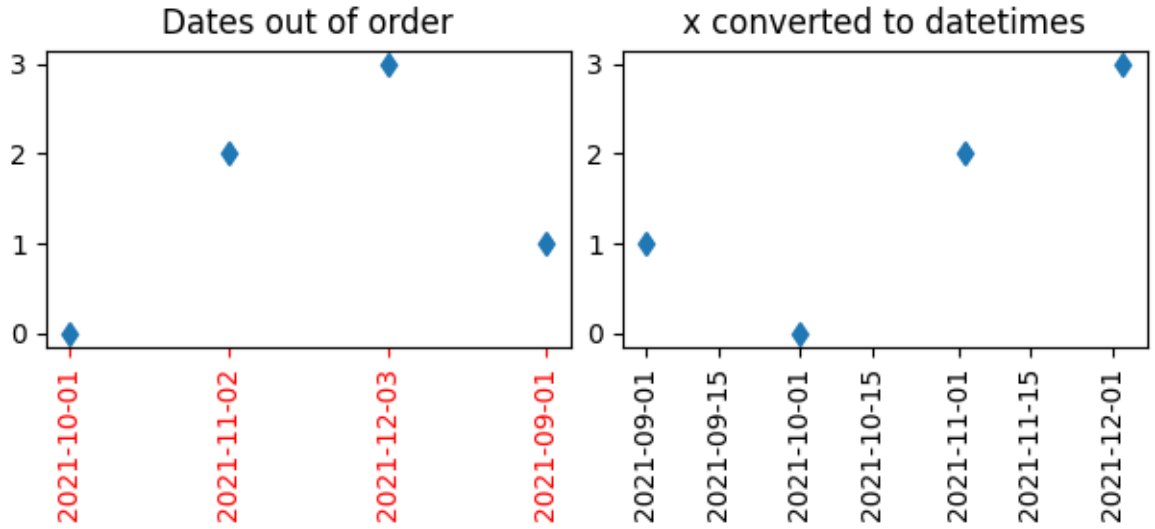
A common case is when dates are read from a CSV file, they need to be converted from strings to datetime objects to get the proper date locators and formatters.

```
fig, ax = plt.subplots(1, 2, layout='constrained', figsize=(6, 2.75))
x = ['2021-10-01', '2021-11-02', '2021-12-03', '2021-09-01']
y = [0, 2, 3, 1]
ax[0].plot(x, y, 'd')
ax[0].tick_params(axis='x', labelrotation=90, color='r', labelcolor='r')
ax[0].set_title('Dates out of order')

# convert to datetime64
x = np.asarray(x, dtype='datetime64[s]')
ax[1].plot(x, y, 'd')
ax[1].tick_params(axis='x', labelrotation=90)
ax[1].set_title('x converted to datetimes')
```

(continues on next page)

(continued from previous page)

```
plt.show()
```



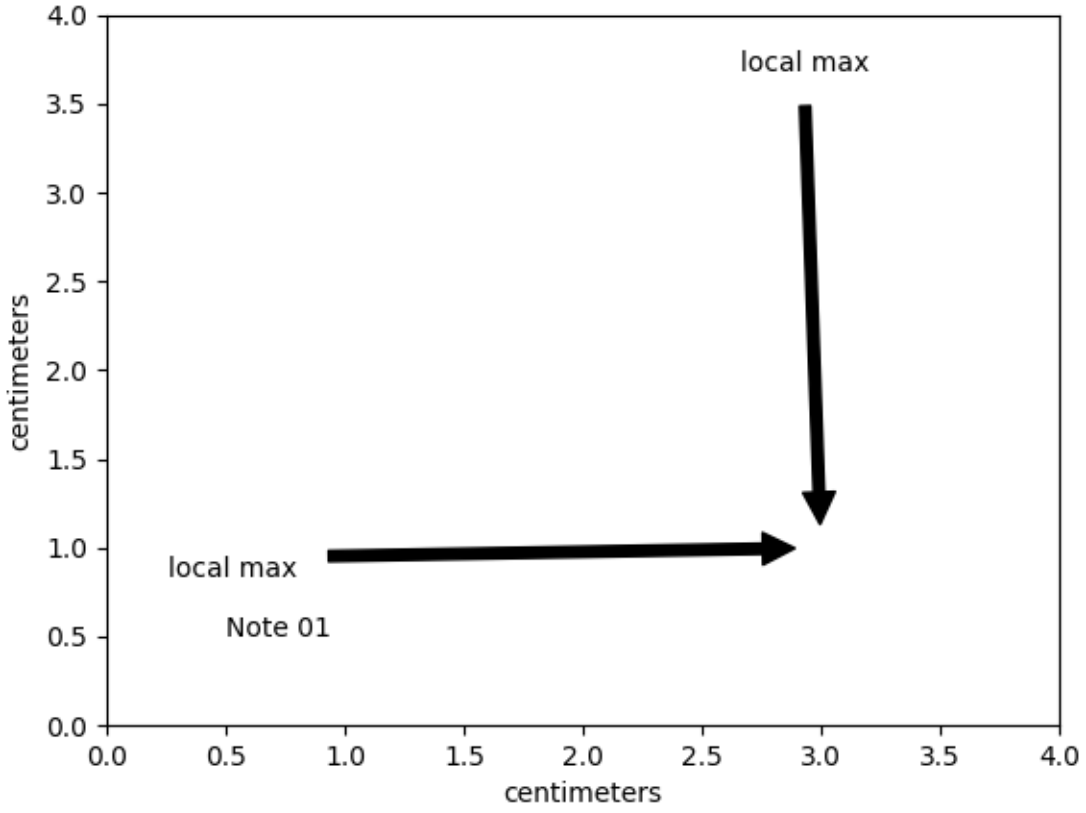

**Total running time of the script:** (0 minutes 1.075 seconds)

### 6.25.22 Units

These examples cover the many representations of units in Matplotlib.

#### Annotation with units

The example illustrates how to create text and arrow annotations using a centimeter-scale plot.




```
from basic_units import cm

import matplotlib.pyplot as plt

fig, ax = plt.subplots()

ax.annotate("Note 01", [0.5*cm, 0.5*cm])

# xy and text both unitized
ax.annotate('local max', xy=(3*cm, 1*cm), xycoords='data',
            xytext=(0.8*cm, 0.95*cm), textcoords='data',
            arrowprops=dict(facecolor='black', shrink=0.05),
            horizontalalignment='right', verticalalignment='top')

# mixing units w/ nonunits
ax.annotate('local max', xy=(3*cm, 1*cm), xycoords='data',
            xytext=(0.8, 0.95), textcoords='axes fraction',
            arrowprops=dict(facecolor='black', shrink=0.05),
            horizontalalignment='right', verticalalignment='top')


ax.set_xlim(0*cm, 4*cm)
ax.set_ylim(0*cm, 4*cm)
```

(continues on next page)

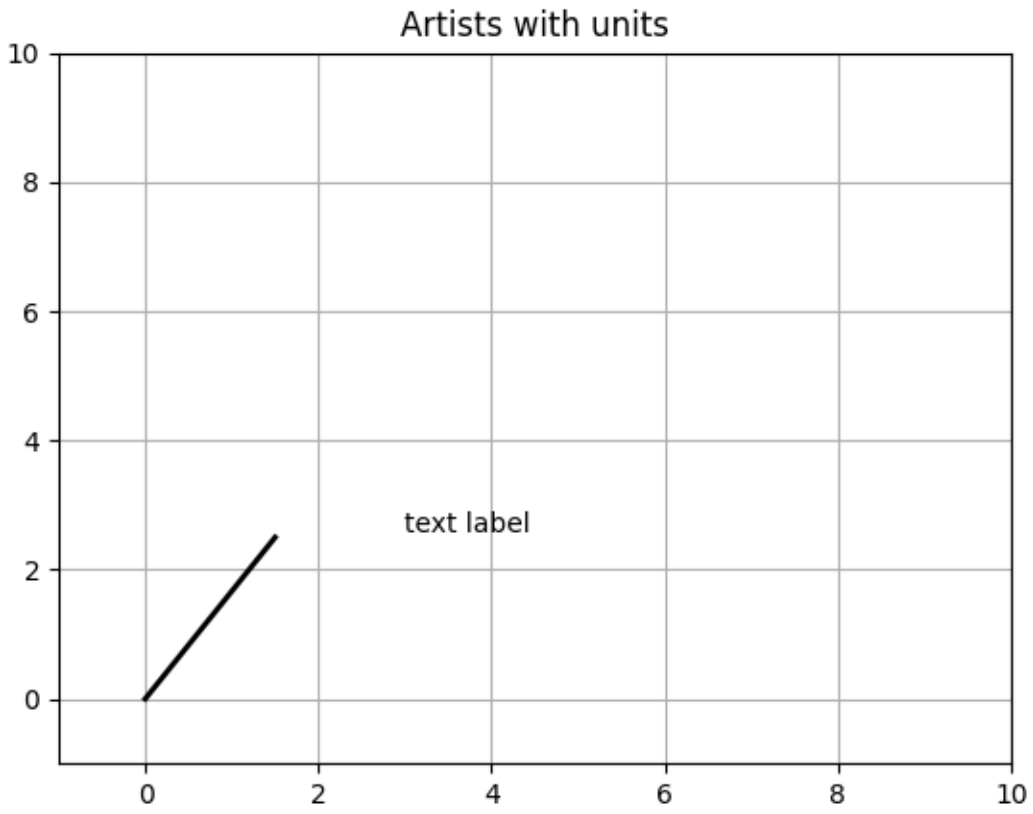
(continued from previous page)

```
plt.show()
```

### Artist tests

Test unit support with each of the Matplotlib primitive artist types.

The axis handles unit conversions and the artists keep a pointer to their axis parent. You must initialize the artists with the axis instance if you want to use them with unit data, or else they will not know how to convert the units to scalars.




```
import random

from basic_units import cm, inch

import matplotlib.pyplot as plt
import numpy as np

import matplotlib.collections as collections
import matplotlib.lines as lines
import matplotlib.patches as patches
```

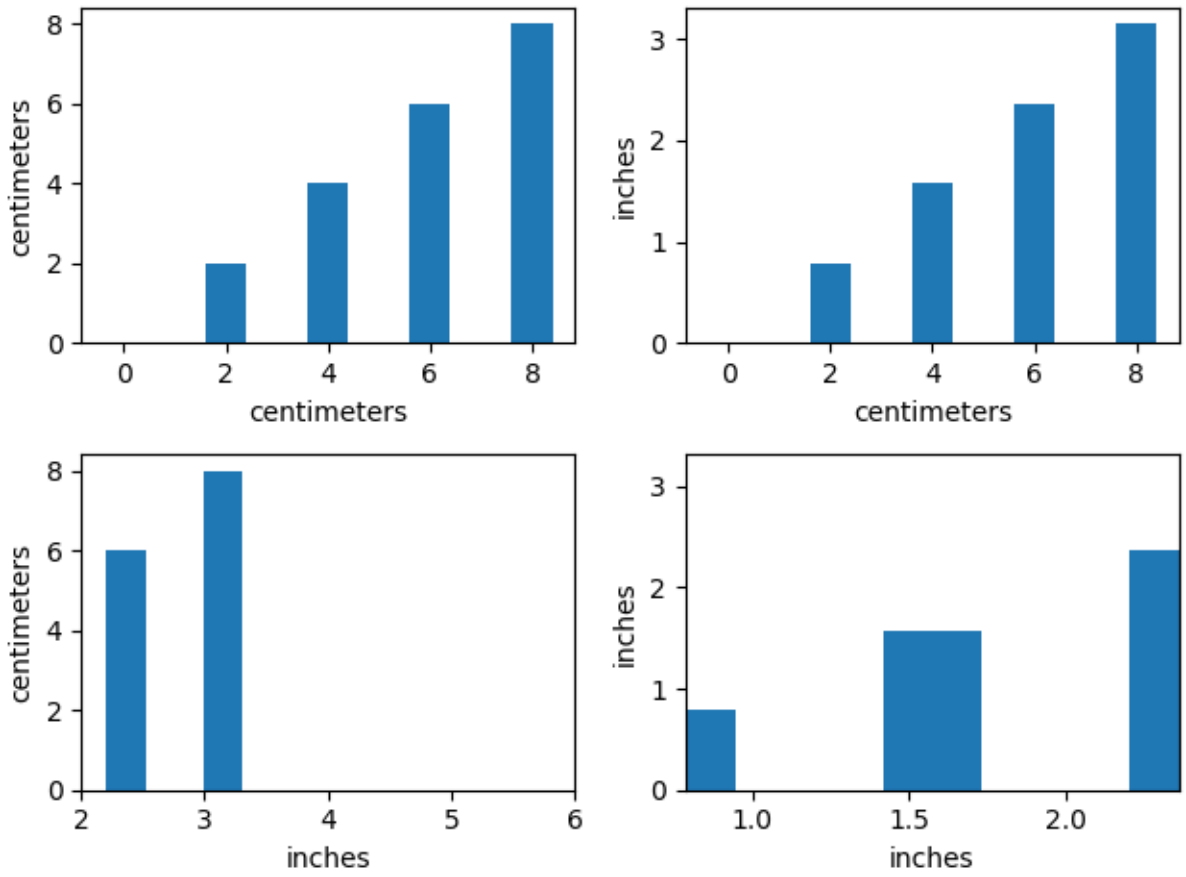
(continues on next page)

(continued from previous page)

```
import matplotlib.text as text

fig, ax = plt.subplots()
ax.xaxis.set_units(cm)
ax.yaxis.set_units(cm)

# Fixing random state for reproducibility
np.random.seed(19680801)

if 0:
    # test a line collection
    # Not supported at present.
    verts = []
    for i in range(10):
        # a random line segment in inches
        verts.append(zip(*inch*10*np.random.rand(2, random.randint(2, 15))))
    lc = collections.LineCollection(verts, axes=ax)
    ax.add_collection(lc)

# test a plain-ol-line
line = lines.Line2D([0*cm, 1.5*cm], [0*cm, 2.5*cm],
                    lw=2, color='black', axes=ax)
ax.add_line(line)

if 0:
    # test a patch
    # Not supported at present.
    rect = patches.Rectangle((1*cm, 1*cm), width=5*cm, height=2*cm,
                             alpha=0.2, axes=ax)
    ax.add_patch(rect)


t = text.Text(3*cm, 2.5*cm, 'text label', ha='left', va='bottom', axes=ax)
ax.add_artist(t)

ax.set_xlim(-1*cm, 10*cm)
ax.set_ylim(-1*cm, 10*cm)
# ax.xaxis.set_units(inch)
ax.grid(True)
ax.set_title("Artists with units")
plt.show()
```

### Bar demo with units

A plot using a variety of centimetre and inch conversions. This example shows how default unit introspection works (ax1), how various keywords can be used to set the x and y units to override the defaults (ax2, ax3, ax4) and how one can set the xlimits using scalars (ax3, current units assumed) or units (conversions applied to get the numbers to current units).




```
from basic_units import cm, inch

import matplotlib.pyplot as plt
import numpy as np

cms = cm * np.arange(0, 10, 2)
bottom = 0 * cm
width = 0.8 * cm

fig, axs = plt.subplots(2, 2)

axs[0, 0].bar(cms, cms, bottom=bottom)

axs[0, 1].bar(cms, cms, bottom=bottom, width=width, xunits=cm, yunits=inch)

axs[1, 0].bar(cms, cms, bottom=bottom, width=width, xunits=inch, yunits=cm)
```

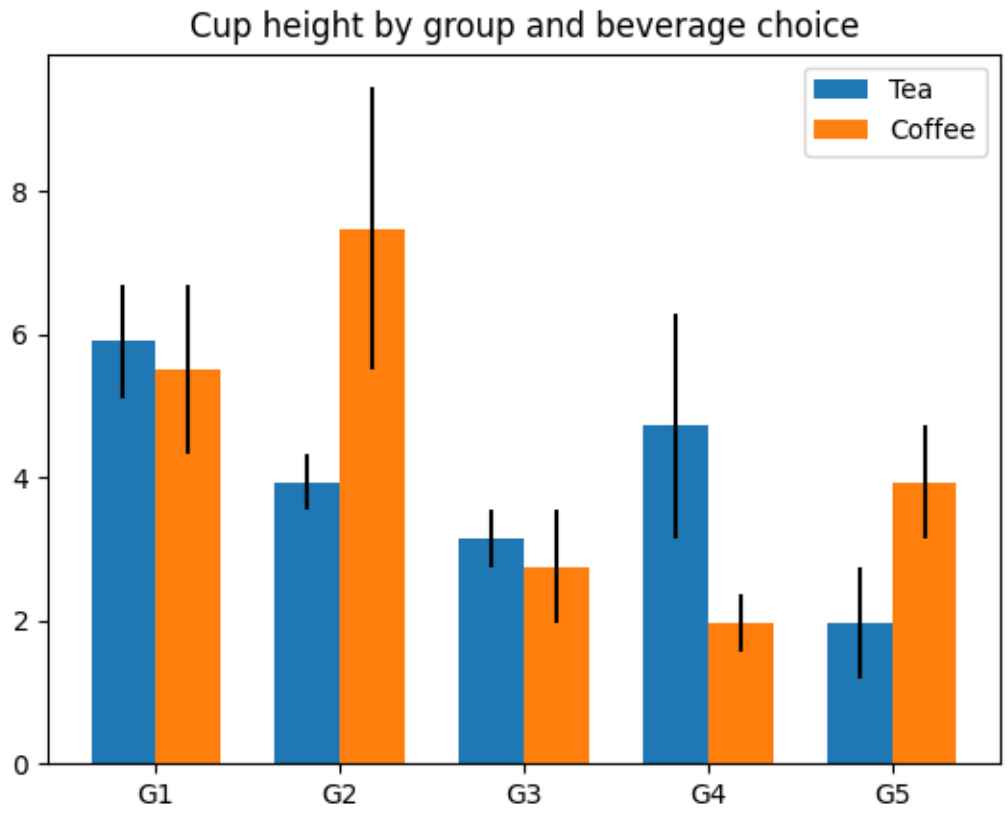
(continues on next page)

(continued from previous page)

```
axs[1, 0].set_xlim(2, 6)  # scalars are interpreted in current units

axs[1, 1].bar(cms, cms, bottom=bottom, width=width, xunits=inch, yunits=inch)
axs[1, 1].set_xlim(2 * cm, 6 * cm)  # cm are converted to inches

fig.tight_layout()
plt.show()
```

### Group barchart with units

This is the same example as *the barchart* in centimeters.




```
from basic_units import cm, inch

import matplotlib.pyplot as plt
import numpy as np

N = 5
tea_means = [15*cm, 10*cm, 8*cm, 12*cm, 5*cm]
tea_std = [2*cm, 1*cm, 1*cm, 4*cm, 2*cm]
```

(continues on next page)

(continued from previous page)

```
fig, ax = plt.subplots()
ax.yaxis.set_units(inch)

ind = np.arange(N)    # the x locations for the groups
width = 0.35          # the width of the bars
ax.bar(ind, tea_means, width, bottom=0*cm, yerr=tea_std, label='Tea')

coffee_means = (14*cm, 19*cm, 7*cm, 5*cm, 10*cm)
coffee_std = (3*cm, 5*cm, 2*cm, 1*cm, 2*cm)
ax.bar(ind + width, coffee_means, width, bottom=0*cm, yerr=coffee_std,
       label='Coffee')

ax.set_title('Cup height by group and beverage choice')
ax.set_xticks(ind + width / 2, labels=['G1', 'G2', 'G3', 'G4', 'G5'])

ax.legend()
ax.autoscale_view()

plt.show()
```

### Basic Units

```
import math

from packaging.version import parse as parse_version

import numpy as np

import matplotlib.ticker as ticker
import matplotlib.units as units


class ProxyDelegate:
    def __init__(self, fn_name, proxy_type):
        self.proxy_type = proxy_type
        self.fn_name = fn_name

    def __get__(self, obj, objtype=None):
        return self.proxy_type(self.fn_name, obj)


class TaggedValueMeta(type):
    def __init__(self, name, bases, dict):
        for fn_name in self._proxies:
            if not hasattr(self, fn_name):
                setattr(self, fn_name,
                        ProxyDelegate(fn_name, self._proxies[fn_name]))
```

(continues on next page)

(continued from previous page)

```
class PassThroughProxy:
    def __init__(self, fn_name, obj):
        self.fn_name = fn_name
        self.target = obj.proxy_target

    def __call__(self, *args):
        fn = getattr(self.target, self.fn_name)
        ret = fn(*args)
        return ret


class ConvertArgsProxy(PassThroughProxy):
    def __init__(self, fn_name, obj):
        super().__init__(fn_name, obj)
        self.unit = obj.unit

    def __call__(self, *args):
        converted_args = []
        for a in args:
            try:
                converted_args.append(a.convert_to(self.unit))
            except AttributeError:
                converted_args.append(TaggedValue(a, self.unit))
        converted_args = tuple([c.get_value() for c in converted_args])
        return super().__call__(*converted_args)


class ConvertReturnProxy(PassThroughProxy):
    def __init__(self, fn_name, obj):
        super().__init__(fn_name, obj)
        self.unit = obj.unit

    def __call__(self, *args):
        ret = super().__call__(*args)
        return (NotImplemented if ret is NotImplemented
                else TaggedValue(ret, self.unit))


class ConvertAllProxy(PassThroughProxy):
    def __init__(self, fn_name, obj):
        super().__init__(fn_name, obj)
        self.unit = obj.unit

    def __call__(self, *args):
        converted_args = []
        arg_units = [self.unit]
        for a in args:
            if hasattr(a, 'get_unit') and not hasattr(a, 'convert_to'):
                # If this argument has a unit type but no conversion ability,
                # this operation is prohibited.
                return NotImplemented
```

(continues on next page)

(continued from previous page)

```
            if hasattr(a, 'convert_to'):
                try:
                    a = a.convert_to(self.unit)
                except Exception:
                    pass
                arg_units.append(a.get_unit())
                converted_args.append(a.get_value())
            else:
                converted_args.append(a)
                if hasattr(a, 'get_unit'):
                    arg_units.append(a.get_unit())
                else:
                    arg_units.append(None)
        converted_args = tuple(converted_args)
        ret = super().__call__(*converted_args)
        if ret is NotImplemented:
            return NotImplemented
        ret_unit = unit_resolver(self.fn_name, arg_units)
        if ret_unit is NotImplemented:
            return NotImplemented
        return TaggedValue(ret, ret_unit)


class TaggedValue(metaclass=TaggedValueMeta):

    _proxies = {'__add__': ConvertAllProxy,
                '__sub__': ConvertAllProxy,
                '__mul__': ConvertAllProxy,
                '__rmul__': ConvertAllProxy,
                '__cmp__': ConvertAllProxy,
                '__lt__': ConvertAllProxy,
                '__gt__': ConvertAllProxy,
                '__len__': PassThroughProxy}

    def __new__(cls, value, unit):
        # generate a new subclass for value
        value_class = type(value)
        try:
            subcls = type(f'TaggedValue_of_{value_class.__name__}',
                          (cls, value_class), {})
            return object.__new__(subcls)
        except TypeError:
            return object.__new__(cls)

    def __init__(self, value, unit):
        self.value = value
        self.unit = unit
        self.proxy_target = self.value

    def __copy__(self):
        return TaggedValue(self.value, self.unit)
```

(continues on next page)

(continued from previous page)

```
    def __getattribute__(self, name):
        if name.startswith('__'):
            return object.__getattribute__(self, name)
        variable = object.__getattribute__(self, 'value')
        if hasattr(variable, name) and name not in self.__class__.__dict__:
            return getattr(variable, name)
        return object.__getattribute__(self, name)

    def __array__(self, dtype=object):
        return np.asarray(self.value, dtype)

    def __array_wrap__(self, array, context):
        return TaggedValue(array, self.unit)

    def __repr__(self):
        return f'TaggedValue({self.value!r}, {self.unit!r})'

    def __str__(self):
        return f"{self.value} in {self.unit}"

    def __len__(self):
        return len(self.value)

    if parse_version(np.__version__) >= parse_version('1.20'):
        def __getitem__(self, key):
            return TaggedValue(self.value[key], self.unit)

    def __iter__(self):
        # Return a generator expression rather than use `yield`, so that
        # TypeError is raised by iter(self) if appropriate when checking for
        # iterability.
        return (TaggedValue(inner, self.unit) for inner in self.value)

    def get_compressed_copy(self, mask):
        new_value = np.ma.masked_array(self.value, mask=mask).compressed()
        return TaggedValue(new_value, self.unit)

    def convert_to(self, unit):
        if unit == self.unit or not unit:
            return self
        try:
            new_value = self.unit.convert_value_to(self.value, unit)
        except AttributeError:
            new_value = self
        return TaggedValue(new_value, unit)

    def get_value(self):
        return self.value

    def get_unit(self):
        return self.unit
```

(continues on next page)

(continued from previous page)

```
class BasicUnit:
    def __init__(self, name, fullname=None):
        self.name = name
        if fullname is None:
            fullname = name
        self.fullname = fullname
        self.conversions = dict()

    def __repr__(self):
        return f'BasicUnit({self.name})'

    def __str__(self):
        return self.fullname

    def __call__(self, value):
        return TaggedValue(value, self)

    def __mul__(self, rhs):
        value = rhs
        unit = self
        if hasattr(rhs, 'get_unit'):
            value = rhs.get_value()
            unit = rhs.get_unit()
            unit = unit_resolver('__mul__', (self, unit))
        if unit is NotImplemented:
            return NotImplemented
        return TaggedValue(value, unit)

    def __rmul__(self, lhs):
        return self*lhs

    def __array_wrap__(self, array, context):
        return TaggedValue(array, self)

    def __array__(self, t=None, context=None):
        ret = np.array(1)
        if t is not None:
            return ret.astype(t)
        else:
            return ret

    def add_conversion_factor(self, unit, factor):
        def convert(x):
            return x*factor
        self.conversions[unit] = convert

    def add_conversion_fn(self, unit, fn):
        self.conversions[unit] = fn

    def get_conversion_fn(self, unit):
        return self.conversions[unit]
```

(continues on next page)

(continued from previous page)

```
    def convert_value_to(self, value, unit):
        conversion_fn = self.conversions[unit]
        ret = conversion_fn(value)
        return ret

    def get_unit(self):
        return self


class UnitResolver:
    def addition_rule(self, units):
        for unit_1, unit_2 in zip(units[:-1], units[1:]):
            if unit_1 != unit_2:
                return NotImplemented
        return units[0]

    def multiplication_rule(self, units):
        non_null = [u for u in units if u]
        if len(non_null) > 1:
            return NotImplemented
        return non_null[0]

    op_dict = {
        '__mul__': multiplication_rule,
        '__rmul__': multiplication_rule,
        '__add__': addition_rule,
        '__radd__': addition_rule,
        '__sub__': addition_rule,
        '__rsub__': addition_rule}

    def __call__(self, operation, units):
        if operation not in self.op_dict:
            return NotImplemented

        return self.op_dict[operation](self, units)


unit_resolver = UnitResolver()

cm = BasicUnit('cm', 'centimeters')
inch = BasicUnit('inch', 'inches')
inch.add_conversion_factor(cm, 2.54)
cm.add_conversion_factor(inch, 1/2.54)

radians = BasicUnit('rad', 'radians')
degrees = BasicUnit('deg', 'degrees')
radians.add_conversion_factor(degrees, 180.0/np.pi)
degrees.add_conversion_factor(radians, np.pi/180.0)

secs = BasicUnit('s', 'seconds')
hertz = BasicUnit('Hz', 'Hertz')
```

(continues on next page)

(continued from previous page)

```
minutes = BasicUnit('min', 'minutes')

secs.add_conversion_fn(hertz, lambda x: 1./x)
secs.add_conversion_factor(minutes, 1/60.0)


# radians formatting
def rad_fn(x, pos=None):
    if x >= 0:
        n = int((x / np.pi) * 2.0 + 0.25)
    else:
        n = int((x / np.pi) * 2.0 - 0.25)

    if n == 0:
        return '0'
    elif n == 1:
        return r'$\pi/2$'
    elif n == 2:
        return r'$\pi$'
    elif n == -1:
        return r'$-\pi/2$'
    elif n == -2:
        return r'$-\pi$'
    elif n % 2 == 0:
        return fr'${n//2}\pi$'
    else:
        return fr'${n}\pi/2$'


class BasicUnitConverter(units.ConversionInterface):
    @staticmethod
    def axisinfo(unit, axis):
        """Return AxisInfo instance for x and unit."""

        if unit == radians:
            return units.AxisInfo(
                majloc=ticker.MultipleLocator(base=np.pi/2),
                majfmt=ticker.FuncFormatter(rad_fn),
                label=unit.fullname,
            )
        elif unit == degrees:
            return units.AxisInfo(
                majloc=ticker.AutoLocator(),
                majfmt=ticker.FormatStrFormatter(r'$%i^\circ$'),
                label=unit.fullname,
            )
        elif unit is not None:
            if hasattr(unit, 'fullname'):
                return units.AxisInfo(label=unit.fullname)
            elif hasattr(unit, 'unit'):
                return units.AxisInfo(label=unit.unit.fullname)
        return None
```

(continues on next page)

(continued from previous page)

```
    @staticmethod
    def convert(val, unit, axis):
        if np.iterable(val):
            if isinstance(val, np.ma.MaskedArray):
                val = val.astype(float).filled(np.nan)
            out = np.empty(len(val))
            for i, thisval in enumerate(val):
                if np.ma.is_masked(thisval):
                    out[i] = np.nan
                else:
                    try:
                        out[i] = thisval.convert_to(unit).get_value()
                    except AttributeError:
                        out[i] = thisval
            return out
        if np.ma.is_masked(val):
            return np.nan
        else:
            return val.convert_to(unit).get_value()

    @staticmethod
    def default_units(x, axis):
        """Return the default unit for x or None."""
        if np.iterable(x):
            for thisx in x:
                return thisx.unit
        return x.unit


def cos(x):
    if np.iterable(x):
        return [math.cos(val.convert_to(radians).get_value()) for val in x]
    else:
        return math.cos(x.convert_to(radians).get_value())


units.registry[BasicUnit] = units.registry[TaggedValue] = BasicUnitConverter()
```

### Ellipse with units

Compare the ellipse generated with arcs versus a polygonal approximation.

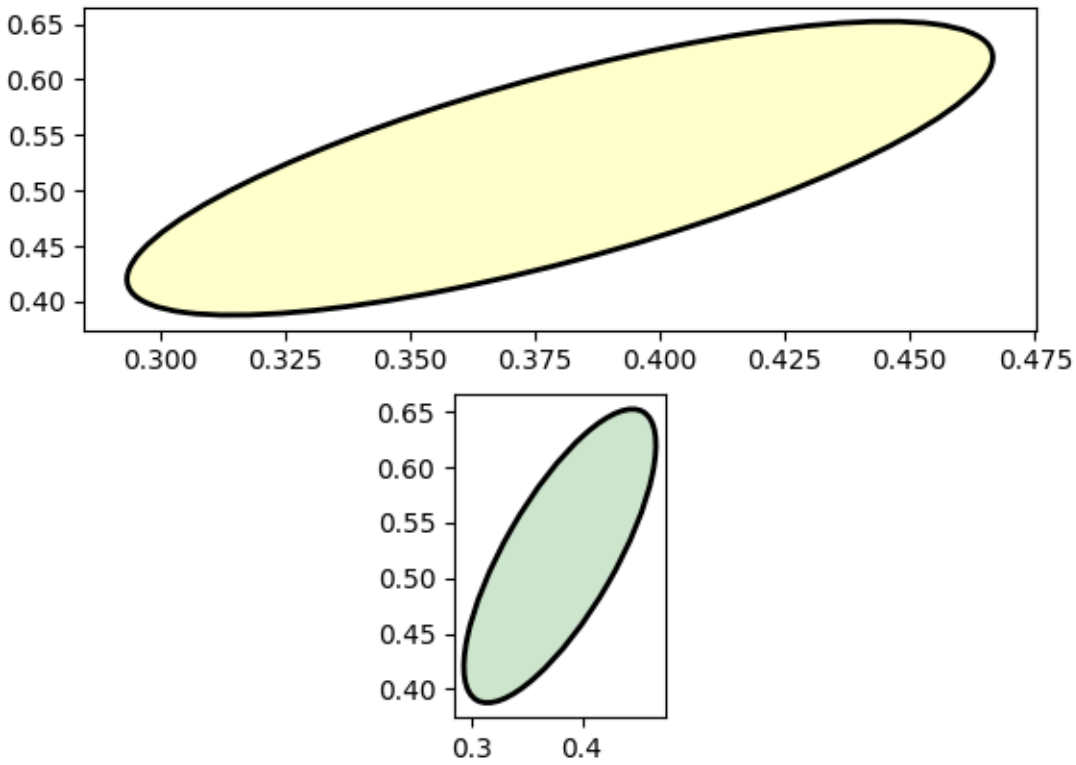
```
from basic_units import cm

import matplotlib.pyplot as plt
import numpy as np

from matplotlib import patches
```

(continues on next page)

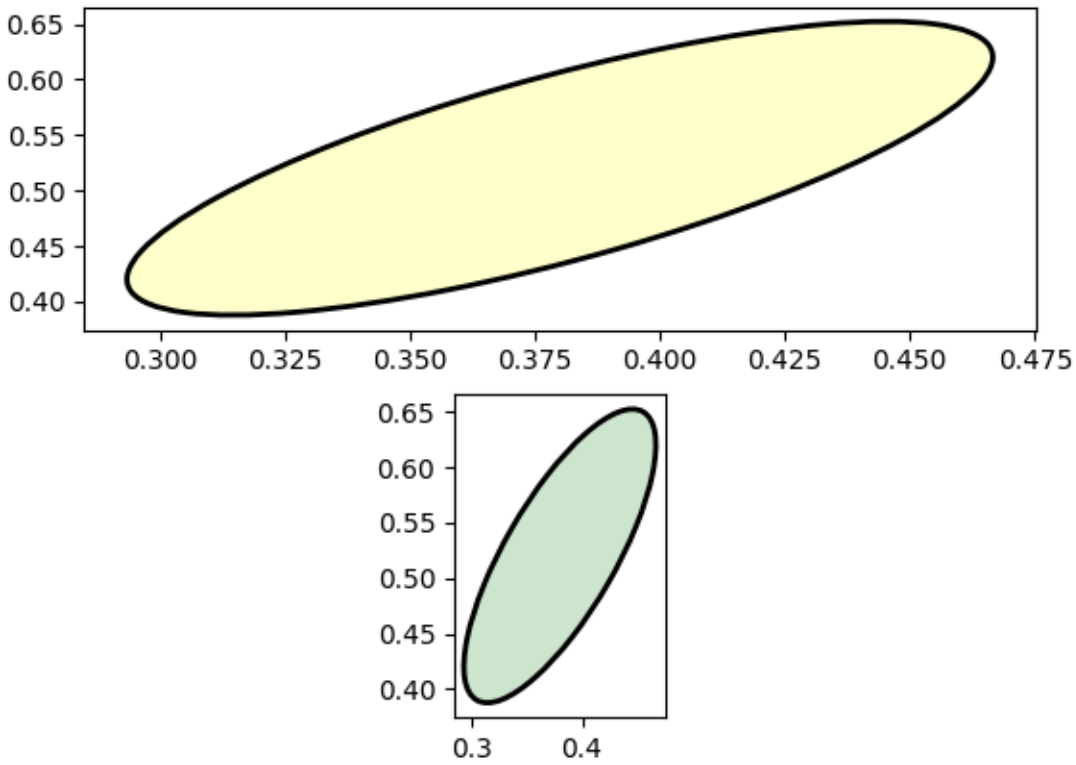
(continued from previous page)

```
xcenter, ycenter = 0.38*cm, 0.52*cm
width, height = 1e-1*cm, 3e-1*cm
angle = -30

theta = np.deg2rad(np.arange(0.0, 360.0, 1.0))
x = 0.5 * width * np.cos(theta)
y = 0.5 * height * np.sin(theta)

rtheta = np.radians(angle)
R = np.array([
    [np.cos(rtheta), -np.sin(rtheta)],
    [np.sin(rtheta),  np.cos(rtheta)],
    ])


x, y = np.dot(R, [x, y])
x += xcenter
y += ycenter
```

```
fig = plt.figure()
ax = fig.add_subplot(211, aspect='auto')
ax.fill(x, y, alpha=0.2, facecolor='yellow',
        edgecolor='yellow', linewidth=1, zorder=1)

e1 = patches.Ellipse((xcenter, ycenter), width, height,
                     angle=angle, linewidth=2, fill=False, zorder=2)

ax.add_patch(e1)

ax = fig.add_subplot(212, aspect='equal')
ax.fill(x, y, alpha=0.2, facecolor='green', edgecolor='green', zorder=1)
e2 = patches.Ellipse((xcenter, ycenter), width, height,
                     angle=angle, linewidth=2, fill=False, zorder=2)


ax.add_patch(e2)
fig.savefig('ellipse_compare')
```

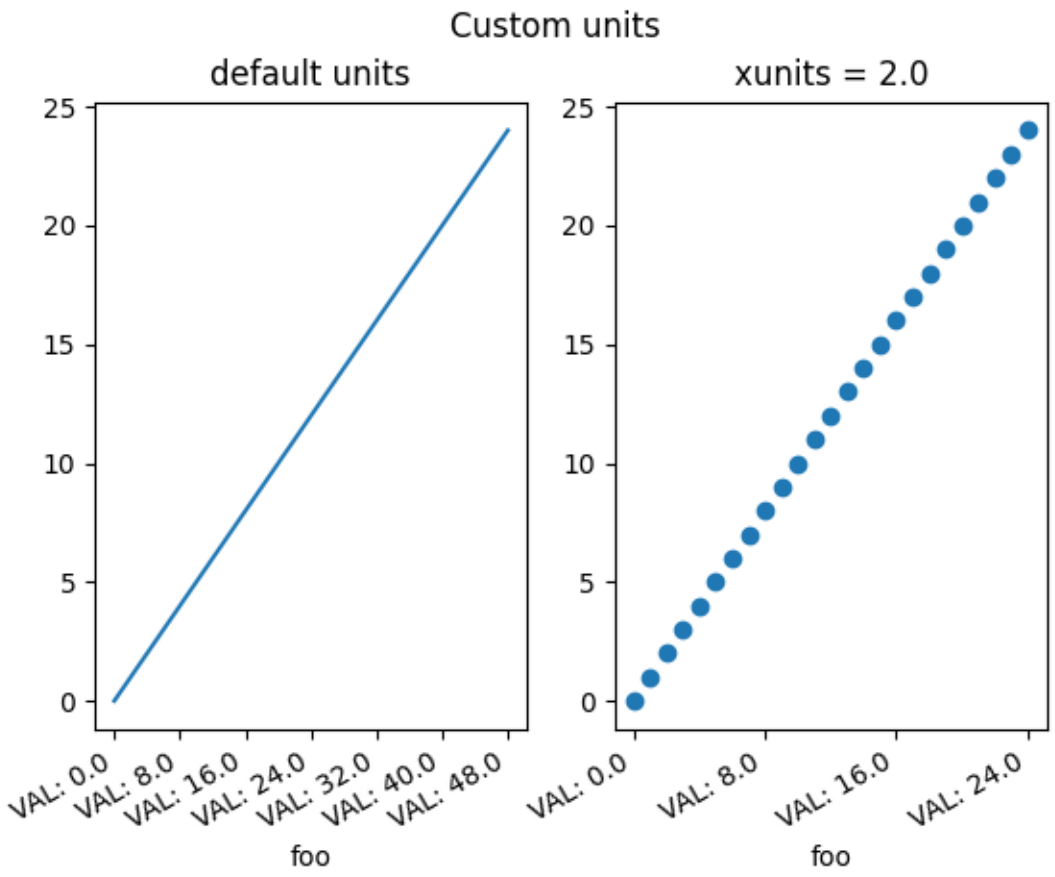



```
fig = plt.figure()
ax = fig.add_subplot(211, aspect='auto')
ax.fill(x, y, alpha=0.2, facecolor='yellow',
        edgecolor='yellow', linewidth=1, zorder=1)

e1 = patches.Arc((xcenter, ycenter), width, height,
                 angle=angle, linewidth=2, fill=False, zorder=2)

ax.add_patch(e1)

ax = fig.add_subplot(212, aspect='equal')
ax.fill(x, y, alpha=0.2, facecolor='green', edgecolor='green', zorder=1)
e2 = patches.Arc((xcenter, ycenter), width, height,
                 angle=angle, linewidth=2, fill=False, zorder=2)


ax.add_patch(e2)
fig.savefig('arc_compare')

plt.show()
```




### Evans test

A mockup "Foo" units class which supports conversion and different tick formatting depending on the "unit". Here the "unit" is just a scalar conversion factor, but this example shows that Matplotlib is entirely agnostic to what kind of units client packages use.




```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib.ticker as ticker
import matplotlib.units as units


class Foo:
    def __init__(self, val, unit=1.0):
        self.unit = unit
        self._val = val * unit

    def value(self, unit):
        if unit is None:
            unit = self.unit
        return self._val / unit


class FooConverter(units.ConversionInterface):
    @staticmethod
    def axisinfo(unit, axis):
        """Return the Foo AxisInfo."""
        if unit == 1.0 or unit == 2.0:
```

(continues on next page)

(continued from previous page)

```
            return units.AxisInfo(
                majloc=ticker.IndexLocator(8, 0),
                majfmt=ticker.FormatStrFormatter("VAL: %s"),
                label='foo',
                )

        else:
            return None

    @staticmethod
    def convert(obj, unit, axis):
        """
        Convert *obj* using *unit*.

        If *obj* is a sequence, return the converted sequence.
        """
        if np.iterable(obj):
            return [o.value(unit) for o in obj]
        else:
            return obj.value(unit)

    @staticmethod
    def default_units(x, axis):
        """Return the default unit for *x* or None."""
        if np.iterable(x):
            for thisx in x:
                return thisx.unit
        else:
            return x.unit


units.registry[Foo] = FooConverter()

# create some Foos
x = [Foo(val, 1.0) for val in range(0, 50, 2)]
# and some arbitrary y data
y = [i for i in range(len(x))]

fig, (ax1, ax2) = plt.subplots(1, 2)
fig.suptitle("Custom units")
fig.subplots_adjust(bottom=0.2)

# plot specifying units
ax2.plot(x, y, 'o', xunits=2.0)
ax2.set_title("xunits = 2.0")
plt.setp(ax2.get_xticklabels(), rotation=30, ha='right')

# plot without specifying units; will use the None branch for axisinfo
ax1.plot(x, y)  # uses default units
ax1.set_title('default units')
plt.setp(ax1.get_xticklabels(), rotation=30, ha='right')
```

(continues on next page)

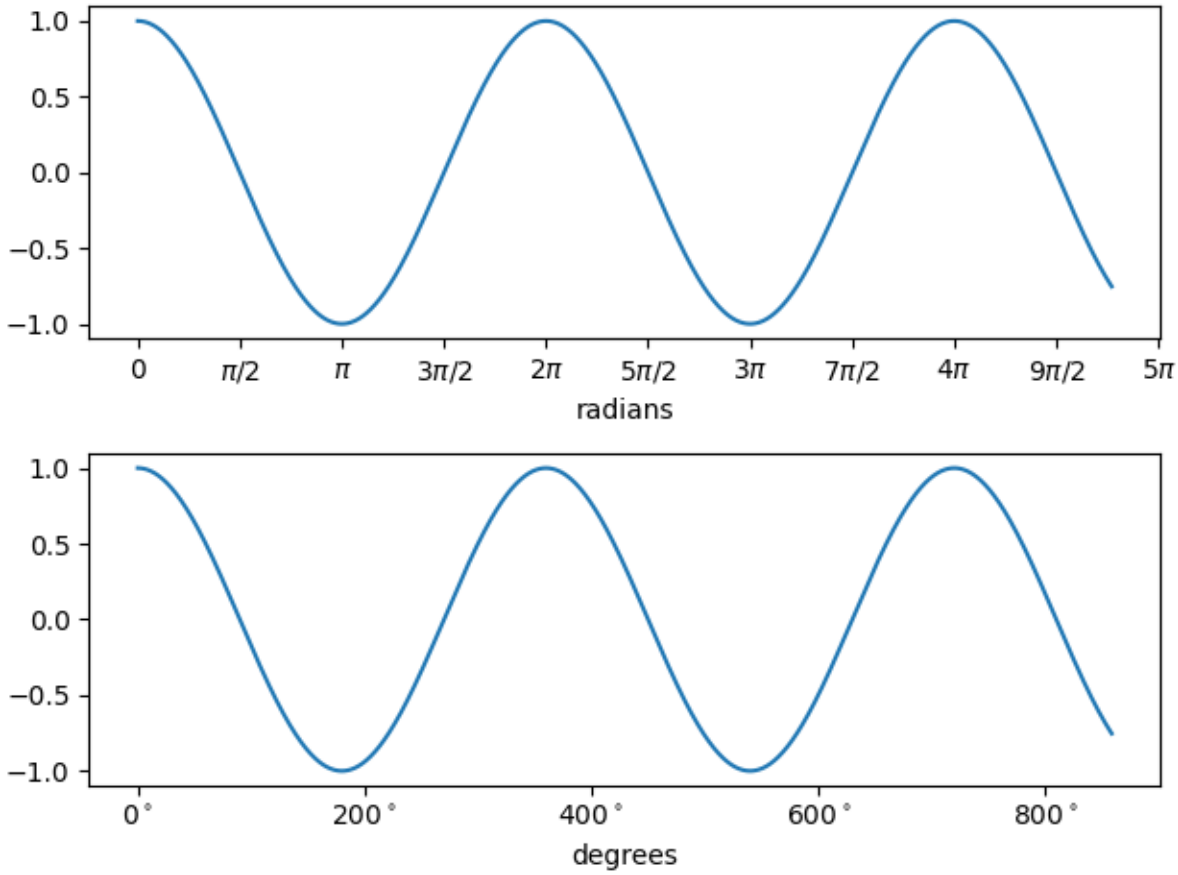
(continued from previous page)

```
plt.show()
```

### Radian ticks

Plot with radians from the basic_units mockup example package.

This example shows how the unit class can determine the tick locating, formatting and axis labeling.




```
from basic_units import cos, degrees, radians

import matplotlib.pyplot as plt
import numpy as np

x = [val*radians for val in np.arange(0, 15, 0.01)]

fig, axs = plt.subplots(2)

axs[0].plot(x, cos(x), xunits=radians)
axs[1].plot(x, cos(x), xunits=degrees)
```

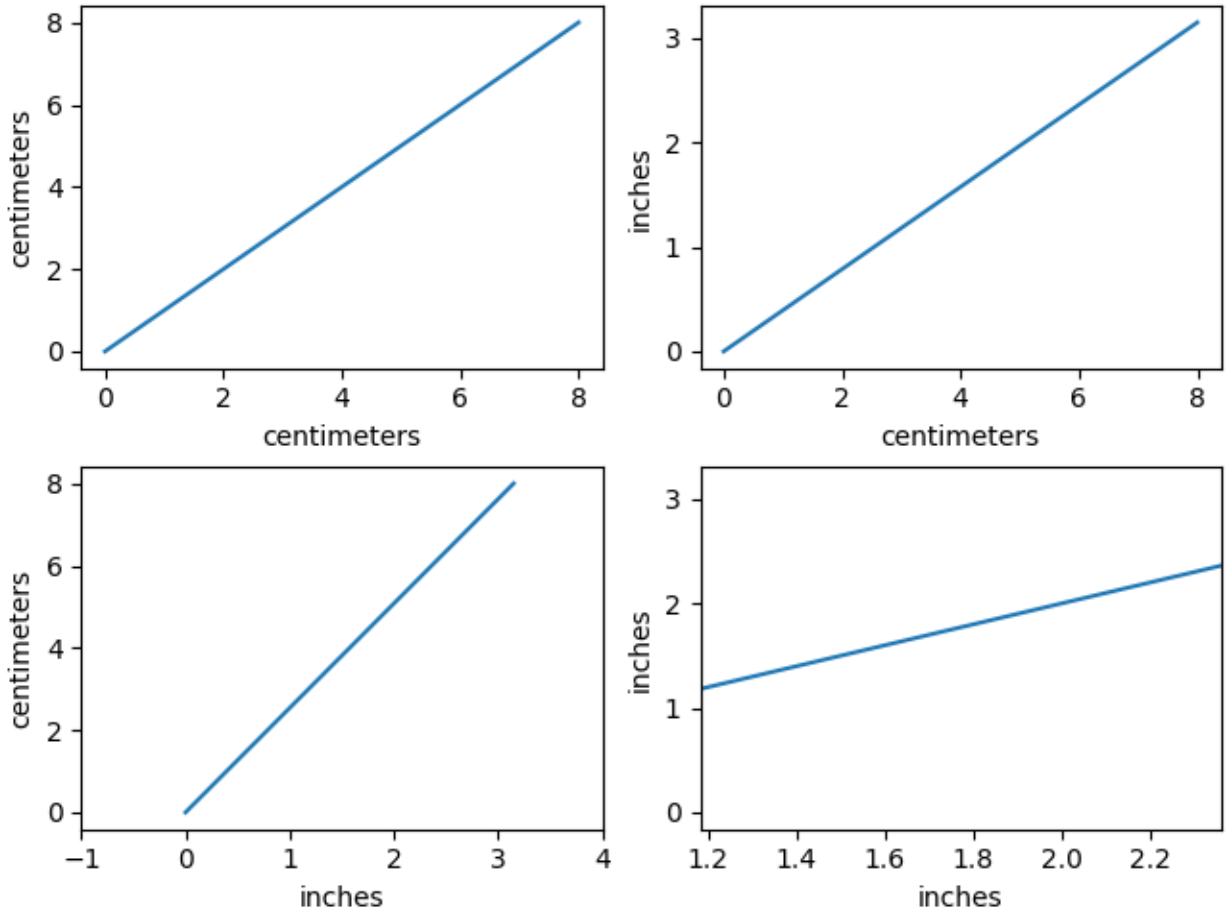
(continues on next page)

(continued from previous page)

```
fig.tight_layout()
plt.show()
```

### Inches and Centimeters

The example illustrates the ability to override default x and y units (ax1) to inches and centimeters using the *xunits* and *yunits* parameters for the `plot` function. Note that conversions are applied to get numbers to correct units.




```
from basic_units import cm, inch

import matplotlib.pyplot as plt
import numpy as np

cms = cm * np.arange(0, 10, 2)

fig, axs = plt.subplots(2, 2, layout='constrained')

axs[0, 0].plot(cms, cms)
```

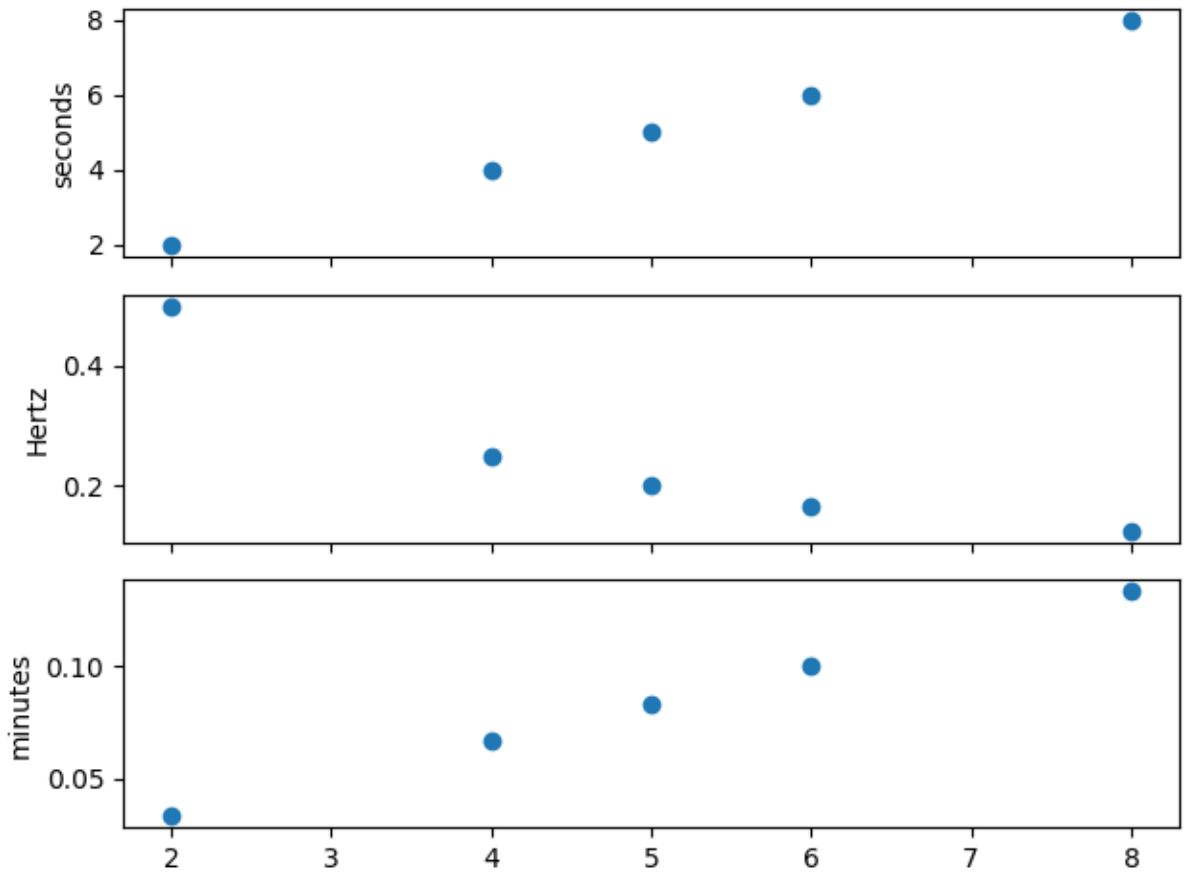
(continues on next page)

(continued from previous page)

```
axs[0, 1].plot(cms, cms, xunits=cm, yunits=inch)

axs[1, 0].plot(cms, cms, xunits=inch, yunits=cm)
axs[1, 0].set_xlim(-1, 4)  # scalars are interpreted in current units

axs[1, 1].plot(cms, cms, xunits=inch, yunits=inch)
axs[1, 1].set_xlim(3*cm, 6*cm)  # cm are converted to inches

plt.show()
```

### Unit handling

The example below shows support for unit conversions over masked arrays.




```
from basic_units import hertz, minutes, secs

import matplotlib.pyplot as plt
import numpy as np

# create masked array
```

(continues on next page)

(continued from previous page)

```
data = (1, 2, 3, 4, 5, 6, 7, 8)
mask = (1, 0, 1, 0, 0, 0, 1, 0)
xsecs = secs * np.ma.MaskedArray(data, mask, float)

fig, (ax1, ax2, ax3) = plt.subplots(nrows=3, sharex=True)

ax1.scatter(xsecs, xsecs)
ax1.yaxis.set_units(secs)
ax2.scatter(xsecs, xsecs, yunits=hertz)
ax3.scatter(xsecs, xsecs, yunits=minutes)

fig.tight_layout()
plt.show()
```

### 6.25.23 Embedding Matplotlib in graphical user interfaces

You can embed Matplotlib directly into a user interface application by following the embedding_in_SOMEGUI.py examples here. Currently Matplotlib supports PyQt/PySide, PyGObject, Tkinter, and wxPython.

When embedding Matplotlib in a GUI, you must use the Matplotlib API directly rather than the pylab/pyplot procedural interface, so take a look at the examples/api directory for some example code working with the API.

#### CanvasAgg demo

This example shows how to use the agg backend directly to create images, which may be of use to web application developers who want full control over their code without using the pyplot interface to manage figures, figure closing etc.

---

**Note:** It is not necessary to avoid using the pyplot interface in order to create figures without a graphical front-end - simply setting the backend to "Agg" would be sufficient.

---

In this example, we show how to save the contents of the agg canvas to a file, and how to extract them to a numpy array, which can in turn be passed off to Pillow. The latter functionality allows e.g. to use Matplotlib inside a cgi-script *without* needing to write a figure to disk, and to write images in any format supported by Pillow.

```
from PIL import Image

import numpy as np

from matplotlib.backends.backend_agg import FigureCanvasAgg
from matplotlib.figure import Figure

fig = Figure(figsize=(5, 4), dpi=100)
```

(continues on next page)

(continued from previous page)

```
# A canvas must be manually attached to the figure (pyplot would automatically
# do it).  This is done by instantiating the canvas with the figure as
# argument.
canvas = FigureCanvasAgg(fig)

# Do some plotting.
ax = fig.add_subplot()
ax.plot([1, 2, 3])

# Option 1: Save the figure to a file; can also be a file-like object␣
 ↪(BytesIO,
# etc.).
fig.savefig("test.png")

# Option 2: Retrieve a memoryview on the renderer buffer, and convert it to a
# numpy array.
canvas.draw()
rgba = np.asarray(canvas.buffer_rgba())
# ... and pass it to PIL.
im = Image.fromarray(rgba)
# This image can then be saved to any format supported by Pillow, e.g.:
im.save("test.bmp")

# Uncomment this line to display the image using ImageMagick's `display` tool.
# im.show()
```

**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.backends.backend_agg.FigureCanvasAgg*
- *matplotlib.figure.Figure*
- *matplotlib.figure.Figure.add_subplot*
- *matplotlib.figure.Figure.savefig* / *matplotlib.pyplot.savefig*
- *matplotlib.axes.Axes.plot* / *matplotlib.pyplot.plot*

### Embedding in GTK3 with a navigation toolbar

Demonstrate NavigationToolbar with GTK3 accessed via pygobject.

```
import gi

gi.require_version('Gtk', '3.0')
from gi.repository import Gtk

import numpy as np
```

(continues on next page)

(continued from previous page)

```
from matplotlib.backends.backend_gtk3 import \
    NavigationToolbar2GTK3 as NavigationToolbar
from matplotlib.backends.backend_gtk3agg import \
    FigureCanvasGTK3Agg as FigureCanvas
from matplotlib.figure import Figure

win = Gtk.Window()
win.connect("delete-event", Gtk.main_quit)
win.set_default_size(400, 300)
win.set_title("Embedding in GTK3")

fig = Figure(figsize=(5, 4), dpi=100)
ax = fig.add_subplot(1, 1, 1)
t = np.arange(0.0, 3.0, 0.01)
s = np.sin(2*np.pi*t)
ax.plot(t, s)

vbox = Gtk.VBox()
win.add(vbox)

# Add canvas to vbox
canvas = FigureCanvas(fig)  # a Gtk.DrawingArea
vbox.pack_start(canvas, True, True, 0)

# Create toolbar
toolbar = NavigationToolbar(canvas)
vbox.pack_start(toolbar, False, False, 0)

win.show_all()
Gtk.main()
```

### Embedding in GTK3

Demonstrate adding a FigureCanvasGTK3Agg widget to a Gtk.ScrolledWindow using GTK3 accessed via pygobject.

```
import gi

gi.require_version('Gtk', '3.0')
from gi.repository import Gtk

import numpy as np

from matplotlib.backends.backend_gtk3agg import \
    FigureCanvasGTK3Agg as FigureCanvas
from matplotlib.figure import Figure

win = Gtk.Window()
win.connect("delete-event", Gtk.main_quit)
```

(continues on next page)

(continued from previous page)

```
win.set_default_size(400, 300)
win.set_title("Embedding in GTK3")

fig = Figure(figsize=(5, 4), dpi=100)
ax = fig.add_subplot()
t = np.arange(0.0, 3.0, 0.01)
s = np.sin(2*np.pi*t)
ax.plot(t, s)

sw = Gtk.ScrolledWindow()
win.add(sw)
# A scrolled window border goes outside the scrollbars and viewport
sw.set_border_width(10)

canvas = FigureCanvas(fig)  # a Gtk.DrawingArea
canvas.set_size_request(800, 600)
sw.add(canvas)

win.show_all()
Gtk.main()
```

### Embedding in GTK4 with a navigation toolbar

Demonstrate NavigationToolbar with GTK4 accessed via pygobject.

```
import gi

gi.require_version('Gtk', '4.0')
from gi.repository import Gtk

import numpy as np

from matplotlib.backends.backend_gtk4 import \
    NavigationToolbar2GTK4 as NavigationToolbar
from matplotlib.backends.backend_gtk4agg import \
    FigureCanvasGTK4Agg as FigureCanvas
from matplotlib.figure import Figure


def on_activate(app):
    win = Gtk.ApplicationWindow(application=app)
    win.set_default_size(400, 300)
    win.set_title("Embedding in GTK4")

    fig = Figure(figsize=(5, 4), dpi=100)
    ax = fig.add_subplot(1, 1, 1)
    t = np.arange(0.0, 3.0, 0.01)
    s = np.sin(2*np.pi*t)
    ax.plot(t, s)
```

(continues on next page)

(continued from previous page)

```
    vbox = Gtk.Box(orientation=Gtk.Orientation.VERTICAL)
    win.set_child(vbox)

    # Add canvas to vbox
    canvas = FigureCanvas(fig)  # a Gtk.DrawingArea
    canvas.set_hexpand(True)
    canvas.set_vexpand(True)
    vbox.append(canvas)

    # Create toolbar
    toolbar = NavigationToolbar(canvas)
    vbox.append(toolbar)

    win.show()


app = Gtk.Application(
    application_id='org.matplotlib.examples.EmbeddingInGTK4PanZoom')
app.connect('activate', on_activate)
app.run(None)
```

### Embedding in GTK4

Demonstrate adding a FigureCanvasGTK4Agg widget to a Gtk.ScrolledWindow using GTK4 accessed via pygobject.

```
import gi

gi.require_version('Gtk', '4.0')
from gi.repository import Gtk

import numpy as np

from matplotlib.backends.backend_gtk4agg import \
    FigureCanvasGTK4Agg as FigureCanvas
from matplotlib.figure import Figure


def on_activate(app):
    win = Gtk.ApplicationWindow(application=app)
    win.set_default_size(400, 300)
    win.set_title("Embedding in GTK4")

    fig = Figure(figsize=(5, 4), dpi=100)
    ax = fig.add_subplot()
    t = np.arange(0.0, 3.0, 0.01)
    s = np.sin(2*np.pi*t)
    ax.plot(t, s)

    # A scrolled margin goes outside the scrollbars and viewport.
```

(continues on next page)

(continued from previous page)

```
    sw = Gtk.ScrolledWindow(margin_top=10, margin_bottom=10,
                            margin_start=10, margin_end=10)
    win.set_child(sw)

    canvas = FigureCanvas(fig)  # a Gtk.DrawingArea
    canvas.set_size_request(800, 600)
    sw.set_child(canvas)

    win.show()


app = Gtk.Application(application_id='org.matplotlib.examples.EmbeddingInGTK4
 ↪')
app.connect('activate', on_activate)
app.run(None)
```

### Embedding in Qt

Simple Qt application embedding Matplotlib canvases. This program will work equally well using any Qt binding (PyQt6, PySide6, PyQt5, PySide2). The binding can be selected by setting the *QT_API* environment variable to the binding name, or by first importing it.

```
import sys
import time

import numpy as np

from matplotlib.backends.backend_qtagg import FigureCanvas
from matplotlib.backends.backend_qtagg import \
    NavigationToolbar2QT as NavigationToolbar
from matplotlib.backends.qt_compat import QtWidgets
from matplotlib.figure import Figure


class ApplicationWindow(QtWidgets.QMainWindow):
    def __init__(self):
        super().__init__()
        self._main = QtWidgets.QWidget()
        self.setCentralWidget(self._main)
        layout = QtWidgets.QVBoxLayout(self._main)

        static_canvas = FigureCanvas(Figure(figsize=(5, 3)))
        # Ideally one would use self.addToolBar here, but it is slightly
        # incompatible between PyQt6 and other bindings, so we just add the
        # toolbar as a plain widget instead.
        layout.addWidget(NavigationToolbar(static_canvas, self))
        layout.addWidget(static_canvas)

        dynamic_canvas = FigureCanvas(Figure(figsize=(5, 3)))
        layout.addWidget(dynamic_canvas)
```

(continues on next page)

(continued from previous page)

```
        layout.addWidget(NavigationToolbar(dynamic_canvas, self))

        self._static_ax = static_canvas.figure.subplots()
        t = np.linspace(0, 10, 501)
        self._static_ax.plot(t, np.tan(t), ".")

        self._dynamic_ax = dynamic_canvas.figure.subplots()
        t = np.linspace(0, 10, 101)
        # Set up a Line2D.
        self._line, = self._dynamic_ax.plot(t, np.sin(t + time.time()))
        self._timer = dynamic_canvas.new_timer(50)
        self._timer.add_callback(self._update_canvas)
        self._timer.start()

    def _update_canvas(self):
        t = np.linspace(0, 10, 101)
        # Shift the sinusoid as a function of time.
        self._line.set_data(t, np.sin(t + time.time()))
        self._line.figure.canvas.draw()


if __name__ == "__main__":
    # Check whether there is already a running QApplication (e.g., if running
    # from an IDE).
    qapp = QtWidgets.QApplication.instance()
    if not qapp:
        qapp = QtWidgets.QApplication(sys.argv)

    app = ApplicationWindow()
    app.show()
    app.activateWindow()
    app.raise_()
    qapp.exec()
```

### Embedding in Tk

```
import tkinter

import numpy as np

# Implement the default Matplotlib key bindings.
from matplotlib.backend_bases import key_press_handler
from matplotlib.backends.backend_tkagg import (FigureCanvasTkAgg,
                                               NavigationToolbar2Tk)
from matplotlib.figure import Figure

root = tkinter.Tk()
root.wm_title("Embedding in Tk")

fig = Figure(figsize=(5, 4), dpi=100)
```

(continues on next page)

(continued from previous page)

```
t = np.arange(0, 3, .01)
ax = fig.add_subplot()
line, = ax.plot(t, 2 * np.sin(2 * np.pi * t))
ax.set_xlabel("time [s]")
ax.set_ylabel("f(t)")

canvas = FigureCanvasTkAgg(fig, master=root)  # A tk.DrawingArea.
canvas.draw()

# pack_toolbar=False will make it easier to use a layout manager later on.
toolbar = NavigationToolbar2Tk(canvas, root, pack_toolbar=False)
toolbar.update()

canvas.mpl_connect(
    "key_press_event", lambda event: print(f"you pressed {event.key}"))
canvas.mpl_connect("key_press_event", key_press_handler)

button_quit = tkinter.Button(master=root, text="Quit", command=root.destroy)


def update_frequency(new_val):
    # retrieve frequency
    f = float(new_val)

    # update data
    y = 2 * np.sin(2 * np.pi * f * t)
    line.set_data(t, y)

    # required to update canvas and attached toolbar!
    canvas.draw()


slider_update = tkinter.Scale(root, from_=1, to=5, orient=tkinter.HORIZONTAL,
                              command=update_frequency, label="Frequency [Hz]
 ↪")

# Packing order is important. Widgets are processed sequentially and if there
# is no space left, because the window is too small, they are not displayed.
# The canvas is rather flexible in its size, so we pack it last which makes
# sure the UI controls are displayed as long as possible.
button_quit.pack(side=tkinter.BOTTOM)
slider_update.pack(side=tkinter.BOTTOM)
toolbar.pack(side=tkinter.BOTTOM, fill=tkinter.X)
canvas.get_tk_widget().pack(side=tkinter.TOP, fill=tkinter.BOTH, expand=True)

tkinter.mainloop()
```

### Embedding in wx #2

An example of how to use wxagg in an application with the new toolbar - comment out the add_toolbar line for no toolbar.

```
import wx
import wx.lib.mixins.inspection as WIT

import numpy as np

from matplotlib.backends.backend_wxagg import FigureCanvasWxAgg as␣
 ↪FigureCanvas
from matplotlib.backends.backend_wxagg import \
    NavigationToolbar2WxAgg as NavigationToolbar
from matplotlib.figure import Figure


class CanvasFrame(wx.Frame):
    def __init__(self):
        super().__init__(None, -1, 'CanvasFrame', size=(550, 350))

        self.figure = Figure()
        self.axes = self.figure.add_subplot()
        t = np.arange(0.0, 3.0, 0.01)
        s = np.sin(2 * np.pi * t)

        self.axes.plot(t, s)
        self.canvas = FigureCanvas(self, -1, self.figure)

        self.sizer = wx.BoxSizer(wx.VERTICAL)
        self.sizer.Add(self.canvas, 1, wx.LEFT | wx.TOP | wx.EXPAND)
        self.SetSizer(self.sizer)
        self.Fit()

        self.add_toolbar()  # comment this out for no toolbar

    def add_toolbar(self):
        self.toolbar = NavigationToolbar(self.canvas)
        self.toolbar.Realize()
        # By adding toolbar in sizer, we are able to put it at the bottom
        # of the frame - so appearance is closer to GTK version.
        self.sizer.Add(self.toolbar, 0, wx.LEFT | wx.EXPAND)
        # update the axes menu on the toolbar
        self.toolbar.update()


# Alternatively you could use:
# class App(wx.App):
class App(WIT.InspectableApp):
    def OnInit(self):
        """Create the main window and insert the custom frame."""
        self.Init()
        frame = CanvasFrame()
```

(continues on next page)

(continued from previous page)

```
        frame.Show(True)

        return True


if __name__ == "__main__":
    app = App()
    app.MainLoop()
```

### Embedding in wx #3

Copyright (C) 2003-2004 Andrew Straw, Jeremy O'Donoghue and others

License: This work is licensed under the PSF. A copy should be included with this source code, and is also available at https://docs.python.org/3/license.html

This is yet another example of using matplotlib with wx. Hopefully this is pretty full-featured:

- both matplotlib toolbar and WX buttons manipulate plot
- full wxApp framework, including widget interaction
- XRC (XML wxWidgets resource) file to create GUI (made with XRCed)

This was derived from embedding_in_wx and dynamic_image_wxagg.

Thanks to matplotlib and wx teams for creating such great software!

```
import wx
import wx.xrc as xrc

import numpy as np

from matplotlib.backends.backend_wxagg import FigureCanvasWxAgg as␣
 ↪FigureCanvas
from matplotlib.backends.backend_wxagg import \
    NavigationToolbar2WxAgg as NavigationToolbar
import matplotlib.cbook as cbook
import matplotlib.cm as cm
from matplotlib.figure import Figure

ERR_TOL = 1e-5  # floating point slop for peak-detection


class PlotPanel(wx.Panel):
    def __init__(self, parent):
        super().__init__(parent, -1)

        self.fig = Figure((5, 4), 75)
        self.canvas = FigureCanvas(self, -1, self.fig)
        self.toolbar = NavigationToolbar(self.canvas)  # matplotlib toolbar
        self.toolbar.Realize()
```

(continues on next page)

(continued from previous page)

```
        # Now put all into a sizer
        sizer = wx.BoxSizer(wx.VERTICAL)
        # This way of adding to sizer allows resizing
        sizer.Add(self.canvas, 1, wx.LEFT | wx.TOP | wx.GROW)
        # Best to allow the toolbar to resize!
        sizer.Add(self.toolbar, 0, wx.GROW)
        self.SetSizer(sizer)
        self.Fit()

    def init_plot_data(self):
        ax = self.fig.add_subplot()

        x = np.arange(120.0) * 2 * np.pi / 60.0
        y = np.arange(100.0) * 2 * np.pi / 50.0
        self.x, self.y = np.meshgrid(x, y)
        z = np.sin(self.x) + np.cos(self.y)
        self.im = ax.imshow(z, cmap=cm.RdBu, origin='lower')

        zmax = np.max(z) - ERR_TOL
        ymax_i, xmax_i = np.nonzero(z >= zmax)
        if self.im.origin == 'upper':
            ymax_i = z.shape[0] - ymax_i
        self.lines = ax.plot(xmax_i, ymax_i, 'ko')

        self.toolbar.update()  # Not sure why this is needed - ADS

    def GetToolBar(self):
        # You will need to override GetToolBar if you are using an
        # unmanaged toolbar in your frame
        return self.toolbar

    def OnWhiz(self, event):
        self.x += np.pi / 15
        self.y += np.pi / 20
        z = np.sin(self.x) + np.cos(self.y)
        self.im.set_array(z)

        zmax = np.max(z) - ERR_TOL
        ymax_i, xmax_i = np.nonzero(z >= zmax)
        if self.im.origin == 'upper':
            ymax_i = z.shape[0] - ymax_i
        self.lines[0].set_data(xmax_i, ymax_i)

        self.canvas.draw()


class MyApp(wx.App):
    def OnInit(self):
        xrcfile = cbook.get_sample_data('embedding_in_wx3.xrc',
                                        asfileobj=False)
        print('loading', xrcfile)
```

(continues on next page)

(continued from previous page)

```
        self.res = xrc.XmlResource(xrcfile)

        # main frame and panel ---------

        self.frame = self.res.LoadFrame(None, "MainFrame")
        self.panel = xrc.XRCCTRL(self.frame, "MainPanel")

        # matplotlib panel -------------

        # container for matplotlib panel (I like to make a container
        # panel for our panel so I know where it'll go when in XRCed.)
        plot_container = xrc.XRCCTRL(self.frame, "plot_container_panel")
        sizer = wx.BoxSizer(wx.VERTICAL)

        # matplotlib panel itself
        self.plotpanel = PlotPanel(plot_container)
        self.plotpanel.init_plot_data()

        # wx boilerplate
        sizer.Add(self.plotpanel, 1, wx.EXPAND)
        plot_container.SetSizer(sizer)

        # whiz button ------------------
        whiz_button = xrc.XRCCTRL(self.frame, "whiz_button")
        whiz_button.Bind(wx.EVT_BUTTON, self.plotpanel.OnWhiz)

        # bang button ------------------
        bang_button = xrc.XRCCTRL(self.frame, "bang_button")
        bang_button.Bind(wx.EVT_BUTTON, self.OnBang)

        # final setup ------------------
        self.frame.Show()

        self.SetTopWindow(self.frame)

        return True

    def OnBang(self, event):
        bang_count = xrc.XRCCTRL(self.frame, "bang_count")
        bangs = bang_count.GetValue()
        bangs = int(bangs) + 1
        bang_count.SetValue(str(bangs))


if __name__ == '__main__':
    app = MyApp()
    app.MainLoop()
```

### Embedding in wx #4

An example of how to use wxagg in a wx application with a custom toolbar.

```
import wx

import numpy as np

from matplotlib.backends.backend_wxagg import FigureCanvasWxAgg as␣
 ↪FigureCanvas
from matplotlib.backends.backend_wxagg import \
    NavigationToolbar2WxAgg as NavigationToolbar
from matplotlib.figure import Figure


class MyNavigationToolbar(NavigationToolbar):
    """Extend the default wx toolbar with your own event handlers."""

    def __init__(self, canvas):
        super().__init__(canvas)
        # We use a stock wx bitmap, but you could also use your own image␣
 ↪file.
        bmp = wx.ArtProvider.GetBitmap(wx.ART_CROSS_MARK, wx.ART_TOOLBAR)
        tool = self.AddTool(wx.ID_ANY, 'Click me', bmp,
                            'Activate custom control')
        self.Bind(wx.EVT_TOOL, self._on_custom, id=tool.GetId())

    def _on_custom(self, event):
        # add some text to the Axes in a random location in axes coords with a
        # random color
        ax = self.canvas.figure.axes[0]
        x, y = np.random.rand(2)  # generate a random location
        rgb = np.random.rand(3)  # generate a random color
        ax.text(x, y, 'You clicked me', transform=ax.transAxes, color=rgb)
        self.canvas.draw()
        event.Skip()


class CanvasFrame(wx.Frame):
    def __init__(self):
        super().__init__(None, -1, 'CanvasFrame', size=(550, 350))

        self.figure = Figure(figsize=(5, 4), dpi=100)
        self.axes = self.figure.add_subplot()
        t = np.arange(0.0, 3.0, 0.01)
        s = np.sin(2 * np.pi * t)

        self.axes.plot(t, s)

        self.canvas = FigureCanvas(self, -1, self.figure)

        self.sizer = wx.BoxSizer(wx.VERTICAL)
        self.sizer.Add(self.canvas, 1, wx.TOP | wx.LEFT | wx.EXPAND)
```

(continues on next page)

(continued from previous page)

```
        self.toolbar = MyNavigationToolbar(self.canvas)
        self.toolbar.Realize()
        # By adding toolbar in sizer, we are able to put it at the bottom
        # of the frame - so appearance is closer to GTK version.
        self.sizer.Add(self.toolbar, 0, wx.LEFT | wx.EXPAND)

        # update the axes menu on the toolbar
        self.toolbar.update()
        self.SetSizer(self.sizer)
        self.Fit()


class App(wx.App):
    def OnInit(self):
        """Create the main window and insert the custom frame."""
        frame = CanvasFrame()
        frame.Show(True)
        return True


if __name__ == "__main__":
    app = App()
    app.MainLoop()
```

### Embedding in wx #5

```
import wx
import wx.lib.agw.aui as aui
import wx.lib.mixins.inspection as wit

from matplotlib.backends.backend_wxagg import FigureCanvasWxAgg as␣
 ↪FigureCanvas
from matplotlib.backends.backend_wxagg import \
    NavigationToolbar2WxAgg as NavigationToolbar
from matplotlib.figure import Figure


class Plot(wx.Panel):
    def __init__(self, parent, id=-1, dpi=None, **kwargs):
        super().__init__(parent, id=id, **kwargs)
        self.figure = Figure(dpi=dpi, figsize=(2, 2))
        self.canvas = FigureCanvas(self, -1, self.figure)
        self.toolbar = NavigationToolbar(self.canvas)
        self.toolbar.Realize()

        sizer = wx.BoxSizer(wx.VERTICAL)
        sizer.Add(self.canvas, 1, wx.EXPAND)
        sizer.Add(self.toolbar, 0, wx.LEFT | wx.EXPAND)
        self.SetSizer(sizer)
```

(continues on next page)

(continued from previous page)

```
class PlotNotebook(wx.Panel):
    def __init__(self, parent, id=-1):
        super().__init__(parent, id=id)
        self.nb = aui.AuiNotebook(self)
        sizer = wx.BoxSizer()
        sizer.Add(self.nb, 1, wx.EXPAND)
        self.SetSizer(sizer)

    def add(self, name="plot"):
        page = Plot(self.nb)
        self.nb.AddPage(page, name)
        return page.figure


def demo():
    # Alternatively you could use:
    # app = wx.App()
    # InspectableApp is a great debug tool, see:
    # http://wiki.wxpython.org/Widget%20Inspection%20Tool
    app = wit.InspectableApp()
    frame = wx.Frame(None, -1, 'Plotter')
    plotter = PlotNotebook(frame)
    axes1 = plotter.add('figure 1').add_subplot()
    axes1.plot([1, 2, 3], [2, 1, 4])
    axes2 = plotter.add('figure 2').add_subplot()
    axes2.plot([1, 2, 3, 4, 5], [2, 1, 4, 2, 3])
    frame.Show()
    app.MainLoop()

if __name__ == "__main__":
    demo()
```

## Embedding WebAgg

This example demonstrates how to embed Matplotlib WebAgg interactive plotting in your own web application and framework. It is not necessary to do all this if you merely want to display a plot in a browser or use Matplotlib's built-in Tornado-based server "on the side".

The framework being used must support web sockets.

```
import argparse
import io
import json
import mimetypes
from pathlib import Path
import signal
import socket
```

(continues on next page)

(continued from previous page)

```
try:
    import tornado
except ImportError as err:
    raise RuntimeError("This example requires tornado.") from err
import tornado.httpserver
import tornado.ioloop
import tornado.web
import tornado.websocket

import numpy as np

import matplotlib as mpl
from matplotlib.backends.backend_webagg import (
    FigureManagerWebAgg, new_figure_manager_given_figure)
from matplotlib.figure import Figure


def create_figure():
    """
    Creates a simple example figure.
    """
    fig = Figure()
    ax = fig.add_subplot()
    t = np.arange(0.0, 3.0, 0.01)
    s = np.sin(2 * np.pi * t)
    ax.plot(t, s)
    return fig


# The following is the content of the web page.  You would normally
# generate this using some sort of template facility in your web
# framework, but here we just use Python string formatting.
html_content = """<!DOCTYPE html>
<html lang="en">
  <head>
    <!-- TODO: There should be a way to include all of the required javascript
               and CSS so matplotlib can add to the set in the future if it
               needs to. -->
    <link rel="stylesheet" href="_static/css/page.css" type="text/css">
    <link rel="stylesheet" href="_static/css/boilerplate.css" type="text/css">
    <link rel="stylesheet" href="_static/css/fbm.css" type="text/css">
    <link rel="stylesheet" href="_static/css/mpl.css" type="text/css">
    <script src="mpl.js"></script>

    <script>
      /* This is a callback that is called when the user saves
         (downloads) a file.  Its purpose is really to map from a
         figure and file format to a url in the application. */
      function ondownload(figure, format) {
        window.open('download.' + format, '_blank');
      };
```

(continues on next page)

(continued from previous page)

```
      function ready(fn) {
        if (document.readyState != "loading") {
          fn();
        } else {
          document.addEventListener("DOMContentLoaded", fn);
        }
      }

      ready(
        function() {
          /* It is up to the application to provide a websocket that the␣
 ↪figure
             will use to communicate to the server.  This websocket object can
             also be a "fake" websocket that underneath multiplexes messages
             from multiple figures, if necessary. */
          var websocket_type = mpl.get_websocket_type();
          var websocket = new websocket_type("%(ws_uri)sws");

          // mpl.figure creates a new figure on the webpage.
          var fig = new mpl.figure(
              // A unique numeric identifier for the figure
              %(fig_id)s,
              // A websocket object (or something that behaves like one)
              websocket,
              // A function called when a file type is selected for download
              ondownload,
              // The HTML element in which to place the figure
              document.getElementById("figure"));
        }
      );
    </script>

    <title>matplotlib</title>
  </head>

  <body>
    <div id="figure">
    </div>
  </body>
</html>
"""


class MyApplication(tornado.web.Application):
    class MainPage(tornado.web.RequestHandler):
        """
        Serves the main HTML page.
        """

        def get(self):
            manager = self.application.manager
            ws_uri = f"ws://{self.request.host}/"
```

(continues on next page)

(continued from previous page)

```
            content = html_content % {
                "ws_uri": ws_uri, "fig_id": manager.num}
            self.write(content)

    class MplJs(tornado.web.RequestHandler):
        """
        Serves the generated matplotlib javascript file.  The content
        is dynamically generated based on which toolbar functions the
        user has defined.  Call `FigureManagerWebAgg` to get its
        content.
        """

        def get(self):
            self.set_header('Content-Type', 'application/javascript')
            js_content = FigureManagerWebAgg.get_javascript()

            self.write(js_content)

    class Download(tornado.web.RequestHandler):
        """
        Handles downloading of the figure in various file formats.
        """

        def get(self, fmt):
            manager = self.application.manager
            self.set_header(
                'Content-Type', mimetypes.types_map.get(fmt, 'binary'))
            buff = io.BytesIO()
            manager.canvas.figure.savefig(buff, format=fmt)
            self.write(buff.getvalue())

    class WebSocket(tornado.websocket.WebSocketHandler):
        """
        A websocket for interactive communication between the plot in
        the browser and the server.

        In addition to the methods required by tornado, it is required to
        have two callback methods:

            - ``send_json(json_content)`` is called by matplotlib when
              it needs to send json to the browser.  `json_content` is
              a JSON tree (Python dictionary), and it is the responsibility
              of this implementation to encode it as a string to send over
              the socket.

            - ``send_binary(blob)`` is called to send binary image data
              to the browser.
        """
        supports_binary = True

        def open(self):
            # Register the websocket with the FigureManager.
```

(continues on next page)

(continued from previous page)

```
            manager = self.application.manager
            manager.add_web_socket(self)
            if hasattr(self, 'set_nodelay'):
                self.set_nodelay(True)

        def on_close(self):
            # When the socket is closed, deregister the websocket with
            # the FigureManager.
            manager = self.application.manager
            manager.remove_web_socket(self)

        def on_message(self, message):
            # The 'supports_binary' message is relevant to the
            # websocket itself.  The other messages get passed along
            # to matplotlib as-is.

            # Every message has a "type" and a "figure_id".
            message = json.loads(message)
            if message['type'] == 'supports_binary':
                self.supports_binary = message['value']
            else:
                manager = self.application.manager
                manager.handle_json(message)

        def send_json(self, content):
            self.write_message(json.dumps(content))

        def send_binary(self, blob):
            if self.supports_binary:
                self.write_message(blob, binary=True)
            else:
                data_uri = ("" +
                            blob.encode('base64').replace('\n', ''))
                self.write_message(data_uri)

    def __init__(self, figure):
        self.figure = figure
        self.manager = new_figure_manager_given_figure(id(figure), figure)

        super().__init__([
            # Static files for the CSS and JS
            (r'/_static/(.*)',
             tornado.web.StaticFileHandler,
             {'path': FigureManagerWebAgg.get_static_file_path()}),

            # Static images for the toolbar
            (r'/_images/(.*)',
             tornado.web.StaticFileHandler,
             {'path': Path(mpl.get_data_path(), 'images')}),

            # The page that contains all of the pieces
            ('/', self.MainPage),
```

(continues on next page)

(continued from previous page)

```
            ('/mpl.js', self.MplJs),

            # Sends images and events to the browser, and receives
            # events from the browser
            ('/ws', self.WebSocket),

            # Handles the downloading (i.e., saving) of static images
            (r'/download.([a-z0-9.]+)', self.Download),
        ])


if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument('-p', '--port', type=int, default=8080,
                        help='Port to listen on (0 for a random port).')
    args = parser.parse_args()

    figure = create_figure()
    application = MyApplication(figure)

    http_server = tornado.httpserver.HTTPServer(application)
    sockets = tornado.netutil.bind_sockets(args.port, '')
    http_server.add_sockets(sockets)

    for s in sockets:
        addr, port = s.getsockname()[:2]
        if s.family is socket.AF_INET6:
            addr = f'[{addr}]'
        print(f"Listening on http://{addr}:{port}/")
    print("Press Ctrl+C to quit")

    ioloop = tornado.ioloop.IOLoop.instance()

    def shutdown():
        ioloop.stop()
        print("Server stopped")

    old_handler = signal.signal(
        signal.SIGINT,
        lambda sig, frame: ioloop.add_callback_from_signal(shutdown))

    try:
        ioloop.start()
    finally:
        signal.signal(signal.SIGINT, old_handler)
```

### Fourier Demo WX

```
import wx

import numpy as np

from matplotlib.backends.backend_wxagg import FigureCanvasWxAgg as␣
↪FigureCanvas
from matplotlib.figure import Figure


class Knob:
    """
    Knob - simple class with a "setKnob" method.
    A Knob instance is attached to a Param instance, e.g., param.attach(knob)
    Base class is for documentation purposes.
    """

    def setKnob(self, value):
        pass


class Param:
    """
    The idea of the "Param" class is that some parameter in the GUI may have
    several knobs that both control it and reflect the parameter's state, e.g.
    a slider, text, and dragging can all change the value of the frequency in
    the waveform of this example.
    The class allows a cleaner way to update/"feedback" to the other knobs␣
↪when
    one is being changed.  Also, this class handles min/max constraints for␣
↪all
    the knobs.
    Idea - knob list - in "set" method, knob object is passed as well
      - the other knobs in the knob list have a "set" method which gets
        called for the others.
    """

    def __init__(self, initialValue=None, minimum=0., maximum=1.):
        self.minimum = minimum
        self.maximum = maximum
        if initialValue != self.constrain(initialValue):
            raise ValueError('illegal initial value')
        self.value = initialValue
        self.knobs = []

    def attach(self, knob):
        self.knobs += [knob]

    def set(self, value, knob=None):
        self.value = value
        self.value = self.constrain(value)
```

(continues on next page)

(continued from previous page)

```
        for feedbackKnob in self.knobs:
            if feedbackKnob != knob:
                feedbackKnob.setKnob(self.value)
        return self.value

    def constrain(self, value):
        if value <= self.minimum:
            value = self.minimum
        if value >= self.maximum:
            value = self.maximum
        return value


class SliderGroup(Knob):
    def __init__(self, parent, label, param):
        self.sliderLabel = wx.StaticText(parent, label=label)
        self.sliderText = wx.TextCtrl(parent, -1, style=wx.TE_PROCESS_ENTER)
        self.slider = wx.Slider(parent, -1)
        # self.slider.SetMax(param.maximum*1000)
        self.slider.SetRange(0, int(param.maximum * 1000))
        self.setKnob(param.value)

        sizer = wx.BoxSizer(wx.HORIZONTAL)
        sizer.Add(self.sliderLabel, 0,
                  wx.EXPAND | wx.ALL,
                  border=2)
        sizer.Add(self.sliderText, 0,
                  wx.EXPAND | wx.ALL,
                  border=2)
        sizer.Add(self.slider, 1, wx.EXPAND)
        self.sizer = sizer

        self.slider.Bind(wx.EVT_SLIDER, self.sliderHandler)
        self.sliderText.Bind(wx.EVT_TEXT_ENTER, self.sliderTextHandler)

        self.param = param
        self.param.attach(self)

    def sliderHandler(self, event):
        value = event.GetInt() / 1000.
        self.param.set(value)

    def sliderTextHandler(self, event):
        value = float(self.sliderText.GetValue())
        self.param.set(value)

    def setKnob(self, value):
        self.sliderText.SetValue(f'{value:g}')
        self.slider.SetValue(int(value * 1000))


class FourierDemoFrame(wx.Frame):
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        panel = wx.Panel(self)

        # create the GUI elements
        self.createCanvas(panel)
        self.createSliders(panel)

        # place them in a sizer for the Layout
        sizer = wx.BoxSizer(wx.VERTICAL)
        sizer.Add(self.canvas, 1, wx.EXPAND)
        sizer.Add(self.frequencySliderGroup.sizer, 0,
                  wx.EXPAND | wx.ALL, border=5)
        sizer.Add(self.amplitudeSliderGroup.sizer, 0,
                  wx.EXPAND | wx.ALL, border=5)
        panel.SetSizer(sizer)

    def createCanvas(self, parent):
        self.lines = []
        self.figure = Figure()
        self.canvas = FigureCanvas(parent, -1, self.figure)
        self.canvas.callbacks.connect('button_press_event', self.mouseDown)
        self.canvas.callbacks.connect('motion_notify_event', self.mouseMotion)
        self.canvas.callbacks.connect('button_release_event', self.mouseUp)
        self.state = ''
        self.mouseInfo = (None, None, None, None)
        self.f0 = Param(2., minimum=0., maximum=6.)
        self.A = Param(1., minimum=0.01, maximum=2.)
        self.createPlots()

        # Not sure I like having two params attached to the same Knob,
        # but that is what we have here... it works but feels kludgy -
        # although maybe it's not too bad since the knob changes both params
        # at the same time (both f0 and A are affected during a drag)
        self.f0.attach(self)
        self.A.attach(self)

    def createSliders(self, panel):
        self.frequencySliderGroup = SliderGroup(
            panel,
            label='Frequency f0:',
            param=self.f0)
        self.amplitudeSliderGroup = SliderGroup(panel, label=' Amplitude a:',
                                                param=self.A)

    def mouseDown(self, event):
        if self.lines[0].contains(event)[0]:
            self.state = 'frequency'
        elif self.lines[1].contains(event)[0]:
            self.state = 'time'
        else:
            self.state = ''
```

(continues on next page)

(continued from previous page)

```
        self.mouseInfo = (event.xdata, event.ydata,
                          max(self.f0.value, .1),
                          self.A.value)

    def mouseMotion(self, event):
        if self.state == '':
            return
        x, y = event.xdata, event.ydata
        if x is None:  # outside the Axes
            return
        x0, y0, f0Init, AInit = self.mouseInfo
        self.A.set(AInit + (AInit * (y - y0) / y0), self)
        if self.state == 'frequency':
            self.f0.set(f0Init + (f0Init * (x - x0) / x0))
        elif self.state == 'time':
            if (x - x0) / x0 != -1.:
                self.f0.set(1. / (1. / f0Init + (1. / f0Init * (x - x0) /␣
↪x0)))

    def mouseUp(self, event):
        self.state = ''

    def createPlots(self):
        # This method creates the subplots, waveforms and labels.
        # Later, when the waveforms or sliders are dragged, only the
        # waveform data will be updated (not here, but below in setKnob).
        self.subplot1, self.subplot2 = self.figure.subplots(2)
        x1, y1, x2, y2 = self.compute(self.f0.value, self.A.value)
        color = (1., 0., 0.)
        self.lines += self.subplot1.plot(x1, y1, color=color, linewidth=2)
        self.lines += self.subplot2.plot(x2, y2, color=color, linewidth=2)
        # Set some plot attributes
        self.subplot1.set_title(
            "Click and drag waveforms to change frequency and amplitude",
            fontsize=12)
        self.subplot1.set_ylabel("Frequency Domain Waveform X(f)", fontsize=8)
        self.subplot1.set_xlabel("frequency f", fontsize=8)
        self.subplot2.set_ylabel("Time Domain Waveform x(t)", fontsize=8)
        self.subplot2.set_xlabel("time t", fontsize=8)
        self.subplot1.set_xlim([-6, 6])
        self.subplot1.set_ylim([0, 1])
        self.subplot2.set_xlim([-2, 2])
        self.subplot2.set_ylim([-2, 2])
        self.subplot1.text(0.05, .95,
                           r'$X(f) = \mathcal{F}\{x(t)\}$',
                           verticalalignment='top',
                           transform=self.subplot1.transAxes)
        self.subplot2.text(0.05, .95,
                           r'$x(t) = a \cdot \cos(2\pi f_0 t) e^{-\pi t^2}$',
                           verticalalignment='top',
                           transform=self.subplot2.transAxes)
```

(continues on next page)

(continued from previous page)

```
    def compute(self, f0, A):
        f = np.arange(-6., 6., 0.02)
        t = np.arange(-2., 2., 0.01)
        x = A * np.cos(2 * np.pi * f0 * t) * np.exp(-np.pi * t ** 2)
        X = A / 2 * \
            (np.exp(-np.pi * (f - f0) ** 2) + np.exp(-np.pi * (f + f0) ** 2))
        return f, X, t, x

    def setKnob(self, value):
        # Note, we ignore value arg here and just go by state of the params
        x1, y1, x2, y2 = self.compute(self.f0.value, self.A.value)
        # update the data of the two waveforms
        self.lines[0].set(xdata=x1, ydata=y1)
        self.lines[1].set(xdata=x2, ydata=y2)
        # make the canvas draw its contents again with the new data
        self.canvas.draw()


class App(wx.App):
    def OnInit(self):
        self.frame1 = FourierDemoFrame(parent=None, title="Fourier Demo",
                                       size=(640, 480))
        self.frame1.Show()
        return True


if __name__ == "__main__":
    app = App()
    app.MainLoop()
```

### GTK3 spreadsheet

Example of embedding Matplotlib in an application and interacting with a treeview to store data. Double-click on an entry to update plot data.

```
import gi

gi.require_version('Gtk', '3.0')
gi.require_version('Gdk', '3.0')
from gi.repository import Gdk, Gtk

from numpy.random import random

from matplotlib.backends.backend_gtk3agg import FigureCanvas  # or gtk3cairo.
from matplotlib.figure import Figure


class DataManager(Gtk.Window):
    num_rows, num_cols = 20, 10
```

(continues on next page)

(continued from previous page)

```
    data = random((num_rows, num_cols))

    def __init__(self):
        super().__init__()
        self.set_default_size(600, 600)
        self.connect('destroy', lambda win: Gtk.main_quit())

        self.set_title('GtkListStore demo')
        self.set_border_width(8)

        vbox = Gtk.VBox(homogeneous=False, spacing=8)
        self.add(vbox)

        label = Gtk.Label(label='Double click a row to plot the data')

        vbox.pack_start(label, False, False, 0)

        sw = Gtk.ScrolledWindow()
        sw.set_shadow_type(Gtk.ShadowType.ETCHED_IN)
        sw.set_policy(Gtk.PolicyType.NEVER, Gtk.PolicyType.AUTOMATIC)
        vbox.pack_start(sw, True, True, 0)

        model = self.create_model()

        self.treeview = Gtk.TreeView(model=model)

        # Matplotlib stuff
        fig = Figure(figsize=(6, 4))

        self.canvas = FigureCanvas(fig)  # a Gtk.DrawingArea
        vbox.pack_start(self.canvas, True, True, 0)
        ax = fig.add_subplot()
        self.line, = ax.plot(self.data[0, :], 'go')  # plot the first row

        self.treeview.connect('row-activated', self.plot_row)
        sw.add(self.treeview)

        self.add_columns()

        self.add_events(Gdk.EventMask.BUTTON_PRESS_MASK |
                        Gdk.EventMask.KEY_PRESS_MASK |
                        Gdk.EventMask.KEY_RELEASE_MASK)

    def plot_row(self, treeview, path, view_column):
        ind, = path  # get the index into data
        points = self.data[ind, :]
        self.line.set_ydata(points)
        self.canvas.draw()

    def add_columns(self):
        for i in range(self.num_cols):
            column = Gtk.TreeViewColumn(str(i), Gtk.CellRendererText(),␣
```

(continues on next page)

(continued from previous page)

```
 ↪text=i)
            self.treeview.append_column(column)

    def create_model(self):
        types = [float] * self.num_cols
        store = Gtk.ListStore(*types)
        for row in self.data:
            store.append(tuple(row))
        return store


manager = DataManager()
manager.show_all()
Gtk.main()
```

### GTK4 spreadsheet

Example of embedding Matplotlib in an application and interacting with a treeview to store data. Double-click on an entry to update plot data.

```
import gi

gi.require_version('Gtk', '4.0')
gi.require_version('Gdk', '4.0')
from gi.repository import Gtk

from numpy.random import random

from matplotlib.backends.backend_gtk4agg import FigureCanvas  # or gtk4cairo.
from matplotlib.figure import Figure


class DataManager(Gtk.ApplicationWindow):
    num_rows, num_cols = 20, 10

    data = random((num_rows, num_cols))

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.set_default_size(600, 600)

        self.set_title('GtkListStore demo')

        vbox = Gtk.Box(orientation=Gtk.Orientation.VERTICAL,␣
 ↪homogeneous=False,
                       spacing=8)
        self.set_child(vbox)

        label = Gtk.Label(label='Double click a row to plot the data')
        vbox.append(label)
```

(continues on next page)

(continued from previous page)

```
        sw = Gtk.ScrolledWindow()
        sw.set_has_frame(True)
        sw.set_policy(Gtk.PolicyType.NEVER, Gtk.PolicyType.AUTOMATIC)
        sw.set_hexpand(True)
        sw.set_vexpand(True)
        vbox.append(sw)

        model = self.create_model()
        self.treeview = Gtk.TreeView(model=model)
        self.treeview.connect('row-activated', self.plot_row)
        sw.set_child(self.treeview)

        # Matplotlib stuff
        fig = Figure(figsize=(6, 4), layout='constrained')

        self.canvas = FigureCanvas(fig)  # a Gtk.DrawingArea
        self.canvas.set_hexpand(True)
        self.canvas.set_vexpand(True)
        vbox.append(self.canvas)
        ax = fig.add_subplot()
        self.line, = ax.plot(self.data[0, :], 'go')  # plot the first row

        self.add_columns()

    def plot_row(self, treeview, path, view_column):
        ind, = path  # get the index into data
        points = self.data[ind, :]
        self.line.set_ydata(points)
        self.canvas.draw()

    def add_columns(self):
        for i in range(self.num_cols):
            column = Gtk.TreeViewColumn(str(i), Gtk.CellRendererText(),␣
↪text=i)
            self.treeview.append_column(column)

    def create_model(self):
        types = [float] * self.num_cols
        store = Gtk.ListStore(*types)
        for row in self.data:
            # Gtk.ListStore.append is broken in PyGObject, so insert manually.
            it = store.insert(-1)
            store.set(it, {i: val for i, val in enumerate(row)})
        return store


def on_activate(app):
    manager = DataManager(application=app)
    manager.show()
```

(continues on next page)

(continued from previous page)

```
app = Gtk.Application(application_id='org.matplotlib.examples.GTK4Spreadsheet
 ↪')
app.connect('activate', on_activate)
app.run()
```

### Display mathtext in WX

Demonstrates how to convert (math)text to a wx.Bitmap for display in various controls on wxPython.

```
from io import BytesIO

import wx

import numpy as np

from matplotlib.backends.backend_wx import NavigationToolbar2Wx
from matplotlib.backends.backend_wxagg import FigureCanvasWxAgg as␣
 ↪FigureCanvas
from matplotlib.figure import Figure

IS_WIN = 'wxMSW' in wx.PlatformInfo


def mathtext_to_wxbitmap(s):
    # We draw the text at position (0, 0) but then rely on
    # ``facecolor="none"`` and ``bbox_inches="tight", pad_inches=0`` to get a
    # transparent mask that is then loaded into a wx.Bitmap.
    fig = Figure(facecolor="none")
    text_color = (
        np.array(wx.SystemSettings.GetColour(wx.SYS_COLOUR_WINDOWTEXT)) / 255)
    fig.text(0, 0, s, fontsize=10, color=text_color)
    buf = BytesIO()
    fig.savefig(buf, format="png", dpi=150, bbox_inches="tight", pad_inches=0)
    s = buf.getvalue()
    return wx.Bitmap.NewFromPNGData(s, len(s))


functions = [
    (r'$\sin(2 \pi x)$', lambda x: np.sin(2*np.pi*x)),
    (r'$\frac{4}{3}\pi x^3$', lambda x: (4/3)*np.pi*x**3),
    (r'$\cos(2 \pi x)$', lambda x: np.cos(2*np.pi*x)),
    (r'$\log(x)$', lambda x: np.log(x))
]


class CanvasFrame(wx.Frame):
    def __init__(self, parent, title):
        super().__init__(parent, -1, title, size=(550, 350))

        self.figure = Figure()
```

(continues on next page)

(continued from previous page)

```
        self.axes = self.figure.add_subplot()

        self.canvas = FigureCanvas(self, -1, self.figure)

        self.change_plot(0)

        self.sizer = wx.BoxSizer(wx.VERTICAL)
        self.add_buttonbar()
        self.sizer.Add(self.canvas, 1, wx.LEFT | wx.TOP | wx.GROW)
        self.add_toolbar()  # comment this out for no toolbar

        menuBar = wx.MenuBar()

        # File Menu
        menu = wx.Menu()
        m_exit = menu.Append(
            wx.ID_EXIT, "E&xit\tAlt-X", "Exit this simple sample")
        menuBar.Append(menu, "&File")
        self.Bind(wx.EVT_MENU, self.OnClose, m_exit)

        if IS_WIN:
            # Equation Menu
            menu = wx.Menu()
            for i, (mt, func) in enumerate(functions):
                bm = mathtext_to_wxbitmap(mt)
                item = wx.MenuItem(menu, 1000 + i, " ")
                item.SetBitmap(bm)
                menu.Append(item)
                self.Bind(wx.EVT_MENU, self.OnChangePlot, item)
            menuBar.Append(menu, "&Functions")

        self.SetMenuBar(menuBar)

        self.SetSizer(self.sizer)
        self.Fit()

    def add_buttonbar(self):
        self.button_bar = wx.Panel(self)
        self.button_bar_sizer = wx.BoxSizer(wx.HORIZONTAL)
        self.sizer.Add(self.button_bar, 0, wx.LEFT | wx.TOP | wx.GROW)

        for i, (mt, func) in enumerate(functions):
            bm = mathtext_to_wxbitmap(mt)
            button = wx.BitmapButton(self.button_bar, 1000 + i, bm)
            self.button_bar_sizer.Add(button, 1, wx.GROW)
            self.Bind(wx.EVT_BUTTON, self.OnChangePlot, button)

        self.button_bar.SetSizer(self.button_bar_sizer)

    def add_toolbar(self):
        """Copied verbatim from embedding_wx2.py"""
        self.toolbar = NavigationToolbar2Wx(self.canvas)
```

(continues on next page)

(continued from previous page)

```
        self.toolbar.Realize()
        # By adding toolbar in sizer, we are able to put it at the bottom
        # of the frame - so appearance is closer to GTK version.
        self.sizer.Add(self.toolbar, 0, wx.LEFT | wx.EXPAND)
        # update the axes menu on the toolbar
        self.toolbar.update()

    def OnChangePlot(self, event):
        self.change_plot(event.GetId() - 1000)

    def change_plot(self, plot_number):
        t = np.arange(1.0, 3.0, 0.01)
        s = functions[plot_number][1](t)
        self.axes.clear()
        self.axes.plot(t, s)
        self.canvas.draw()

    def OnClose(self, event):
        self.Destroy()


class MyApp(wx.App):
    def OnInit(self):
        frame = CanvasFrame(None, "wxPython mathtext demo app")
        self.SetTopWindow(frame)
        frame.Show(True)
        return True


if __name__ == "__main__":
    app = MyApp()
    app.MainLoop()
```

### Matplotlib with Glade 3

```
from pathlib import Path

import gi

gi.require_version('Gtk', '3.0')
from gi.repository import Gtk

import numpy as np

from matplotlib.backends.backend_gtk3agg import \
    FigureCanvasGTK3Agg as FigureCanvas
from matplotlib.figure import Figure


class Window1Signals:
```

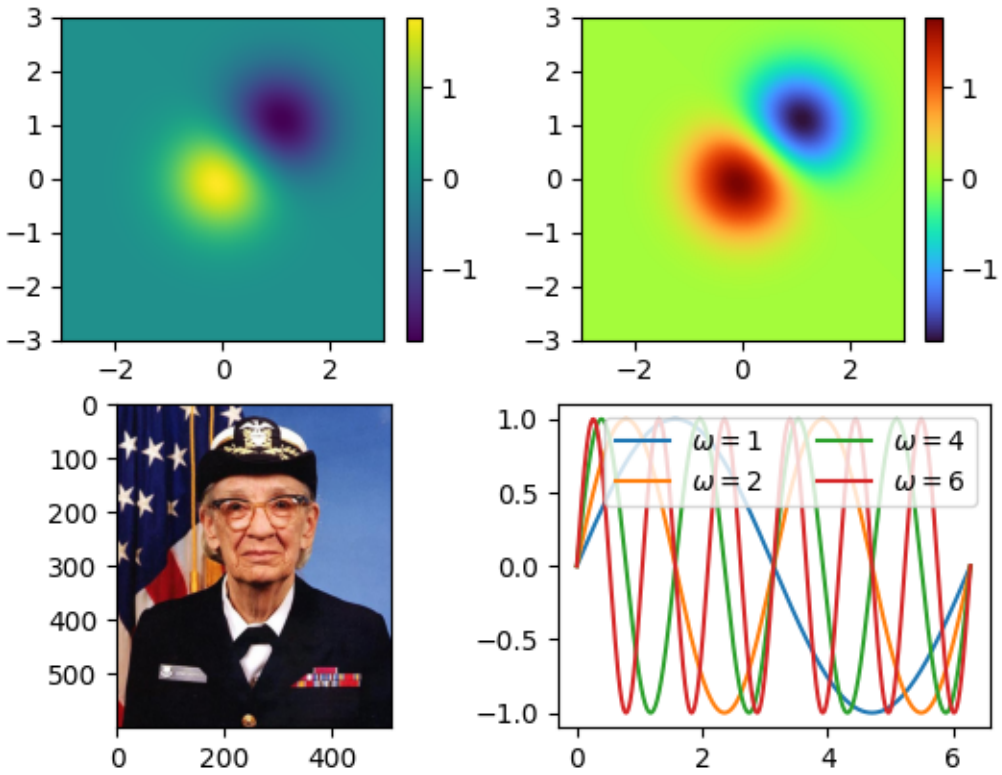
(continues on next page)

(continued from previous page)

```
    def on_window1_destroy(self, widget):
        Gtk.main_quit()


def main():
    builder = Gtk.Builder()
    builder.add_objects_from_file(
        str(Path(__file__).parent / "mpl_with_glade3.glade"),
        ("window1", ""))
    builder.connect_signals(Window1Signals())
    window = builder.get_object("window1")
    sw = builder.get_object("scrolledwindow1")

    # Start of Matplotlib specific code
    figure = Figure(figsize=(8, 6), dpi=71)
    axis = figure.add_subplot()
    t = np.arange(0.0, 3.0, 0.01)
    s = np.sin(2*np.pi*t)
    axis.plot(t, s)

    axis.set_xlabel('time [s]')
    axis.set_ylabel('voltage [V]')

    canvas = FigureCanvas(figure)  # a Gtk.DrawingArea
    canvas.set_size_request(800, 600)
    sw.add(canvas)
    # End of Matplotlib specific code

    window.show_all()
    Gtk.main()

if __name__ == "__main__":
    main()
```

### mplcvd -- an example of figure hook

To use this hook, ensure that this module is in your `PYTHONPATH`, and set `rcParams["figure.hooks"] = ["mplcvd:setup"]`. This hook depends on the `colorspacious` third-party module.




```
import functools
from pathlib import Path

import colorspacious

import numpy as np

_BUTTON_NAME = "Filter"
_BUTTON_HELP = "Simulate color vision deficiencies"
_MENU_ENTRIES = {
    "None": None,
    "Greyscale": "greyscale",
    "Deuteranopia": "deuteranomaly",
    "Protanopia": "protanomaly",
    "Tritanopia": "tritanomaly",
}


def _get_color_filter(name):
    """
    Given a color filter name, create a color filter function.

    Parameters
```

(continues on next page)

(continued from previous page)

```
    ----------
    name : str
        The color filter name, one of the following:

        - ``"none"``: ...
        - ``"greyscale"``: Convert the input to luminosity.
        - ``"deuteranopia"``: Simulate the most common form of red-green
          colorblindness.
        - ``"protanopia"``: Simulate a rarer form of red-green colorblindness.
        - ``"tritanopia"``: Simulate the rare form of blue-yellow
          colorblindness.

        Color conversions use `colorspacious`_.

    Returns
    -------
    callable
        A color filter function that has the form:

        def filter(input: np.ndarray[M, N, D])-> np.ndarray[M, N, D]

        where (M, N) are the image dimensions, and D is the color depth (3 for
        RGB, 4 for RGBA). Alpha is passed through unchanged and otherwise
        ignored.
    """
    if name not in _MENU_ENTRIES:
        raise ValueError(f"Unsupported filter name: {name!r}")
    name = _MENU_ENTRIES[name]

    if name is None:
        return None

    elif name == "greyscale":
        rgb_to_jch = colorspacious.cspace_converter("sRGB1", "JCh")
        jch_to_rgb = colorspacious.cspace_converter("JCh", "sRGB1")

        def convert(im):
            greyscale_JCh = rgb_to_jch(im)
            greyscale_JCh[..., 1] = 0
            im = jch_to_rgb(greyscale_JCh)
            return im

    else:
        cvd_space = {"name": "sRGB1+CVD", "cvd_type": name, "severity": 100}
        convert = colorspacious.cspace_converter(cvd_space, "sRGB1")

    def filter_func(im, dpi):
        alpha = None
        if im.shape[-1] == 4:
            im, alpha = im[..., :3], im[..., 3]
        im = convert(im)
        if alpha is not None:
```

(continues on next page)

(continued from previous page)

```
            im = np.dstack((im, alpha))
        return np.clip(im, 0, 1), 0, 0

    return filter_func


def _set_menu_entry(tb, name):
    tb.canvas.figure.set_agg_filter(_get_color_filter(name))
    tb.canvas.draw_idle()


def setup(figure):
    tb = figure.canvas.toolbar
    if tb is None:
        return
    for cls in type(tb).__mro__:
        pkg = cls.__module__.split(".")[0]
        if pkg != "matplotlib":
            break
    if pkg == "gi":
        _setup_gtk(tb)
    elif pkg in ("PyQt5", "PySide2", "PyQt6", "PySide6"):
        _setup_qt(tb)
    elif pkg == "tkinter":
        _setup_tk(tb)
    elif pkg == "wx":
        _setup_wx(tb)
    else:
        raise NotImplementedError("The current backend is not supported")


def _setup_gtk(tb):
    from gi.repository import Gio, GLib, Gtk

    for idx in range(tb.get_n_items()):
        children = tb.get_nth_item(idx).get_children()
        if children and isinstance(children[0], Gtk.Label):
            break

    toolitem = Gtk.SeparatorToolItem()
    tb.insert(toolitem, idx)

    image = Gtk.Image.new_from_gicon(
        Gio.Icon.new_for_string(
            str(Path(__file__).parent / "images/eye-symbolic.svg")),
        Gtk.IconSize.LARGE_TOOLBAR)

    # The type of menu is progressively downgraded depending on GTK version.
    if Gtk.check_version(3, 6, 0) is None:

        group = Gio.SimpleActionGroup.new()
        action = Gio.SimpleAction.new_stateful("cvdsim",
```

(continues on next page)

(continued from previous page)

```
                                                GLib.VariantType("s"),
                                                GLib.Variant("s", "none"))
        group.add_action(action)

        @functools.partial(action.connect, "activate")
        def set_filter(action, parameter):
            _set_menu_entry(tb, parameter.get_string())
            action.set_state(parameter)

        menu = Gio.Menu()
        for name in _MENU_ENTRIES:
            menu.append(name, f"local.cvdsim::{name}")

        button = Gtk.MenuButton.new()
        button.remove(button.get_children()[0])
        button.add(image)
        button.insert_action_group("local", group)
        button.set_menu_model(menu)
        button.get_style_context().add_class("flat")

        item = Gtk.ToolItem()
        item.add(button)
        tb.insert(item, idx + 1)

    else:

        menu = Gtk.Menu()
        group = []
        for name in _MENU_ENTRIES:
            item = Gtk.RadioMenuItem.new_with_label(group, name)
            item.set_active(name == "None")
            item.connect(
                "activate", lambda item: _set_menu_entry(tb, item.get_
↪label()))
            group.append(item)
            menu.append(item)
        menu.show_all()

        tbutton = Gtk.MenuToolButton.new(image, _BUTTON_NAME)
        tbutton.set_menu(menu)
        tb.insert(tbutton, idx + 1)

    tb.show_all()


def _setup_qt(tb):
    from matplotlib.backends.qt_compat import QtGui, QtWidgets

    menu = QtWidgets.QMenu()
    try:
        QActionGroup = QtGui.QActionGroup  # Qt6
    except AttributeError:
```

(continues on next page)

(continued from previous page)

```
        QActionGroup = QtWidgets.QActionGroup  # Qt5
    group = QActionGroup(menu)
    group.triggered.connect(lambda action: _set_menu_entry(tb, action.text()))

    for name in _MENU_ENTRIES:
        action = menu.addAction(name)
        action.setCheckable(True)
        action.setActionGroup(group)
        action.setChecked(name == "None")

    actions = tb.actions()
    before = next(
        (action for action in actions
         if isinstance(tb.widgetForAction(action), QtWidgets.QLabel)), None)

    tb.insertSeparator(before)
    button = QtWidgets.QToolButton()
    # FIXME: _icon needs public API.
    button.setIcon(tb._icon(str(Path(__file__).parent / "images/eye.png")))
    button.setText(_BUTTON_NAME)
    button.setToolTip(_BUTTON_HELP)
    button.setPopupMode(QtWidgets.QToolButton.ToolButtonPopupMode.
↪InstantPopup)
    button.setMenu(menu)
    tb.insertWidget(before, button)


def _setup_tk(tb):
    import tkinter as tk

    tb._Spacer()  # FIXME: _Spacer needs public API.

    button = tk.Menubutton(master=tb, relief="raised")
    button._image_file = str(Path(__file__).parent / "images/eye.png")
    # FIXME: _set_image_for_button needs public API (perhaps like _icon).
    tb._set_image_for_button(button)
    button.pack(side=tk.LEFT)

    menu = tk.Menu(master=button, tearoff=False)
    for name in _MENU_ENTRIES:
        menu.add("radiobutton", label=name,
                 command=lambda _name=name: _set_menu_entry(tb, _name))
    menu.invoke(0)
    button.config(menu=menu)


def _setup_wx(tb):
    import wx

    idx = next(idx for idx in range(tb.ToolsCount)
               if tb.GetToolByPos(idx).IsStretchableSpace())
    tb.InsertSeparator(idx)
```

(continues on next page)

(continued from previous page)

```
    tool = tb.InsertTool(
        idx + 1, -1, _BUTTON_NAME,
        # FIXME: _icon needs public API.
        tb._icon(str(Path(__file__).parent / "images/eye.png")),
        # FIXME: ITEM_DROPDOWN is not supported on macOS.
        kind=wx.ITEM_DROPDOWN, shortHelp=_BUTTON_HELP)

    menu = wx.Menu()
    for name in _MENU_ENTRIES:
        item = menu.AppendRadioItem(-1, name)
        menu.Bind(
            wx.EVT_MENU,
            lambda event, _name=name: _set_menu_entry(tb, _name),
            id=item.Id,
        )
    tb.SetDropdownMenu(tool.Id, menu)


if __name__ == '__main__':
    import matplotlib.pyplot as plt

    from matplotlib import cbook

    plt.rcParams['figure.hooks'].append('mplcvd:setup')

    fig, axd = plt.subplot_mosaic(
        [
            ['viridis', 'turbo'],
            ['photo', 'lines']
        ]
    )

    delta = 0.025
    x = y = np.arange(-3.0, 3.0, delta)
    X, Y = np.meshgrid(x, y)
    Z1 = np.exp(-X**2 - Y**2)
    Z2 = np.exp(-(X - 1)**2 - (Y - 1)**2)
    Z = (Z1 - Z2) * 2

    imv = axd['viridis'].imshow(
        Z, interpolation='bilinear',
        origin='lower', extent=[-3, 3, -3, 3],
        vmax=abs(Z).max(), vmin=-abs(Z).max()
    )
    fig.colorbar(imv)
    imt = axd['turbo'].imshow(
        Z, interpolation='bilinear', cmap='turbo',
        origin='lower', extent=[-3, 3, -3, 3],
        vmax=abs(Z).max(), vmin=-abs(Z).max()
    )
    fig.colorbar(imt)
```

(continues on next page)

(continued from previous page)

```
    # A sample image
    with cbook.get_sample_data('grace_hopper.jpg') as image_file:
        photo = plt.imread(image_file)
    axd['photo'].imshow(photo)

    th = np.linspace(0, 2*np.pi, 1024)
    for j in [1, 2, 4, 6]:
        axd['lines'].plot(th, np.sin(th * j), label=f'$\\omega={j}$')
    axd['lines'].legend(ncols=2, loc='upper right')
    plt.show()
```

### pyplot with GTK3

An example of how to use pyplot to manage your figure windows, but modify the GUI by accessing the underlying GTK widgets.

```
import matplotlib

matplotlib.use('GTK3Agg')  # or 'GTK3Cairo'
import gi

import matplotlib.pyplot as plt

gi.require_version('Gtk', '3.0')
from gi.repository import Gtk

fig, ax = plt.subplots()
ax.plot([1, 2, 3], 'ro-', label='easy as 1 2 3')
ax.plot([1, 4, 9], 'gs--', label='easy as 1 2 3 squared')
ax.legend()

manager = fig.canvas.manager
# you can access the window or vbox attributes this way
toolbar = manager.toolbar
vbox = manager.vbox

# now let's add a button to the toolbar
button = Gtk.Button(label='Click me')
button.show()
button.connect('clicked', lambda button: print('hi mom'))

toolitem = Gtk.ToolItem()
toolitem.show()
toolitem.set_tooltip_text('Click me for fun and profit')
toolitem.add(button)

pos = 8  # where to insert this in the toolbar
toolbar.insert(toolitem, pos)

# now let's add a widget to the vbox
```

(continues on next page)

(continued from previous page)

```
label = Gtk.Label()
label.set_markup('Drag mouse over axes for position')
label.show()
vbox.pack_start(label, False, False, 0)
vbox.reorder_child(toolbar, -1)


def update(event):
    if event.xdata is None:
        label.set_markup('Drag mouse over axes for position')
    else:
        label.set_markup(
            f'<span color="#ef0000">x,y=({event.xdata}, {event.ydata})</span>
 ↪')


fig.canvas.mpl_connect('motion_notify_event', update)

plt.show()
```

### pyplot with GTK4

An example of how to use pyplot to manage your figure windows, but modify the GUI by accessing the underlying GTK widgets.

```
import matplotlib

matplotlib.use('GTK4Agg')  # or 'GTK4Cairo'
import gi

import matplotlib.pyplot as plt

gi.require_version('Gtk', '4.0')
from gi.repository import Gtk

fig, ax = plt.subplots()
ax.plot([1, 2, 3], 'ro-', label='easy as 1 2 3')
ax.plot([1, 4, 9], 'gs--', label='easy as 1 2 3 squared')
ax.legend()

manager = fig.canvas.manager
# you can access the window or vbox attributes this way
toolbar = manager.toolbar
vbox = manager.vbox

# now let's add a button to the toolbar
button = Gtk.Button(label='Click me')
button.connect('clicked', lambda button: print('hi mom'))
button.set_tooltip_text('Click me for fun and profit')
toolbar.append(button)
```

(continues on next page)

(continued from previous page)

```
# now let's add a widget to the vbox
label = Gtk.Label()
label.set_markup('Drag mouse over axes for position')
vbox.insert_child_after(label, fig.canvas)


def update(event):
    if event.xdata is None:
        label.set_markup('Drag mouse over axes for position')
    else:
        label.set_markup(
            f'<span color="#ef0000">x,y=({event.xdata}, {event.ydata})</span>
 ↪')


fig.canvas.mpl_connect('motion_notify_event', update)

plt.show()
```

## SVG Histogram

Demonstrate how to create an interactive histogram, in which bars are hidden or shown by clicking on legend markers.

The interactivity is encoded in ecmascript (javascript) and inserted in the SVG code in a post-processing step. To render the image, open it in a web browser. SVG is supported in most web browsers used by Linux and macOS users. Windows IE9 supports SVG, but earlier versions do not.

## Notes

The matplotlib backend lets us assign ids to each object. This is the mechanism used here to relate matplotlib objects created in python and the corresponding SVG constructs that are parsed in the second step. While flexible, ids are cumbersome to use for large collection of objects. Two mechanisms could be used to simplify things:

- systematic grouping of objects into SVG <g> tags,
- assigning classes to each SVG object according to its origin.

For example, instead of modifying the properties of each individual bar, the bars from the `hist` function could either be grouped in a PatchCollection, or be assigned a class="hist_##" attribute.

CSS could also be used more extensively to replace repetitive markup throughout the generated SVG.

Author: david.huard@gmail.com

```
from io import BytesIO
import json
```

(continues on next page)

(continued from previous page)

```
import xml.etree.ElementTree as ET

import matplotlib.pyplot as plt
import numpy as np

plt.rcParams['svg.fonttype'] = 'none'

# Apparently, this `register_namespace` method is necessary to avoid garbling
# the XML namespace with ns0.
ET.register_namespace("", "http://www.w3.org/2000/svg")

# Fixing random state for reproducibility
np.random.seed(19680801)

# --- Create histogram, legend and title ---
plt.figure()
r = np.random.randn(100)
r1 = r + 1
labels = ['Rabbits', 'Frogs']
H = plt.hist([r, r1], label=labels)
containers = H[-1]
leg = plt.legend(frameon=False)
plt.title("From a web browser, click on the legend\n"
          "marker to toggle the corresponding histogram.")


# --- Add ids to the svg objects we'll modify

hist_patches = {}
for ic, c in enumerate(containers):
    hist_patches[f'hist_{ic}'] = []
    for il, element in enumerate(c):
        element.set_gid(f'hist_{ic}_patch_{il}')
        hist_patches[f'hist_{ic}'].append(f'hist_{ic}_patch_{il}')

# Set ids for the legend patches
for i, t in enumerate(leg.get_patches()):
    t.set_gid(f'leg_patch_{i}')

# Set ids for the text patches
for i, t in enumerate(leg.get_texts()):
    t.set_gid(f'leg_text_{i}')

# Save SVG in a fake file object.
f = BytesIO()
plt.savefig(f, format="svg")

# Create XML tree from the SVG file.
tree, xmlid = ET.XMLID(f.getvalue())


# --- Add interactivity ---
```

(continues on next page)

(continued from previous page)

```
# Add attributes to the patch objects.
for i, t in enumerate(leg.get_patches()):
    el = xmlid[f'leg_patch_{i}']
    el.set('cursor', 'pointer')
    el.set('onclick', "toggle_hist(this)")

# Add attributes to the text objects.
for i, t in enumerate(leg.get_texts()):
    el = xmlid[f'leg_text_{i}']
    el.set('cursor', 'pointer')
    el.set('onclick', "toggle_hist(this)")

# Create script defining the function `toggle_hist`.
# We create a global variable `container` that stores the patches id
# belonging to each histogram. Then a function "toggle_element" sets the
# visibility attribute of all patches of each histogram and the opacity
# of the marker itself.

script = """
<script type="text/ecmascript">
<![CDATA[
var container = %s

function toggle(oid, attribute, values) {
    /* Toggle the style attribute of an object between two values.

    Parameters
    ----------
    oid : str
      Object identifier.
    attribute : str
      Name of style attribute.
    values : [on state, off state]
      The two values that are switched between.
    */
    var obj = document.getElementById(oid);
    var a = obj.style[attribute];

    a = (a == values[0] || a == "") ? values[1] : values[0];
    obj.style[attribute] = a;
    }

function toggle_hist(obj) {

    var num = obj.id.slice(-1);

    toggle('leg_patch_' + num, 'opacity', [1, 0.3]);
    toggle('leg_text_' + num, 'opacity', [1, 0.5]);

    var names = container['hist_'+num]
```

(continues on next page)

(continued from previous page)

```
    for (var i=0; i < names.length; i++) {
        toggle(names[i], 'opacity', [1, 0])
    };
    }
]]>
</script>
""" % json.dumps(hist_patches)

# Add a transition effect
css = tree.find('.//{http://www.w3.org/2000/svg}style')
css.text = css.text + "g {-webkit-transition:opacity 0.4s ease-out;" + \
    "-moz-transition:opacity 0.4s ease-out;}"

# Insert the script and save to file.
tree.insert(0, ET.XML(script))

ET.ElementTree(tree).write("svg_histogram.svg")
```

### SVG Tooltip

This example shows how to create a tooltip that will show up when hovering over a matplotlib patch.

Although it is possible to create the tooltip from CSS or javascript, here we create it in matplotlib and simply toggle its visibility on when hovering over the patch. This approach provides total control over the tooltip placement and appearance, at the expense of more code up front.

The alternative approach would be to put the tooltip content in `title` attributes of SVG objects. Then, using an existing js/CSS library, it would be relatively straightforward to create the tooltip in the browser. The content would be dictated by the `title` attribute, and the appearance by the CSS.

**author**

David Huard

```
from io import BytesIO
import xml.etree.ElementTree as ET

import matplotlib.pyplot as plt

ET.register_namespace("", "http://www.w3.org/2000/svg")

fig, ax = plt.subplots()

# Create patches to which tooltips will be assigned.
rect1 = plt.Rectangle((10, -20), 10, 5, fc='blue')
rect2 = plt.Rectangle((-20, 15), 10, 5, fc='green')

shapes = [rect1, rect2]
labels = ['This is a blue rectangle.', 'This is a green rectangle']

for i, (item, label) in enumerate(zip(shapes, labels)):
```

(continues on next page)

(continued from previous page)

```
    patch = ax.add_patch(item)
    annotate = ax.annotate(labels[i], xy=item.get_xy(), xytext=(0, 0),
                           textcoords='offset points', color='w', ha='center',
                           fontsize=8, bbox=dict(boxstyle='round, pad=.5',
                                                 fc=(.1, .1, .1, .92),
                                                 ec=(1., 1., 1.), lw=1,
                                                 zorder=1))

    ax.add_patch(patch)
    patch.set_gid(f'mypatch_{i:03d}')
    annotate.set_gid(f'mytooltip_{i:03d}')

# Save the figure in a fake file object
ax.set_xlim(-30, 30)
ax.set_ylim(-30, 30)
ax.set_aspect('equal')

f = BytesIO()
plt.savefig(f, format="svg")

# --- Add interactivity ---

# Create XML tree from the SVG file.
tree, xmlid = ET.XMLID(f.getvalue())
tree.set('onload', 'init(event)')

for i in shapes:
    # Get the index of the shape
    index = shapes.index(i)
    # Hide the tooltips
    tooltip = xmlid[f'mytooltip_{index:03d}']
    tooltip.set('visibility', 'hidden')
    # Assign onmouseover and onmouseout callbacks to patches.
    mypatch = xmlid[f'mypatch_{index:03d}']
    mypatch.set('onmouseover', "ShowTooltip(this)")
    mypatch.set('onmouseout', "HideTooltip(this)")

# This is the script defining the ShowTooltip and HideTooltip functions.
script = """
    <script type="text/ecmascript">
    <![CDATA[

    function init(event) {
        if ( window.svgDocument == null ) {
            svgDocument = event.target.ownerDocument;
            }
        }

    function ShowTooltip(obj) {
        var cur = obj.id.split("_")[1];
        var tip = svgDocument.getElementById('mytooltip_' + cur);
        tip.setAttribute('visibility', "visible")
```

(continues on next page)

(continued from previous page)

```
        }

    function HideTooltip(obj) {
        var cur = obj.id.split("_")[1];
        var tip = svgDocument.getElementById('mytooltip_' + cur);
        tip.setAttribute('visibility', "hidden")
        }

    ]]>
    </script>
    """

# Insert the script at the top of the file and save it.
tree.insert(0, ET.XML(script))
ET.ElementTree(tree).write('svg_tooltip.svg')
```

### Tool Manager

This example demonstrates how to

- modify the Toolbar
- create tools
- add tools
- remove tools

using *matplotlib.backend_managers.ToolManager*.

```
import matplotlib.pyplot as plt

from matplotlib.backend_tools import ToolBase, ToolToggleBase

plt.rcParams['toolbar'] = 'toolmanager'


class ListTools(ToolBase):
    """List all the tools controlled by the `ToolManager`."""
    default_keymap = 'm'  # keyboard shortcut
    description = 'List Tools'

    def trigger(self, *args, **kwargs):
        print('_' * 80)
        fmt_tool = "{:12} {:45} {}".format
        print(fmt_tool('Name (id)', 'Tool description', 'Keymap'))
        print('-' * 80)
        tools = self.toolmanager.tools
        for name in sorted(tools):
            if not tools[name].description:
                continue
```

(continues on next page)

(continued from previous page)

```
            keys = ', '.join(sorted(self.toolmanager.get_tool_keymap(name)))
            print(fmt_tool(name, tools[name].description, keys))
        print('_' * 80)
        fmt_active_toggle = "{!s:12} {!s:45}".format
        print("Active Toggle tools")
        print(fmt_active_toggle("Group", "Active"))
        print('-' * 80)
        for group, active in self.toolmanager.active_toggle.items():
            print(fmt_active_toggle(group, active))


class GroupHideTool(ToolToggleBase):
    """Show lines with a given gid."""
    default_keymap = 'S'
    description = 'Show by gid'
    default_toggled = True

    def __init__(self, *args, gid, **kwargs):
        self.gid = gid
        super().__init__(*args, **kwargs)

    def enable(self, *args):
        self.set_lines_visibility(True)

    def disable(self, *args):
        self.set_lines_visibility(False)

    def set_lines_visibility(self, state):
        for ax in self.figure.get_axes():
            for line in ax.get_lines():
                if line.get_gid() == self.gid:
                    line.set_visible(state)
        self.figure.canvas.draw()


fig = plt.figure()
plt.plot([1, 2, 3], gid='mygroup')
plt.plot([2, 3, 4], gid='unknown')
plt.plot([3, 2, 1], gid='mygroup')

# Add the custom tools that we created
fig.canvas.manager.toolmanager.add_tool('List', ListTools)
fig.canvas.manager.toolmanager.add_tool('Show', GroupHideTool, gid='mygroup')

# Add an existing tool to new group `foo`.
# It can be added as many times as we want
fig.canvas.manager.toolbar.add_tool('zoom', 'foo')

# Remove the forward button
fig.canvas.manager.toolmanager.remove_tool('forward')

# To add a custom tool to the toolbar at specific location inside
```

(continues on next page)

(continued from previous page)

```
# the navigation group
fig.canvas.manager.toolbar.add_tool('Show', 'navigation', 1)

plt.show()
```

### Embedding in a web application server (Flask)

When using Matplotlib in a web server it is strongly recommended to not use pyplot (pyplot maintains references to the opened figures to make *show* work, but this will cause memory leaks unless the figures are properly closed).

Since Matplotlib 3.1, one can directly create figures using the *Figure* constructor and save them to in-memory buffers. In older versions, it was necessary to explicitly instantiate an Agg canvas (see e.g. *CanvasAgg demo*).

The following example uses Flask, but other frameworks work similarly:

```
import base64
from io import BytesIO

from flask import Flask

from matplotlib.figure import Figure

app = Flask(__name__)


@app.route("/")
def hello():
    # Generate the figure **without using pyplot**.
    fig = Figure()
    ax = fig.subplots()
    ax.plot([1, 2])
    # Save it to a temporary buffer.
    buf = BytesIO()
    fig.savefig(buf, format="png")
    # Embed the result in the html output.
    data = base64.b64encode(buf.getbuffer()).decode("ascii")
    return f"<img src='{data}'/>"
```

Since the above code is a Flask application, it should be run using the flask command-line tool Assuming that the working directory contains this script:

Unix-like systems

```
FLASK_APP=web_application_server_sgskip flask run
```

Windows

```
set FLASK_APP=web_application_server_sgskip
flask run
```

### Clickable images for HTML

Andrew Dalke of Dalke Scientific has written a nice article on how to make html click maps with Matplotlib agg PNGs. We would also like to add this functionality to SVG. If you are interested in contributing to these efforts that would be great.

### Adding a cursor in WX

Example to draw a cursor and report the data coords in wx.

```
import wx

import numpy as np

from matplotlib.backends.backend_wx import NavigationToolbar2Wx
from matplotlib.backends.backend_wxagg import FigureCanvasWxAgg as␣
 ↪FigureCanvas
from matplotlib.figure import Figure


class CanvasFrame(wx.Frame):
    def __init__(self, ):
        super().__init__(None, -1, 'CanvasFrame', size=(550, 350))

        self.figure = Figure()
        self.axes = self.figure.add_subplot()
        t = np.arange(0.0, 3.0, 0.01)
        s = np.sin(2*np.pi*t)

        self.axes.plot(t, s)
        self.axes.set_xlabel('t')
        self.axes.set_ylabel('sin(t)')
        self.figure_canvas = FigureCanvas(self, -1, self.figure)

        # Note that event is a MplEvent
        self.figure_canvas.mpl_connect(
            'motion_notify_event', self.UpdateStatusBar)
        self.figure_canvas.Bind(wx.EVT_ENTER_WINDOW, self.ChangeCursor)

        self.sizer = wx.BoxSizer(wx.VERTICAL)
        self.sizer.Add(self.figure_canvas, 1, wx.LEFT | wx.TOP | wx.GROW)
        self.SetSizer(self.sizer)
        self.Fit()

        self.statusBar = wx.StatusBar(self, -1)
        self.SetStatusBar(self.statusBar)
```

(continues on next page)

(continued from previous page)

```
        self.toolbar = NavigationToolbar2Wx(self.figure_canvas)
        self.sizer.Add(self.toolbar, 0, wx.LEFT | wx.EXPAND)
        self.toolbar.Show()

    def ChangeCursor(self, event):
        self.figure_canvas.SetCursor(wx.Cursor(wx.CURSOR_BULLSEYE))

    def UpdateStatusBar(self, event):
        if event.inaxes:
            self.statusBar.SetStatusText(f"x={event.xdata}  y={event.ydata}")


class App(wx.App):
    def OnInit(self):
        """Create the main window and insert the custom frame."""
        frame = CanvasFrame()
        self.SetTopWindow(frame)
        frame.Show(True)
        return True


if __name__ == '__main__':
    app = App()
    app.MainLoop()
```

### 6.25.24 Widgets

Examples of how to write primitive, but GUI agnostic, widgets in matplotlib

#### Annotated cursor

Display a data cursor including a text box, which shows the plot point close to the mouse pointer.

The new cursor inherits from *Cursor* and demonstrates the creation of new widgets and their event callbacks.

See also the *cross hair cursor*, which implements a cursor tracking the plotted data, but without using inheritance and without displaying the currently tracked coordinates.

---

**Note:** The figure related to this example does not show the cursor, because that figure is automatically created in a build queue, where the first mouse movement, which triggers the cursor creation, is missing.

---

```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.backend_bases import MouseEvent
```

(continues on next page)

(continued from previous page)

```
from matplotlib.widgets import Cursor


class AnnotatedCursor(Cursor):
    """
    A crosshair cursor like `~matplotlib.widgets.Cursor` with a text showing \
    the current coordinates.

    For the cursor to remain responsive you must keep a reference to it.
    The data of the axis specified as *dataaxis* must be in ascending
    order. Otherwise, the `numpy.searchsorted` call might fail and the text
    disappears. You can satisfy the requirement by sorting the data you plot.
    Usually the data is already sorted (if it was created e.g. using
    `numpy.linspace`), but e.g. scatter plots might cause this problem.
    The cursor sticks to the plotted line.

    Parameters
    ----------
    line : `matplotlib.lines.Line2D`
        The plot line from which the data coordinates are displayed.

    numberformat : `python format string <https://docs.python.org/3/\
    library/string.html#formatstrings>`_, optional, default: "{0:.4g};{1:.4g}"
        The displayed text is created by calling *format()* on this string
        with the two coordinates.

    offset : (float, float) default: (5, 5)
        The offset in display (pixel) coordinates of the text position
        relative to the cross-hair.

    dataaxis : {"x", "y"}, optional, default: "x"
        If "x" is specified, the vertical cursor line sticks to the mouse
        pointer. The horizontal cursor line sticks to *line*
        at that x value. The text shows the data coordinates of *line*
        at the pointed x value. If you specify "y", it works in the opposite
        manner. But: For the "y" value, where the mouse points to, there might
        be multiple matching x values, if the plotted function is not␣
↪biunique.
        Cursor and text coordinate will always refer to only one x value.
        So if you use the parameter value "y", ensure that your function is
        biunique.

    Other Parameters
    ----------------
    textprops : `matplotlib.text` properties as dictionary
        Specifies the appearance of the rendered text object.

    **cursorargs : `matplotlib.widgets.Cursor` properties
        Arguments passed to the internal `~matplotlib.widgets.Cursor`␣
↪instance.
        The `matplotlib.axes.Axes` argument is mandatory! The parameter
        *useblit* can be set to *True* in order to achieve faster rendering.
```

(continues on next page)

(continued from previous page)

```
    """

    def __init__(self, line, numberformat="{0:.4g};{1:.4g}", offset=(5, 5),
                 dataaxis='x', textprops=None, **cursorargs):
        if textprops is None:
            textprops = {}
        # The line object, for which the coordinates are displayed
        self.line = line
        # The format string, on which .format() is called for creating the␣
↪text
        self.numberformat = numberformat
        # Text position offset
        self.offset = np.array(offset)
        # The axis in which the cursor position is looked up
        self.dataaxis = dataaxis

        # First call baseclass constructor.
        # Draws cursor and remembers background for blitting.
        # Saves ax as class attribute.
        super().__init__(**cursorargs)

        # Default value for position of text.
        self.set_position(self.line.get_xdata()[0], self.line.get_ydata()[0])
        # Create invisible animated text
        self.text = self.ax.text(
            self.ax.get_xbound()[0],
            self.ax.get_ybound()[0],
            "0, 0",
            animated=bool(self.useblit),
            visible=False, **textprops)
        # The position at which the cursor was last drawn
        self.lastdrawnplotpoint = None

    def onmove(self, event):
        """
        Overridden draw callback for cursor. Called when moving the mouse.
        """

        # Leave method under the same conditions as in overridden method
        if self.ignore(event):
            self.lastdrawnplotpoint = None
            return
        if not self.canvas.widgetlock.available(self):
            self.lastdrawnplotpoint = None
            return

        # If the mouse left drawable area, we now make the text invisible.
        # Baseclass will redraw complete canvas after, which makes both text
        # and cursor disappear.
        if event.inaxes != self.ax:
            self.lastdrawnplotpoint = None
```

(continues on next page)

(continued from previous page)

```
            self.text.set_visible(False)
            super().onmove(event)
            return

        # Get the coordinates, which should be displayed as text,
        # if the event coordinates are valid.
        plotpoint = None
        if event.xdata is not None and event.ydata is not None:
            # Get plot point related to current x position.
            # These coordinates are displayed in text.
            plotpoint = self.set_position(event.xdata, event.ydata)
            # Modify event, such that the cursor is displayed on the
            # plotted line, not at the mouse pointer,
            # if the returned plot point is valid
            if plotpoint is not None:
                event.xdata = plotpoint[0]
                event.ydata = plotpoint[1]

        # If the plotpoint is given, compare to last drawn plotpoint and
        # return if they are the same.
        # Skip even the call of the base class, because this would restore the
        # background, draw the cursor lines and would leave us the job to
        # re-draw the text.
        if plotpoint is not None and plotpoint == self.lastdrawnplotpoint:
            return

        # Baseclass redraws canvas and cursor. Due to blitting,
        # the added text is removed in this call, because the
        # background is redrawn.
        super().onmove(event)

        # Check if the display of text is still necessary.
        # If not, just return.
        # This behaviour is also cloned from the base class.
        if not self.get_active() or not self.visible:
            return

        # Draw the widget, if event coordinates are valid.
        if plotpoint is not None:
            # Update position and displayed text.
            # Position: Where the event occurred.
            # Text: Determined by set_position() method earlier
            # Position is transformed to pixel coordinates,
            # an offset is added there and this is transformed back.
            temp = [event.xdata, event.ydata]
            temp = self.ax.transData.transform(temp)
            temp = temp + self.offset
            temp = self.ax.transData.inverted().transform(temp)
            self.text.set_position(temp)
            self.text.set_text(self.numberformat.format(*plotpoint))
            self.text.set_visible(self.visible)
```

(continues on next page)

(continued from previous page)

```
            # Tell base class, that we have drawn something.
            # Baseclass needs to know, that it needs to restore a clean
            # background, if the cursor leaves our figure context.
            self.needclear = True

            # Remember the recently drawn cursor position, so events for the
            # same position (mouse moves slightly between two plot points)
            # can be skipped
            self.lastdrawnplotpoint = plotpoint
        # otherwise, make text invisible
        else:
            self.text.set_visible(False)

        # Draw changes. Cannot use _update method of baseclass,
        # because it would first restore the background, which
        # is done already and is not necessary.
        if self.useblit:
            self.ax.draw_artist(self.text)
            self.canvas.blit(self.ax.bbox)
        else:
            # If blitting is deactivated, the overridden _update call made
            # by the base class immediately returned.
            # We still have to draw the changes.
            self.canvas.draw_idle()

    def set_position(self, xpos, ypos):
        """
        Finds the coordinates, which have to be shown in text.

        The behaviour depends on the *dataaxis* attribute. Function looks
        up the matching plot coordinate for the given mouse position.

        Parameters
        ----------
        xpos : float
            The current x position of the cursor in data coordinates.
            Important if *dataaxis* is set to 'x'.
        ypos : float
            The current y position of the cursor in data coordinates.
            Important if *dataaxis* is set to 'y'.

        Returns
        -------
        ret : {2D array-like, None}
            The coordinates which should be displayed.
            *None* is the fallback value.
        """

        # Get plot line data
        xdata = self.line.get_xdata()
        ydata = self.line.get_ydata()
```

(continues on next page)

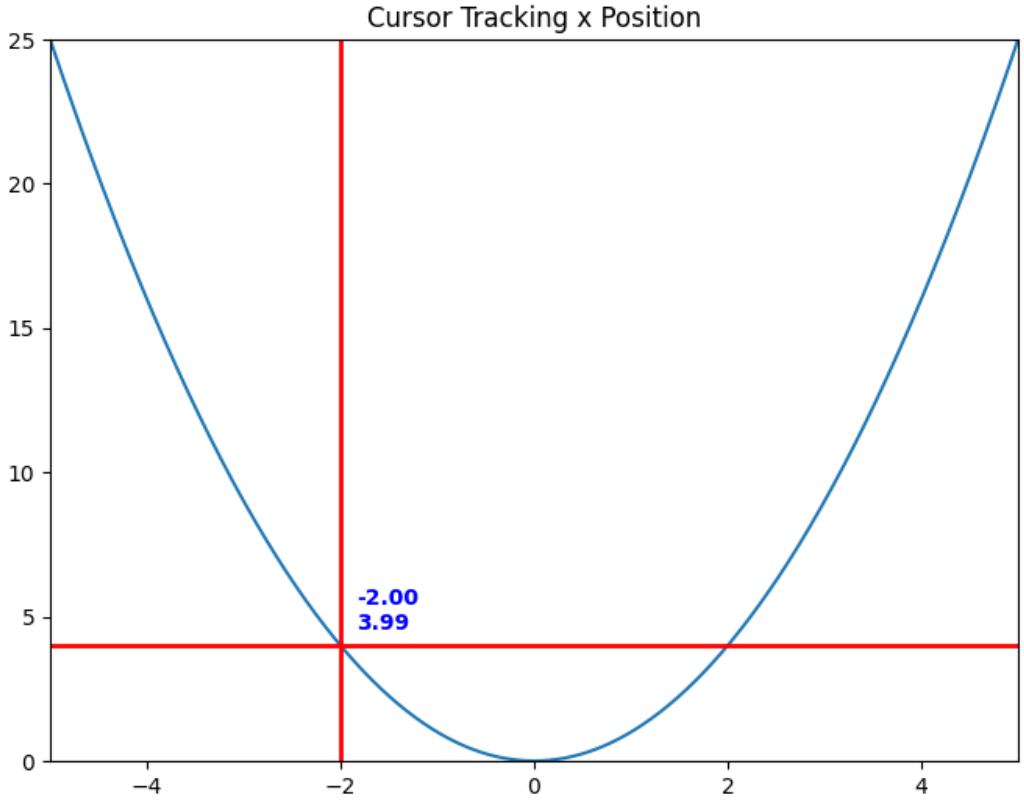
(continued from previous page)

```
        # The dataaxis attribute decides, in which axis we look up which␣
↪cursor
        # coordinate.
        if self.dataaxis == 'x':
            pos = xpos
            data = xdata
            lim = self.ax.get_xlim()
        elif self.dataaxis == 'y':
            pos = ypos
            data = ydata
            lim = self.ax.get_ylim()
        else:
            raise ValueError(f"The data axis specifier {self.dataaxis} should
↪"
                             f"be 'x' or 'y'")

        # If position is valid and in valid plot data range.
        if pos is not None and lim[0] <= pos <= lim[-1]:
            # Find closest x value in sorted x vector.
            # This requires the plotted data to be sorted.
            index = np.searchsorted(data, pos)
            # Return none, if this index is out of range.
            if index < 0 or index >= len(data):
                return None
            # Return plot point as tuple.
            return (xdata[index], ydata[index])

        # Return none if there is no good related point for this x position.
        return None

    def clear(self, event):
        """
        Overridden clear callback for cursor, called before drawing the␣
↪figure.
        """

        # The base class saves the clean background for blitting.
        # Text and cursor are invisible,
        # until the first mouse move event occurs.
        super().clear(event)
        if self.ignore(event):
            return
        self.text.set_visible(False)

    def _update(self):
        """
        Overridden method for either blitting or drawing the widget canvas.

        Passes call to base class if blitting is activated, only.
        In other cases, one draw_idle call is enough, which is placed
        explicitly in this class (see *onmove()*).
        In that case, `~matplotlib.widgets.Cursor` is not supposed to draw
```

(continues on next page)

(continued from previous page)

```
        something using this method.
        """

        if self.useblit:
            super()._update()


fig, ax = plt.subplots(figsize=(8, 6))
ax.set_title("Cursor Tracking x Position")

x = np.linspace(-5, 5, 1000)
y = x**2

line, = ax.plot(x, y)
ax.set_xlim(-5, 5)
ax.set_ylim(0, 25)

# A minimum call
# Set useblit=True on most backends for enhanced performance
# and pass the ax parameter to the Cursor base class.
# cursor = AnnotatedCursor(line=lin[0], ax=ax, useblit=True)

# A more advanced call. Properties for text and lines are passed.
# Watch the passed color names and the color of cursor line and text, to
# relate the passed options to graphical elements.
# The dataaxis parameter is still the default.
cursor = AnnotatedCursor(
    line=line,
    numberformat="{0:.2f}\n{1:.2f}",
    dataaxis='x', offset=[10, 10],
    textprops={'color': 'blue', 'fontweight': 'bold'},
    ax=ax,
    useblit=True,
    color='red',
    linewidth=2)

# Simulate a mouse move to (-2, 10), needed for online docs
t = ax.transData
MouseEvent(
    "motion_notify_event", ax.figure.canvas, *t.transform((-2, 10))
)._process()

plt.show()
```




### Trouble with non-biunique functions

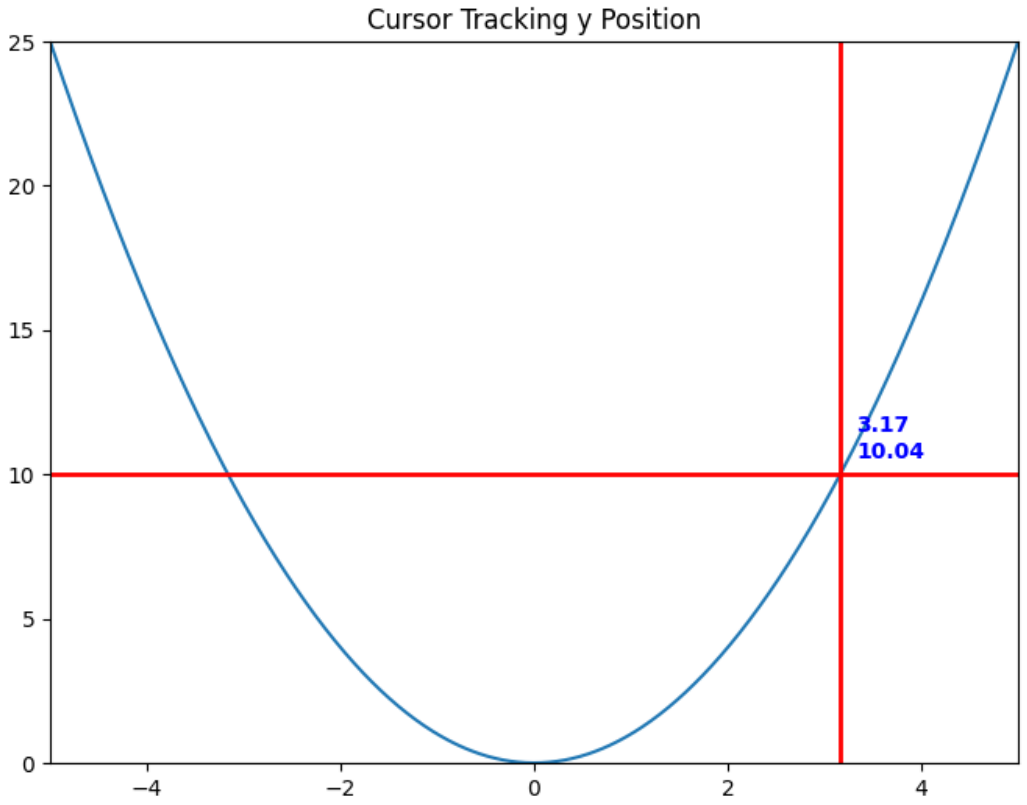
A call demonstrating problems with the *dataaxis=y* parameter. The text now looks up the matching x value for the current cursor y position instead of vice versa. Hover your cursor to y=4. There are two x values producing this y value: -2 and 2. The function is only unique, but not biunique. Only one value is shown in the text.

```
fig, ax = plt.subplots(figsize=(8, 6))
ax.set_title("Cursor Tracking y Position")

line, = ax.plot(x, y)
ax.set_xlim(-5, 5)
ax.set_ylim(0, 25)

cursor = AnnotatedCursor(
    line=line,
    numberformat="{0:.2f}\n{1:.2f}",
    dataaxis='y', offset=[10, 10],
    textprops={'color': 'blue', 'fontweight': 'bold'},
    ax=ax,
    useblit=True,
```

(continues on next page)

(continued from previous page)

```
    color='red', linewidth=2)

# Simulate a mouse move to (-2, 10), needed for online docs
t = ax.transData
MouseEvent(
    "motion_notify_event", ax.figure.canvas, *t.transform((-2, 10))
)._process()

plt.show()
```




## Buttons

Constructing a simple button GUI to modify a sine wave.

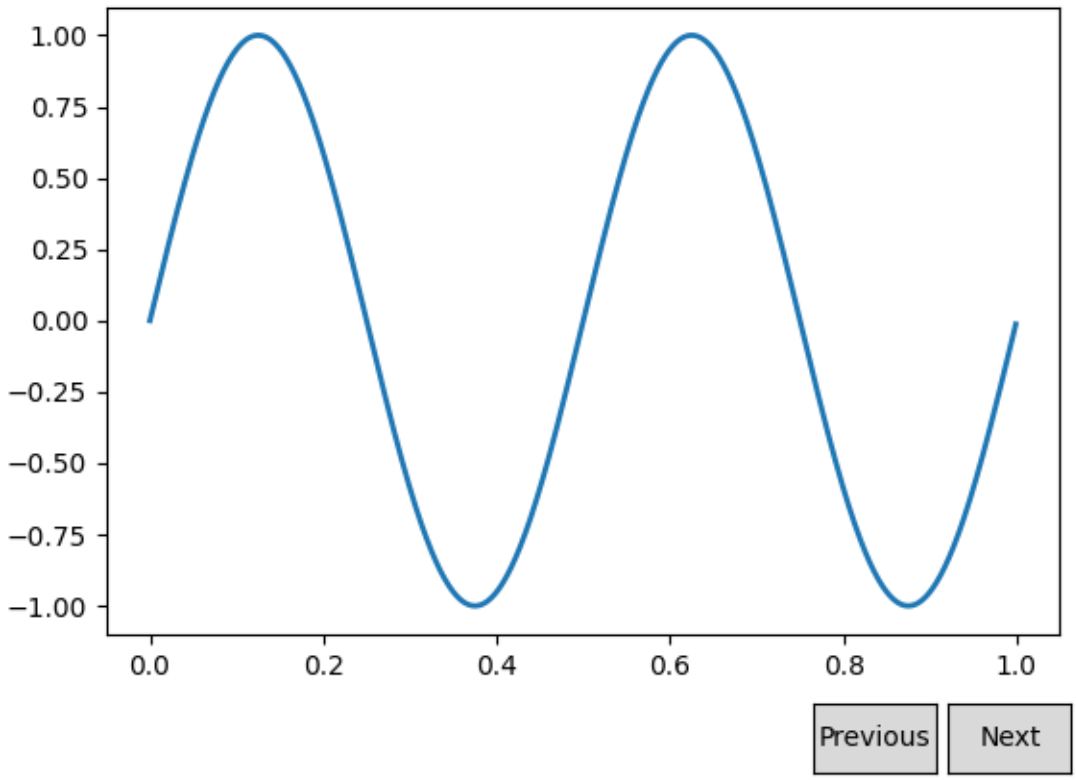
The `next` and `previous` button widget helps visualize the wave with new frequencies.

```
import matplotlib.pyplot as plt
import numpy as np
```

(continues on next page)

(continued from previous page)

```
from matplotlib.widgets import Button

freqs = np.arange(2, 20, 3)

fig, ax = plt.subplots()
fig.subplots_adjust(bottom=0.2)
t = np.arange(0.0, 1.0, 0.001)
s = np.sin(2*np.pi*freqs[0]*t)
l, = ax.plot(t, s, lw=2)


class Index:
    ind = 0

    def next(self, event):
        self.ind += 1
        i = self.ind % len(freqs)
        ydata = np.sin(2*np.pi*freqs[i]*t)
        l.set_ydata(ydata)
        plt.draw()

    def prev(self, event):
        self.ind -= 1
        i = self.ind % len(freqs)
        ydata = np.sin(2*np.pi*freqs[i]*t)
        l.set_ydata(ydata)
        plt.draw()

callback = Index()
axprev = fig.add_axes([0.7, 0.05, 0.1, 0.075])
axnext = fig.add_axes([0.81, 0.05, 0.1, 0.075])
bnext = Button(axnext, 'Next')
bnext.on_clicked(callback.next)
bprev = Button(axprev, 'Previous')
bprev.on_clicked(callback.prev)

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.widgets.Button*

## Check buttons

Turning visual elements on and off with check buttons.

This program shows the use of *CheckButtons* which is similar to check boxes. There are 3 different sine waves shown, and we can choose which waves are displayed with the check buttons.

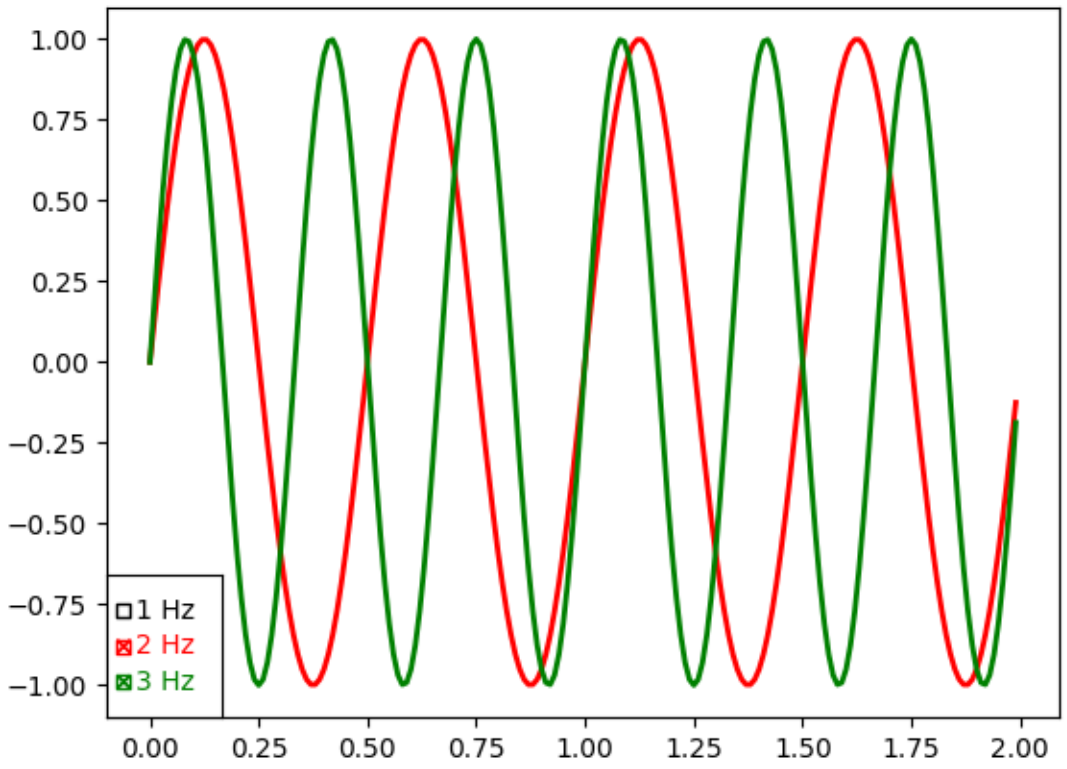
Check buttons may be styled using the *check_props*, *frame_props*, and *label_props* parameters. The parameters each take a dictionary with keys of artist property names and values of lists of settings with length matching the number of buttons.

```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.widgets import CheckButtons
```

(continues on next page)

(continued from previous page)

```
t = np.arange(0.0, 2.0, 0.01)
s0 = np.sin(2*np.pi*t)
s1 = np.sin(4*np.pi*t)
s2 = np.sin(6*np.pi*t)

fig, ax = plt.subplots()
l0, = ax.plot(t, s0, visible=False, lw=2, color='black', label='1 Hz')
l1, = ax.plot(t, s1, lw=2, color='red', label='2 Hz')
l2, = ax.plot(t, s2, lw=2, color='green', label='3 Hz')

lines_by_label = {l.get_label(): l for l in [l0, l1, l2]}
line_colors = [l.get_color() for l in lines_by_label.values()]

# Make checkbuttons with all plotted lines with correct visibility
rax = ax.inset_axes([0.0, 0.0, 0.12, 0.2])
check = CheckButtons(
    ax=rax,
    labels=lines_by_label.keys(),
    actives=[l.get_visible() for l in lines_by_label.values()],
    label_props={'color': line_colors},
    frame_props={'edgecolor': line_colors},
    check_props={'facecolor': line_colors},
)


def callback(label):
    ln = lines_by_label[label]
    ln.set_visible(not ln.get_visible())
    ln.figure.canvas.draw_idle()

check.on_clicked(callback)

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.widgets.CheckButtons*

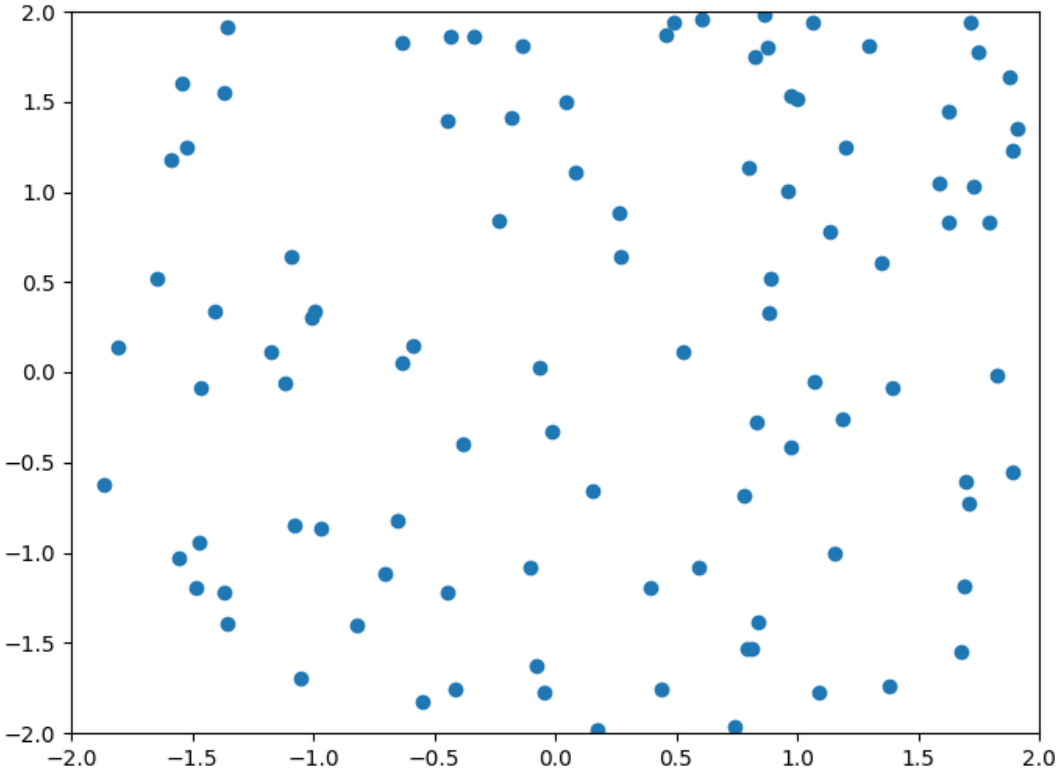
## Cursor

```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.widgets import Cursor

# Fixing random state for reproducibility
np.random.seed(19680801)

fig, ax = plt.subplots(figsize=(8, 6))

x, y = 4*(np.random.rand(2, 100) - .5)
ax.plot(x, y, 'o')
```

(continues on next page)

(continued from previous page)

```
ax.set_xlim(-2, 2)
ax.set_ylim(-2, 2)

# Set useblit=True on most backends for enhanced performance.
cursor = Cursor(ax, useblit=True, color='red', linewidth=2)

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.widgets.Cursor*

### Lasso Selector

Interactively selecting data points with the lasso tool.

This examples plots a scatter plot. You can then select a few points by drawing a lasso loop around the points on the graph. To draw, just click on the graph, hold, and drag it around the points you need to select.

```
import numpy as np

from matplotlib.path import Path
from matplotlib.widgets import LassoSelector


class SelectFromCollection:
    """
    Select indices from a matplotlib collection using `LassoSelector`.

    Selected indices are saved in the `ind` attribute. This tool fades out the
    points that are not part of the selection (i.e., reduces their alpha
    values). If your collection has alpha < 1, this tool will permanently
    alter the alpha values.

    Note that this tool selects collection objects based on their *origins*
    (i.e., `offsets`).

    Parameters
    ----------
    ax : `~matplotlib.axes.Axes`
        Axes to interact with.
    collection : `matplotlib.collections.Collection` subclass
        Collection you want to select from.
    alpha_other : 0 <= float <= 1
        To highlight a selection, this tool sets all selected points to an
        alpha value of 1 and non-selected points to *alpha_other*.
    """

    def __init__(self, ax, collection, alpha_other=0.3):
        self.canvas = ax.figure.canvas
        self.collection = collection
        self.alpha_other = alpha_other

        self.xys = collection.get_offsets()
        self.Npts = len(self.xys)

        # Ensure that we have separate colors for each object
        self.fc = collection.get_facecolors()
        if len(self.fc) == 0:
            raise ValueError('Collection must have a facecolor')
        elif len(self.fc) == 1:
            self.fc = np.tile(self.fc, (self.Npts, 1))

        self.lasso = LassoSelector(ax, onselect=self.onselect)
        self.ind = []
```

(continues on next page)

(continued from previous page)

```
    def onselect(self, verts):
        path = Path(verts)
        self.ind = np.nonzero(path.contains_points(self.xys))[0]
        self.fc[:, -1] = self.alpha_other
        self.fc[self.ind, -1] = 1
        self.collection.set_facecolors(self.fc)
        self.canvas.draw_idle()

    def disconnect(self):
        self.lasso.disconnect_events()
        self.fc[:, -1] = 1
        self.collection.set_facecolors(self.fc)
        self.canvas.draw_idle()


if __name__ == '__main__':
    import matplotlib.pyplot as plt

    # Fixing random state for reproducibility
    np.random.seed(19680801)

    data = np.random.rand(100, 2)

    subplot_kw = dict(xlim=(0, 1), ylim=(0, 1), autoscale_on=False)
    fig, ax = plt.subplots(subplot_kw=subplot_kw)

    pts = ax.scatter(data[:, 0], data[:, 1], s=80)
    selector = SelectFromCollection(ax, pts)

    def accept(event):
        if event.key == "enter":
            print("Selected points:")
            print(selector.xys[selector.ind])
            selector.disconnect()
            ax.set_title("")
            fig.canvas.draw()

    fig.canvas.mpl_connect("key_press_event", accept)
    ax.set_title("Press enter to accept selected points.")

    plt.show()
```

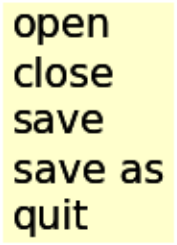
**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.widgets.LassoSelector*
- *matplotlib.path.Path*

### Menu

Using texts to construct a simple menu.




```
from dataclasses import dataclass

import matplotlib.pyplot as plt

import matplotlib.artist as artist
import matplotlib.patches as patches
from matplotlib.typing import ColorType


@dataclass
class ItemProperties:
    fontsize: float = 14
    labelcolor: ColorType = 'black'
    bgcolor: ColorType = 'yellow'
    alpha: float = 1.0


class MenuItem(artist.Artist):
    padx = 0.05  # inches
```

(continues on next page)

(continued from previous page)

```
    pady = 0.05

    def __init__(self, fig, labelstr, props=None, hoverprops=None,
                 on_select=None):
        super().__init__()

        self.set_figure(fig)
        self.labelstr = labelstr

        self.props = props if props is not None else ItemProperties()
        self.hoverprops = (
            hoverprops if hoverprops is not None else ItemProperties())
        if self.props.fontsize != self.hoverprops.fontsize:
            raise NotImplementedError(
                'support for different font sizes not implemented')

        self.on_select = on_select

        # specify coordinates in inches.
        self.label = fig.text(0, 0, labelstr, transform=fig.dpi_scale_trans,
                              size=props.fontsize)
        self.text_bbox = self.label.get_window_extent(
            fig.canvas.get_renderer())
        self.text_bbox = fig.dpi_scale_trans.inverted().transform_bbox(self.
↪text_bbox)

        self.rect = patches.Rectangle(
            (0, 0), 1, 1, transform=fig.dpi_scale_trans
        )  # Will be updated later.

        self.set_hover_props(False)

        fig.canvas.mpl_connect('button_release_event', self.check_select)

    def check_select(self, event):
        over, _ = self.rect.contains(event)
        if not over:
            return
        if self.on_select is not None:
            self.on_select(self)

    def set_extent(self, x, y, w, h, depth):
        self.rect.set(x=x, y=y, width=w, height=h)
        self.label.set(position=(x + self.padx, y + depth + self.pady / 2))
        self.hover = False

    def draw(self, renderer):
        self.rect.draw(renderer)
        self.label.draw(renderer)

    def set_hover_props(self, b):
        props = self.hoverprops if b else self.props
```

(continues on next page)

(continued from previous page)

```
        self.label.set(color=props.labelcolor)
        self.rect.set(facecolor=props.bgcolor, alpha=props.alpha)

    def set_hover(self, event):
        """
        Update the hover status of event and return whether it was changed.
        """
        b, _ = self.rect.contains(event)
        changed = (b != self.hover)
        if changed:
            self.set_hover_props(b)
        self.hover = b
        return changed


class Menu:
    def __init__(self, fig, menuitems):
        self.figure = fig

        self.menuitems = menuitems

        maxw = max(item.text_bbox.width for item in menuitems)
        maxh = max(item.text_bbox.height for item in menuitems)
        depth = max(-item.text_bbox.y0 for item in menuitems)

        x0 = 1
        y0 = 4

        width = maxw + 2 * MenuItem.padx
        height = maxh + MenuItem.pady

        for item in menuitems:
            left = x0
            bottom = y0 - maxh - MenuItem.pady

            item.set_extent(left, bottom, width, height, depth)

            fig.artists.append(item)
            y0 -= maxh + MenuItem.pady

        fig.canvas.mpl_connect('motion_notify_event', self.on_move)

    def on_move(self, event):
        if any(item.set_hover(event) for item in self.menuitems):
            self.figure.canvas.draw()


fig = plt.figure()
fig.subplots_adjust(left=0.3)
props = ItemProperties(labelcolor='black', bgcolor='yellow',
                       fontsize=15, alpha=0.2)
hoverprops = ItemProperties(labelcolor='white', bgcolor='blue',
```

(continues on next page)

(continued from previous page)

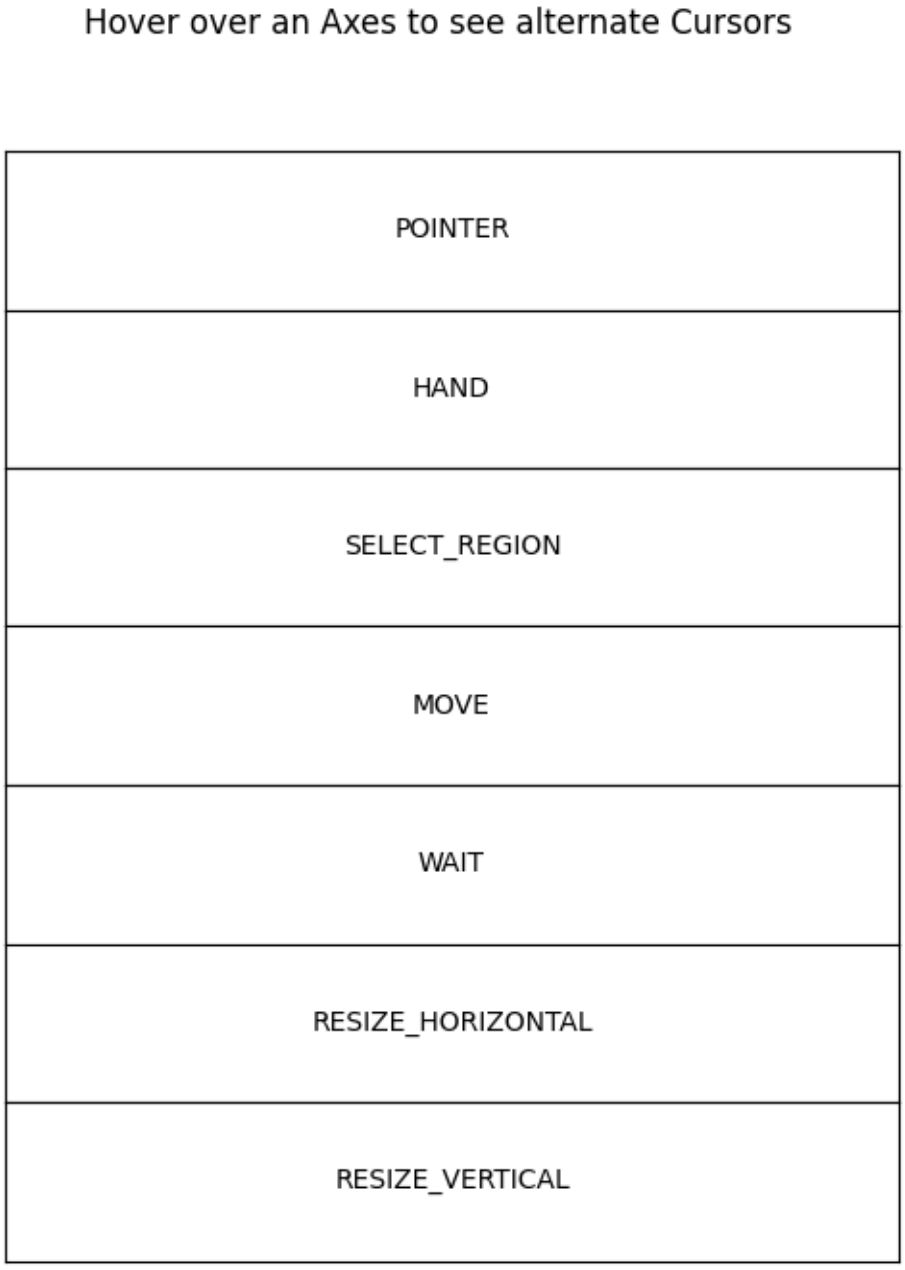
```
                              fontsize=15, alpha=0.2)

menuitems = []
for label in ('open', 'close', 'save', 'save as', 'quit'):
    def on_select(item):
        print(f'you selected {item.labelstr}')
    item = MenuItem(fig, label, props=props, hoverprops=hoverprops,
                    on_select=on_select)
    menuitems.append(item)

menu = Menu(fig, menuitems)
plt.show()
```

## Mouse Cursor

This example sets an alternative cursor on a figure canvas.

Note, this is an interactive example, and must be run to see the effect.

```
import matplotlib.pyplot as plt

from matplotlib.backend_tools import Cursors

fig, axs = plt.subplots(len(Cursors), figsize=(6, len(Cursors) + 0.5),
                        gridspec_kw={'hspace': 0})
fig.suptitle('Hover over an Axes to see alternate Cursors')

for cursor, ax in zip(Cursors, axs):
    ax.cursor_to_use = cursor
    ax.text(0.5, 0.5, cursor.name,
            horizontalalignment='center', verticalalignment='center')
    ax.set(xticks=[], yticks=[])


def hover(event):
    if fig.canvas.widgetlock.locked():
        # Don't do anything if the zoom/pan tools have been enabled.
        return

    fig.canvas.set_cursor(
        event.inaxes.cursor_to_use if event.inaxes else Cursors.POINTER)


fig.canvas.mpl_connect('motion_notify_event', hover)

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

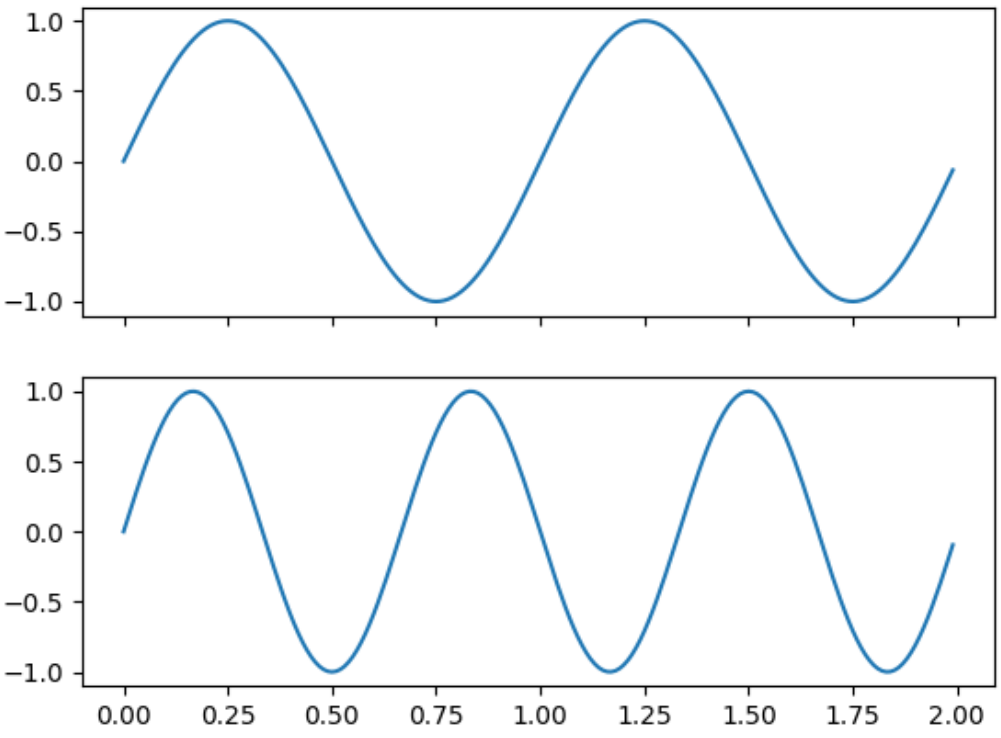
- *matplotlib.backend_bases.FigureCanvasBase.set_cursor*

### Multicursor

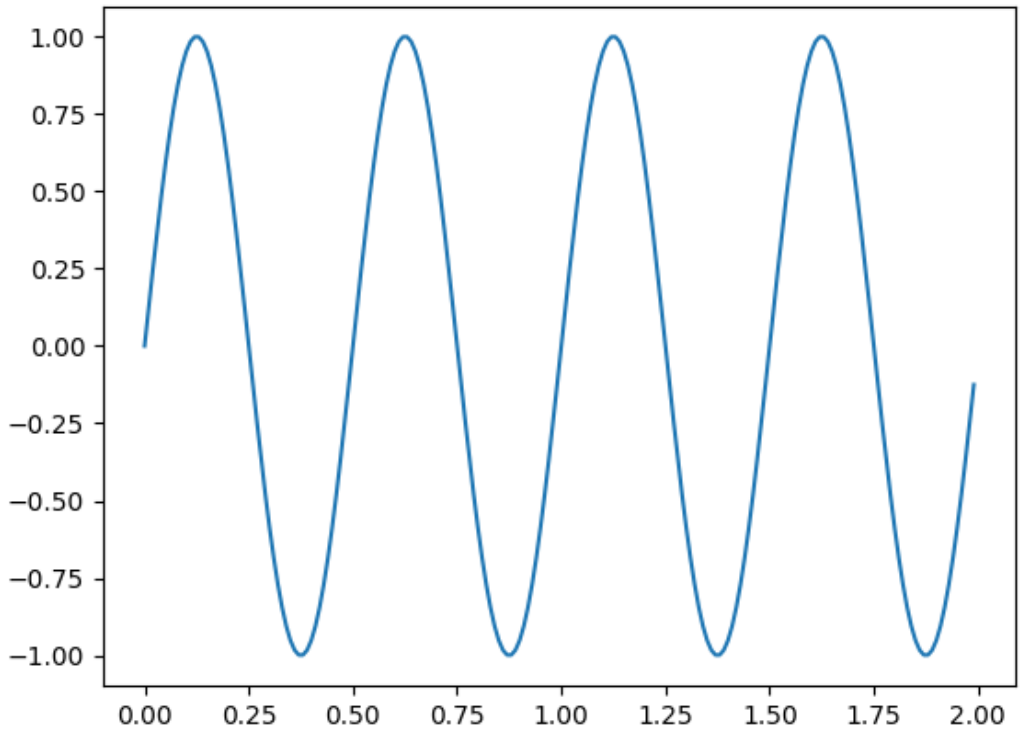
Showing a cursor on multiple plots simultaneously.

This example generates three Axes split over two different figures. On hovering the cursor over data in one subplot, the values of that datapoint are shown in all Axes.

```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.widgets import MultiCursor

t = np.arange(0.0, 2.0, 0.01)
s1 = np.sin(2*np.pi*t)
s2 = np.sin(3*np.pi*t)
s3 = np.sin(4*np.pi*t)

fig, (ax1, ax2) = plt.subplots(2, sharex=True)
ax1.plot(t, s1)
ax2.plot(t, s2)
fig, ax3 = plt.subplots()
ax3.plot(t, s3)

multi = MultiCursor(None, (ax1, ax2, ax3), color='r', lw=1)
plt.show()
```




-




- 

**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.widgets.MultiCursor*

### Select indices from a collection using polygon selector

Shows how one can select indices of a polygon interactively.

```
import numpy as np

from matplotlib.path import Path
from matplotlib.widgets import PolygonSelector


class SelectFromCollection:
    """
    Select indices from a matplotlib collection using `PolygonSelector`.

    Selected indices are saved in the `ind` attribute. This tool fades out the
    points that are not part of the selection (i.e., reduces their alpha
```

(continues on next page)

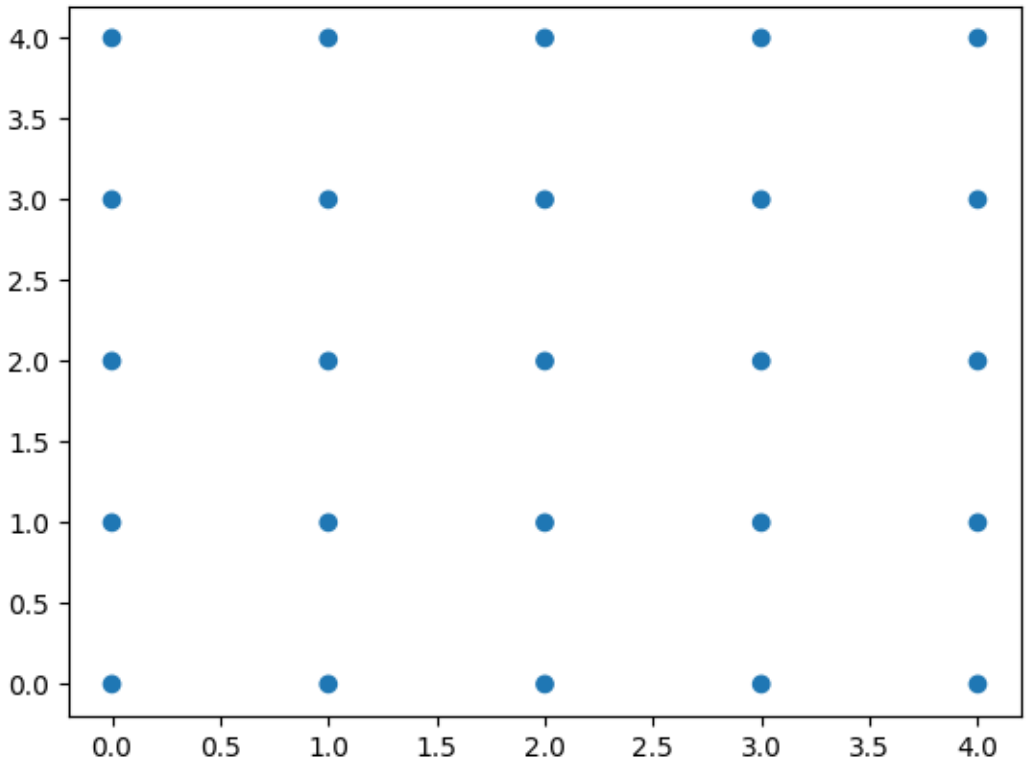
(continued from previous page)

```
    values). If your collection has alpha < 1, this tool will permanently
    alter the alpha values.

    Note that this tool selects collection objects based on their *origins*
    (i.e., `offsets`).

    Parameters
    ----------
    ax : `~matplotlib.axes.Axes`
        Axes to interact with.
    collection : `matplotlib.collections.Collection` subclass
        Collection you want to select from.
    alpha_other : 0 <= float <= 1
        To highlight a selection, this tool sets all selected points to an
        alpha value of 1 and non-selected points to *alpha_other*.
    """

    def __init__(self, ax, collection, alpha_other=0.3):
        self.canvas = ax.figure.canvas
        self.collection = collection
        self.alpha_other = alpha_other

        self.xys = collection.get_offsets()
        self.Npts = len(self.xys)

        # Ensure that we have separate colors for each object
        self.fc = collection.get_facecolors()
        if len(self.fc) == 0:
            raise ValueError('Collection must have a facecolor')
        elif len(self.fc) == 1:
            self.fc = np.tile(self.fc, (self.Npts, 1))

        self.poly = PolygonSelector(ax, self.onselect, draw_bounding_box=True)
        self.ind = []

    def onselect(self, verts):
        path = Path(verts)
        self.ind = np.nonzero(path.contains_points(self.xys))[0]
        self.fc[:, -1] = self.alpha_other
        self.fc[self.ind, -1] = 1
        self.collection.set_facecolors(self.fc)
        self.canvas.draw_idle()

    def disconnect(self):
        self.poly.disconnect_events()
        self.fc[:, -1] = 1
        self.collection.set_facecolors(self.fc)
        self.canvas.draw_idle()


if __name__ == '__main__':
    import matplotlib.pyplot as plt
```

(continues on next page)

(continued from previous page)

```
    fig, ax = plt.subplots()
    grid_size = 5
    grid_x = np.tile(np.arange(grid_size), grid_size)
    grid_y = np.repeat(np.arange(grid_size), grid_size)
    pts = ax.scatter(grid_x, grid_y)

    selector = SelectFromCollection(ax, pts)

    print("Select points in the figure by enclosing them within a polygon.")
    print("Press the 'esc' key to start a new polygon.")
    print("Try holding the 'shift' key to move all of the vertices.")
    print("Try holding the 'ctrl' key to move a single vertex.")

    plt.show()

    selector.disconnect()

    # After figure is closed print the coordinates of the selected points
    print('\nSelected points:')
    print(selector.xys[selector.ind])
```



```
Select points in the figure by enclosing them within a polygon.
Press the 'esc' key to start a new polygon.
Try holding the 'shift' key to move all of the vertices.
Try holding the 'ctrl' key to move a single vertex.

Selected points:
[]
```

**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.widgets.PolygonSelector*
- *matplotlib.path.Path*

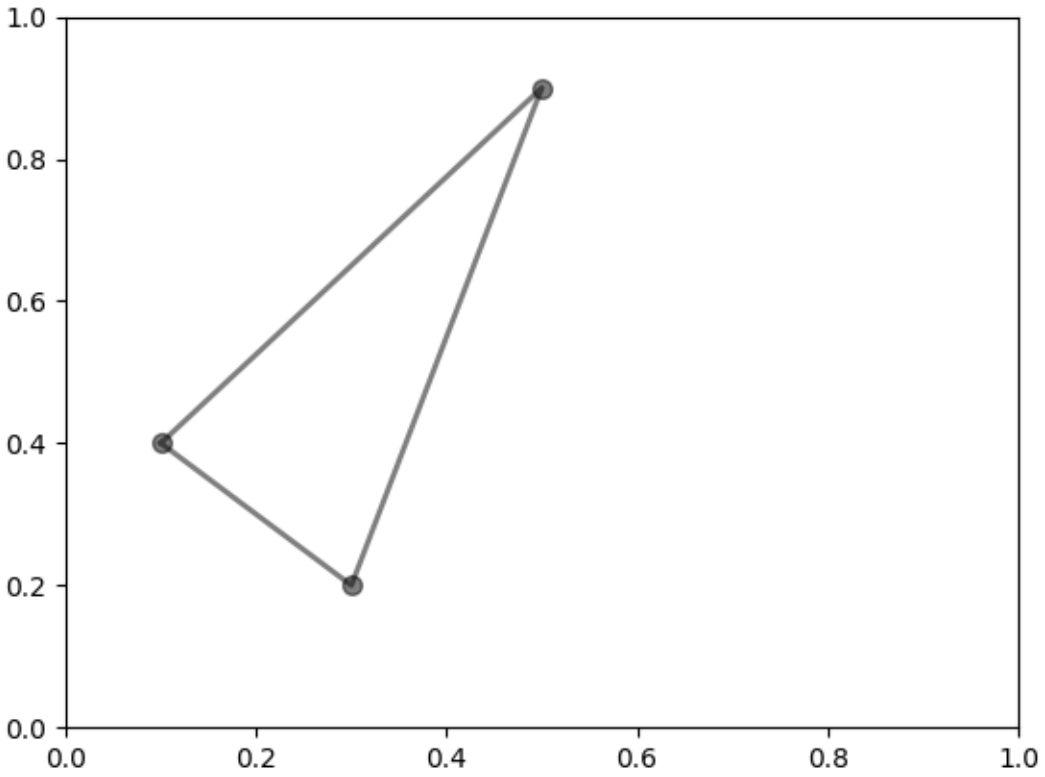
## Polygon Selector

Shows how to create a polygon programmatically or interactively

```
import matplotlib.pyplot as plt

from matplotlib.widgets import PolygonSelector
```

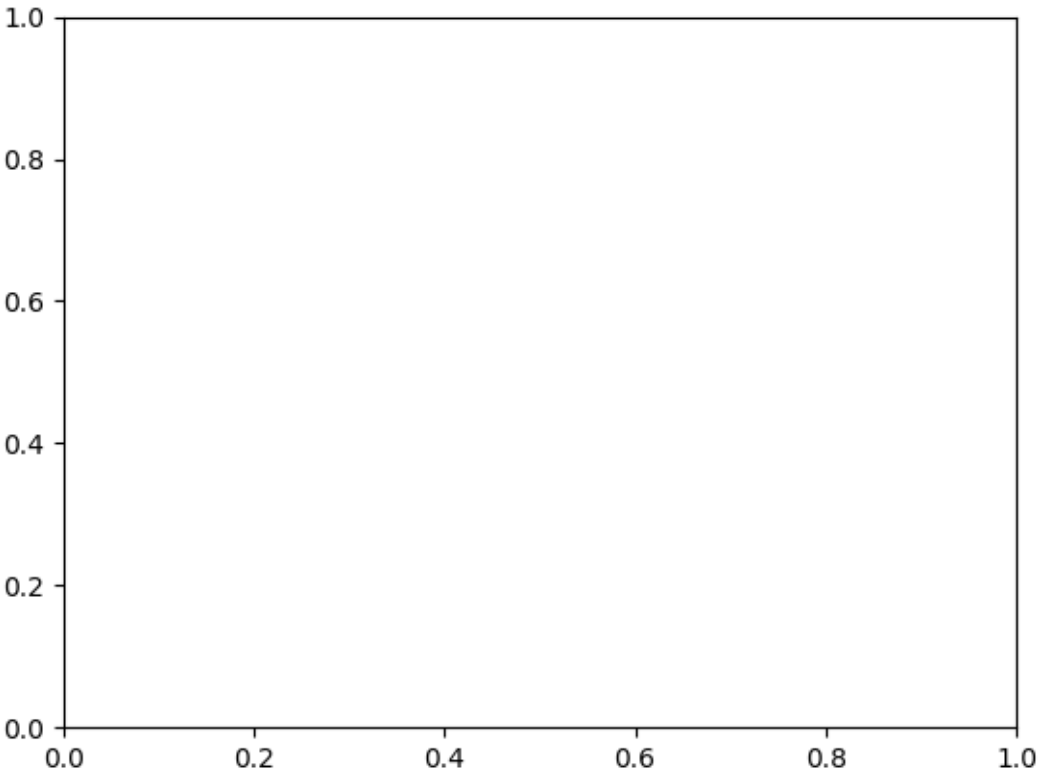
To create the polygon programmatically

```
fig, ax = plt.subplots()
fig.show()

selector = PolygonSelector(ax, lambda *args: None)

# Add three vertices
selector.verts = [(0.1, 0.4), (0.5, 0.9), (0.3, 0.2)]
```




To create the polygon interactively

```
fig2, ax2 = plt.subplots()
fig2.show()

selector2 = PolygonSelector(ax2, lambda *args: None)

print("Click on the figure to create a polygon.")
print("Press the 'esc' key to start a new polygon.")
print("Try holding the 'shift' key to move all of the vertices.")
print("Try holding the 'ctrl' key to move a single vertex.")
```




```
Click on the figure to create a polygon.
Press the 'esc' key to start a new polygon.
Try holding the 'shift' key to move all of the vertices.
Try holding the 'ctrl' key to move a single vertex.
```

**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.widgets.PolygonSelector*

## Radio Buttons

Using radio buttons to choose properties of your plot.

Radio buttons let you choose between multiple options in a visualization. In this case, the buttons let the user choose one of the three different sine waves to be shown in the plot.

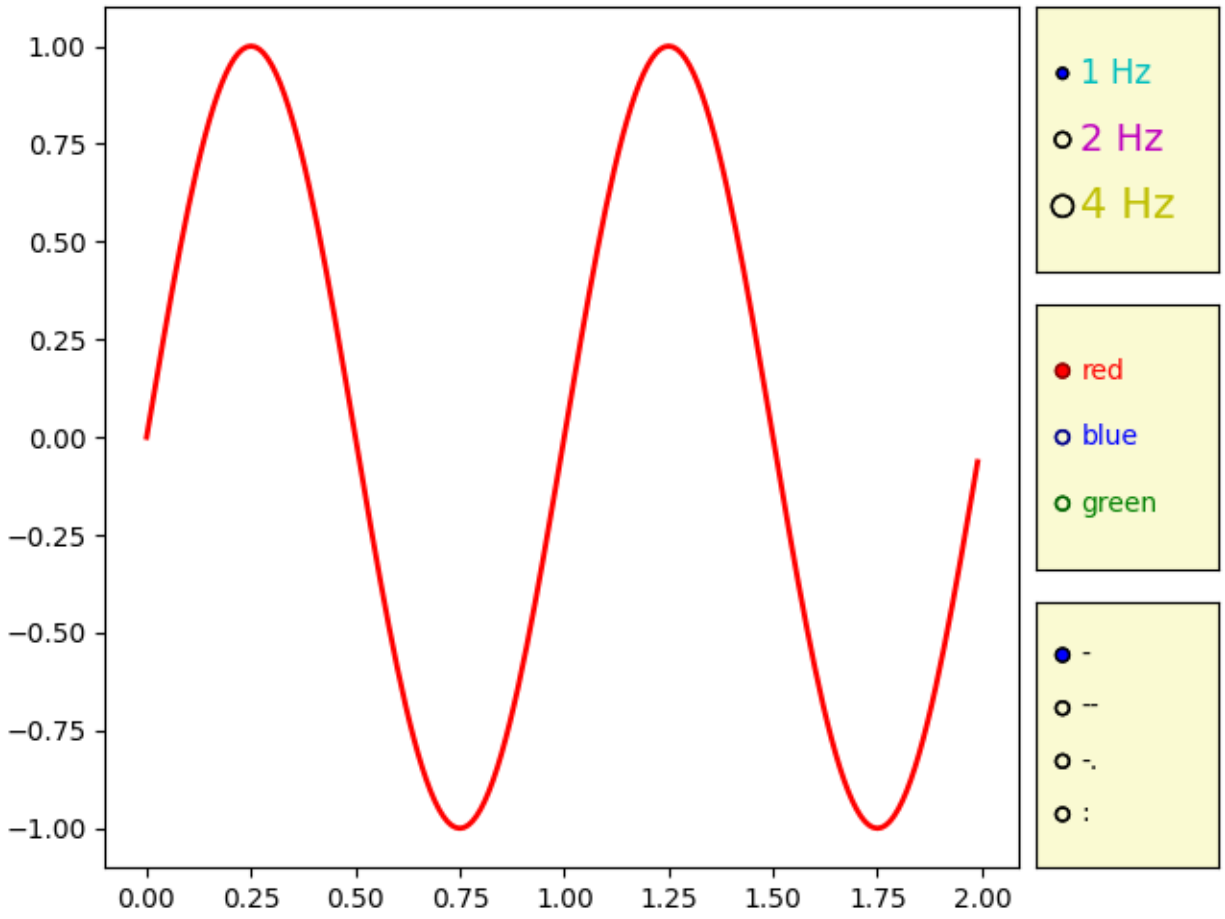
Radio buttons may be styled using the *label_props* and *radio_props* parameters, which each take a dictionary with keys of artist property names and values of lists of settings with length matching the number of buttons.

```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.widgets import RadioButtons

t = np.arange(0.0, 2.0, 0.01)
s0 = np.sin(2*np.pi*t)
s1 = np.sin(4*np.pi*t)
s2 = np.sin(8*np.pi*t)

fig, ax = plt.subplot_mosaic(
    [
        ['main', 'freq'],
        ['main', 'color'],
        ['main', 'linestyle'],
    ],
    width_ratios=[5, 1],
    layout='constrained',
)
l, = ax['main'].plot(t, s0, lw=2, color='red')

radio_background = 'lightgoldenrodyellow'

ax['freq'].set_facecolor(radio_background)
radio = RadioButtons(ax['freq'], ('1 Hz', '2 Hz', '4 Hz'),
                     label_props={'color': 'cmy', 'fontsize': [12, 14, 16]},
                     radio_props={'s': [16, 32, 64]})


def hzfunc(label):
    hzdict = {'1 Hz': s0, '2 Hz': s1, '4 Hz': s2}
    ydata = hzdict[label]
    l.set_ydata(ydata)
    fig.canvas.draw()
radio.on_clicked(hzfunc)

ax['color'].set_facecolor(radio_background)
radio2 = RadioButtons(
    ax['color'], ('red', 'blue', 'green'),
    label_props={'color': ['red', 'blue', 'green']},
    radio_props={
        'facecolor': ['red', 'blue', 'green'],
        'edgecolor': ['darkred', 'darkblue', 'darkgreen'],
    })


def colorfunc(label):
    l.set_color(label)
    fig.canvas.draw()
radio2.on_clicked(colorfunc)

ax['linestyle'].set_facecolor(radio_background)
```

(continues on next page)

(continued from previous page)

```
radio3 = RadioButtons(ax['linestyle'], ('-', '--', '-.', ':'))


def stylefunc(label):
    l.set_linestyle(label)
    fig.canvas.draw()
radio3.on_clicked(stylefunc)

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.widgets.RadioButtons*

### Thresholding an Image with RangeSlider

Using the RangeSlider widget to control the thresholding of an image.

The RangeSlider widget can be used similarly to the *widgets.Slider* widget. The major difference is that RangeSlider's `val` attribute is a tuple of floats `(lower val, upper val)` rather than a single float.

See *Slider* for an example of using a `Slider` to control a single float.

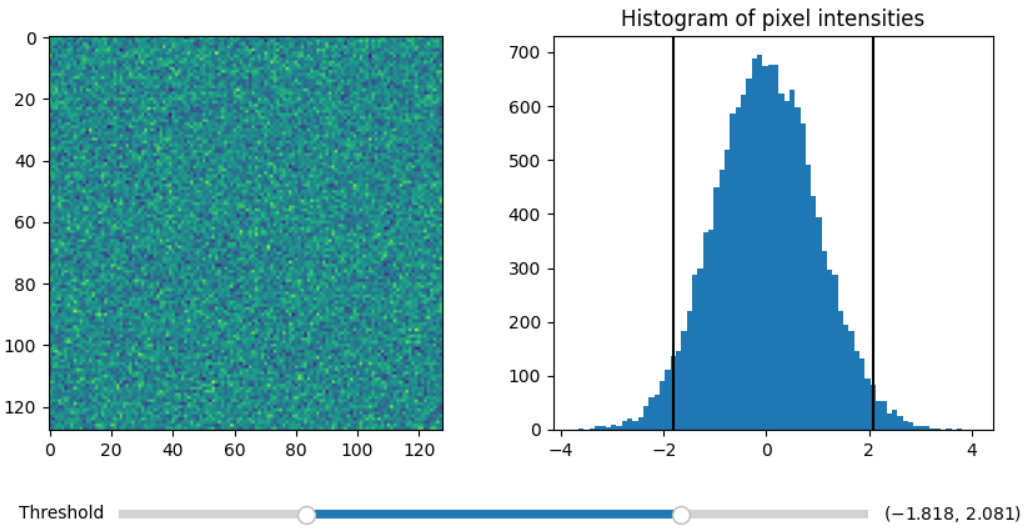
See *Snapping Sliders to Discrete Values* for an example of having the `Slider` snap to discrete values.

```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.widgets import RangeSlider

# generate a fake image
np.random.seed(19680801)
N = 128
img = np.random.randn(N, N)

fig, axs = plt.subplots(1, 2, figsize=(10, 5))
fig.subplots_adjust(bottom=0.25)

im = axs[0].imshow(img)
axs[1].hist(img.flatten(), bins='auto')
axs[1].set_title('Histogram of pixel intensities')

# Create the RangeSlider
slider_ax = fig.add_axes([0.20, 0.1, 0.60, 0.03])
slider = RangeSlider(slider_ax, "Threshold", img.min(), img.max())

# Create the Vertical lines on the histogram
lower_limit_line = axs[1].axvline(slider.val[0], color='k')
upper_limit_line = axs[1].axvline(slider.val[1], color='k')


def update(val):
    # The val passed to a callback by the RangeSlider will
    # be a tuple of (min, max)

    # Update the image's colormap
    im.norm.vmin = val[0]
    im.norm.vmax = val[1]

    # Update the position of the vertical lines
    lower_limit_line.set_xdata([val[0], val[0]])
    upper_limit_line.set_xdata([val[1], val[1]])

    # Redraw the figure to ensure it updates
    fig.canvas.draw_idle()
```

(continues on next page)

(continued from previous page)

```
slider.on_changed(update)
plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.widgets.RangeSlider*

## Rectangle and ellipse selectors

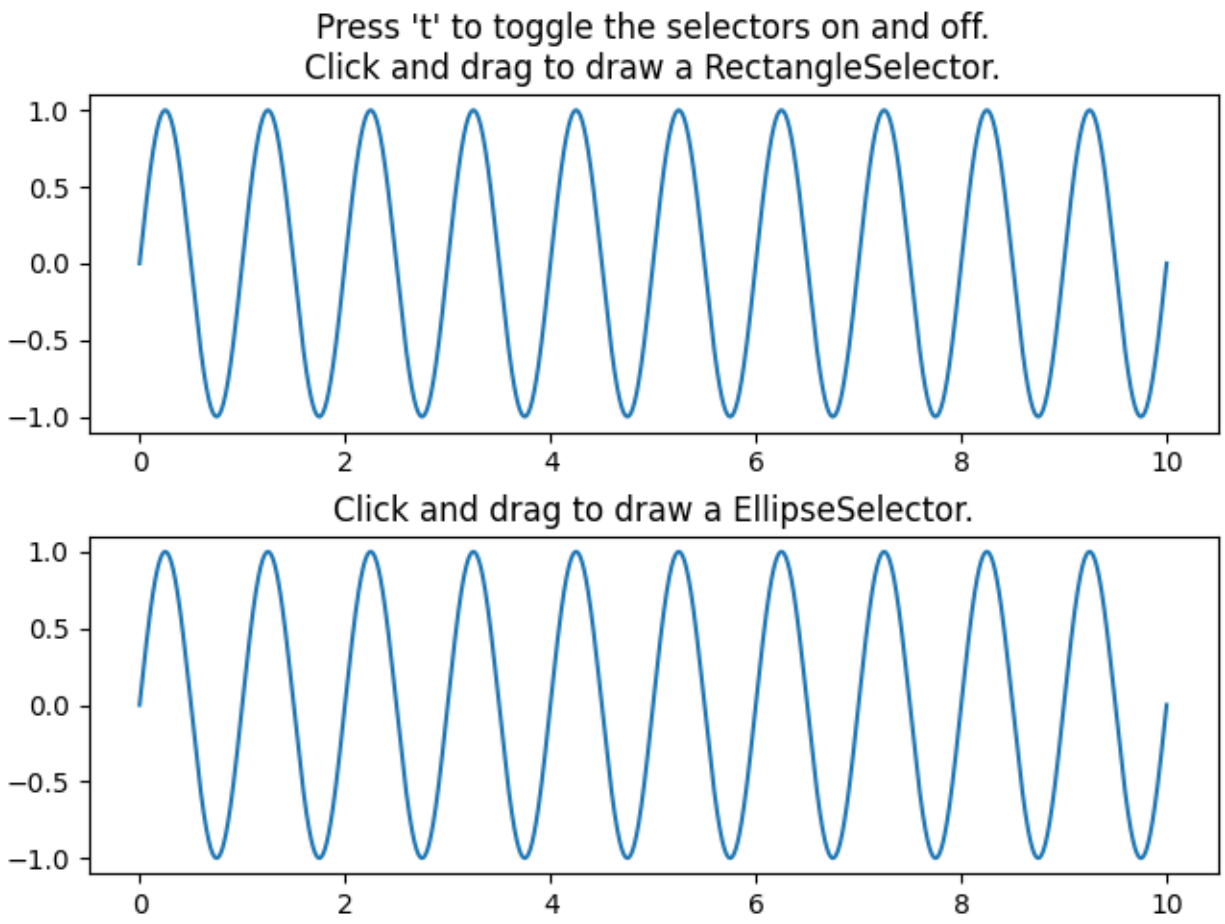
Click somewhere, move the mouse, and release the mouse button. *RectangleSelector* and *EllipseSelector* draw a rectangle or an ellipse from the initial click position to the current mouse position (within the same axes) until the button is released. A connected callback receives the click- and release-events.

```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.widgets import EllipseSelector, RectangleSelector


def select_callback(eclick, erelease):
    """
    Callback for line selection.

    *eclick* and *erelease* are the press and release events.
```

(continues on next page)

(continued from previous page)

```
    """
    x1, y1 = eclick.xdata, eclick.ydata
    x2, y2 = erelease.xdata, erelease.ydata
    print(f"({x1:3.2f}, {y1:3.2f}) --> ({x2:3.2f}, {y2:3.2f})")
    print(f"The buttons you used were: {eclick.button} {erelease.button}")


def toggle_selector(event):
    print('Key pressed.')
    if event.key == 't':
        for selector in selectors:
            name = type(selector).__name__
            if selector.active:
                print(f'{name} deactivated.')
                selector.set_active(False)
            else:
                print(f'{name} activated.')
                selector.set_active(True)


fig = plt.figure(layout='constrained')
axs = fig.subplots(2)

N = 100000  # If N is large one can see improvement by using blitting.
x = np.linspace(0, 10, N)

selectors = []
for ax, selector_class in zip(axs, [RectangleSelector, EllipseSelector]):
    ax.plot(x, np.sin(2*np.pi*x))  # plot something
    ax.set_title(f"Click and drag to draw a {selector_class.__name__}.")
    selectors.append(selector_class(
        ax, select_callback,
        useblit=True,
        button=[1, 3],  # disable middle button
        minspanx=5, minspany=5,
        spancoords='pixels',
        interactive=True))
    fig.canvas.mpl_connect('key_press_event', toggle_selector)
axs[0].set_title("Press 't' to toggle the selectors on and off.\n"
                 + axs[0].get_title())
plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.widgets.RectangleSelector*
- *matplotlib.widgets.EllipseSelector*

## Slider

In this example, sliders are used to control the frequency and amplitude of a sine wave.

See *Snapping Sliders to Discrete Values* for an example of having the `Slider` snap to discrete values.

See *Thresholding an Image with RangeSlider* for an example of using a `RangeSlider` to define a range of values.

```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.widgets import Button, Slider
```

(continues on next page)

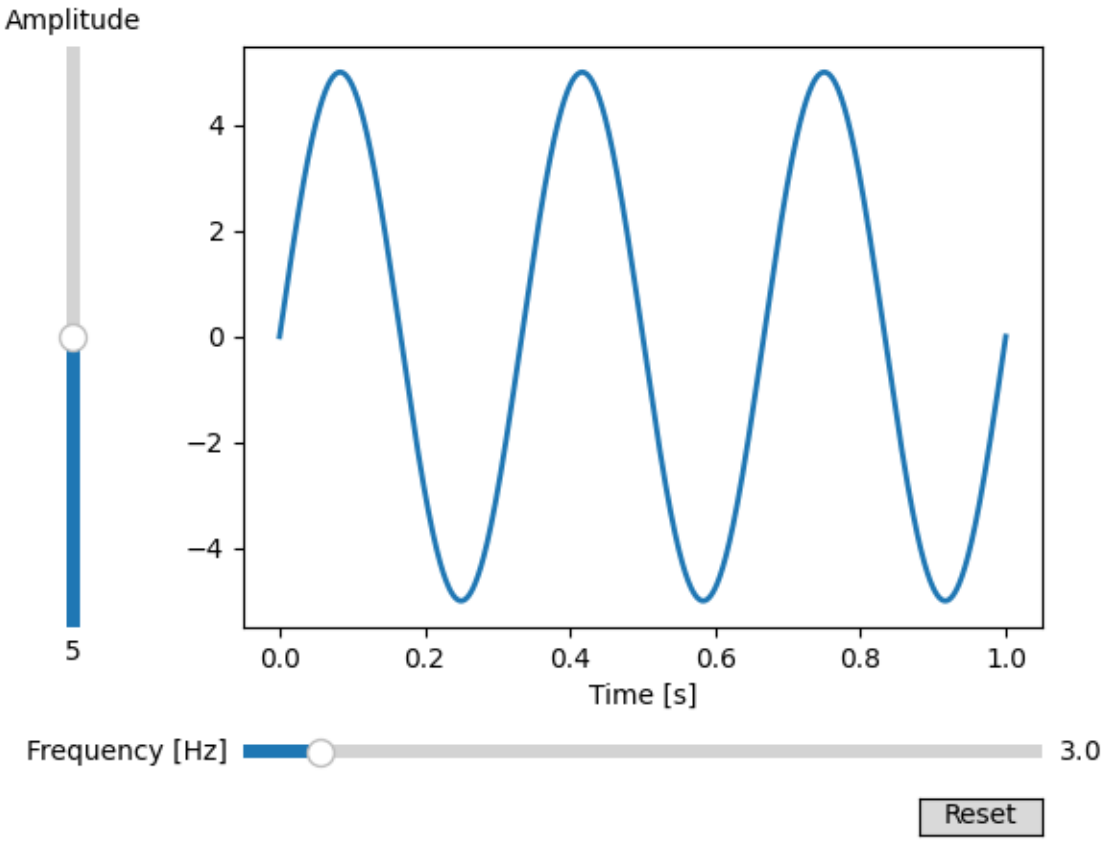
(continued from previous page)

```
# The parametrized function to be plotted
def f(t, amplitude, frequency):
    return amplitude * np.sin(2 * np.pi * frequency * t)

t = np.linspace(0, 1, 1000)

# Define initial parameters
init_amplitude = 5
init_frequency = 3

# Create the figure and the line that we will manipulate
fig, ax = plt.subplots()
line, = ax.plot(t, f(t, init_amplitude, init_frequency), lw=2)
ax.set_xlabel('Time [s]')

# adjust the main plot to make room for the sliders
fig.subplots_adjust(left=0.25, bottom=0.25)

# Make a horizontal slider to control the frequency.
axfreq = fig.add_axes([0.25, 0.1, 0.65, 0.03])
freq_slider = Slider(
    ax=axfreq,
    label='Frequency [Hz]',
    valmin=0.1,
    valmax=30,
    valinit=init_frequency,
)

# Make a vertically oriented slider to control the amplitude
axamp = fig.add_axes([0.1, 0.25, 0.0225, 0.63])
amp_slider = Slider(
    ax=axamp,
    label="Amplitude",
    valmin=0,
    valmax=10,
    valinit=init_amplitude,
    orientation="vertical"
)


# The function to be called anytime a slider's value changes
def update(val):
    line.set_ydata(f(t, amp_slider.val, freq_slider.val))
    fig.canvas.draw_idle()


# register the update function with each slider
freq_slider.on_changed(update)
amp_slider.on_changed(update)
```

(continues on next page)

(continued from previous page)

```
# Create a `matplotlib.widgets.Button` to reset the sliders to initial values.
resetax = fig.add_axes([0.8, 0.025, 0.1, 0.04])
button = Button(resetax, 'Reset', hovercolor='0.975')


def reset(event):
    freq_slider.reset()
    amp_slider.reset()
button.on_clicked(reset)

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.widgets.Button*
- *matplotlib.widgets.Slider*

### Snapping Sliders to Discrete Values

You can snap slider values to discrete values using the `valstep` argument.

In this example the Freq slider is constrained to be multiples of pi, and the Amp slider uses an array as the `valstep` argument to more densely sample the first part of its range.

See *Slider* for an example of using a `Slider` to control a single float.

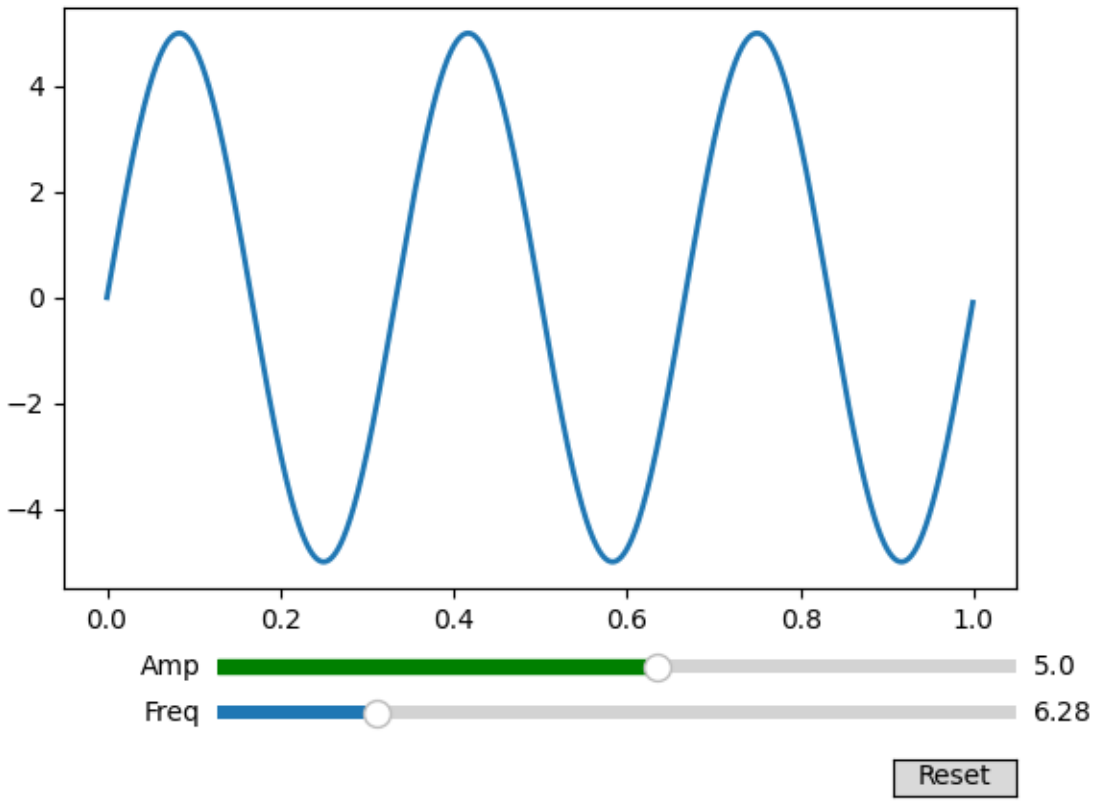
See *Thresholding an Image with RangeSlider* for an example of using a `RangeSlider` to define a range of values.

```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.widgets import Button, Slider

t = np.arange(0.0, 1.0, 0.001)
a0 = 5
f0 = 3
s = a0 * np.sin(2 * np.pi * f0 * t)

fig, ax = plt.subplots()
fig.subplots_adjust(bottom=0.25)
l, = ax.plot(t, s, lw=2)

ax_freq = fig.add_axes([0.25, 0.1, 0.65, 0.03])
ax_amp = fig.add_axes([0.25, 0.15, 0.65, 0.03])

# define the values to use for snapping
allowed_amplitudes = np.concatenate([np.linspace(.1, 5, 100), [6, 7, 8, 9]])

# create the sliders
samp = Slider(
    ax_amp, "Amp", 0.1, 9.0,
    valinit=a0, valstep=allowed_amplitudes,
    color="green"
)

sfreq = Slider(
    ax_freq, "Freq", 0, 10*np.pi,
    valinit=2*np.pi, valstep=np.pi,
    initcolor='none'  # Remove the line marking the valinit position.
)


def update(val):
    amp = samp.val
    freq = sfreq.val
    l.set_ydata(amp*np.sin(2*np.pi*freq*t))
    fig.canvas.draw_idle()
```

(continues on next page)

(continued from previous page)

```
sfreq.on_changed(update)
samp.on_changed(update)

ax_reset = fig.add_axes([0.8, 0.025, 0.1, 0.04])
button = Button(ax_reset, 'Reset', hovercolor='0.975')


def reset(event):
    sfreq.reset()
    samp.reset()
button.on_clicked(reset)


plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.widgets.Slider*
- *matplotlib.widgets.Button*

### Span Selector

The *SpanSelector* is a mouse widget that enables selecting a range on an axis.

Here, an x-range can be selected on the upper axis; a detailed view of the selected range is then plotted on the lower axis.

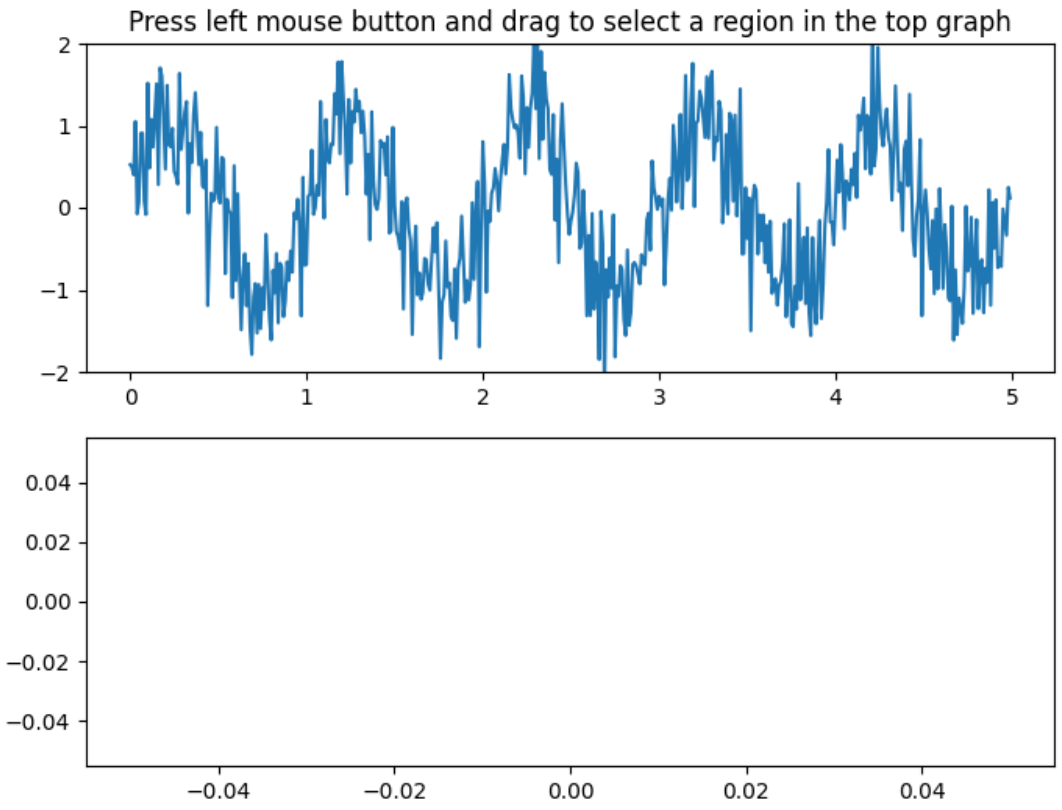
**Note:** If the SpanSelector object is garbage collected you will lose the interactivity. You must keep a hard reference to it to prevent this.

```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.widgets import SpanSelector

# Fixing random state for reproducibility
np.random.seed(19680801)

fig, (ax1, ax2) = plt.subplots(2, figsize=(8, 6))

x = np.arange(0.0, 5.0, 0.01)
y = np.sin(2 * np.pi * x) + 0.5 * np.random.randn(len(x))

ax1.plot(x, y)
ax1.set_ylim(-2, 2)
ax1.set_title('Press left mouse button and drag '
              'to select a region in the top graph')

line2, = ax2.plot([], [])


def onselect(xmin, xmax):
    indmin, indmax = np.searchsorted(x, (xmin, xmax))
    indmax = min(len(x) - 1, indmax)

    region_x = x[indmin:indmax]
    region_y = y[indmin:indmax]

    if len(region_x) >= 2:
        line2.set_data(region_x, region_y)
        ax2.set_xlim(region_x[0], region_x[-1])
        ax2.set_ylim(region_y.min(), region_y.max())
        fig.canvas.draw_idle()


span = SpanSelector(
    ax1,
    onselect,
```

(continues on next page)

(continued from previous page)

```
    "horizontal",
    useblit=True,
    props=dict(alpha=0.5, facecolor="tab:blue"),
    interactive=True,
    drag_from_anywhere=True
)
# Set useblit=True on most backends for enhanced performance.


plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

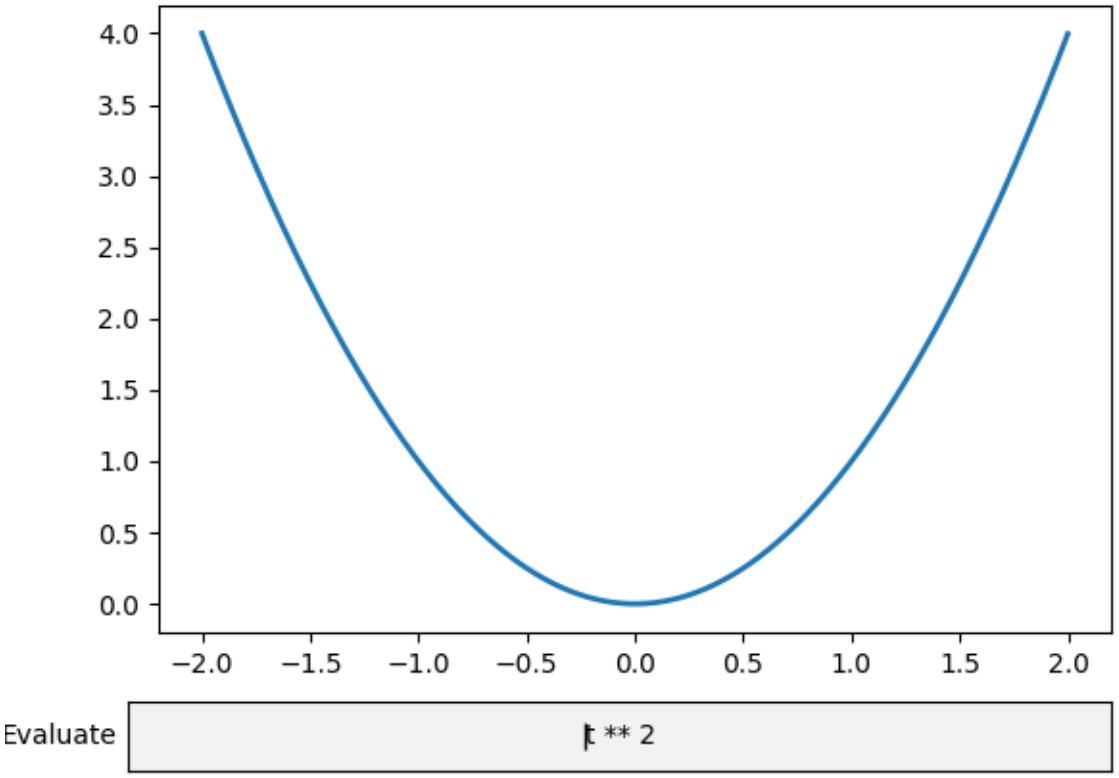
- *matplotlib.widgets.SpanSelector*

### Textbox

The Textbox widget lets users interactively provide text input, including formulas. In this example, the plot is updated using the *on_submit* method. This method triggers the execution of the *submit* function when the user presses enter in the textbox or leaves the textbox.

Note: The *matplotlib.widgets.TextBox* widget is different from the following static elements: *Annotations* and *Placing text boxes*.

```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.widgets import TextBox

fig, ax = plt.subplots()
fig.subplots_adjust(bottom=0.2)

t = np.arange(-2.0, 2.0, 0.001)
l, = ax.plot(t, np.zeros_like(t), lw=2)


def submit(expression):
    """
    Update the plotted function to the new math *expression*.

    *expression* is a string using "t" as its independent variable, e.g.
    "t ** 3".
    """
    ydata = eval(expression, {'np': np}, {'t': t})
    l.set_ydata(ydata)
    ax.relim()
    ax.autoscale_view()
    plt.draw()


axbox = fig.add_axes([0.1, 0.05, 0.8, 0.075])
text_box = TextBox(axbox, "Evaluate", textalignment="center")
text_box.on_submit(submit)
text_box.set_val("t ** 2")  # Trigger `submit` with the initial string.

plt.show()
```



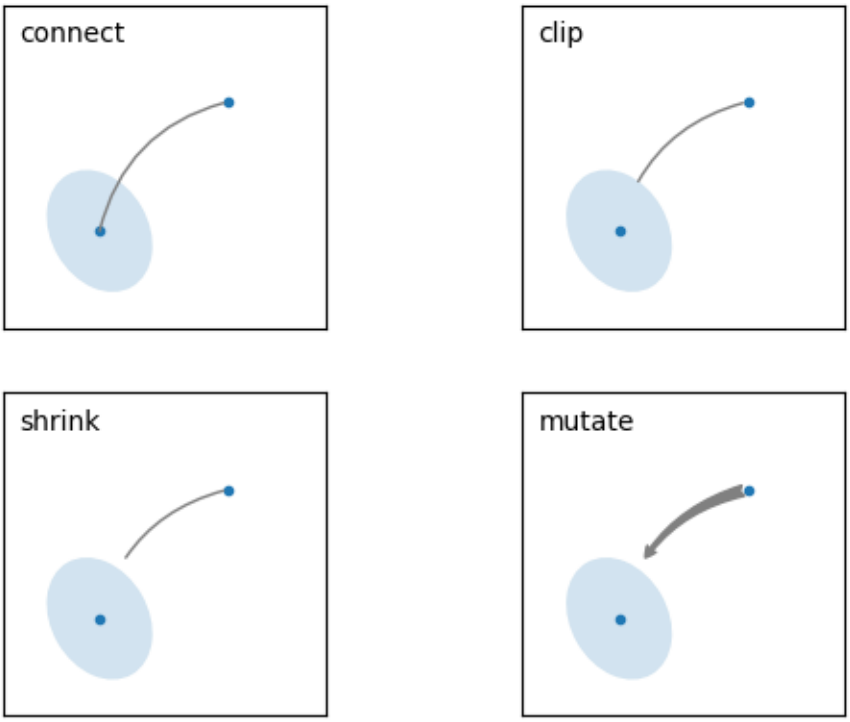

**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.widgets.TextBox*

### 6.25.25 Userdemo

### Annotate Explain




```
import matplotlib.pyplot as plt

import matplotlib.patches as mpatches

fig, axs = plt.subplots(2, 2)
x1, y1 = 0.3, 0.3
x2, y2 = 0.7, 0.7

ax = axs.flat[0]
ax.plot([x1, x2], [y1, y2], ".")
el = mpatches.Ellipse((x1, y1), 0.3, 0.4, angle=30, alpha=0.2)
ax.add_artist(el)
ax.annotate("",
            xy=(x1, y1), xycoords='data',
            xytext=(x2, y2), textcoords='data',
            arrowprops=dict(arrowstyle="-",
                            color="0.5",
                            patchB=None,
                            shrinkB=0,
                            connectionstyle="arc3,rad=0.3",
                            ),
```

(continues on next page)

(continued from previous page)

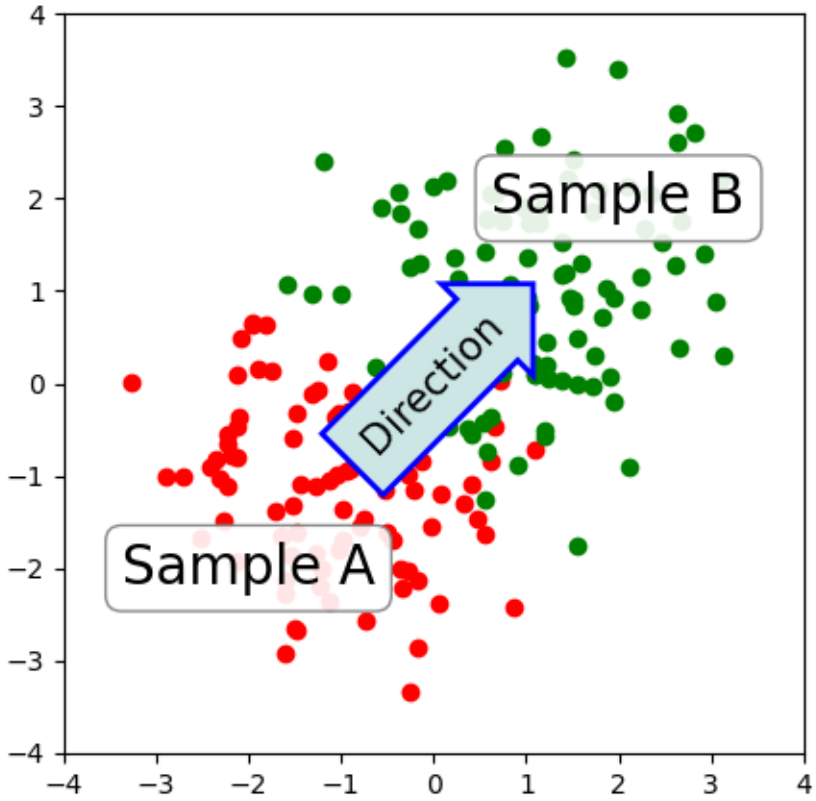
```
            )
ax.text(.05, .95, "connect", transform=ax.transAxes, ha="left", va="top")

ax = axs.flat[1]
ax.plot([x1, x2], [y1, y2], ".")
el = mpatches.Ellipse((x1, y1), 0.3, 0.4, angle=30, alpha=0.2)
ax.add_artist(el)
ax.annotate("",
            xy=(x1, y1), xycoords='data',
            xytext=(x2, y2), textcoords='data',
            arrowprops=dict(arrowstyle="-",
                            color="0.5",
                            patchB=el,
                            shrinkB=0,
                            connectionstyle="arc3,rad=0.3",
                            ),
            )
ax.text(.05, .95, "clip", transform=ax.transAxes, ha="left", va="top")

ax = axs.flat[2]
ax.plot([x1, x2], [y1, y2], ".")
el = mpatches.Ellipse((x1, y1), 0.3, 0.4, angle=30, alpha=0.2)
ax.add_artist(el)
ax.annotate("",
            xy=(x1, y1), xycoords='data',
            xytext=(x2, y2), textcoords='data',
            arrowprops=dict(arrowstyle="-",
                            color="0.5",
                            patchB=el,
                            shrinkB=5,
                            connectionstyle="arc3,rad=0.3",
                            ),
            )
ax.text(.05, .95, "shrink", transform=ax.transAxes, ha="left", va="top")

ax = axs.flat[3]
ax.plot([x1, x2], [y1, y2], ".")
el = mpatches.Ellipse((x1, y1), 0.3, 0.4, angle=30, alpha=0.2)
ax.add_artist(el)
ax.annotate("",
            xy=(x1, y1), xycoords='data',
            xytext=(x2, y2), textcoords='data',
            arrowprops=dict(arrowstyle="fancy",
                            color="0.5",
                            patchB=el,
                            shrinkB=5,
                            connectionstyle="arc3,rad=0.3",
                            ),
            )
ax.text(.05, .95, "mutate", transform=ax.transAxes, ha="left", va="top")

for ax in axs.flat:
```

(continues on next page)

(continued from previous page)

```
    ax.set(xlim=(0, 1), ylim=(0, 1), xticks=[], yticks=[], aspect=1)

plt.show()
```

### Annotate Text Arrow




```
import matplotlib.pyplot as plt
import numpy as np

# Fixing random state for reproducibility
np.random.seed(19680801)

fig, ax = plt.subplots(figsize=(5, 5))
ax.set_aspect(1)

x1 = -1 + np.random.randn(100)
y1 = -1 + np.random.randn(100)
x2 = 1. + np.random.randn(100)
y2 = 1. + np.random.randn(100)
```

(continues on next page)

(continued from previous page)

```
ax.scatter(x1, y1, color="r")
ax.scatter(x2, y2, color="g")

bbox_props = dict(boxstyle="round", fc="w", ec="0.5", alpha=0.9)
ax.text(-2, -2, "Sample A", ha="center", va="center", size=20,
        bbox=bbox_props)
ax.text(2, 2, "Sample B", ha="center", va="center", size=20,
        bbox=bbox_props)


bbox_props = dict(boxstyle="rarrow", fc=(0.8, 0.9, 0.9), ec="b", lw=2)
t = ax.text(0, 0, "Direction", ha="center", va="center", rotation=45,
            size=15,
            bbox=bbox_props)

bb = t.get_bbox_patch()
bb.set_boxstyle("rarrow", pad=0.6)

ax.set_xlim(-4, 4)
ax.set_ylim(-4, 4)

plt.show()
```

### Connection styles for annotations

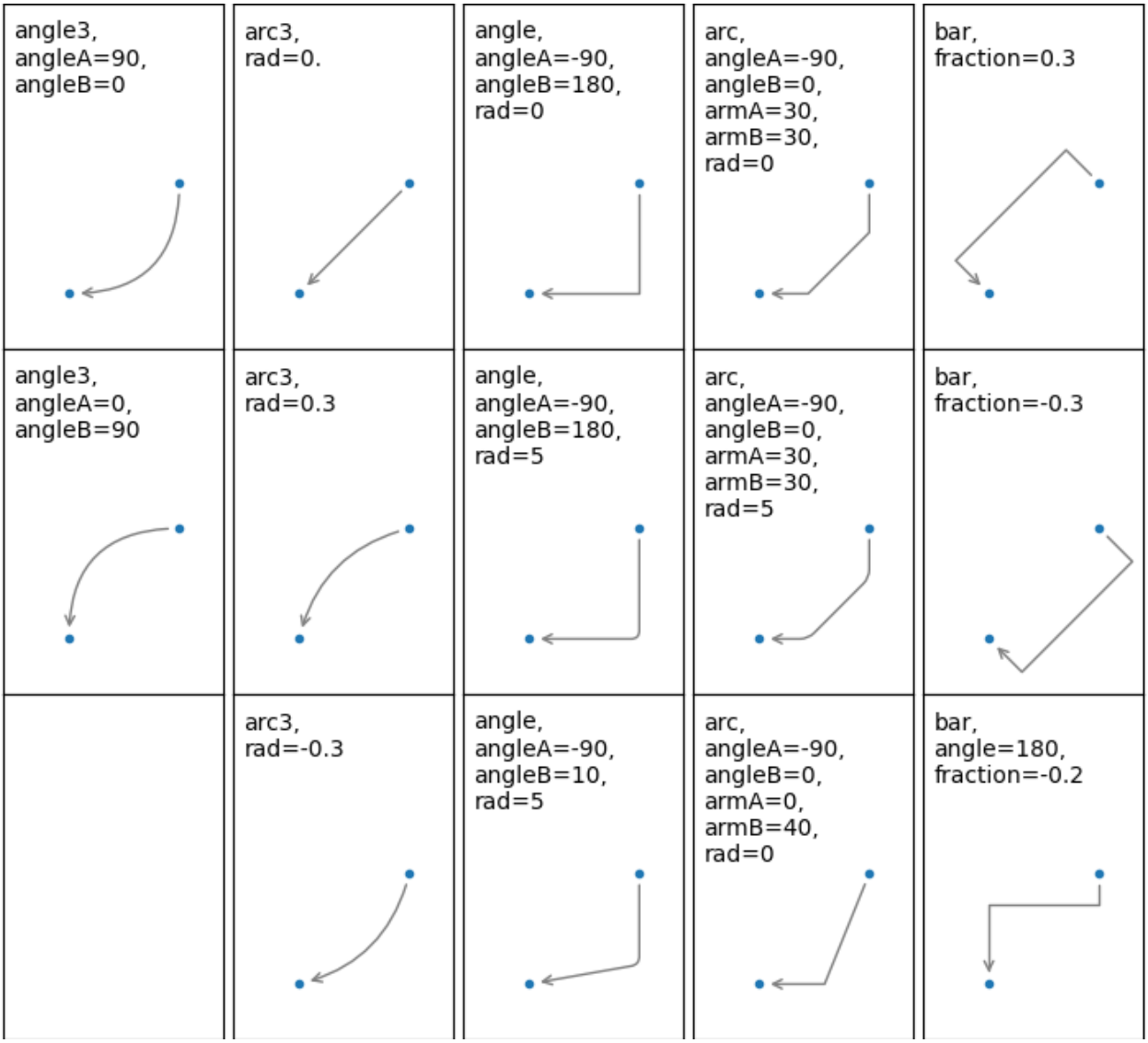
When creating an annotation using *annotate*, the arrow shape can be controlled via the *connectionstyle* parameter of *arrowprops*. For further details see the description of *FancyArrowPatch*.

```
import matplotlib.pyplot as plt


def demo_con_style(ax, connectionstyle):
    x1, y1 = 0.3, 0.2
    x2, y2 = 0.8, 0.6

    ax.plot([x1, x2], [y1, y2], ".")
    ax.annotate("",
                xy=(x1, y1), xycoords='data',
                xytext=(x2, y2), textcoords='data',
                arrowprops=dict(arrowstyle="->", color="0.5",
                                shrinkA=5, shrinkB=5,
                                patchA=None, patchB=None,
                                connectionstyle=connectionstyle,
                                ),
                )

    ax.text(.05, .95, connectionstyle.replace(",", ",\n"),
            transform=ax.transAxes, ha="left", va="top")
```

(continues on next page)

(continued from previous page)

```
fig, axs = plt.subplots(3, 5, figsize=(7, 6.3), layout="constrained")
demo_con_style(axs[0, 0], "angle3,angleA=90,angleB=0")
demo_con_style(axs[1, 0], "angle3,angleA=0,angleB=90")
demo_con_style(axs[0, 1], "arc3,rad=0.")
demo_con_style(axs[1, 1], "arc3,rad=0.3")
demo_con_style(axs[2, 1], "arc3,rad=-0.3")
demo_con_style(axs[0, 2], "angle,angleA=-90,angleB=180,rad=0")
demo_con_style(axs[1, 2], "angle,angleA=-90,angleB=180,rad=5")
demo_con_style(axs[2, 2], "angle,angleA=-90,angleB=10,rad=5")
demo_con_style(axs[0, 3], "arc,angleA=-90,angleB=0,armA=30,armB=30,rad=0")
demo_con_style(axs[1, 3], "arc,angleA=-90,angleB=0,armA=30,armB=30,rad=5")
demo_con_style(axs[2, 3], "arc,angleA=-90,angleB=0,armA=0,armB=40,rad=0")
demo_con_style(axs[0, 4], "bar,fraction=0.3")
demo_con_style(axs[1, 4], "bar,fraction=-0.3")
demo_con_style(axs[2, 4], "bar,angle=180,fraction=-0.2")

for ax in axs.flat:
    ax.set(xlim=(0, 1), ylim=(0, 1.25), xticks=[], yticks=[], aspect=1.25)
fig.get_layout_engine().set(wspace=0, hspace=0, w_pad=0, h_pad=0)

plt.show()
```




**References**

The use of the following functions, methods, classes and modules is shown in this example:

- *matplotlib.axes.Axes.annotate*
- *matplotlib.patches.FancyArrowPatch*

### Custom box styles

This example demonstrates the implementation of a custom *BoxStyle*. Custom *ConnectionStyle*s and *ArrowStyle*s can be similarly defined.

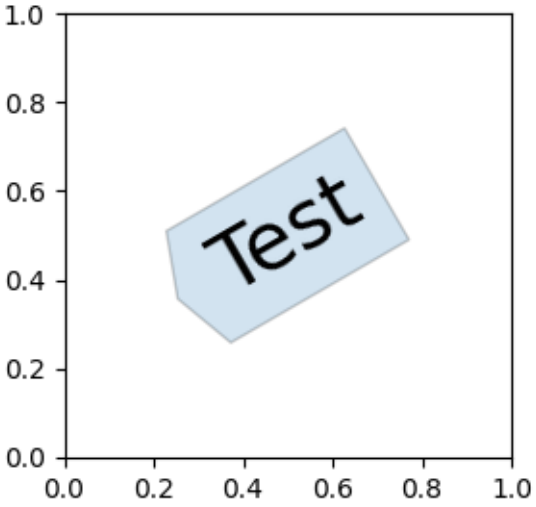
```
import matplotlib.pyplot as plt

from matplotlib.patches import BoxStyle
from matplotlib.path import Path
```

Custom box styles can be implemented as a function that takes arguments specifying both a rectangular box and the amount of "mutation", and returns the "mutated" path. The specific signature is the one of `custom_box_style` below.

Here, we return a new path which adds an "arrow" shape on the left of the box.

The custom box style can then be used by passing `bbox=dict(boxstyle=custom_box_style, ...)` to *Axes.text*.

```
def custom_box_style(x0, y0, width, height, mutation_size):
    """
    Given the location and size of the box, return the path of the box around
    it.

    Rotation is automatically taken care of.

    Parameters
    ----------
    x0, y0, width, height : float
        Box location and size.
    mutation_size : float
        Mutation reference scale, typically the text font size.
    """
    # padding
    mypad = 0.3
    pad = mutation_size * mypad
    # width and height with padding added.
    width = width + 2 * pad
    height = height + 2 * pad
    # boundary of the padded box
    x0, y0 = x0 - pad, y0 - pad
    x1, y1 = x0 + width, y0 + height
    # return the new path
    return Path([(x0, y0),
                 (x1, y0), (x1, y1), (x0, y1),
                 (x0-pad, (y0+y1)/2), (x0, y0),
                 (x0, y0)],
                closed=True)


fig, ax = plt.subplots(figsize=(3, 3))
ax.text(0.5, 0.5, "Test", size=30, va="center", ha="center", rotation=30,
        bbox=dict(boxstyle=custom_box_style, alpha=0.2))
```




Likewise, custom box styles can be implemented as classes that implement `__call__`.

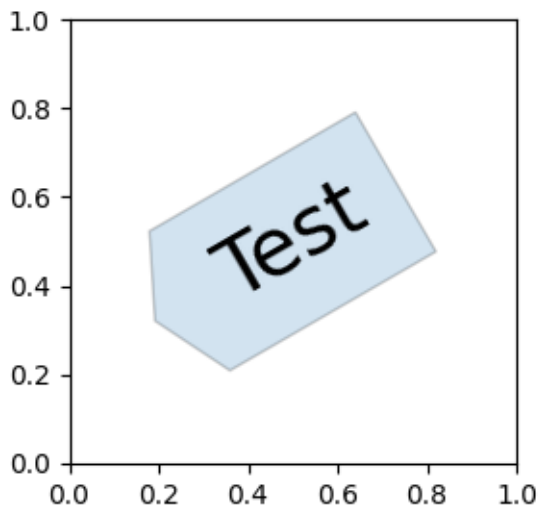
The classes can then be registered into the `BoxStyle._style_list` dict, which allows specifying the box style as a string, `bbox=dict(boxstyle="registered_name,param=value,...", ...)`. Note that this registration relies on internal APIs and is therefore not officially supported.

```
class MyStyle:
    """A simple box."""

    def __init__(self, pad=0.3):
        """
        The arguments must be floats and have default values.

        Parameters
        ----------
        pad : float
            amount of padding
        """
        self.pad = pad
        super().__init__()

    def __call__(self, x0, y0, width, height, mutation_size):
        """
        Given the location and size of the box, return the path of the box
        around it.

        Rotation is automatically taken care of.

        Parameters
        ----------
        x0, y0, width, height : float
            Box location and size.
        mutation_size : float
            Reference scale for the mutation, typically the text font size.
```

(continues on next page)

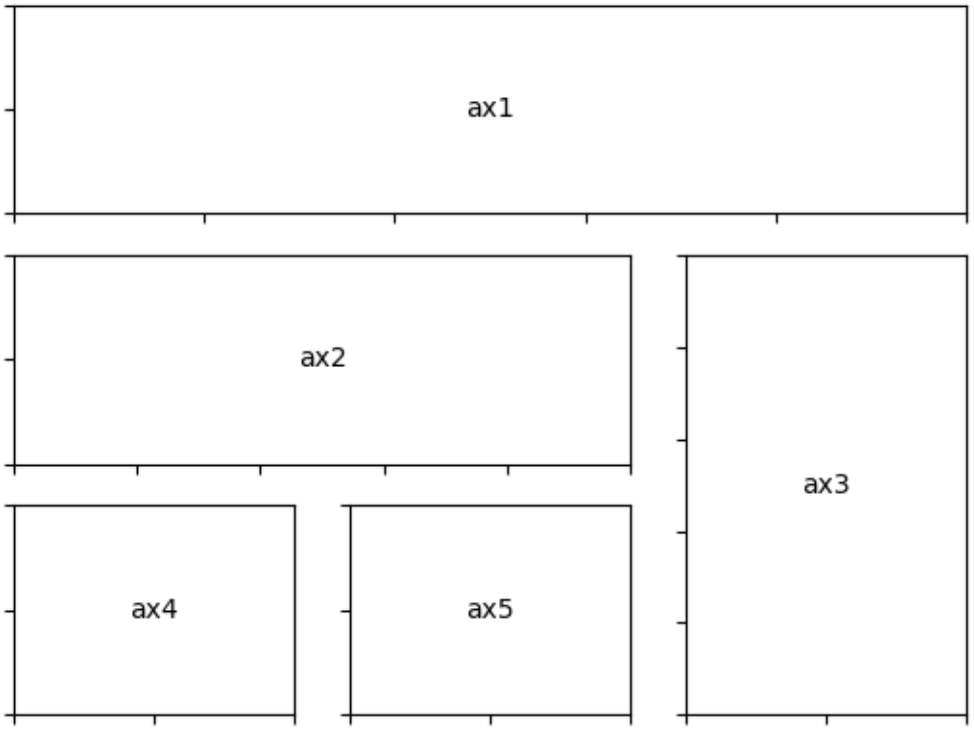
(continued from previous page)

```
        """
        # padding
        pad = mutation_size * self.pad
        # width and height with padding added
        width = width + 2.*pad
        height = height + 2.*pad
        # boundary of the padded box
        x0, y0 = x0 - pad, y0 - pad
        x1, y1 = x0 + width, y0 + height
        # return the new path
        return Path([(x0, y0),
                     (x1, y0), (x1, y1), (x0, y1),
                     (x0-pad, (y0+y1)/2.), (x0, y0),
                     (x0, y0)],
                    closed=True)


BoxStyle._style_list["angled"] = MyStyle  # Register the custom style.

fig, ax = plt.subplots(figsize=(3, 3))
ax.text(0.5, 0.5, "Test", size=30, va="center", ha="center", rotation=30,
        bbox=dict(boxstyle="angled,pad=0.5", alpha=0.2))

del BoxStyle._style_list["angled"]  # Unregister it.

plt.show()
```



### subplot2grid demo

This example demonstrates the use of *pyplot.subplot2grid* to generate subplots. Using *GridSpec*, as demonstrated in *GridSpec demo* is generally preferred.




```
import matplotlib.pyplot as plt


def annotate_axes(fig):
    for i, ax in enumerate(fig.axes):
        ax.text(0.5, 0.5, "ax%d" % (i+1), va="center", ha="center")
        ax.tick_params(labelbottom=False, labelleft=False)


fig = plt.figure()
ax1 = plt.subplot2grid((3, 3), (0, 0), colspan=3)
ax2 = plt.subplot2grid((3, 3), (1, 0), colspan=2)
ax3 = plt.subplot2grid((3, 3), (1, 2), rowspan=2)
ax4 = plt.subplot2grid((3, 3), (2, 0))
ax5 = plt.subplot2grid((3, 3), (2, 1))

annotate_axes(fig)
```

(continues on next page)

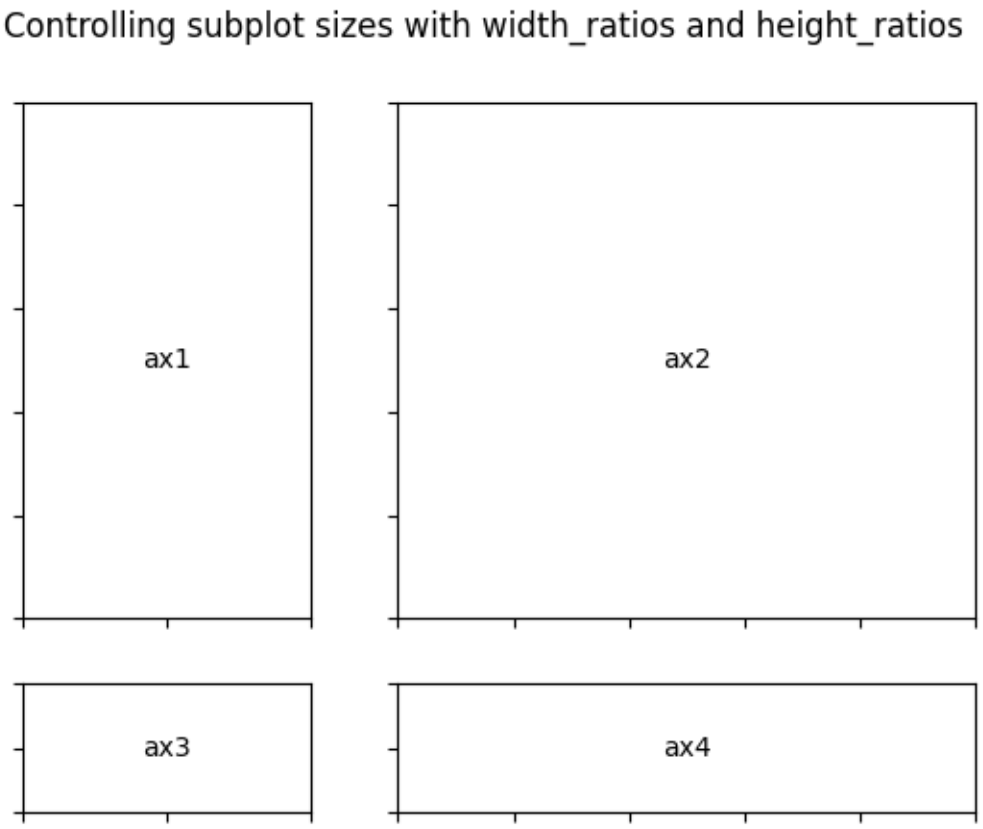
(continued from previous page)

```
plt.show()
```

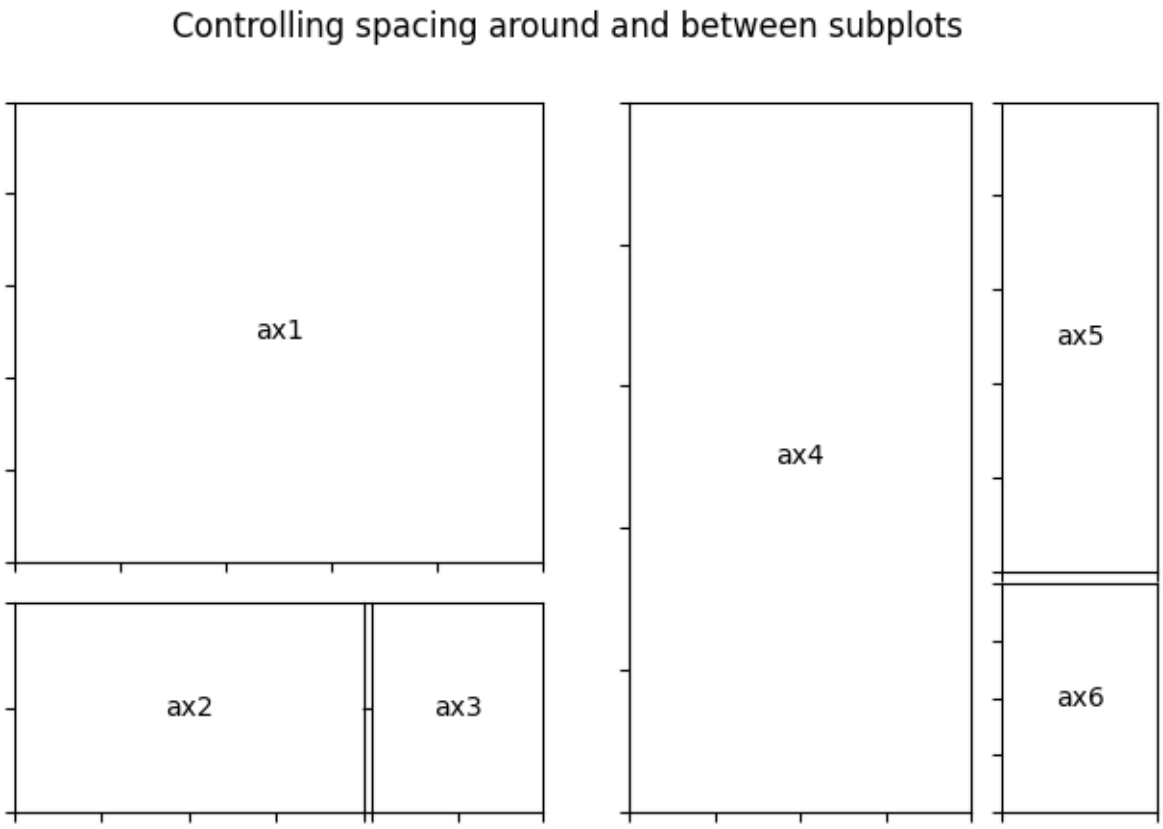
### GridSpec demo

This example demonstrates the use of *GridSpec* to generate subplots, the control of the relative sizes of subplots with *width_ratios* and *height_ratios*, and the control of the spacing around and between subplots using subplot params (*left*, *right*, *bottom*, *top*, *wspace*, and *hspace*).

```
import matplotlib.pyplot as plt

from matplotlib.gridspec import GridSpec


def annotate_axes(fig):
    for i, ax in enumerate(fig.axes):
        ax.text(0.5, 0.5, "ax%d" % (i+1), va="center", ha="center")
        ax.tick_params(labelbottom=False, labelleft=False)


fig = plt.figure()
fig.suptitle("Controlling subplot sizes with width_ratios and height_ratios")

gs = GridSpec(2, 2, width_ratios=[1, 2], height_ratios=[4, 1])
ax1 = fig.add_subplot(gs[0])
ax2 = fig.add_subplot(gs[1])
ax3 = fig.add_subplot(gs[2])
ax4 = fig.add_subplot(gs[3])

annotate_axes(fig)
```

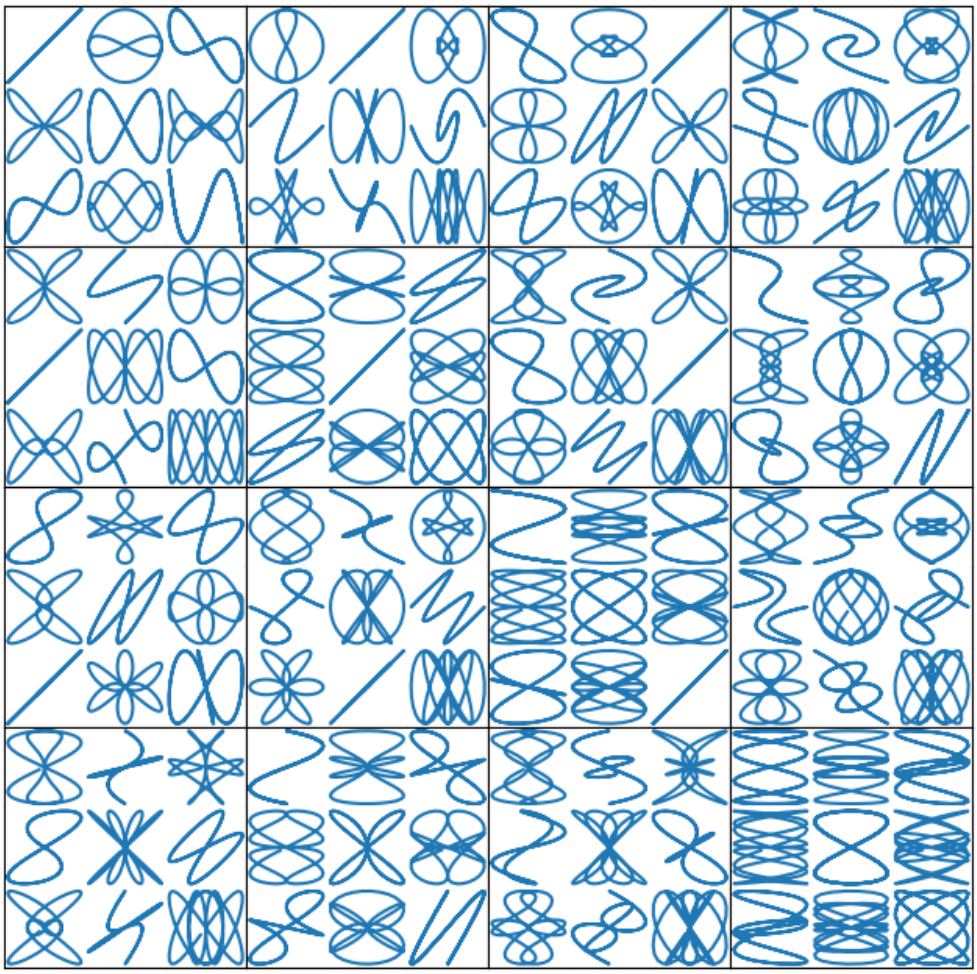



```
fig = plt.figure()
fig.suptitle("Controlling spacing around and between subplots")

gs1 = GridSpec(3, 3, left=0.05, right=0.48, wspace=0.05)
ax1 = fig.add_subplot(gs1[:-1, :])
ax2 = fig.add_subplot(gs1[-1, :-1])
ax3 = fig.add_subplot(gs1[-1, -1])

gs2 = GridSpec(3, 3, left=0.55, right=0.98, hspace=0.05)
ax4 = fig.add_subplot(gs2[:, :-1])
ax5 = fig.add_subplot(gs2[:-1, -1])
ax6 = fig.add_subplot(gs2[-1, -1])

annotate_axes(fig)

plt.show()
```




### Nested GridSpecs

This example demonstrates the use of nested *GridSpec*s.



```
import matplotlib.pyplot as plt
import numpy as np


def squiggle_xy(a, b, c, d):
    i = np.arange(0.0, 2*np.pi, 0.05)
    return np.sin(i*a)*np.cos(i*b), np.sin(i*c)*np.cos(i*d)


fig = plt.figure(figsize=(8, 8))
outer_grid = fig.add_gridspec(4, 4, wspace=0, hspace=0)

for a in range(4):
```

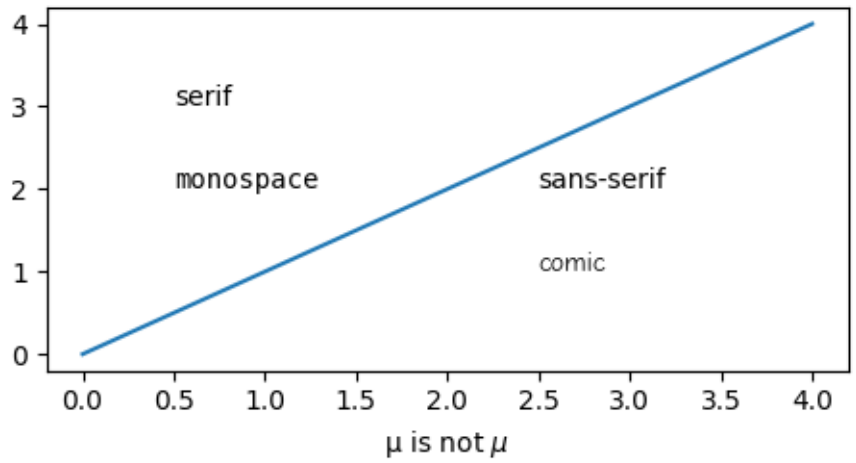
(continues on next page)

(continued from previous page)

```
    for b in range(4):
        # gridspec inside gridspec
        inner_grid = outer_grid[a, b].subgridspec(3, 3, wspace=0, hspace=0)
        axs = inner_grid.subplots()  # Create all subplots for the inner grid.
        for (c, d), ax in np.ndenumerate(axs):
            ax.plot(*squiggle_xy(a + 1, b + 1, c + 1, d + 1))
            ax.set(xticks=[], yticks=[])

# show only the outside spines
for ax in fig.get_axes():
    ss = ax.get_subplotspec()
    ax.spines.top.set_visible(ss.is_first_row())
    ax.spines.bottom.set_visible(ss.is_last_row())
    ax.spines.left.set_visible(ss.is_first_col())
    ax.spines.right.set_visible(ss.is_last_col())

plt.show()
```

**Total running time of the script:** (0 minutes 2.572 seconds)

## PGF fonts




```
findfont: Generic family 'serif' not found because none of the following␣
 ↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
 ↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
 ↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
 ↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
 ↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
 ↪families were found:
```

(continues on next page)

(continued from previous page)

```
findfont: Font family ['serif'] not found. Falling back to DejaVu Sans.
findfont: Generic family 'serif' not found because none of the following␣
 ↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
 ↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
 ↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
 ↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
 ↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
 ↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
 ↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
 ↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
 ↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
 ↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
 ↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
 ↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
 ↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
 ↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
 ↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
 ↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
 ↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
 ↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
 ↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
 ↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
 ↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
 ↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
 ↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
 ↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
 ↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
 ↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
 ↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
```

(continues on next page)

(continued from previous page)

```
↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
```

(continues on next page)

(continued from previous page)

```
↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
↪families were found:
findfont: Generic family 'serif' not found because none of the following␣
↪families were found:
```

```
import matplotlib.pyplot as plt
```

(continues on next page)

(continued from previous page)

```
plt.rcParams.update({
    "font.family": "serif",
    # Use LaTeX default serif font.
    "font.serif": [],
    # Use specific cursive fonts.
    "font.cursive": ["Comic Neue", "Comic Sans MS"],
})

fig, ax = plt.subplots(figsize=(4.5, 2.5))

ax.plot(range(5))

ax.text(0.5, 3., "serif")
ax.text(0.5, 2., "monospace", family="monospace")
ax.text(2.5, 2., "sans-serif", family="DejaVu Sans")  # Use specific sans␣
↪font.
ax.text(2.5, 1., "comic", family="cursive")
ax.set_xlabel("µ is not $\\mu$")

fig.tight_layout(pad=.5)

fig.savefig("pgf_fonts.pdf")
fig.savefig("pgf_fonts.png")
```

### PGF preamble

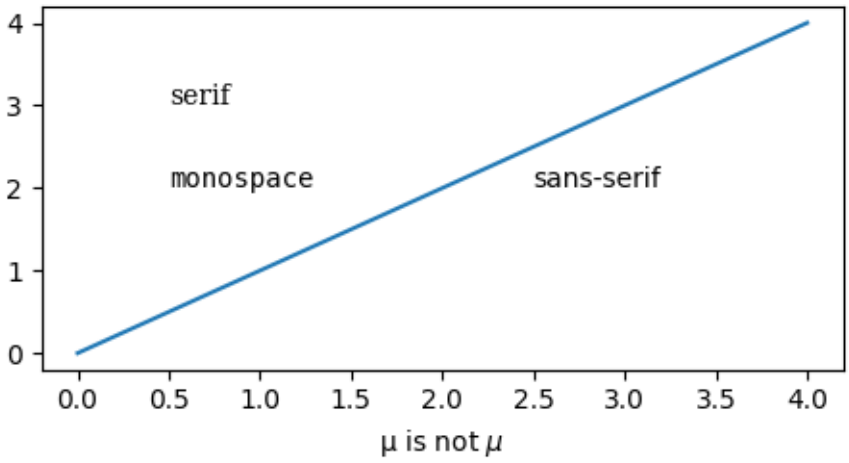
```
import matplotlib as mpl

mpl.use("pgf")
import matplotlib.pyplot as plt

plt.rcParams.update({
    "font.family": "serif",  # use serif/main font for text elements
    "text.usetex": True,     # use inline math for ticks
    "pgf.rcfonts": False,    # don't setup fonts from rc parameters
    "pgf.preamble": "\n".join([
         r"\usepackage{url}",            # load additional packages
         r"\usepackage{unicode-math}",   # unicode math setup
         r"\setmainfont{DejaVu Serif}",  # serif font via preamble
    ])
})

fig, ax = plt.subplots(figsize=(4.5, 2.5))

ax.plot(range(5))

ax.set_xlabel("unicode text: я, ψ, €, ü")
ax.set_ylabel(r"\url{https://matplotlib.org}")
ax.legend(["unicode math: $λ=∑_i^∞ µ_i^2$"])
```

(continues on next page)

(continued from previous page)

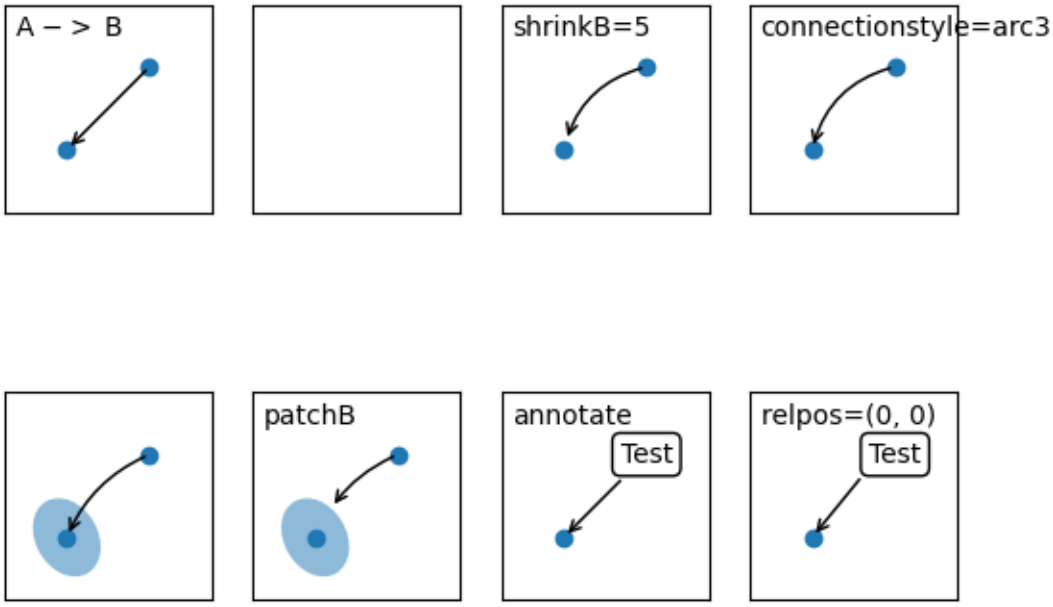
```
fig.tight_layout(pad=.5)

fig.savefig("pgf_preamble.pdf")
fig.savefig("pgf_preamble.png")
```

### PGF texsystem




```
import matplotlib.pyplot as plt

plt.rcParams.update({
    "pgf.texsystem": "pdflatex",
    "pgf.preamble": "\n".join([
         r"\usepackage[utf8x]{inputenc}",
         r"\usepackage[T1]{fontenc}",
         r"\usepackage{cmbright}",
    ]),
})

fig, ax = plt.subplots(figsize=(4.5, 2.5))

ax.plot(range(5))

ax.text(0.5, 3., "serif", family="serif")
ax.text(0.5, 2., "monospace", family="monospace")
ax.text(2.5, 2., "sans-serif", family="sans-serif")
ax.set_xlabel(r"μ is not $\mu$")

fig.tight_layout(pad=.5)

fig.savefig("pgf_texsystem.pdf")
fig.savefig("pgf_texsystem.png")
```

### Simple Annotate01




```
import matplotlib.pyplot as plt

import matplotlib.patches as mpatches

fig, axs = plt.subplots(2, 4)
x1, y1 = 0.3, 0.3
x2, y2 = 0.7, 0.7

ax = axs.flat[0]
ax.plot([x1, x2], [y1, y2], "o")
ax.annotate("",
            xy=(x1, y1), xycoords='data',
            xytext=(x2, y2), textcoords='data',
            arrowprops=dict(arrowstyle="->"))
ax.text(.05, .95, "A $->$ B",
        transform=ax.transAxes, ha="left", va="top")

ax = axs.flat[2]
ax.plot([x1, x2], [y1, y2], "o")
ax.annotate("",
            xy=(x1, y1), xycoords='data',
```

(continues on next page)

(continued from previous page)

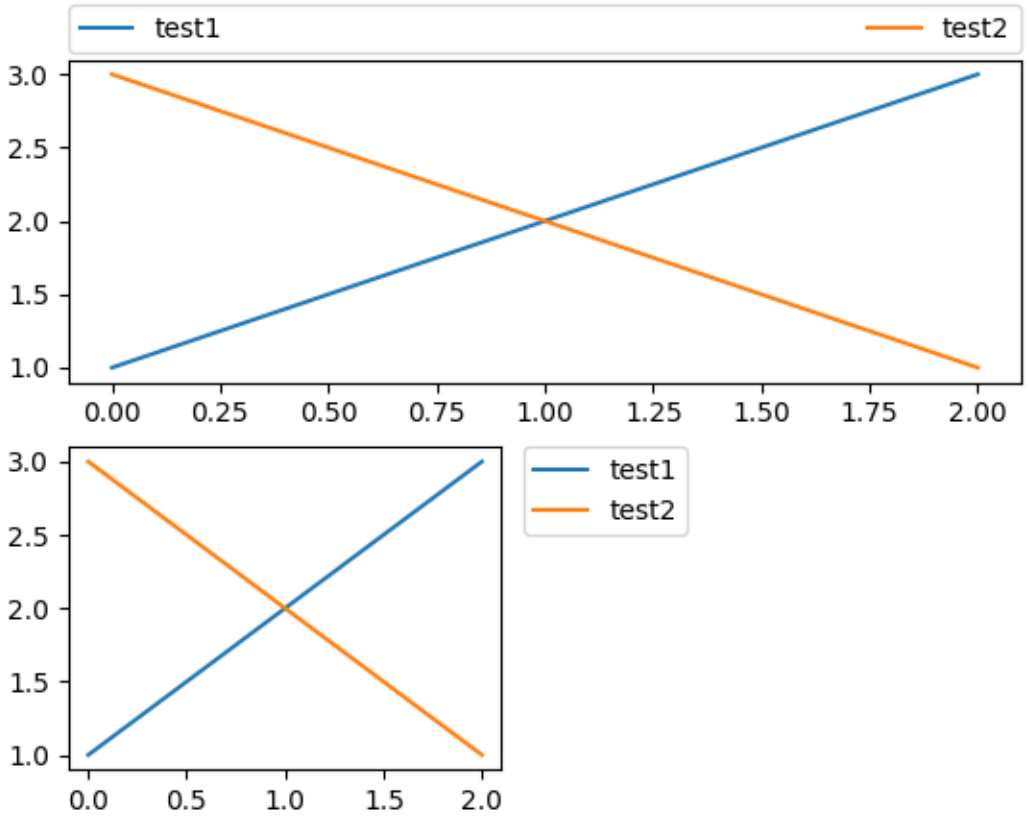
```
            xytext=(x2, y2), textcoords='data',
            arrowprops=dict(arrowstyle="->", connectionstyle="arc3,rad=0.3",
                            shrinkB=5))
ax.text(.05, .95, "shrinkB=5",
        transform=ax.transAxes, ha="left", va="top")

ax = axs.flat[3]
ax.plot([x1, x2], [y1, y2], "o")
ax.annotate("",
            xy=(x1, y1), xycoords='data',
            xytext=(x2, y2), textcoords='data',
            arrowprops=dict(arrowstyle="->", connectionstyle="arc3,rad=0.3"))
ax.text(.05, .95, "connectionstyle=arc3",
        transform=ax.transAxes, ha="left", va="top")

ax = axs.flat[4]
ax.plot([x1, x2], [y1, y2], "o")
el = mpatches.Ellipse((x1, y1), 0.3, 0.4, angle=30, alpha=0.5)
ax.add_artist(el)
ax.annotate("",
            xy=(x1, y1), xycoords='data',
            xytext=(x2, y2), textcoords='data',
            arrowprops=dict(arrowstyle="->", connectionstyle="arc3,rad=0.2"))

ax = axs.flat[5]
ax.plot([x1, x2], [y1, y2], "o")
el = mpatches.Ellipse((x1, y1), 0.3, 0.4, angle=30, alpha=0.5)
ax.add_artist(el)
ax.annotate("",
            xy=(x1, y1), xycoords='data',
            xytext=(x2, y2), textcoords='data',
            arrowprops=dict(arrowstyle="->", connectionstyle="arc3,rad=0.2",
                            patchB=el))
ax.text(.05, .95, "patchB",
        transform=ax.transAxes, ha="left", va="top")

ax = axs.flat[6]
ax.plot([x1], [y1], "o")
ax.annotate("Test",
            xy=(x1, y1), xycoords='data',
            xytext=(x2, y2), textcoords='data',
            ha="center", va="center",
            bbox=dict(boxstyle="round", fc="w"),
            arrowprops=dict(arrowstyle="->"))
ax.text(.05, .95, "annotate",
        transform=ax.transAxes, ha="left", va="top")

ax = axs.flat[7]
ax.plot([x1], [y1], "o")
ax.annotate("Test",
            xy=(x1, y1), xycoords='data',
            xytext=(x2, y2), textcoords='data',
```

(continues on next page)

(continued from previous page)

```
            ha="center", va="center",
            bbox=dict(boxstyle="round", fc="w", ),
            arrowprops=dict(arrowstyle="->", relpos=(0., 0.)))
ax.text(.05, .95, "relpos=(0, 0)",
        transform=ax.transAxes, ha="left", va="top")

for ax in axs.flat:
    ax.set(xlim=(0, 1), ylim=(0, 1), xticks=[], yticks=[], aspect=1)

plt.show()
```

### Simple Legend01



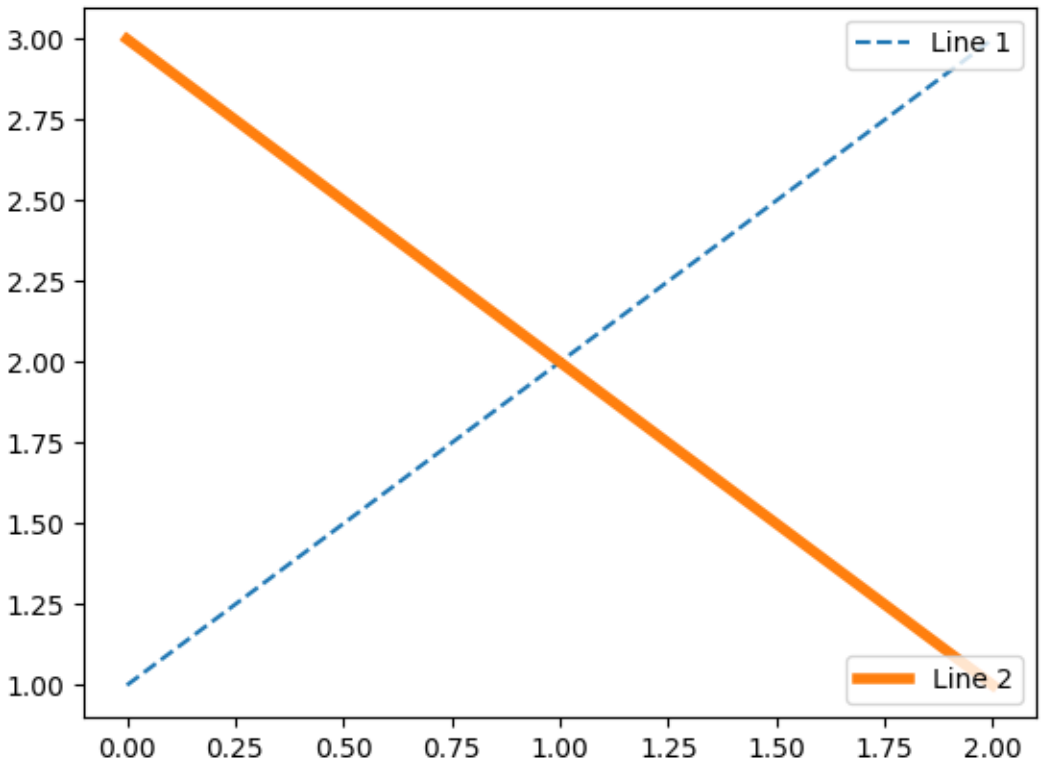

```
import matplotlib.pyplot as plt

fig = plt.figure()

ax = fig.add_subplot(211)
ax.plot([1, 2, 3], label="test1")
ax.plot([3, 2, 1], label="test2")
```

(continues on next page)

(continued from previous page)

```
# Place a legend above this subplot, expanding itself to
# fully use the given bounding box.
ax.legend(bbox_to_anchor=(0., 1.02, 1., .102), loc='lower left',
          ncols=2, mode="expand", borderaxespad=0.)

ax = fig.add_subplot(223)
ax.plot([1, 2, 3], label="test1")
ax.plot([3, 2, 1], label="test2")
# Place a legend to the right of this smaller subplot.
ax.legend(bbox_to_anchor=(1.05, 1), loc='upper left', borderaxespad=0.)

plt.show()
```

### Simple Legend02




```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()

line1, = ax.plot([1, 2, 3], label="Line 1", linestyle='--')
```

(continues on next page)

(continued from previous page)

```
line2, = ax.plot([3, 2, 1], label="Line 2", linewidth=4)

# Create a legend for the first line.
first_legend = ax.legend(handles=[line1], loc='upper right')

# Add the legend manually to the current Axes.
ax.add_artist(first_legend)

# Create another legend for the second line.
ax.legend(handles=[line2], loc='lower right')

plt.show()
```

**Reference**

CHAPTER

SEVEN

# API REFERENCE

## 7.1 Matplotlib interfaces

Matplotlib has two interfaces. See *Matplotlib Application Interfaces (APIs)* for a more detailed description of both and their recommended use cases.

**Axes interface** (object-based, explicit)

create a *Figure* and one or more *Axes* objects, then *explicitly* use methods on these objects to add data, configure limits, set labels etc.

API:

- *subplots*: create Figure and Axes
- *axes*: add data, limits, labels etc.
- *Figure*: for figure-level methods

Example:

```
fig, ax = plt.subplots()
ax.plot(x, y)
ax.set_title("Sample plot")
plt.show()
```

**pyplot interface** (function-based, implicit)

consists of functions in the *pyplot* module. Figure and Axes are manipulated through these functions and are only *implicitly* present in the background.

API:

- *matplotlib.pyplot*

Example:

```
plt.plot(x, y)
plt.title("Sample plot")
plt.show()
```

## 7.2 Modules

Alphabetical list of modules:

### 7.2.1 matplotlib

An object-oriented plotting library.

A procedural interface is provided by the companion pyplot module, which may be imported directly, e.g.:

```
import matplotlib.pyplot as plt
```

or using ipython:

```
ipython
```

at your terminal, followed by:

```
In [1]: %matplotlib
In [2]: import matplotlib.pyplot as plt
```

at the ipython shell prompt.

For the most part, direct use of the explicit object-oriented library is encouraged when programming; the implicit pyplot interface is primarily for working interactively. The exceptions to this suggestion are the pyplot functions *pyplot.figure*, *pyplot.subplot*, *pyplot.subplots*, and *pyplot.savefig*, which can greatly simplify scripting. See *Matplotlib Application Interfaces (APIs)* for an explanation of the tradeoffs between the implicit and explicit interfaces.

Modules include:

***matplotlib.axes***

The *Axes* class. Most pyplot functions are wrappers for *Axes* methods. The axes module is the highest level of OO access to the library.

***matplotlib.figure***

The *Figure* class.

***matplotlib.artist***

The *Artist* base class for all classes that draw things.

***matplotlib.lines***

The *Line2D* class for drawing lines and markers.

***matplotlib.patches***

Classes for drawing polygons.

***matplotlib.text***

The *Text* and *Annotation* classes.

***matplotlib.image***

The *AxesImage* and *FigureImage* classes.

***matplotlib.collections***

Classes for efficient drawing of groups of lines or polygons.

***matplotlib.colors***

Color specifications and making colormaps.

***matplotlib.cm***

Colormaps, and the *ScalarMappable* mixin class for providing color mapping functionality to other classes.

***matplotlib.ticker***

Calculation of tick mark locations and formatting of tick labels.

***matplotlib.backends***

A subpackage with modules for various GUI libraries and output formats.

The base matplotlib namespace includes:

***rcParams***

Default configuration settings; their defaults may be overridden using a `matplotlibrc` file.

***use***

Setting the Matplotlib backend. This should be called before any figure is created, because it is not possible to switch between different GUI backends after that.

The following environment variables can be used to customize the behavior:

***MPLBACKEND***

This optional variable can be set to choose the Matplotlib backend. See *What is a backend?*.

***MPLCONFIGDIR***

This is the directory used to store user customizations to Matplotlib, as well as some caches to improve performance. If *MPLCONFIGDIR* is not defined, *HOME*`/.config/matplotlib` and *HOME*`/.cache/matplotlib` are used on Linux, and *HOME*`/.matplotlib` on other platforms, if they are writable. Otherwise, the Python standard library's `tempfile.gettempdir` is used to find a base directory in which the `matplotlib` subdirectory is created.

Matplotlib was initially written by John D. Hunter (1968-2012) and is now developed and maintained by a host of others.

Occasionally the internal documentation (python docstrings) will refer to MATLAB®, a registered trademark of The MathWorks, Inc.

### Backend management

matplotlib.**use**(*backend*, *, *force=True*)

Select the backend used for rendering and GUI integration.

If pyplot is already imported, *switch_backend* is used and if the new backend is different than the current backend, all Figures will be closed.

**Parameters**

**backend**

[str] The backend to switch to. This can either be one of the standard backend names, which are case-insensitive:

- interactive backends: GTK3Agg, GTK3Cairo, GTK4Agg, GTK4Cairo, MacOSX, nbAgg, notebook, QtAgg, QtCairo, TkAgg, TkCairo, WebAgg, WX, WXAgg, WXCairo, Qt5Agg, Qt5Cairo
- non-interactive backends: agg, cairo, pdf, pgf, ps, svg, template

or a string of the form: `module://my.module.name`.

notebook is a synonym for nbAgg.

Switching to an interactive backend is not possible if an unrelated event loop has already been started (e.g., switching to GTK3Agg if a TkAgg window has already been opened). Switching to a non-interactive backend is always possible.

**force**

[bool, default: True] If True (the default), raise an `ImportError` if the backend cannot be set up (either because it fails to import, or because an incompatible GUI interactive framework is already running); if False, silently ignore the failure.

**See also:**

***Backends***
***matplotlib.get_backend***
***matplotlib.pyplot.switch_backend***

matplotlib.**get_backend**()

Return the name of the current backend.

**See also:**

***matplotlib.use***

matplotlib.**interactive**(*b*)

Set whether to redraw after every plotting command (e.g. *pyplot.xlabel*).

matplotlib.**is_interactive**()

Return whether to redraw after every plotting command.

**Note:** This function is only intended for use in backends. End users should use *pyplot.isinteractive* instead.

### Default values and styling

matplotlib.**rcParams**

An instance of *RcParams* for handling default Matplotlib values.

**class** matplotlib.**RcParams**(**args*, ***kwargs*)

A dict-like key-value store for config parameters, including validation.

Validating functions are defined and associated with rc parameters in *matplotlib.rcsetup*.

The list of rcParams is:

- _internal.classic_mode
- agg.path.chunksize
- animation.bitrate
- animation.codec
- animation.convert_args
- animation.convert_path
- animation.embed_limit
- animation.ffmpeg_args
- animation.ffmpeg_path
- animation.frame_format
- animation.html
- animation.writer
- axes.autolimit_mode
- axes.axisbelow
- axes.edgecolor
- axes.facecolor
- axes.formatter.limits
- axes.formatter.min_exponent
- axes.formatter.offset_threshold

- axes.formatter.use_locale
- axes.formatter.use_mathtext
- axes.formatter.useoffset
- axes.grid
- axes.grid.axis
- axes.grid.which
- axes.labelcolor
- axes.labelpad
- axes.labelsize
- axes.labelweight
- axes.linewidth
- axes.prop_cycle
- axes.spines.bottom
- axes.spines.left
- axes.spines.right
- axes.spines.top
- axes.titlecolor
- axes.titlelocation
- axes.titlepad
- axes.titlesize
- axes.titleweight
- axes.titley
- axes.unicode_minus
- axes.xmargin
- axes.ymargin
- axes.zmargin
- axes3d.automargin
- axes3d.grid
- axes3d.xaxis.panecolor
- axes3d.yaxis.panecolor
- axes3d.zaxis.panecolor
- backend

- backend_fallback
- boxplot.bootstrap
- boxplot.boxprops.color
- boxplot.boxprops.linestyle
- boxplot.boxprops.linewidth
- boxplot.capprops.color
- boxplot.capprops.linestyle
- boxplot.capprops.linewidth
- boxplot.flierprops.color
- boxplot.flierprops.linestyle
- boxplot.flierprops.linewidth
- boxplot.flierprops.marker
- boxplot.flierprops.markeredgecolor
- boxplot.flierprops.markeredgewidth
- boxplot.flierprops.markerfacecolor
- boxplot.flierprops.markersize
- boxplot.meanline
- boxplot.meanprops.color
- boxplot.meanprops.linestyle
- boxplot.meanprops.linewidth
- boxplot.meanprops.marker
- boxplot.meanprops.markeredgecolor
- boxplot.meanprops.markerfacecolor
- boxplot.meanprops.markersize
- boxplot.medianprops.color
- boxplot.medianprops.linestyle
- boxplot.medianprops.linewidth
- boxplot.notch
- boxplot.patchartist
- boxplot.showbox
- boxplot.showcaps
- boxplot.showfliers

- boxplot.showmeans
- boxplot.vertical
- boxplot.whiskerprops.color
- boxplot.whiskerprops.linestyle
- boxplot.whiskerprops.linewidth
- boxplot.whiskers
- contour.algorithm
- contour.corner_mask
- contour.linewidth
- contour.negative_linestyle
- date.autoformatter.day
- date.autoformatter.hour
- date.autoformatter.microsecond
- date.autoformatter.minute
- date.autoformatter.month
- date.autoformatter.second
- date.autoformatter.year
- date.converter
- date.epoch
- date.interval_multiples
- docstring.hardcopy
- errorbar.capsize
- figure.autolayout
- figure.constrained_layout.h_pad
- figure.constrained_layout.hspace
- figure.constrained_layout.use
- figure.constrained_layout.w_pad
- figure.constrained_layout.wspace
- figure.dpi
- figure.edgecolor
- figure.facecolor
- figure.figsize

- figure.frameon
- figure.hooks
- figure.labelsize
- figure.labelweight
- figure.max_open_warning
- figure.raise_window
- figure.subplot.bottom
- figure.subplot.hspace
- figure.subplot.left
- figure.subplot.right
- figure.subplot.top
- figure.subplot.wspace
- figure.titlesize
- figure.titleweight
- font.cursive
- font.family
- font.fantasy
- font.monospace
- font.sans-serif
- font.serif
- font.size
- font.stretch
- font.style
- font.variant
- font.weight
- grid.alpha
- grid.color
- grid.linestyle
- grid.linewidth
- hatch.color
- hatch.linewidth
- hist.bins

- image.aspect
- image.cmap
- image.composite_image
- image.interpolation
- image.interpolation_stage
- image.lut
- image.origin
- image.resample
- interactive
- keymap.back
- keymap.copy
- keymap.forward
- keymap.fullscreen
- keymap.grid
- keymap.grid_minor
- keymap.help
- keymap.home
- keymap.pan
- keymap.quit
- keymap.quit_all
- keymap.save
- keymap.xscale
- keymap.yscale
- keymap.zoom
- legend.borderaxespad
- legend.borderpad
- legend.columnspacing
- legend.edgecolor
- legend.facecolor
- legend.fancybox
- legend.fontsize
- legend.framealpha

- legend.frameon
- legend.handleheight
- legend.handlelength
- legend.handletextpad
- legend.labelcolor
- legend.labelspacing
- legend.loc
- legend.markerscale
- legend.numpoints
- legend.scatterpoints
- legend.shadow
- legend.title_fontsize
- lines.antialiased
- lines.color
- lines.dash_capstyle
- lines.dash_joinstyle
- lines.dashdot_pattern
- lines.dashed_pattern
- lines.dotted_pattern
- lines.linestyle
- lines.linewidth
- lines.marker
- lines.markeredgecolor
- lines.markeredgewidth
- lines.markerfacecolor
- lines.markersize
- lines.scale_dashes
- lines.solid_capstyle
- lines.solid_joinstyle
- macosx.window_mode
- markers.fillstyle
- mathtext.bf

- mathtext.bfit
- mathtext.cal
- mathtext.default
- mathtext.fallback
- mathtext.fontset
- mathtext.it
- mathtext.rm
- mathtext.sf
- mathtext.tt
- patch.antialiased
- patch.edgecolor
- patch.facecolor
- patch.force_edgecolor
- patch.linewidth
- path.effects
- path.simplify
- path.simplify_threshold
- path.sketch
- path.snap
- pcolor.shading
- pcolormesh.snap
- pdf.compression
- pdf.fonttype
- pdf.inheritcolor
- pdf.use14corefonts
- pgf.preamble
- pgf.rcfonts
- pgf.texsystem
- polaraxes.grid
- ps.distiller.res
- ps.fonttype
- ps.papersize

- ps.useafm
- ps.usedistiller
- savefig.bbox
- savefig.directory
- savefig.dpi
- savefig.edgecolor
- savefig.facecolor
- savefig.format
- savefig.orientation
- savefig.pad_inches
- savefig.transparent
- scatter.edgecolors
- scatter.marker
- svg.fonttype
- svg.hashsalt
- svg.image_inline
- text.antialiased
- text.color
- text.hinting
- text.hinting_factor
- text.kerning_factor
- text.latex.preamble
- text.parse_math
- text.usetex
- timezone
- tk.window_focus
- toolbar
- webagg.address
- webagg.open_in_browser
- webagg.port
- webagg.port_retries
- xaxis.labellocation

- xtick.alignment
- xtick.bottom
- xtick.color
- xtick.direction
- xtick.labelbottom
- xtick.labelcolor
- xtick.labelsize
- xtick.labeltop
- xtick.major.bottom
- xtick.major.pad
- xtick.major.size
- xtick.major.top
- xtick.major.width
- xtick.minor.bottom
- xtick.minor.ndivs
- xtick.minor.pad
- xtick.minor.size
- xtick.minor.top
- xtick.minor.visible
- xtick.minor.width
- xtick.top
- yaxis.labellocation
- ytick.alignment
- ytick.color
- ytick.direction
- ytick.labelcolor
- ytick.labelleft
- ytick.labelright
- ytick.labelsize
- ytick.left
- ytick.major.left
- ytick.major.pad

- ytick.major.right
- ytick.major.size
- ytick.major.width
- ytick.minor.left
- ytick.minor.ndivs
- ytick.minor.pad
- ytick.minor.right
- ytick.minor.size
- ytick.minor.visible
- ytick.minor.width
- ytick.right

**See also:**

*The matplotlibrc file*

**find_all**(*pattern*)

Return the subset of this RcParams dictionary whose keys match, using `re.search()`, the given `pattern`.

**Note:** Changes to the returned dictionary are *not* propagated to the parent RcParams dictionary.

**copy**()

Copy this RcParams instance.

matplotlib.**rc_context**(*rc=None*, *fname=None*)

Return a context manager for temporarily changing rcParams.

The *rcParams["backend"]* will not be reset by the context manager.

rcParams changed both through the context manager invocation and in the body of the context will be reset on context exit.

**Parameters**

**rc**

[dict] The rcParams to temporarily set.

**fname**

[str or path-like] A file with Matplotlib rc settings. If both *fname* and *rc* are given, settings from *rc* take precedence.

**See also:**

*The matplotlibrc file*

### Examples

Passing explicit values via a dict:

```
with mpl.rc_context({'interactive': False}):
    fig, ax = plt.subplots()
    ax.plot(range(3), range(3))
    fig.savefig('example.png')
    plt.close(fig)
```

Loading settings from a file:

```
with mpl.rc_context(fname='print.rc'):
    plt.plot(x, y)  # uses 'print.rc'
```

Setting in the context body:

```
with mpl.rc_context():
    # will be reset
    mpl.rcParams['lines.linewidth'] = 5
    plt.plot(x, y)
```

`matplotlib.`**`rc`**(*group*, ***kwargs*)

Set the current *rcParams*. *group* is the grouping for the rc, e.g., for `lines.linewidth` the group is `lines`, for `axes.facecolor`, the group is `axes`, and so on. Group may also be a list or tuple of group names, e.g., (*xtick*, *ytick*). *kwargs* is a dictionary attribute name/value pairs, e.g.,:

```
rc('lines', linewidth=2, color='r')
```

sets the current *rcParams* and is equivalent to:

```
rcParams['lines.linewidth'] = 2
rcParams['lines.color'] = 'r'
```

The following aliases are available to save typing for interactive users:

| Alias | Property |
|---|---|
| 'lw' | 'linewidth' |
| 'ls' | 'linestyle' |
| 'c' | 'color' |
| 'fc' | 'facecolor' |
| 'ec' | 'edgecolor' |
| 'mew' | 'markeredgewidth' |
| 'aa' | 'antialiased' |

Thus you could abbreviate the above call as:

```
rc('lines', lw=2, c='r')
```

Note you can use python's kwargs dictionary facility to store dictionaries of default parameters. e.g., you can customize the font rc as follows:

```
font = {'family' : 'monospace',
        'weight' : 'bold',
        'size'   : 'larger'}
rc('font', **font)  # pass in the font dict as kwargs
```

This enables you to easily switch between several configurations. Use `matplotlib.style.use('default')` or *rcdefaults()* to restore the default *rcParams* after changes.

### Notes

Similar functionality is available by using the normal dict interface, i.e. `rcParams.update({"lines.linewidth": 2, ...})` (but `rcParams.update` does not support abbreviations or grouping).

`matplotlib.`**`rcdefaults`**`()`

Restore the *rcParams* from Matplotlib's internal default style.

Style-blacklisted *rcParams* (defined in `matplotlib.style.core.STYLE_BLACKLIST`) are not updated.

**See also:**

***matplotlib.rc_file_defaults***

Restore the *rcParams* from the rc file originally loaded by Matplotlib.

***matplotlib.style.use***

Use a specific style file. Call `style.use('default')` to restore the default style.

`matplotlib.`**`rc_file_defaults`**`()`

Restore the *rcParams* from the original rc file loaded by Matplotlib.

Style-blacklisted *rcParams* (defined in `matplotlib.style.core.STYLE_BLACKLIST`) are not updated.

`matplotlib.`**`rc_file`**`(`*fname*, *, *use_default_template=True*`)`

Update *rcParams* from file.

Style-blacklisted *rcParams* (defined in `matplotlib.style.core.STYLE_BLACKLIST`) are not updated.

**Parameters**

**fname**

[str or path-like] A file with Matplotlib rc settings.

**use_default_template**

[bool] If True, initialize with default parameters before updating with those in the given file. If False, the current configuration persists and only the parameters specified in the file are updated.

matplotlib.**rc_params**(*fail_on_error=False*)

Construct a *RcParams* instance from the default Matplotlib rc file.

matplotlib.**rc_params_from_file**(*fname*, *fail_on_error=False*, *use_default_template=True*)

Construct a *RcParams* from file *fname*.

**Parameters**

**fname**

[str or path-like] A file with Matplotlib rc settings.

**fail_on_error**

[bool] If True, raise an error when the parser fails to convert a parameter.

**use_default_template**

[bool] If True, initialize with default parameters before updating with those in the given file. If False, the configuration class only contains the parameters specified in the file. (Useful for updating dicts.)

matplotlib.**get_configdir**()

Return the string path of the configuration directory.

The directory is chosen as follows:

1. If the MPLCONFIGDIR environment variable is supplied, choose that.
2. On Linux, follow the XDG specification and look first in `$XDG_CONFIG_HOME`, if defined, or `$HOME/.config`. On other platforms, choose `$HOME/.matplotlib`.
3. If the chosen directory exists and is writable, use that as the configuration directory.
4. Else, create a temporary directory, and use it as the configuration directory.

matplotlib.**matplotlib_fname**()

Get the location of the config file.

The file location is determined in the following order

- `$PWD/matplotlibrc`
- `$MATPLOTLIBRC` if it is not a directory
- `$MATPLOTLIBRC/matplotlibrc`
- `$MPLCONFIGDIR/matplotlibrc`
- **On Linux,**

  - `$XDG_CONFIG_HOME/matplotlib/matplotlibrc` (if `$XDG_CONFIG_HOME` is defined)
  - or `$HOME/.config/matplotlib/matplotlibrc` (if `$XDG_CONFIG_HOME` is not defined)

- On other platforms, - `$HOME/.matplotlib/matplotlibrc` if `$HOME` is defined
- Lastly, it looks in `$MATPLOTLIBDATA/matplotlibrc`, which should always exist.

`matplotlib.`**`get_data_path`**`()`

Return the path to Matplotlib data.

### Logging

`matplotlib.`**`set_loglevel`**(*level*)

Configure Matplotlib's logging levels.

Matplotlib uses the standard library `logging` framework under the root logger 'matplotlib'. This is a helper function to:

- set Matplotlib's root logger level
- set the root logger handler's level, creating the handler if it does not exist yet

Typically, one should call `set_loglevel("info")` or `set_loglevel("debug")` to get additional debugging information.

Users or applications that are installing their own logging handlers may want to directly manipulate `logging.getLogger('matplotlib')` rather than use this function.

**Parameters**

**level**

[{"notset", "debug", "info", "warning", "error", "critical"}] The log level of the handler.

#### Notes

The first time this function is called, an additional handler is attached to Matplotlib's root handler; this handler is reused every time and this function simply manipulates the logger and handler's level.

### Colormaps and color sequences

matplotlib.**colormaps**

Container for colormaps that are known to Matplotlib by name.

The universal registry instance is *matplotlib.colormaps*. There should be no need for users to instantiate *ColormapRegistry* themselves.

Read access uses a dict-like interface mapping names to *Colormap*s:

```
import matplotlib as mpl
cmap = mpl.colormaps['viridis']
```

Returned *Colormap*s are copies, so that their modification does not change the global definition of the colormap.

Additional colormaps can be added via *ColormapRegistry.register*:

```
mpl.colormaps.register(my_colormap)
```

To get a list of all registered colormaps, you can do:

```
from matplotlib import colormaps
list(colormaps)
```

matplotlib.**color_sequences**

Container for sequences of colors that are known to Matplotlib by name.

The universal registry instance is *matplotlib.color_sequences*. There should be no need for users to instantiate *ColorSequenceRegistry* themselves.

Read access uses a dict-like interface mapping names to lists of colors:

```
import matplotlib as mpl
cmap = mpl.color_sequences['tab10']
```

The returned lists are copies, so that their modification does not change the global definition of the color sequence.

Additional color sequences can be added via *ColorSequenceRegistry.register*:

```
mpl.color_sequences.register('rgb', ['r', 'g', 'b'])
```

### Miscellaneous

**class** matplotlib.**MatplotlibDeprecationWarning**

A class for issuing deprecation warnings for Matplotlib users.

matplotlib.**get_cachedir**()

Return the string path of the cache directory.

The procedure used to find the directory is the same as for *get_configdir*, except using `$XDG_CACHE_HOME/$HOME/.cache` instead.

## 7.2.2 matplotlib.afm

**Attention:** This module is considered internal.

Its use is deprecated and it will be removed in a future version.

A python interface to Adobe Font Metrics Files.

Although a number of other Python implementations exist, and may be more complete than this, it was decided not to go with them because they were either:

1) copyrighted or used a non-BSD compatible license
2) had too many dependencies and a free standing lib was needed
3) did more than needed and it was easier to write afresh rather than figure out how to get just what was needed.

It is pretty easy to use, and has no external dependencies:

```
>>> import matplotlib as mpl
>>> from pathlib import Path
>>> afm_path = Path(mpl.get_data_path(), 'fonts', 'afm', 'ptmr8a.afm')
>>>
>>> from matplotlib.afm import AFM
>>> with afm_path.open('rb') as fh:
...     afm = AFM(fh)
>>> afm.string_width_height('What the heck?')
(6220.0, 694)
>>> afm.get_fontname()
'Times-Roman'
>>> afm.get_kern_dist('A', 'f')
0
>>> afm.get_kern_dist('A', 'y')
-92.0
```

(continues on next page)

(continued from previous page)

```
>>> afm.get_bbox_char('!')
[130, -9, 238, 676]
```

As in the Adobe Font Metrics File Format Specification, all dimensions are given in units of 1/1000 of the scale factor (point size) of the font being used.

**class** matplotlib._afm.**AFM**(*fh*)

Bases: object

Parse the AFM file in file object *fh*.

**property family_name**

The font family name, e.g., 'Times'.

**get_angle**()

Return the fontangle as float.

**get_bbox_char**(*c*, *isord=False*)

**get_capheight**()

Return the cap height as float.

**get_familyname**()

Return the font family name, e.g., 'Times'.

**get_fontname**()

Return the font name, e.g., 'Times-Roman'.

**get_fullname**()

Return the font full name, e.g., 'Times-Roman'.

**get_height_char**(*c*, *isord=False*)

Get the bounding box (ink) height of character *c* (space is 0).

**get_horizontal_stem_width**()

Return the standard horizontal stem width as float, or *None* if not specified in AFM file.

**get_kern_dist**(*c1*, *c2*)

Return the kerning pair distance (possibly 0) for chars *c1* and *c2*.

**get_kern_dist_from_name**(*name1*, *name2*)

Return the kerning pair distance (possibly 0) for chars *name1* and *name2*.

**get_name_char**(*c*, *isord=False*)

Get the name of the character, i.e., ';' is 'semicolon'.

**get_str_bbox**(*s*)

Return the string bounding box.

**get_str_bbox_and_descent**(*s*)

Return the string bounding box and the maximal descent.

**get_underline_thickness**()

Return the underline thickness as float.

**get_vertical_stem_width**()

Return the standard vertical stem width as float, or *None* if not specified in AFM file.

**get_weight**()

Return the font weight, e.g., 'Bold' or 'Roman'.

**get_width_char**(*c*, *isord=False*)

Get the width of the character from the character metric WX field.

**get_width_from_char_name**(*name*)

Get the width of the character from a type1 character name.

**get_xheight**()

Return the xheight as float.

**property postscript_name**

**string_width_height**(*s*)

Return the string width (including kerning) and string height as a (*w*, *h*) tuple.

**class** matplotlib._afm.**CharMetrics**(*width*, *name*, *bbox*)

Bases: tuple

Represents the character metrics of a single character.

### Notes

The fields do currently only describe a subset of character metrics information defined in the AFM standard.

Create new instance of CharMetrics(width, name, bbox)

**bbox**

The bbox of the character (B) as a tuple (*llx*, *lly*, *urx*, *ury*).

**name**

The character name (N).

**width**

The character width (WX).

**class** matplotlib._afm.**CompositePart**(*name*, *dx*, *dy*)

Bases: tuple

Represents the information on a composite element of a composite char.

Create new instance of CompositePart(name, dx, dy)

**dx**

x-displacement of the part from the origin.

**dy**

y-displacement of the part from the origin.

**name**

Name of the part, e.g. 'acute'.

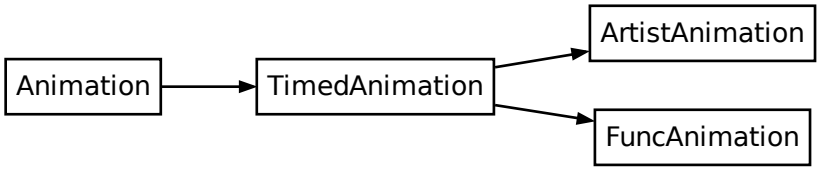
### 7.2.3 matplotlib.animation

**Table of Contents**


- *Animation*
- *Writer Classes*
- *Helper Classes*


#### Animation

The easiest way to make a live animation in Matplotlib is to use one of the `Animation` classes.




| | |
|---|---|
| `Animation` | A base class for Animations. |
| `FuncAnimation` | `TimedAnimation` subclass that makes an animation by repeatedly calling a function *func*. |
| `ArtistAnimation` | `TimedAnimation` subclass that creates an animation by using a fixed set of `Artist` objects. |

### matplotlib.animation.Animation

**class** matplotlib.animation.**Animation**(*fig*, *event_source=None*, *blit=False*)

A base class for Animations.

This class is not usable as is, and should be subclassed to provide needed behavior.

---

**Note:** You must store the created Animation in a variable that lives as long as the animation should run. Otherwise, the Animation object will be garbage-collected and the animation stops.

---

**Parameters**

**fig**

[*Figure*] The figure object used to get needed events, such as draw or resize.

**event_source**

[object, optional] A class that can run a callback when desired events are generated, as well as be stopped and started.

Examples include timers (see *TimedAnimation*) and file system notifications.

**blit**

[bool, default: False] Whether blitting is used to optimize drawing. If the backend does not support blitting, then this parameter has no effect.

**See also:**

***FuncAnimation***, ***ArtistAnimation***

**__init__**(*fig*, *event_source=None*, *blit=False*)

### Methods

| | |
|---|---|
| *__init__*(fig[, event_source, blit]) | |
| *new_frame_seq*() | Return a new sequence of frame information. |
| *new_saved_frame_seq*() | Return a new sequence of saved/cached frame information. |
| *pause*() | Pause the animation. |
| *resume*() | Resume the animation. |
| *save*(filename[, writer, fps, dpi, codec, ...]) | Save the animation as a movie file by drawing every frame. |
| *to_html5_video*([embed_limit]) | Convert the animation to an HTML5 `<video>` tag. |
| *to_jshtml*([fps, embed_frames, default_mode]) | Generate HTML representation of the animation. |

**new_frame_seq**()

Return a new sequence of frame information.

**new_saved_frame_seq**()

Return a new sequence of saved/cached frame information.

**pause**()

Pause the animation.

**resume**()

Resume the animation.

**save**(*filename*, *writer=None*, *fps=None*, *dpi=None*, *codec=None*, *bitrate=None*, *extra_args=None*, *metadata=None*, *extra_anim=None*, *savefig_kwargs=None*, *, *progress_callback=None*)

Save the animation as a movie file by drawing every frame.

**Parameters**

**filename**

[str] The output filename, e.g., `mymovie.mp4`.

**writer**

[*MovieWriter* or str, default: *rcParams["animation.writer"]* (default: `'ffmpeg'`)] A *MovieWriter* instance to use or a key that identifies a class to use, such as 'ffmpeg'.

**fps**

[int, optional] Movie frame rate (per second). If not set, the frame rate from the animation's frame interval.

**dpi**

[float, default: *rcParams["savefig.dpi"]* (default: 'figure')] Controls the dots per inch for the movie frames. Together with the figure's size in inches, this controls the size of the movie.

**codec**

[str, default: *rcParams["animation.codec"]* (default: 'h264').] The video codec to use. Not all codecs are supported by a given *MovieWriter*.

**bitrate**

[int, default: *rcParams["animation.bitrate"]* (default: -1)] The bitrate of the movie, in kilobits per second. Higher values means higher quality movies, but increase the file size. A value of -1 lets the underlying movie encoder select the bitrate.

**extra_args**

[list of str or None, optional] Extra command-line arguments passed to the underlying movie encoder. These arguments are passed last to the encoder, just before the output filename. The default, None, means to use *rcParams["animation.[name-of-encoder]_args"]* for the builtin writers.

**metadata**

[dict[str, str], default: {}] Dictionary of keys and values for metadata to include in the output file. Some keys that may be of use include: title, artist, genre, subject, copyright, srcform, comment.

**extra_anim**

[list, default: []] Additional *Animation* objects that should be included in the saved movie file. These need to be from the same *Figure* instance. Also, animation frames will just be simply combined, so there should be a 1:1 correspondence between the frames from the different animations.

**savefig_kwargs**

[dict, default: {}] Keyword arguments passed to each *savefig* call used to save the individual frames.

**progress_callback**

[function, optional] A callback function that will be called for every frame to notify the saving progress. It must have the signature

```
def func(current_frame: int, total_frames: int) -> Any
```

where *current_frame* is the current frame number and *total_frames* is the total number of frames to be saved. *total_frames* is set to None, if the total number of frames cannot be determined. Return values may exist but are ignored.

Example code to write the progress to stdout:

```
progress_callback = lambda i, n: print(f'Saving frame {i}
↪/{n}')
```

### Notes

*fps*, *codec*, *bitrate*, *extra_args* and *metadata* are used to construct a `MovieWriter` instance and can only be passed if *writer* is a string. If they are passed as non-*None* and *writer* is a `MovieWriter`, a `RuntimeError` will be raised.

**to_html5_video**(*embed_limit=None*)

Convert the animation to an HTML5 `<video>` tag.

This saves the animation as an h264 video, encoded in base64 directly into the HTML5 video tag. This respects `rcParams["animation.writer"]` (default: `'ffmpeg'`) and `rcParams["animation.bitrate"]` (default: `-1`). This also makes use of the *interval* to control the speed, and uses the *repeat* parameter to decide whether to loop.

**Parameters**

**embed_limit**

[float, optional] Limit, in MB, of the returned animation. No animation is created if the limit is exceeded. Defaults to `rcParams["animation.embed_limit"]` (default: `20.0`) = 20.0.

**Returns**

**str**

An HTML5 video tag with the animation embedded as base64 encoded h264 video. If the *embed_limit* is exceeded, this returns the string "Video too large to embed."

**to_jshtml**(*fps=None*, *embed_frames=True*, *default_mode=None*)

Generate HTML representation of the animation.

**Parameters**

**fps**

[int, optional] Movie frame rate (per second). If not set, the frame rate from the animation's frame interval.

**embed_frames**

[bool, optional]

**default_mode**

[str, optional] What to do when the animation ends. Must be one of `{'loop', 'once', 'reflect'}`. Defaults to `'loop'` if the *repeat* parameter is True, otherwise `'once'`.

### matplotlib.animation.FuncAnimation

**class** matplotlib.animation.**FuncAnimation**(*fig*, *func*, *frames=None*, *init_func=None*, *fargs=None*, *save_count=None*, *, *cache_frame_data=True*, ***kwargs*)

*TimedAnimation* subclass that makes an animation by repeatedly calling a function *func*.

---

**Note:** You must store the created Animation in a variable that lives as long as the animation should run. Otherwise, the Animation object will be garbage-collected and the animation stops.

---

#### Parameters

**fig**

[*Figure*] The figure object used to get needed events, such as draw or resize.

**func**

[callable] The function to call at each frame. The first argument will be the next value in *frames*. Any additional positional arguments can be supplied using `functools.partial` or via the *fargs* parameter.

The required signature is:

```
def func(frame, *fargs) -> iterable_of_artists
```

It is often more convenient to provide the arguments using `functools.partial`. In this way it is also possible to pass keyword arguments. To pass a function with both positional and keyword arguments, set all arguments as keyword arguments, just leaving the *frame* argument unset:

```
def func(frame, art, *, y=None):
    ...

ani = FuncAnimation(fig, partial(func, art=ln, y='foo'))
```

If `blit == True`, *func* must return an iterable of all artists that were modified or created. This information is used by the blitting algorithm to determine which parts of the figure have to be updated. The return value is unused if `blit == False` and may be omitted in that case.

**frames**

[iterable, int, generator function, or None, optional] Source of data to pass *func* and each frame of the animation

- If an iterable, then simply use the values provided. If the iterable has a length, it will override the *save_count* kwarg.
- If an integer, then equivalent to passing `range(frames)`
- If a generator function, then must have the signature:

```
def gen_function() -> obj
```

- If *None*, then equivalent to passing `itertools.count`.

In all of these cases, the values in *frames* is simply passed through to the user-supplied *func* and thus can be of any type.

**init_func**

[callable, optional] A function used to draw a clear frame. If not given, the results of drawing from the first item in the frames sequence will be used. This function will be called once before the first frame.

The required signature is:

```
def init_func() -> iterable_of_artists
```

If `blit == True`, *init_func* must return an iterable of artists to be re-drawn. This information is used by the blitting algorithm to determine which parts of the figure have to be updated. The return value is unused if `blit == False` and may be omitted in that case.

**fargs**

[tuple or None, optional] Additional arguments to pass to each call to *func*. Note: the use of `functools.partial` is preferred over *fargs*. See *func* for details.

**save_count**

[int, optional] Fallback for the number of values from *frames* to cache. This is only used if the number of frames cannot be inferred from *frames*, i.e. when it's an iterator without length or a generator.

**interval**

[int, default: 200] Delay between frames in milliseconds.

**repeat_delay**

[int, default: 0] The delay in milliseconds between consecutive animation runs, if *repeat* is True.

**repeat**

[bool, default: True] Whether the animation repeats when the sequence of frames is completed.

**blit**

[bool, default: False] Whether blitting is used to optimize drawing. Note: when using blitting, any animated artists will be drawn according to their zorder; however, they will be drawn on top of any previous artists, regardless of their zorder.

**cache_frame_data**

[bool, default: True] Whether frame data is cached. Disabling cache might be helpful when frames contain large objects.

**__init__** (*fig*, *func*, *frames=None*, *init_func=None*, *fargs=None*, *save_count=None*, *, *cache_frame_data=True*, ***kwargs*)

### Methods

| | |
|---|---|
| *__init__*(fig, func[, frames, init_func, ...]) | |
| *new_frame_seq*() | Return a new sequence of frame information. |
| *new_saved_frame_seq*() | Return a new sequence of saved/cached frame information. |
| pause() | Pause the animation. |
| resume() | Resume the animation. |
| save(filename[, writer, fps, dpi, codec, ...]) | Save the animation as a movie file by drawing every frame. |
| to_html5_video([embed_limit]) | Convert the animation to an HTML5 <video> tag. |
| to_jshtml([fps, embed_frames, default_mode]) | Generate HTML representation of the animation. |

**new_frame_seq**()

Return a new sequence of frame information.

**new_saved_frame_seq**()

Return a new sequence of saved/cached frame information.

## matplotlib.animation.ArtistAnimation

**class** matplotlib.animation.**ArtistAnimation** (*fig*, *artists*, **args*, ***kwargs*)

*TimedAnimation* subclass that creates an animation by using a fixed set of *Artist* objects.

Before creating an instance, all plotting should have taken place and the relevant artists saved.

**Note:** You must store the created Animation in a variable that lives as long as the animation should run. Otherwise, the Animation object will be garbage-collected and the animation stops.

**Parameters**

**fig**

[*Figure*] The figure object used to get needed events, such as draw or resize.

**artists**

[list] Each list entry is a collection of *Artist* objects that are made visible on the corresponding frame. Other artists are made invisible.

**interval**

[int, default: 200] Delay between frames in milliseconds.

**repeat_delay**

[int, default: 0] The delay in milliseconds between consecutive animation runs, if *repeat* is True.

**repeat**

[bool, default: True] Whether the animation repeats when the sequence of frames is completed.

**blit**

[bool, default: False] Whether blitting is used to optimize drawing.

**__init__**(*fig*, *artists*, **args*, ***kwargs*)

## Methods

| | |
|---|---|
| *__init__*(fig, artists, *args, **kwargs) | |
| new_frame_seq() | Return a new sequence of frame information. |
| new_saved_frame_seq() | Return a new sequence of saved/cached frame information. |
| pause() | Pause the animation. |
| resume() | Resume the animation. |
| save(filename[, writer, fps, dpi, codec, ...]) | Save the animation as a movie file by drawing every frame. |
| to_html5_video([embed_limit]) | Convert the animation to an HTML5 <video> tag. |
| to_jshtml([fps, embed_frames, default_mode]) | Generate HTML representation of the animation. |

In both cases it is critical to keep a reference to the instance object. The animation is advanced by a timer (typically from the host GUI framework) which the *Animation* object holds the only reference to. If you do not hold a reference to the *Animation* object, it (and hence the timers) will be garbage collected which will stop the animation.

To save an animation use *Animation.save*, *Animation.to_html5_video*, or *Animation.to_jshtml*.

See *Helper Classes* below for details about what movie formats are supported.

### FuncAnimation

The inner workings of *FuncAnimation* is more-or-less:

```
for d in frames:
    artists = func(d, *fargs)
    fig.canvas.draw_idle()
    fig.canvas.start_event_loop(interval)
```

with details to handle 'blitting' (to dramatically improve the live performance), to be non-blocking, not repeatedly start/stop the GUI event loop, handle repeats, multiple animated axes, and easily save the animation to a movie file.

'Blitting' is a standard technique in computer graphics. The general gist is to take an existing bit map (in our case a mostly rasterized figure) and then 'blit' one more artist on top. Thus, by managing a saved 'clean' bitmap, we can only re-draw the few artists that are changing at each frame and possibly save significant amounts of time. When we use blitting (by passing `blit=True`), the core loop of *FuncAnimation* gets a bit more complicated:

```
ax = fig.gca()

def update_blit(artists):
    fig.canvas.restore_region(bg_cache)
    for a in artists:
        a.axes.draw_artist(a)

    ax.figure.canvas.blit(ax.bbox)

artists = init_func()

for a in artists:
   a.set_animated(True)

fig.canvas.draw()
bg_cache = fig.canvas.copy_from_bbox(ax.bbox)

for f in frames:
    artists = func(f, *fargs)
    update_blit(artists)
    fig.canvas.start_event_loop(interval)
```

This is of course leaving out many details (such as updating the background when the figure is resized or fully re-drawn). However, this hopefully minimalist example gives a sense of how `init_func` and `func` are used inside of *FuncAnimation* and the theory of how 'blitting' works.

**Note:** The zorder of artists is not taken into account when 'blitting' because the 'blitted' artists are always

drawn on top.

The expected signature on `func` and `init_func` is very simple to keep *`FuncAnimation`* out of your book keeping and plotting logic, but this means that the callable objects you pass in must know what artists they should be working on. There are several approaches to handling this, of varying complexity and encapsulation. The simplest approach, which works quite well in the case of a script, is to define the artist at a global scope and let Python sort things out. For example:

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation

fig, ax = plt.subplots()
xdata, ydata = [], []
ln, = ax.plot([], [], 'ro')

def init():
    ax.set_xlim(0, 2*np.pi)
    ax.set_ylim(-1, 1)
    return ln,

def update(frame):
    xdata.append(frame)
    ydata.append(np.sin(frame))
    ln.set_data(xdata, ydata)
    return ln,

ani = FuncAnimation(fig, update, frames=np.linspace(0, 2*np.pi, 128),
                    init_func=init, blit=True)
plt.show()
```

The second method is to use `functools.partial` to pass arguments to the function:

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
from functools import partial

fig, ax = plt.subplots()
line1, = ax.plot([], [], 'ro')

def init():
    ax.set_xlim(0, 2*np.pi)
    ax.set_ylim(-1, 1)
    return line1,

def update(frame, ln, x, y):
    x.append(frame)
    y.append(np.sin(frame))
    ln.set_data(x, y)
    return ln,
```

(continues on next page)

(continued from previous page)

```
ani = FuncAnimation(
    fig, partial(update, ln=line1, x=[], y=[]),
    frames=np.linspace(0, 2*np.pi, 128),
    init_func=init, blit=True)

plt.show()
```

A third method is to use closures to build up the required artists and functions. A fourth method is to create a class.

## Examples

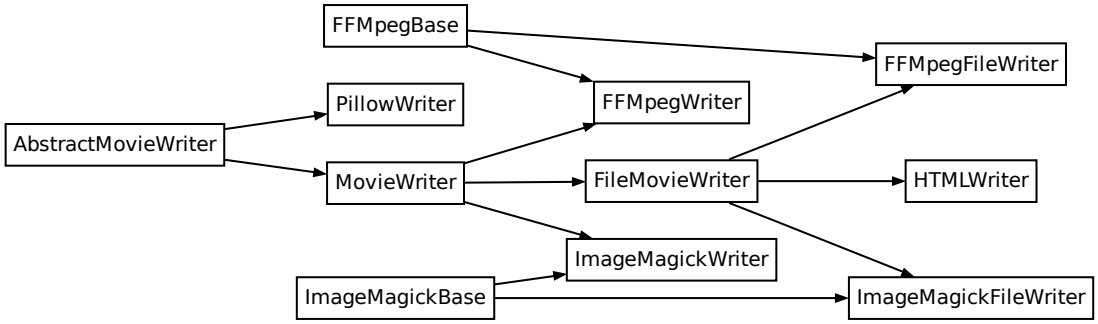
- *Decay*
- *The Bayes update*
- *The double pendulum problem*
- *Animated histogram*
- *Rain simulation*
- *Animated 3D random walk*
- *Animated line plot*
- *Oscilloscope*
- *MATPLOTLIB UNCHAINED*

### `ArtistAnimation`

## Examples

- *Animated image using a precomputed list of images*

### Writer Classes




The provided writers fall into a few broad categories.

The Pillow writer relies on the Pillow library to write the animation, keeping all data in memory.

*PillowWriter*

### matplotlib.animation.PillowWriter

**class** matplotlib.animation.**PillowWriter**(*fps=5*, *metadata=None*, *codec=None*, *bitrate=None*)

**__init__**(*fps=5*, *metadata=None*, *codec=None*, *bitrate=None*)

#### Methods

| | |
|---|---|
| *__init__*([fps, metadata, codec, bitrate]) | |
| *finish*() | Finish any processing for writing the movie. |
| *grab_frame*(**savefig_kwargs) | Grab the image information from the figure and save as a movie frame. |
| *isAvailable*() | |
| saving(fig, outfile, dpi, *args, **kwargs) | Context manager to facilitate writing the movie file. |
| *setup*(fig, outfile[, dpi]) | Setup for writing the movie file. |

### Attributes

| | |
|---|---|
| `frame_size` | A tuple `(width, height)` in pixels of a movie frame. |

**finish**()

Finish any processing for writing the movie.

**grab_frame**(***savefig_kwargs*)

Grab the image information from the figure and save as a movie frame.

All keyword arguments in *savefig_kwargs* are passed on to the *savefig* call that saves the figure. However, several keyword arguments that are supported by *savefig* may not be passed as they are controlled by the MovieWriter:

- ***dpi*, *bbox_inches*: These may not be passed because each frame of the**

  animation much be exactly the same size in pixels.

- *format*: This is controlled by the MovieWriter.

**classmethod isAvailable**()

**setup**(*fig*, *outfile*, *dpi=None*)

Setup for writing the movie file.

**Parameters**

**fig**

[*Figure*] The figure object that contains the information for frames.

**outfile**

[str] The filename of the resulting movie file.

**dpi**

[float, default: `fig.dpi`] The DPI (or resolution) for the file. This controls the size in pixels of the resulting movie file.

The HTML writer generates JavaScript-based animations.

| | |
|---|---|
| *HTMLWriter* | Writer for JavaScript-based HTML movies. |

### matplotlib.animation.HTMLWriter

**class** matplotlib.animation.**HTMLWriter**(*fps=30*, *codec=None*, *bitrate=None*, *extra_args=None*, *metadata=None*, *embed_frames=False*, *default_mode='loop'*, *embed_limit=None*)

Writer for JavaScript-based HTML movies.

**Parameters**

**fps**

[int, default: 5] Movie frame rate (per second).

**codec**

[str or None, default: *rcParams["animation.codec"]* (default: `'h264'`)] The codec to use.

**bitrate**

[int, default: *rcParams["animation.bitrate"]* (default: `-1`)] The bitrate of the movie, in kilobits per second. Higher values means higher quality movies, but increase the file size. A value of -1 lets the underlying movie encoder select the bitrate.

**extra_args**

[list of str or None, optional] Extra command-line arguments passed to the underlying movie encoder. These arguments are passed last to the encoder, just before the filename. The default, None, means to use *rcParams["animation.[name-of-encoder]_args"]* for the builtin writers.

**metadata**

[dict[str, str], default: {}] A dictionary of keys and values for metadata to include in the output file. Some keys that may be of use include: title, artist, genre, subject, copyright, srcform, comment.

**__init__**(*fps=30*, *codec=None*, *bitrate=None*, *extra_args=None*, *metadata=None*, *embed_frames=False*, *default_mode='loop'*, *embed_limit=None*)

**Parameters**

**fps**

[int, default: 5] Movie frame rate (per second).

**codec**

[str or None, default: *rcParams["animation.codec"]* (default: `'h264'`)] The codec to use.

**bitrate**

[int, default: *rcParams["animation.bitrate"]* (default: -1)] The bitrate of the movie, in kilobits per second. Higher values means higher quality movies, but increase the file size. A value of -1 lets the underlying movie encoder select the bitrate.

**extra_args**

[list of str or None, optional] Extra command-line arguments passed to the underlying movie encoder. These arguments are passed last to the encoder, just before the filename. The default, None, means to use *rcParams["animation.[name-of-encoder]_args"]* for the builtin writers.

**metadata**

[dict[str, str], default: {}] A dictionary of keys and values for metadata to include in the output file. Some keys that may be of use include: title, artist, genre, subject, copyright, srcform, comment.

## Methods

| *__init__*([fps, codec, bitrate, extra_args, ...]) | **Parameters** |
|---|---|
| bin_path() | Return the binary path to the commandline tool used by a specific subclass. |
| *finish*() | Finish any processing for writing the movie. |
| *grab_frame*(**savefig_kwargs) | Grab the image information from the figure and save as a movie frame. |
| *isAvailable*() | Return whether a MovieWriter subclass is actually available. |
| saving(fig, outfile, dpi, *args, **kwargs) | Context manager to facilitate writing the movie file. |
| *setup*(fig, outfile[, dpi, frame_dir]) | Setup for writing the movie file. |

## Attributes

| | |
|---|---|
| frame_format | Format (png, jpeg, etc.) to use for saving the frames, which can be decided by the individual subclasses. |
| frame_size | A tuple (width, height) in pixels of a movie frame. |
| *supported_formats* | |

**finish**()

Finish any processing for writing the movie.

**grab_frame**(***savefig_kwargs*)

Grab the image information from the figure and save as a movie frame.

All keyword arguments in *savefig_kwargs* are passed on to the *savefig* call that saves the figure. However, several keyword arguments that are supported by *savefig* may not be passed as they are controlled by the MovieWriter:

- ***dpi*, *bbox_inches*: These may not be passed because each frame of the**

  animation much be exactly the same size in pixels.

- *format*: This is controlled by the MovieWriter.

**classmethod isAvailable**()

Return whether a MovieWriter subclass is actually available.

**setup**(*fig*, *outfile*, *dpi=None*, *frame_dir=None*)

Setup for writing the movie file.

**Parameters**

**fig**

[*Figure*] The figure to grab the rendered frames from.

**outfile**

[str] The filename of the resulting movie file.

**dpi**

[float, default: `fig.dpi`] The dpi of the output file. This, with the figure size, controls the size in pixels of the resulting movie file.

**frame_prefix**

[str, optional] The filename prefix to use for temporary files. If *None* (the default), files are written to a temporary directory which is deleted by *finish*; if not *None*, no temporary files are deleted.

**supported_formats = ['png', 'jpeg', 'tiff', 'svg']**

The pipe-based writers stream the captured frames over a pipe to an external process. The pipe-based variants tend to be more performant, but may not work on all systems.

| | |
|---|---|
| *FFMpegWriter* | Pipe-based ffmpeg writer. |
| *ImageMagickWriter* | Pipe-based animated gif writer. |

### matplotlib.animation.FFMpegWriter

**class** matplotlib.animation.**FFMpegWriter**(*fps=5*, *codec=None*, *bitrate=None*, *extra_args=None*, *metadata=None*)

Pipe-based ffmpeg writer.

Frames are streamed directly to ffmpeg via a pipe and written in a single pass.

This effectively works as a slideshow input to ffmpeg with the fps passed as `-framerate`, so see also their notes on frame rates for further details.

**Parameters**

**fps**

[int, default: 5] Movie frame rate (per second).

**codec**

[str or None, default: *rcParams["animation.codec"]* (default: `'h264'`)] The codec to use.

**bitrate**

[int, default: *rcParams["animation.bitrate"]* (default: `-1`)] The bitrate of the movie, in kilobits per second. Higher values means higher quality movies, but increase the file size. A value of -1 lets the underlying movie encoder select the bitrate.

**extra_args**

[list of str or None, optional] Extra command-line arguments passed to the underlying movie encoder. These arguments are passed last to the encoder, just before the filename. The default, None, means to use *rcParams["animation.[name-of-encoder]_args"]* for the builtin writers.

**metadata**

[dict[str, str], default: {}] A dictionary of keys and values for metadata to include in the output file. Some keys that may be of use include: title, artist, genre, subject, copyright, srcform, comment.

**__init__**(*fps=5*, *codec=None*, *bitrate=None*, *extra_args=None*, *metadata=None*)

**Parameters**

**fps**

[int, default: 5] Movie frame rate (per second).

**codec**

[str or None, default: *rcParams["animation.codec"]* (default: `'h264'`)] The codec to use.

**bitrate**

[int, default: *rcParams["animation.bitrate"]* (default: -1)] The bitrate of the movie, in kilobits per second. Higher values means higher quality movies, but increase the file size. A value of -1 lets the underlying movie encoder select the bitrate.

**extra_args**

[list of str or None, optional] Extra command-line arguments passed to the underlying movie encoder. These arguments are passed last to the encoder, just before the filename. The default, None, means to use *rcParams["animation.[name-of-encoder]_args"]* for the builtin writers.

**metadata**

[dict[str, str], default: {}] A dictionary of keys and values for metadata to include in the output file. Some keys that may be of use include: title, artist, genre, subject, copyright, srcform, comment.

## Methods

| *__init__*([fps, codec, bitrate, extra_args, ...]) | **Parameters** |
|---|---|
| bin_path() | Return the binary path to the commandline tool used by a specific subclass. |
| finish() | Finish any processing for writing the movie. |
| grab_frame(**savefig_kwargs) | Grab the image information from the figure and save as a movie frame. |
| isAvailable() | Return whether a MovieWriter subclass is actually available. |
| saving(fig, outfile, dpi, *args, **kwargs) | Context manager to facilitate writing the movie file. |
| setup(fig, outfile[, dpi]) | Setup for writing the movie file. |

## Attributes

| | |
|---|---|
| frame_size | A tuple (width, height) in pixels of a movie frame. |
| output_args | |
| supported_formats | |

### matplotlib.animation.ImageMagickWriter

**class** matplotlib.animation.**ImageMagickWriter**(*fps=5*, *codec=None*, *bitrate=None*, *extra_args=None*, *metadata=None*)

Pipe-based animated gif writer.

Frames are streamed directly to ImageMagick via a pipe and written in a single pass.

**Parameters**

**fps**

[int, default: 5] Movie frame rate (per second).

**codec**

[str or None, default: *rcParams["animation.codec"]* (default: `'h264'`)] The codec to use.

**bitrate**

[int, default: *rcParams["animation.bitrate"]* (default: `-1`)] The bitrate of the movie, in kilobits per second. Higher values means higher quality movies, but increase the file size. A value of -1 lets the underlying movie encoder select the bitrate.

**extra_args**

[list of str or None, optional] Extra command-line arguments passed to the underlying movie encoder. These arguments are passed last to the encoder, just before the filename. The default, None, means to use *rcParams["animation.[name-of-encoder]_args"]* for the builtin writers.

**metadata**

[dict[str, str], default: {}] A dictionary of keys and values for metadata to include in the output file. Some keys that may be of use include: title, artist, genre, subject, copyright, srcform, comment.

**__init__**(*fps=5*, *codec=None*, *bitrate=None*, *extra_args=None*, *metadata=None*)

**Parameters**

**fps**

[int, default: 5] Movie frame rate (per second).

**codec**

[str or None, default: *rcParams["animation.codec"]* (default: `'h264'`)] The codec to use.

**bitrate**

[int, default: *rcParams["animation.bitrate"]* (default: -1)] The bitrate of the movie, in kilobits per second. Higher values means higher quality movies, but increase the file size. A value of -1 lets the underlying movie encoder select the bitrate.

**extra_args**

[list of str or None, optional] Extra command-line arguments passed to the underlying movie encoder. These arguments are passed last to the encoder, just before the filename. The default, None, means to use *rcParams["animation.[name-of-encoder]_args"]* for the builtin writers.

**metadata**

[dict[str, str], default: {}] A dictionary of keys and values for metadata to include in the output file. Some keys that may be of use include: title, artist, genre, subject, copyright, srcform, comment.

### Methods

| *__init__*([fps, codec, bitrate, extra_args, ...]) | **Parameters** |
|---|---|
| bin_path() | Return the binary path to the commandline tool used by a specific subclass. |
| finish() | Finish any processing for writing the movie. |
| grab_frame(**savefig_kwargs) | Grab the image information from the figure and save as a movie frame. |
| isAvailable() | Return whether a MovieWriter subclass is actually available. |
| saving(fig, outfile, dpi, *args, **kwargs) | Context manager to facilitate writing the movie file. |
| setup(fig, outfile[, dpi]) | Setup for writing the movie file. |

### Attributes

| | |
|---|---|
| frame_size | A tuple (width, height) in pixels of a movie frame. |
| *input_names* | |
| supported_formats | |

**input_names = '-'**

The file-based writers save temporary files for each frame which are stitched into a single file at the end. Although slower, these writers can be easier to debug.

| | |
|---|---|
| *FFMpegFileWriter* | File-based ffmpeg writer. |
| *ImageMagickFileWriter* | File-based animated gif writer. |

### matplotlib.animation.FFMpegFileWriter

**class** matplotlib.animation.**FFMpegFileWriter**(**args*, ***kwargs*)

File-based ffmpeg writer.

Frames are written to temporary files on disk and then stitched together at the end.

This effectively works as a slideshow input to ffmpeg with the fps passed as `-framerate`, so see also their notes on frame rates for further details.

**__init__**(**args*, ***kwargs*)

#### Methods

| | |
|---|---|
| *__init__*(*args, **kwargs) | |
| bin_path() | Return the binary path to the commandline tool used by a specific subclass. |
| finish() | Finish any processing for writing the movie. |
| grab_frame(**savefig_kwargs) | Grab the image information from the figure and save as a movie frame. |
| isAvailable() | Return whether a MovieWriter subclass is actually available. |
| saving(fig, outfile, dpi, *args, **kwargs) | Context manager to facilitate writing the movie file. |
| setup(fig, outfile[, dpi, frame_prefix]) | Setup for writing the movie file. |

### Attributes

| | |
|---|---|
| frame_format | Format (png, jpeg, etc.) to use for saving the frames, which can be decided by the individual subclasses. |
| frame_size | A tuple (width, height) in pixels of a movie frame. |
| output_args | |
| *supported_formats* | |

**supported_formats = ['png', 'jpeg', 'tiff', 'raw', 'rgba']**

## matplotlib.animation.ImageMagickFileWriter

**class** matplotlib.animation.**ImageMagickFileWriter**(**args*, ***kwargs*)

File-based animated gif writer.

Frames are written to temporary files on disk and then stitched together at the end.

**__init__**(**args*, ***kwargs*)

### Methods

| | |
|---|---|
| *__init__*(*args, **kwargs) | |
| bin_path() | Return the binary path to the commandline tool used by a specific subclass. |
| finish() | Finish any processing for writing the movie. |
| grab_frame(**savefig_kwargs) | Grab the image information from the figure and save as a movie frame. |
| isAvailable() | Return whether a MovieWriter subclass is actually available. |
| saving(fig, outfile, dpi, *args, **kwargs) | Context manager to facilitate writing the movie file. |
| setup(fig, outfile[, dpi, frame_prefix]) | Setup for writing the movie file. |

### Attributes

| | |
|---|---|
| `frame_format` | Format (png, jpeg, etc.) to use for saving the frames, which can be decided by the individual subclasses. |
| `frame_size` | A tuple `(width, height)` in pixels of a movie frame. |
| *`input_names`* | |
| *`supported_formats`* | |

**property input_names**

**supported_formats = ['png', 'jpeg', 'tiff', 'raw', 'rgba']**

The writer classes provide a way to grab sequential frames from the same underlying *`Figure`*. They all provide three methods that must be called in sequence:

- *`setup`* prepares the writer (e.g. opening a pipe). Pipe-based and file-based writers take different arguments to `setup()`.
- *`grab_frame`* can then be called as often as needed to capture a single frame at a time
- *`finish`* finalizes the movie and writes the output file to disk.

Example:

```
moviewriter = MovieWriter(...)
moviewriter.setup(fig, 'my_movie.ext', dpi=100)
for j in range(n):
    update_figure(j)
    moviewriter.grab_frame()
moviewriter.finish()
```

If using the writer classes directly (not through *`Animation.save`*), it is strongly encouraged to use the *`saving`* context manager:

```
with moviewriter.saving(fig, 'myfile.mp4', dpi=100):
    for j in range(n):
        update_figure(j)
        moviewriter.grab_frame()
```

to ensure that setup and cleanup are performed as necessary.

### Examples

- *Frame grabbing*

### Helper Classes

### Animation Base Classes

| | |
|---|---|
| `Animation` | A base class for Animations. |
| `TimedAnimation` | `Animation` subclass for time-based animation. |

### matplotlib.animation.TimedAnimation

**class** matplotlib.animation.**TimedAnimation**(*fig*, *interval=200*, *repeat_delay=0*, *repeat=True*, *event_source=None*, **args*, ***kwargs*)

`Animation` subclass for time-based animation.

A new frame is drawn every *interval* milliseconds.

**Note:** You must store the created Animation in a variable that lives as long as the animation should run. Otherwise, the Animation object will be garbage-collected and the animation stops.

**Parameters**

**fig**

[`Figure`] The figure object used to get needed events, such as draw or resize.

**interval**

[int, default: 200] Delay between frames in milliseconds.

**repeat_delay**

[int, default: 0] The delay in milliseconds between consecutive animation runs, if *repeat* is True.

**repeat**

[bool, default: True] Whether the animation repeats when the sequence of frames is completed.

**blit**

[bool, default: False] Whether blitting is used to optimize drawing.

**__init__**(*fig*, *interval=200*, *repeat_delay=0*, *repeat=True*, *event_source=None*, **args*, ***kwargs*)

### Methods

| | |
|---|---|
| *__init__*(fig[, interval, repeat_delay, ...]) | |
| new_frame_seq() | Return a new sequence of frame information. |
| new_saved_frame_seq() | Return a new sequence of saved/cached frame information. |
| pause() | Pause the animation. |
| resume() | Resume the animation. |
| save(filename[, writer, fps, dpi, codec, ...]) | Save the animation as a movie file by drawing every frame. |
| to_html5_video([embed_limit]) | Convert the animation to an HTML5 <video> tag. |
| to_jshtml([fps, embed_frames, default_mode]) | Generate HTML representation of the animation. |

## Writer Registry

A module-level registry is provided to map between the name of the writer and the class to allow a string to be passed to *Animation.save* instead of a writer instance.

| | |
|---|---|
| *MovieWriterRegistry* | Registry of available writer classes by human readable name. |

## matplotlib.animation.MovieWriterRegistry

**class** matplotlib.animation.**MovieWriterRegistry**

Registry of available writer classes by human readable name.

**__init__**()

### Methods

| | |
|---|---|
| *__init__*() | |
| *is_available*(name) | Check if given writer is available by name. |
| *list*() | Get a list of available MovieWriters. |
| *register*(name) | Decorator for registering a class under a name. |

**is_available**(*name*)

Check if given writer is available by name.

**Parameters**

**name**

[str]

**Returns**

**bool**

**list**()

Get a list of available MovieWriters.

**register**(*name*)

Decorator for registering a class under a name.

Example use:

```
@registry.register(name)
class Foo:
    pass
```

## Writer Base Classes

To reduce code duplication base classes

| | |
|---|---|
| *AbstractMovieWriter* | Abstract base class for writing movies, providing a way to grab frames by calling *grab_frame*. |
| *MovieWriter* | Base class for writing movies. |
| *FileMovieWriter* | *MovieWriter* for writing to individual files and stitching at the end. |

### matplotlib.animation.AbstractMovieWriter

**class** matplotlib.animation.**AbstractMovieWriter**(*fps=5*, *metadata=None*, *codec=None*, *bitrate=None*)

Abstract base class for writing movies, providing a way to grab frames by calling *grab_frame*.

*setup* is called to start the process and *finish* is called afterwards. *saving* is provided as a context manager to facilitate this process as

```
with moviewriter.saving(fig, outfile='myfile.mp4', dpi=100):
    # Iterate over frames
    moviewriter.grab_frame(**savefig_kwargs)
```

The use of the context manager ensures that *setup* and *finish* are performed as necessary.

An instance of a concrete subclass of this class can be given as the `writer` argument of *Animation.save()*.

**__init__**(*fps=5*, *metadata=None*, *codec=None*, *bitrate=None*)

#### Methods

| | |
|---|---|
| *__init__*([fps, metadata, codec, bitrate]) | |
| *finish*() | Finish any processing for writing the movie. |
| *grab_frame*(**savefig_kwargs) | Grab the image information from the figure and save as a movie frame. |
| *saving*(fig, outfile, dpi, *args, **kwargs) | Context manager to facilitate writing the movie file. |
| *setup*(fig, outfile[, dpi]) | Setup for writing the movie file. |

#### Attributes

| | |
|---|---|
| *frame_size* | A tuple `(width, height)` in pixels of a movie frame. |

**abstract finish**()

Finish any processing for writing the movie.

**property frame_size**

A tuple `(width, height)` in pixels of a movie frame.

**abstract grab_frame**(***savefig_kwargs*)

Grab the image information from the figure and save as a movie frame.

All keyword arguments in *savefig_kwargs* are passed on to the `savefig` call that saves the figure. However, several keyword arguments that are supported by `savefig` may not be passed as they are controlled by the MovieWriter:

- ***dpi*, *bbox_inches*: These may not be passed because each frame of the**

  animation much be exactly the same size in pixels.

- *format*: This is controlled by the MovieWriter.

**saving** (*fig*, *outfile*, *dpi*, **args*, ***kwargs*)

Context manager to facilitate writing the movie file.

`*args, **kw` are any parameters that should be passed to `setup`.

**abstract setup** (*fig*, *outfile*, *dpi=None*)

Setup for writing the movie file.

**Parameters**

**fig**

[`Figure`] The figure object that contains the information for frames.

**outfile**

[str] The filename of the resulting movie file.

**dpi**

[float, default: `fig.dpi`] The DPI (or resolution) for the file. This controls the size in pixels of the resulting movie file.

### matplotlib.animation.MovieWriter

**class** matplotlib.animation.**MovieWriter** (*fps=5*, *codec=None*, *bitrate=None*, *extra_args=None*, *metadata=None*)

Base class for writing movies.

This is a base class for MovieWriter subclasses that write a movie frame data to a pipe. You cannot instantiate this class directly. See examples for how to use its subclasses.

**Attributes**

**frame_format**

[str] The format used in writing frame data, defaults to 'rgba'.

**fig**

[`Figure`] The figure to capture data from. This must be provided by the subclasses.

**Parameters**

**fps**

[int, default: 5] Movie frame rate (per second).

**codec**

[str or None, default: `rcParams["animation.codec"]` (default: `'h264'`)] The codec to use.

**bitrate**

[int, default: `rcParams["animation.bitrate"]` (default: `-1`)] The bitrate of the movie, in kilobits per second. Higher values means higher quality movies, but increase the file size. A value of -1 lets the underlying movie encoder select the bitrate.

**extra_args**

[list of str or None, optional] Extra command-line arguments passed to the underlying movie encoder. These arguments are passed last to the encoder, just before the filename. The default, None, means to use `rcParams["animation.[name-of-encoder]_args"]` for the builtin writers.

**metadata**

[dict[str, str], default: {}] A dictionary of keys and values for metadata to include in the output file. Some keys that may be of use include: title, artist, genre, subject, copyright, srcform, comment.

**__init__**(*fps=5*, *codec=None*, *bitrate=None*, *extra_args=None*, *metadata=None*)

**Parameters**

**fps**

[int, default: 5] Movie frame rate (per second).

**codec**

[str or None, default: `rcParams["animation.codec"]` (default: `'h264'`)] The codec to use.

**bitrate**

[int, default: `rcParams["animation.bitrate"]` (default: `-1`)] The bitrate of the movie, in kilobits per second. Higher values means higher quality movies, but increase the file size. A value of -1 lets the underlying movie encoder select the bitrate.

**extra_args**

[list of str or None, optional] Extra command-line arguments passed to the underlying movie encoder. These arguments are passed last to the encoder, just before the filename. The default, None, means to use `rcParams["animation.[name-of-encoder]_args"]` for the builtin writers.

**metadata**

[dict[str, str], default: {}] A dictionary of keys and values for metadata to include in the output file. Some keys that may be of use include: title, artist, genre, subject, copyright, srcform, comment.

## Methods

| *__init__*([fps, codec, bitrate, extra_args, ...]) | **Parameters** |
|---|---|
| *bin_path*() | Return the binary path to the commandline tool used by a specific subclass. |
| *finish*() | Finish any processing for writing the movie. |
| *grab_frame*(**savefig_kwargs) | Grab the image information from the figure and save as a movie frame. |
| *isAvailable*() | Return whether a MovieWriter subclass is actually available. |
| saving(fig, outfile, dpi, *args, **kwargs) | Context manager to facilitate writing the movie file. |
| *setup*(fig, outfile[, dpi]) | Setup for writing the movie file. |

## Attributes

| | |
|---|---|
| frame_size | A tuple (width, height) in pixels of a movie frame. |
| *supported_formats* | |

**classmethod bin_path**()

Return the binary path to the commandline tool used by a specific subclass. This is a class method so that the tool can be looked for before making a particular MovieWriter subclass available.

**finish**()

Finish any processing for writing the movie.

**grab_frame**(***savefig_kwargs*)

Grab the image information from the figure and save as a movie frame.

All keyword arguments in *savefig_kwargs* are passed on to the *savefig* call that saves the figure. However, several keyword arguments that are supported by *savefig* may not be passed as they are controlled by the MovieWriter:

- ***dpi*, *bbox_inches*: These may not be passed because each frame of the**

  animation much be exactly the same size in pixels.

- *format*: This is controlled by the MovieWriter.

**classmethod isAvailable**()

Return whether a MovieWriter subclass is actually available.

**setup**(*fig*, *outfile*, *dpi=None*)

Setup for writing the movie file.

**Parameters**

**fig**

[*Figure*] The figure object that contains the information for frames.

**outfile**

[str] The filename of the resulting movie file.

**dpi**

[float, default: `fig.dpi`] The DPI (or resolution) for the file. This controls the size in pixels of the resulting movie file.

**supported_formats = ['rgba']**

### matplotlib.animation.FileMovieWriter

**class** matplotlib.animation.**FileMovieWriter**(**args*, ***kwargs*)

*MovieWriter* for writing to individual files and stitching at the end.

This must be sub-classed to be useful.

**Parameters**

**fps**

[int, default: 5] Movie frame rate (per second).

**codec**

[str or None, default: *rcParams["animation.codec"]* (default: `'h264'`)] The codec to use.

**bitrate**

[int, default: *rcParams["animation.bitrate"]* (default: `-1`)] The bitrate of the movie, in kilobits per second. Higher values means higher quality movies, but increase the file size. A value of -1 lets the underlying movie encoder select the bitrate.

**extra_args**

[list of str or None, optional] Extra command-line arguments passed to the underlying movie encoder. These arguments are passed last to the encoder, just before the filename. The default, None, means to use *rcParams["animation.[name-of-encoder]_args"]* for the builtin writers.

**metadata**

[dict[str, str], default: {}] A dictionary of keys and values for metadata to include in the output file. Some keys that may be of use include: title, artist, genre, subject, copyright, srcform, comment.

**__init__**(**args*, ***kwargs*)

**Parameters**

**fps**

[int, default: 5] Movie frame rate (per second).

**codec**

[str or None, default: *rcParams["animation.codec"]* (default: `'h264'`)] The codec to use.

**bitrate**

[int, default: *rcParams["animation.bitrate"]* (default: `-1`)] The bitrate of the movie, in kilobits per second. Higher values means higher quality movies, but increase the file size. A value of -1 lets the underlying movie encoder select the bitrate.

**extra_args**

[list of str or None, optional] Extra command-line arguments passed to the underlying movie encoder. These arguments are passed last to the encoder, just before the filename. The default, None, means to use *rcParams["animation.[name-of-encoder]_args"]* for the builtin writers.

**metadata**

[dict[str, str], default: {}] A dictionary of keys and values for metadata to include in the output file. Some keys that may be of use include: title, artist, genre, subject, copyright, srcform, comment.

### Methods

| __init__(*args, **kwargs) | Parameters |
|---|---|
| bin_path() | Return the binary path to the commandline tool used by a specific subclass. |
| finish() | Finish any processing for writing the movie. |
| grab_frame(**savefig_kwargs) | Grab the image information from the figure and save as a movie frame. |
| isAvailable() | Return whether a MovieWriter subclass is actually available. |
| saving(fig, outfile, dpi, *args, **kwargs) | Context manager to facilitate writing the movie file. |
| setup(fig, outfile[, dpi, frame_prefix]) | Setup for writing the movie file. |

### Attributes

| | |
|---|---|
| frame_format | Format (png, jpeg, etc.) to use for saving the frames, which can be decided by the individual subclasses. |
| frame_size | A tuple (width, height) in pixels of a movie frame. |
| supported_formats | |

**finish**()

Finish any processing for writing the movie.

**property frame_format**

Format (png, jpeg, etc.) to use for saving the frames, which can be decided by the individual subclasses.

**grab_frame**(***savefig_kwargs*)

Grab the image information from the figure and save as a movie frame.

All keyword arguments in *savefig_kwargs* are passed on to the *savefig* call that saves the figure. However, several keyword arguments that are supported by *savefig* may not be passed as they are controlled by the MovieWriter:

- ***dpi*, *bbox_inches*: These may not be passed because each frame of the**

  animation much be exactly the same size in pixels.

- *format*: This is controlled by the MovieWriter.

**setup** (*fig*, *outfile*, *dpi=None*, *frame_prefix=None*)

Setup for writing the movie file.

**Parameters**

**fig**

[*Figure*] The figure to grab the rendered frames from.

**outfile**

[str] The filename of the resulting movie file.

**dpi**

[float, default: `fig.dpi`] The dpi of the output file. This, with the figure size, controls the size in pixels of the resulting movie file.

**frame_prefix**

[str, optional] The filename prefix to use for temporary files. If *None* (the default), files are written to a temporary directory which is deleted by *finish*; if not *None*, no temporary files are deleted.

and mixins

| | |
|---|---|
| *FFMpegBase* | Mixin class for FFMpeg output. |
| *ImageMagickBase* | Mixin class for ImageMagick output. |

## matplotlib.animation.FFMpegBase

**class** matplotlib.animation.**FFMpegBase**

Mixin class for FFMpeg output.

This is a base class for the concrete *FFMpegWriter* and *FFMpegFileWriter* classes.

**__init__** (**args*, ***kwargs*)

### Methods

| | |
|---|---|
| *__init__*(*args, **kwargs) | |

#### Attributes

| |
|---|
| *output_args* |

**property output_args**

### matplotlib.animation.ImageMagickBase

**class** matplotlib.animation.**ImageMagickBase**

Mixin class for ImageMagick output.

This is a base class for the concrete *ImageMagickWriter* and *ImageMagickFileWriter* classes, which define an `input_names` attribute (or property) specifying the input names passed to ImageMagick.

**__init__**(**args*, ***kwargs*)

#### Methods

| |
|---|
| *__init__*(*args, **kwargs) |
| *bin_path*() |
| *isAvailable*() |

**classmethod bin_path**()

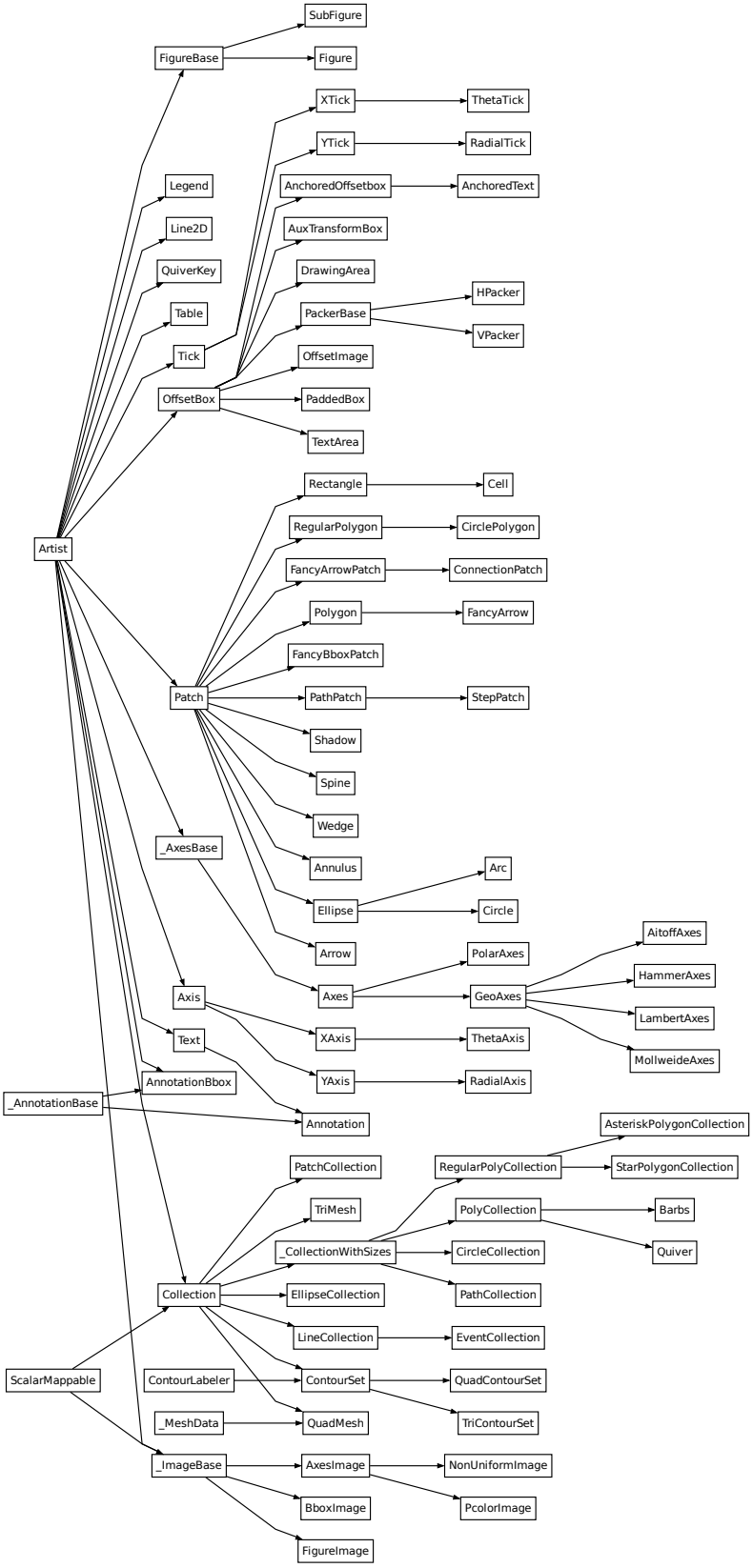
**classmethod isAvailable**()

are provided.

See the source code for how to easily implement new *MovieWriter* classes.

## 7.2.4 matplotlib.artist

## Inheritance Diagrams



### `Artist` class

**class** matplotlib.artist.**Artist**

Abstract base class for objects that render into a FigureCanvas.

Typically, all visible elements in a figure are subclasses of Artist.

### Interactive

| | |
|---|---|
| *Artist.add_callback* | Add a callback function that will be called whenever one of the *Artist*'s properties changes. |
| *Artist.remove_callback* | Remove a callback based on its observer id. |
| *Artist.pchanged* | Call all of the registered callbacks. |
| *Artist.get_cursor_data* | Return the cursor data for a given event. |
| *Artist.format_cursor_data* | Return a string representation of *data*. |
| *Artist.set_mouseover* | Set whether this artist is queried for custom context information when the mouse cursor moves over it. |
| *Artist.get_mouseover* | Return whether this artist is queried for custom context information when the mouse cursor moves over it. |
| *Artist.mouseover* | Return whether this artist is queried for custom context information when the mouse cursor moves over it. |
| *Artist.contains* | Test whether the artist contains the mouse event. |
| *Artist.pick* | Process a pick event. |
| *Artist.pickable* | Return whether the artist is pickable. |
| *Artist.set_picker* | Define the picking behavior of the artist. |
| *Artist.get_picker* | Return the picking behavior of the artist. |

### matplotlib.artist.Artist.add_callback

Artist.**add_callback**(*func*)

Add a callback function that will be called whenever one of the *Artist*'s properties changes.

**Parameters**

**func**

[callable] The callback function. It must have the signature:

```
def func(artist: Artist) -> Any
```

where *artist* is the calling *Artist*. Return values may exist but are ignored.

**Returns**

**int**

The observer id associated with the callback. This id can be used for removing the callback with *remove_callback* later.

**See also:**

***remove_callback***

### matplotlib.artist.Artist.remove_callback

Artist.**remove_callback**(*oid*)

Remove a callback based on its observer id.

**See also:**

***add_callback***

### matplotlib.artist.Artist.pchanged

Artist.**pchanged**()

Call all of the registered callbacks.

This function is triggered internally when a property is changed.

**See also:**

***add_callback***
***remove_callback***

### matplotlib.artist.Artist.get_cursor_data

Artist.**get_cursor_data**(*event*)

Return the cursor data for a given event.

---

**Note:** This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

---

Cursor data can be used by Artists to provide additional context information for a given event. The default implementation just returns *None*.

Subclasses can override the method and return arbitrary data. However, when doing so, they must ensure that *format_cursor_data* can convert the data to a string representation.

The only current use case is displaying the z-value of an *AxesImage* in the status bar of a plot window, while moving the mouse.

**Parameters**

**event**

[*MouseEvent*]

**See also:**

***format_cursor_data***

### matplotlib.artist.Artist.format_cursor_data

Artist.**format_cursor_data**(*data*)

Return a string representation of *data*.

---

**Note:** This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

---

The default implementation converts ints and floats and arrays of ints and floats into a comma-separated string enclosed in square brackets, unless the artist has an associated colorbar, in which case scalar values are formatted using the colorbar's formatter.

**See also:**

***get_cursor_data***

### matplotlib.artist.Artist.set_mouseover

Artist.**set_mouseover**(*mouseover*)

Set whether this artist is queried for custom context information when the mouse cursor moves over it.

**Parameters**

**mouseover**

[bool]

**See also:**

***get_cursor_data***
***ToolCursorPosition***
***NavigationToolbar2***

### matplotlib.artist.Artist.get_mouseover

Artist.**get_mouseover**()

Return whether this artist is queried for custom context information when the mouse cursor moves over it.

### matplotlib.artist.Artist.mouseover

**property** Artist.**mouseover**

Return whether this artist is queried for custom context information when the mouse cursor moves over it.

### matplotlib.artist.Artist.contains

Artist.**contains**(*mouseevent*)

Test whether the artist contains the mouse event.

**Parameters**

**mouseevent**

[*MouseEvent*]

**Returns**

**contains**

[bool] Whether any values are within the radius.

**details**

[dict] An artist-specific dictionary of details of the event context, such as which points are contained in the pick radius. See the individual Artist subclasses for details.

### Examples using `matplotlib.artist.Artist.contains`

- *Looking Glass*

### matplotlib.artist.Artist.pick

Artist.**pick**(*mouseevent*)

Process a pick event.

Each child artist will fire a pick event if *mouseevent* is over the artist and the artist has picker set.

**See also:**

***Artist.set_picker***, ***Artist.get_picker***, ***Artist.pickable***

### matplotlib.artist.Artist.pickable

Artist.**pickable**()

Return whether the artist is pickable.

**See also:**

***Artist.set_picker***, ***Artist.get_picker***, ***Artist.pick***

### matplotlib.artist.Artist.set_picker

Artist.**set_picker**(*picker*)

Define the picking behavior of the artist.

**Parameters**

**picker**

[None or bool or float or callable] This can be one of the following:

- *None*: Picking is disabled for this artist (default).
- A boolean: If *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist.
- A float: If picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if its data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event
- A function: If picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

to determine the hit test. if the mouse event is over the artist, return *hit=True* and props is a dictionary of properties you want added to the PickEvent attributes.

### Examples using `matplotlib.artist.Artist.set_picker`

- *Legend picking*
- *Pick event demo*

### matplotlib.artist.Artist.get_picker

Artist.**get_picker**()

Return the picking behavior of the artist.

The possible values are described in *Artist.set_picker*.

**See also:**

***Artist.set_picker***, ***Artist.pickable***, ***Artist.pick***

### Clipping

| | |
|---|---|
| *Artist.set_clip_on* | Set whether the artist uses clipping. |
| *Artist.get_clip_on* | Return whether the artist uses clipping. |
| *Artist.set_clip_box* | Set the artist's clip *Bbox*. |
| *Artist.get_clip_box* | Return the clipbox. |
| *Artist.set_clip_path* | Set the artist's clip path. |
| *Artist.get_clip_path* | Return the clip path. |

### matplotlib.artist.Artist.set_clip_on

Artist.**set_clip_on**(*b*)

Set whether the artist uses clipping.

When False, artists will be visible outside the Axes which can lead to unexpected results.

**Parameters**

**b**

[bool]

### Examples using `matplotlib.artist.Artist.set_clip_on`

- *Text alignment*

### matplotlib.artist.Artist.get_clip_on

`Artist.`**`get_clip_on`**`()`

Return whether the artist uses clipping.

### matplotlib.artist.Artist.set_clip_box

`Artist.`**`set_clip_box`**`(`*clipbox*`)`

Set the artist's clip *Bbox*.

**Parameters**

**clipbox**

[*BboxBase* or None] Will typically be created from a *TransformedBbox*. For instance, `TransformedBbox(Bbox([[0, 0], [1, 1]]), ax.transAxes)` is the default clipping for an artist added to an Axes.

### Examples using `matplotlib.artist.Artist.set_clip_box`

- *Annotating Plots*
- *Ellipse Demo*

### matplotlib.artist.Artist.get_clip_box

`Artist.`**`get_clip_box`**`()`

Return the clipbox.

### matplotlib.artist.Artist.set_clip_path

`Artist.`**`set_clip_path`**`(`*path*, *transform=None*`)`

Set the artist's clip path.

**Parameters**

**path**

[*Patch* or *Path* or *TransformedPath* or None] The clip path. If given a *Path*, *transform* must be provided as well. If *None*, a previously set clip path is removed.

**transform**

[*Transform*, optional] Only used if *path* is a *Path*, in which case the given *Path* is converted to a *TransformedPath* using *transform*.

### Notes

For efficiency, if *path* is a *Rectangle* this method will set the clipping box to the corresponding rectangle and set the clipping path to `None`.

For technical reasons (support of *set*), a tuple (*path*, *transform*) is also accepted as a single positional parameter.

## Examples using `matplotlib.artist.Artist.set_clip_path`

- *Clipping images with patches*
- *Many ways to plot images*
- *Dolphins*
- *Looking Glass*

## matplotlib.artist.Artist.get_clip_path

`Artist.`**`get_clip_path`**`()`

Return the clip path.

## Bulk Properties

| | |
|---|---|
| *Artist.update* | Update this artist's properties from the dict *props*. |
| *Artist.update_from* | Copy properties from *other* to *self*. |
| *Artist.properties* | Return a dictionary of all the properties of the artist. |
| *Artist.set* | Set multiple properties at once. |

## matplotlib.artist.Artist.update

`Artist.`**`update`**`(`*`props`*`)`

Update this artist's properties from the dict *props*.

**Parameters**

**props**

[dict]

### matplotlib.artist.Artist.update_from

Artist.**update_from**(*other*)

Copy properties from *other* to *self*.

### Examples using `matplotlib.artist.Artist.update_from`

- *Poly Editor*

### matplotlib.artist.Artist.properties

Artist.**properties**()

Return a dictionary of all the properties of the artist.

### matplotlib.artist.Artist.set

Artist.**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *gid=<UNSET>*, *in_layout=<UNSET>*, *label=<UNSET>*, *mouseover=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| agg_fil | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image |
| alpha | scalar or None |
| animated | bool |
| clip_bo | BboxBase or None |
| clip_on | bool |
| clip_pa | Patch or (Path, Transform) or None |
| figure | Figure |
| gid | str |
| in_layo | bool |
| label | object |
| mouseov | bool |
| path_ef | list of AbstractPathEffect |
| picker | None or bool or float or callable |
| rasterized | bool |
| sketch_ | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| transform | Transform |
| url | str |
| visible | bool |
| zorder | float |

### Examples using `matplotlib.artist.Artist.set`

- *Bar Label Demo*
- *Curve with error band*
- *Bar chart with gradients*
- *Scatter plot with histograms*
- *Simple Plot*
- *Creating a timeline with lines, dates, and text*
- *Contouring the solution space of optimizations*
- *Contour plot of irregularly spaced data*
- *Axes Demo*

- *Creating multiple subplots using plt.subplots*
- *Boxplots*
- *Hexagonal binned plot*
- *Nested pie charts*
- *Angle annotations on bracket arrows*
- *Annotating Plots*
- *Arrow Demo*
- *Date tick labels*
- *Text Commands*
- *Usetex Baseline Test*
- *Reference for Matplotlib artists*
- *Ellipse Demo*
- *Drawing fancy boxes*
- *Demo Axes Grid*
- *Inset locator demo*
- *axis_direction demo*
- *Parasite Axes demo*
- *Parasite axis demo*
- *Anatomy of a figure*
- *Animated 3D random walk*
- *Pick event demo*
- *Viewlims*
- *Zoom Window*
- *Manual Contour*
- *Plotting with keywords*
- *Patheffect Demo*
- *TickedStroke patheffect*
- *3D box surface plot*
- *Project contour profiles onto a graph*
- *Project filled contour onto a graph*
- *Generate polygons to fill under 3D line graph*
- *3D stem*

- *3D voxel / volumetric plot with RGB colors*
- *Log Demo*
- *Multiple y-axis with Spines*
- *Mouse Cursor*
- *Annotate Explain*
- *Connection styles for annotations*
- *Nested GridSpecs*
- *Simple Annotate01*
- *The Lifecycle of a Plot*
- *plot(x, y)*
- *scatter(x, y)*
- *bar(x, height)*
- *stem(x, y)*
- *fill_between(x, y1, y2)*
- *stackplot(x, y)*
- *stairs(values)*
- *hist(x)*
- *boxplot(X)*
- *errorbar(x, y, yerr, xerr)*
- *violinplot(D)*
- *eventplot(D)*
- *hist2d(x, y)*
- *hexbin(x, y, C)*
- *pie(x)*
- *barbs(X, Y, U, V)*
- *quiver(X, Y, U, V)*
- *tricontour(x, y, z)*
- *tricontourf(x, y, z)*
- *tripcolor(x, y, z)*
- *triplot(x, y)*
- *scatter(xs, ys, zs)*
- *plot_surface(X, Y, Z)*

- *plot_trisurf(x, y, z)*
- *voxels([x, y, z], filled)*
- *plot_wireframe(X, Y, Z)*
- *Animations using Matplotlib*
- *Arranging multiple Axes in a Figure*
- *Annotations*

### Drawing

| | |
|---|---|
| `Artist.draw` | Draw the Artist (and its children) using the given renderer. |
| `Artist.set_animated` | Set whether the artist is intended to be used in an animation. |
| `Artist.get_animated` | Return whether the artist is animated. |
| `Artist.set_alpha` | Set the alpha value used for blending - not supported on all backends. |
| `Artist.get_alpha` | Return the alpha value used for blending - not supported on all backends. |
| `Artist.set_snap` | Set the snapping behavior. |
| `Artist.get_snap` | Return the snap setting. |
| `Artist.set_visible` | Set the artist's visibility. |
| `Artist.get_visible` | Return the visibility. |
| `Artist.zorder` | |
| `Artist.set_zorder` | Set the zorder for the artist. |
| `Artist.get_zorder` | Return the artist's zorder. |
| `Artist.set_agg_filter` | Set the agg filter. |
| `Artist.set_sketch_params` | Set the sketch parameters. |
| `Artist.get_sketch_params` | Return the sketch parameters for the artist. |
| `Artist.set_rasterized` | Force rasterized (bitmap) drawing for vector graphics output. |
| `Artist.get_rasterized` | Return whether the artist is to be rasterized. |
| `Artist.set_path_effects` | Set the path effects. |
| `Artist.get_path_effects` | |
| `Artist.get_agg_filter` | Return filter function to be used for agg filter. |
| `Artist.get_window_extent` | Get the artist's bounding box in display space. |
| `Artist.get_tightbbox` | Like `Artist.get_window_extent`, but includes any clipping. |
| `Artist.get_transformed_clip_path_ar` | Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation. |

### matplotlib.artist.Artist.draw

Artist.**draw**(*renderer*)

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

**Parameters**

**renderer**

[*RendererBase* subclass.]

#### Notes

This method is overridden in the Artist subclasses.

### Examples using `matplotlib.artist.Artist.draw`

- *Artist within an artist*

### matplotlib.artist.Artist.set_animated

Artist.**set_animated**(*b*)

Set whether the artist is intended to be used in an animation.

If True, the artist is excluded from regular drawing of the figure. You have to call *Figure.draw_artist* / *Axes.draw_artist* explicitly on the artist. This approach is used to speed up animations using blitting.

See also *matplotlib.animation* and *Faster rendering by using blitting*.

**Parameters**

**b**

[bool]

### matplotlib.artist.Artist.get_animated

Artist.**get_animated**()

Return whether the artist is animated.

### matplotlib.artist.Artist.set_alpha

Artist.**set_alpha**(*alpha*)

Set the alpha value used for blending - not supported on all backends.

**Parameters**

**alpha**

[scalar or None] *alpha* must be within the 0-1 range, inclusive.

### Examples using `matplotlib.artist.Artist.set_alpha`

- *Violin plot customization*
- *Ellipse Demo*
- *Legend picking*
- *violinplot(D)*

### matplotlib.artist.Artist.get_alpha

Artist.**get_alpha**()

Return the alpha value used for blending - not supported on all backends.

### matplotlib.artist.Artist.set_snap

Artist.**set_snap**(*snap*)

Set the snapping behavior.

Snapping aligns positions with the pixel grid, which results in clearer images. For example, if a black line of 1px width was defined at a position in between two pixels, the resulting image would contain the interpolated value of that line in the pixel grid, which would be a grey value on both adjacent pixel positions. In contrast, snapping will move the line to the nearest integer pixel value, so that the resulting image will really contain a 1px wide black line.

Snapping is currently only supported by the Agg and MacOSX backends.

**Parameters**

**snap**

[bool or None] Possible values:

- *True*: Snap vertices to the nearest pixel center.
- *False*: Do not modify vertex positions.

- *None*: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center.

### matplotlib.artist.Artist.get_snap

Artist.**get_snap**()

Return the snap setting.

See *set_snap* for details.

### matplotlib.artist.Artist.set_visible

Artist.**set_visible**(*b*)

Set the artist's visibility.

**Parameters**

**b**

[bool]

### Examples using `matplotlib.artist.Artist.set_visible`

- *Creating a timeline with lines, dates, and text*
- *Broken Axis*
- *axis_direction demo*
- *Simple Axis Pad*
- *Pan/zoom events of overlapping axes*
- *Stock prices over 32 years*
- *Multiple Axes animation*
- *Keypress event*
- *Legend picking*
- *Spines*
- *Nested GridSpecs*
- *Tight layout guide*
- *Arranging multiple Axes in a Figure*

### matplotlib.artist.Artist.get_visible

Artist.**get_visible**()
Return the visibility.

### Examples using matplotlib.artist.Artist.get_visible

- *Keypress event*
- *Legend picking*

### matplotlib.artist.Artist.zorder

Artist.**zorder = 0**

### matplotlib.artist.Artist.set_zorder

Artist.**set_zorder**(*level*)
Set the zorder for the artist. Artists with lower zorder values are drawn first.

**Parameters**

**level**

[float]

### Examples using matplotlib.artist.Artist.set_zorder

- *SVG filter pie*
- *Zorder Demo*

### matplotlib.artist.Artist.get_zorder

Artist.**get_zorder**()
Return the artist's zorder.

### Examples using `matplotlib.artist.Artist.get_zorder`

- *SVG Filter Line*
- *SVG filter pie*
- *Transformations Tutorial*

## matplotlib.artist.Artist.set_agg_filter

`Artist.`**`set_agg_filter`**(*filter_func*)

Set the agg filter.

**Parameters**

**filter_func**

[callable] A filter function, which takes a (m, n, depth) float array and a dpi value, and returns a (m, n, depth) array and two offsets from the bottom left corner of the image

## matplotlib.artist.Artist.set_sketch_params

`Artist.`**`set_sketch_params`**(*scale=None*, *length=None*, *randomness=None*)

Set the sketch parameters.

**Parameters**

**scale**

[float, optional] The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is `None`, or not provided, no sketch filter will be provided.

**length**

[float, optional] The length of the wiggle along the line, in pixels (default 128.0)

**randomness**

[float, optional] The scale factor by which the length is shrunken or expanded (default 16.0)

The PGF backend uses this argument as an RNG seed and not as described above. Using the same seed yields the same random shape.

### matplotlib.artist.Artist.get_sketch_params

Artist.**get_sketch_params**()

Return the sketch parameters for the artist.

**Returns**

**tuple or None**

A 3-tuple with the following elements:

- *scale*: The amplitude of the wiggle perpendicular to the source line.
- *length*: The length of the wiggle along the line.
- *randomness*: The scale factor by which the length is shrunken or expanded.

Returns *None* if no sketch parameters were set.

### matplotlib.artist.Artist.set_rasterized

Artist.**set_rasterized**(*rasterized*)

Force rasterized (bitmap) drawing for vector graphics output.

Rasterized drawing is not supported by all artists. If you try to enable this on an artist that does not support it, the command has no effect and a warning will be issued.

This setting is ignored for pixel-based output.

See also *Rasterization for vector graphics*.

**Parameters**

**rasterized**

[bool]

### Examples using matplotlib.artist.Artist.set_rasterized

- *Rasterization for vector graphics*

### matplotlib.artist.Artist.get_rasterized

Artist.**get_rasterized**()

Return whether the artist is to be rasterized.

### matplotlib.artist.Artist.set_path_effects

Artist.**set_path_effects**(*path_effects*)

Set the path effects.

**Parameters**

**path_effects**

[list of *AbstractPathEffect*]

### Examples using `matplotlib.artist.Artist.set_path_effects`

- *Patheffect Demo*
- *Path effects guide*

### matplotlib.artist.Artist.get_path_effects

Artist.**get_path_effects**()

### matplotlib.artist.Artist.get_agg_filter

Artist.**get_agg_filter**()

Return filter function to be used for agg filter.

### matplotlib.artist.Artist.get_window_extent

Artist.**get_window_extent**(*renderer=None*)

Get the artist's bounding box in display space.

The bounding box' width and height are nonnegative.

Subclasses should override for inclusion in the bounding box "tight" calculation. Default is to return an empty bounding box at 0, 0.

Be careful when using this function, the results will not update if the artist window extent of the artist changes. The extent can change due to any changes in the transform stack, such as changing the Axes limits, the figure size, or the canvas used (as is done when saving a figure). This can lead to unexpected behavior where interactive figures will look fine on the screen, but will save incorrectly.

### Examples using `matplotlib.artist.Artist.get_window_extent`

- *BboxImage Demo*
- *Programmatically controlling subplot adjustment*
- *Annotations*

## matplotlib.artist.Artist.get_tightbbox

Artist.**get_tightbbox**(*renderer=None*)

Like *Artist.get_window_extent*, but includes any clipping.

**Parameters**

**renderer**

[*RendererBase* subclass, optional] renderer that will be used to draw the figures (i.e. `fig.canvas.get_renderer()`)

**Returns**

***Bbox* or None**

The enclosing bounding box (in figure pixel coordinates). Returns None if clipping results in no intersection.

## matplotlib.artist.Artist.get_transformed_clip_path_and_affine

Artist.**get_transformed_clip_path_and_affine**()

Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

## Figure and Axes

| | |
|---|---|
| *Artist.remove* | Remove the artist from the figure if possible. |
| *Artist.axes* | The *Axes* instance the artist resides in, or *None*. |
| *Artist.set_figure* | Set the *Figure* instance the artist belongs to. |
| *Artist.get_figure* | Return the *Figure* instance the artist belongs to. |

### matplotlib.artist.Artist.remove

Artist.**remove**()

Remove the artist from the figure if possible.

The effect will not be visible until the figure is redrawn, e.g., with *FigureCanvasBase.draw_idle*. Call *relim* to update the Axes limits if desired.

Note: *relim* will not see collections even if the collection was added to the Axes with *autolim* = True.

Note: there is no support for removing the artist's legend entry.

### Examples using matplotlib.artist.Artist.remove

- *Combining two subplots using subplots and GridSpec*
- *Figure subfigures*
- *Artist tutorial*

### matplotlib.artist.Artist.axes

**property** Artist.**axes**

The *Axes* instance the artist resides in, or *None*.

### matplotlib.artist.Artist.set_figure

Artist.**set_figure**(*fig*)

Set the *Figure* instance the artist belongs to.

**Parameters**

**fig**

[*Figure*]

### matplotlib.artist.Artist.get_figure

Artist.**get_figure**()

Return the *Figure* instance the artist belongs to.

### Children

| | |
|---|---|
| *Artist.get_children* | Return a list of the child *Artist*s of this *Artist*. |
| *Artist.findobj* | Find artist objects. |

### matplotlib.artist.Artist.get_children

Artist.**get_children**()

Return a list of the child *Artist*s of this *Artist*.

### matplotlib.artist.Artist.findobj

Artist.**findobj**(*match=None*, *include_self=True*)

Find artist objects.

Recursively find all *Artist* instances contained in the artist.

**Parameters**

**match**

A filter criterion for the matches. This can be

- *None*: Return all objects contained in artist.
- A function with signature `def match(artist: Artist) -> bool`. The result will only contain artists for which the function returns *True*.
- A class instance: e.g., *Line2D*. The result will only contain artists of this class or its subclasses (`isinstance` check).

**include_self**

[bool] Include *self* in the list to be checked for a match.

**Returns**

**list of *Artist***

### Examples using `matplotlib.artist.Artist.findobj`

- *Findobj Demo*

### Transform

| | |
|---|---|
| *Artist.set_transform* | Set the artist transform. |
| *Artist.get_transform* | Return the *Transform* instance used by this artist. |
| *Artist.is_transform_set* | Return whether the Artist has an explicitly set transform. |

### matplotlib.artist.Artist.set_transform

Artist.**set_transform**(*t*)

Set the artist transform.

**Parameters**

**t**

[*Transform*]

### Examples using `matplotlib.artist.Artist.set_transform`

- *Artist within an artist*
- *Text alignment*
- *Line, Poly and RegularPoly Collection with autoscaling*

### matplotlib.artist.Artist.get_transform

Artist.**get_transform**()

Return the *Transform* instance used by this artist.

### Examples using `matplotlib.artist.Artist.get_transform`

- *SVG Filter Line*
- *Axis scales*

### matplotlib.artist.Artist.is_transform_set

Artist.**is_transform_set**()

Return whether the Artist has an explicitly set transform.

This is *True* after *set_transform* has been called.

### Units

| | |
|---|---|
| *Artist.convert_xunits* | Convert *x* using the unit type of the xaxis. |
| *Artist.convert_yunits* | Convert *y* using the unit type of the yaxis. |
| *Artist.have_units* | Return whether units are set on any axis. |

### matplotlib.artist.Artist.convert_xunits

Artist.**convert_xunits**(*x*)

Convert *x* using the unit type of the xaxis.

If the artist is not contained in an Axes or if the xaxis does not have units, *x* itself is returned.

### matplotlib.artist.Artist.convert_yunits

Artist.**convert_yunits**(*y*)

Convert *y* using the unit type of the yaxis.

If the artist is not contained in an Axes or if the yaxis does not have units, *y* itself is returned.

### matplotlib.artist.Artist.have_units

Artist.**have_units**()

Return whether units are set on any axis.

### Metadata

| | |
|---|---|
| *Artist.set_gid* | Set the (group) id for the artist. |
| *Artist.get_gid* | Return the group id. |
| *Artist.set_label* | Set a label that will be displayed in the legend. |
| *Artist.get_label* | Return the label used for this artist in the legend. |
| *Artist.set_url* | Set the url for the artist. |
| *Artist.get_url* | Return the url. |

### matplotlib.artist.Artist.set_gid

Artist.**set_gid**(*gid*)

Set the (group) id for the artist.

**Parameters**

**gid**

[str]

### Examples using `matplotlib.artist.Artist.set_gid`

- *SVG filter pie*

### matplotlib.artist.Artist.get_gid

Artist.**get_gid**()

Return the group id.

### Examples using `matplotlib.artist.Artist.get_gid`

- *SVG filter pie*

### matplotlib.artist.Artist.set_label

Artist.**set_label**(*s*)

Set a label that will be displayed in the legend.

**Parameters**

**s**

[object] *s* will be converted to a string by calling `str`.

### matplotlib.artist.Artist.get_label

Artist.**get_label**()

Return the label used for this artist in the legend.

### Examples using `matplotlib.artist.Artist.get_label`

- *Parasite Simple*
- *SVG Filter Line*
- *SVG filter pie*

### matplotlib.artist.Artist.set_url

Artist.**set_url**(*url*)

Set the url for the artist.

**Parameters**

**url**

[str]

### matplotlib.artist.Artist.get_url

Artist.**get_url**()

Return the url.

### Miscellaneous

| | |
|---|---|
| *Artist.sticky_edges* | x and y sticky edge lists for autoscaling. |
| *Artist.set_in_layout* | Set if artist is to be included in layout calculations, E.g. |
| *Artist.get_in_layout* | Return boolean flag, True if artist is included in layout calculations. |
| *Artist.stale* | Whether the artist is 'stale' and needs to be re-drawn for the output to match the internal state of the artist. |

### matplotlib.artist.Artist.sticky_edges

**property** Artist.**sticky_edges**

`x` and `y` sticky edge lists for autoscaling.

When performing autoscaling, if a data limit coincides with a value in the corresponding sticky_edges list, then no margin will be added--the view limit "sticks" to the edge. A typical use case is histograms, where one usually expects no margin on the bottom edge (0) of the histogram.

Moreover, margin expansion "bumps" against sticky edges and cannot cross them. For example, if the upper data limit is 1.0, the upper view limit computed by simple margin application is 1.2, but there is a sticky edge at 1.1, then the actual upper view limit will be 1.1.

This attribute cannot be assigned to; however, the `x` and `y` lists can be modified in place as needed.

#### Examples

```
>>> artist.sticky_edges.x[:] = (xmin, xmax)
>>> artist.sticky_edges.y[:] = (ymin, ymax)
```

### matplotlib.artist.Artist.set_in_layout

Artist.**set_in_layout**(*in_layout*)

Set if artist is to be included in layout calculations, E.g. *Constrained layout guide*, *Figure.tight_layout()*, and `fig.savefig(fname, bbox_inches='tight')`.

**Parameters**

**in_layout**

[bool]

### Examples using `matplotlib.artist.Artist.set_in_layout`

- *Constrained layout guide*
- *Tight layout guide*

### matplotlib.artist.Artist.get_in_layout

Artist.**get_in_layout**()

Return boolean flag, `True` if artist is included in layout calculations.

E.g. *Constrained layout guide*, *`Figure.tight_layout()`*, and `fig.savefig(fname, bbox_inches='tight')`.

### matplotlib.artist.Artist.stale

**property** Artist.**stale**

Whether the artist is 'stale' and needs to be re-drawn for the output to match the internal state of the artist.

## Functions

| | |
|---|---|
| *`allow_rasterization`* | Decorator for Artist.draw method. |
| *`get`* | Return the value of an *`Artist`*'s *property*, or print all of them. |
| *`getp`* | Return the value of an *`Artist`*'s *property*, or print all of them. |
| *`setp`* | Set one or more properties on an *`Artist`*, or list allowed values. |
| *`kwdoc`* | Inspect an *`Artist`* class (using *`ArtistInspector`*) and return information about its settable properties and their current values. |
| *`ArtistInspector`* | A helper class to inspect an *`Artist`* and return information about its settable properties and their current values. |

### matplotlib.artist.allow_rasterization

matplotlib.artist.**allow_rasterization**(*draw*)

Decorator for Artist.draw method. Provides routines that run before and after the draw call. The before and after functions are useful for changing artist-dependent renderer attributes or making other setup function calls, such as starting and flushing a mixed-mode renderer.

### matplotlib.artist.get

matplotlib.artist.**get**(*obj*, *property=None*)

Return the value of an *Artist*'s *property*, or print all of them.

**Parameters**

**obj**

[*Artist*] The queried artist; e.g., a *Line2D*, a *Text*, or an *Axes*.

**property**

[str or None, default: None] If *property* is 'somename', this function returns `obj.get_somename()`.

If it's None (or unset), it *prints* all gettable properties from *obj*. Many properties have aliases for shorter typing, e.g. 'lw' is an alias for 'linewidth'. In the output, aliases and full property names will be listed as:

property or alias = value

e.g.:

linewidth or lw = 2

**See also:**

***setp***

### matplotlib.artist.getp

matplotlib.artist.**getp**(*obj*, *property=None*)

Return the value of an *Artist*'s *property*, or print all of them.

**Parameters**

**obj**

[*Artist*] The queried artist; e.g., a *Line2D*, a *Text*, or an *Axes*.

**property**

[str or None, default: None] If *property* is 'somename', this function returns `obj.get_somename()`.

If it's None (or unset), it *prints* all gettable properties from *obj*. Many properties have aliases for shorter typing, e.g. 'lw' is an alias for 'linewidth'. In the output, aliases and full property names will be listed as:

property or alias = value

e.g.:

linewidth or lw = 2

**See also:**

*setp*

### matplotlib.artist.setp

matplotlib.artist.**setp**(*obj*, **args*, *file=None*, ***kwargs*)

Set one or more properties on an *Artist*, or list allowed values.

**Parameters**

**obj**

[*Artist* or list of *Artist*] The artist(s) whose properties are being set or queried. When setting properties, all artists are affected; when querying the allowed values, only the first instance in the sequence is queried.

For example, two lines can be made thicker and red with a single call:

```
>>> x = arange(0, 1, 0.01)
>>> lines = plot(x, sin(2*pi*x), x, sin(4*pi*x))
>>> setp(lines, linewidth=2, color='r')
```

**file**

[file-like, default: sys.stdout] Where *setp* writes its output when asked to list allowed values.

```
>>> with open('output.log') as file:
...     setp(line, file=file)
```

The default, None, means sys.stdout.

***args, **kwargs**

The properties to set. The following combinations are supported:

- Set the linestyle of a line to be dashed:

```
>>> line, = plot([1, 2, 3])
>>> setp(line, linestyle='--')
```

- Set multiple properties at once:

```
>>> setp(line, linewidth=2, color='r')
```

- List allowed values for a line's linestyle:

```
>>> setp(line, 'linestyle')
linestyle: {'-', '--', '-.', ':', '', (offset, on-off-
 ↪seq), ...}
```

- List all properties that can be set, and their allowed values:

```
>>> setp(line)
agg_filter: a filter function, ...
[long output listing omitted]
```

*setp* also supports MATLAB style string/value pairs. For example, the following are equivalent:

```
>>> setp(lines, 'linewidth', 2, 'color', 'r')  # MATLAB␣
 ↪style
>>> setp(lines, linewidth=2, color='r')        # Python␣
 ↪style
```

**See also:**

***getp***

## matplotlib.artist.kwdoc

matplotlib.artist.**kwdoc**(*artist*)

Inspect an *Artist* class (using *ArtistInspector*) and return information about its settable properties and their current values.

**Parameters**

**artist**

[*Artist* or an iterable of *Artist*s]

**Returns**

**str**

The settable properties of *artist*, as plain text if *rcParams["docstring.hardcopy"]* (default: False) is False and as a rst table (intended for use in Sphinx) if it is True.

### matplotlib.artist.ArtistInspector

**class** matplotlib.artist.**ArtistInspector**(*o*)

Bases: object

A helper class to inspect an *Artist* and return information about its settable properties and their current values.

Initialize the artist inspector with an *Artist* or an iterable of *Artist*s. If an iterable is used, we assume it is a homogeneous sequence (all *Artist*s are of the same type) and it is your responsibility to make sure this is so.

**aliased_name**(*s*)

Return 'PROPNAME or alias' if *s* has an alias, else return 'PROPNAME'.

For example, for the line markerfacecolor property, which has an alias, return 'markerfacecolor or mfc' and for the transform property, which does not, return 'transform'.

**aliased_name_rest**(*s*, *target*)

Return 'PROPNAME or alias' if *s* has an alias, else return 'PROPNAME', formatted for reST.

For example, for the line markerfacecolor property, which has an alias, return 'markerfacecolor or mfc' and for the transform property, which does not, return 'transform'.

**get_aliases**()

Get a dict mapping property fullnames to sets of aliases for each alias in the *ArtistInspector*.

e.g., for lines:

```
{'markerfacecolor': {'mfc'},
 'linewidth'      : {'lw'},
}
```

**get_setters**()

Get the attribute strings with setters for object.

For example, for a line, return `['markerfacecolor', 'linewidth', ....]`.

**get_valid_values**(*attr*)

Get the legal arguments for the setter associated with *attr*.

This is done by querying the docstring of the setter for a line that begins with "ACCEPTS:" or ".. ACCEPTS:", and then by looking for a numpydoc-style documentation for the setter's first argument.

**static is_alias**(*method*)

Return whether the object *method* is an alias for another method.

**static number_of_parameters**(*func*)

Return number of parameters of the callable *func*.

**pprint_getters**()

Return the getters and actual values as list of strings.

**pprint_setters**(*prop=None*, *leadingspace=2*)

If *prop* is *None*, return a list of strings of all settable properties and their valid values.

If *prop* is not *None*, it is a valid property name and that property will be returned as a string of property : valid values.

**pprint_setters_rest**(*prop=None*, *leadingspace=4*)

If *prop* is *None*, return a list of reST-formatted strings of all settable properties and their valid values.

If *prop* is not *None*, it is a valid property name and that property will be returned as a string of "property : valid" values.

**properties**()

Return a dictionary mapping property name -> value.

### 7.2.5 matplotlib.axes

The `Axes` class represents one (sub-)plot in a figure. It contains the plotted data, axis ticks, labels, title, legend, etc. Its methods are the main interface for manipulating the plot.

**Table of Contents**


- *The Axes class*
- *Plotting*
  - *Basic*
  - *Spans*
  - *Spectral*
  - *Statistics*
  - *Binned*
  - *Contours*
  - *2D arrays*
  - *Unstructured triangles*
  - *Text and annotations*
  - *Vector fields*
- *Clearing*
- *Appearance*

- *Property cycle*
- *Axis / limits*
  - *Axis limits and direction*
  - *Axis labels, title, and legend*
  - *Axis scales*
  - *Autoscaling and margins*
  - *Aspect ratio*
  - *Ticks and tick labels*
- *Units*
- *Adding artists*
- *Twinning and sharing*
- *Axes position*
- *Async/event based*
- *Interactive*
- *Children*
- *Drawing*
- *Projection*
- *Other*


### The Axes class

| | |
|---|---|
| `Axes` | An Axes object encapsulates all the elements of an individual (sub-)plot in a figure. |

### matplotlib.axes.Axes

**class** matplotlib.axes.**Axes**(*fig*, **args*, *facecolor=None*, *frameon=True*, *sharex=None*, *sharey=None*, *label=''*, *xscale=None*, *yscale=None*, *box_aspect=None*, *forward_navigation_events='auto'*, ***kwargs*)

An Axes object encapsulates all the elements of an individual (sub-)plot in a figure.

It contains most of the (sub-)plot elements: `Axis`, `Tick`, `Line2D`, `Text`, `Polygon`, etc., and sets the coordinate system.

Like all visible elements in a figure, Axes is an `Artist` subclass.

The *Axes* instance supports callbacks through a callbacks attribute which is a *CallbackRegistry* instance. The events you can connect to are 'xlim_changed' and 'ylim_changed' and the callback will be called with func(*ax*) where *ax* is the *Axes* instance.

---

**Note:** As a user, you do not instantiate Axes directly, but use Axes creation methods instead; e.g. from *pyplot* or *Figure*: *subplots*, *subplot_mosaic* or *Figure.add_axes*.

---

**Attributes**

**dataLim**

[*Bbox*] The bounding box enclosing all data displayed in the Axes.

**viewLim**

[*Bbox*] The view limits in data coordinates.

Build an Axes in a figure.

**Parameters**

**fig**

[*Figure*] The Axes is built in the *Figure* *fig*.

***args**

`*args` can be a single `(left, bottom, width, height)` rectangle or a single *Bbox*. This specifies the rectangle (in figure coordinates) where the Axes is positioned.

`*args` can also consist of three numbers or a single three-digit number; in the latter case, the digits are considered as independent numbers. The numbers are interpreted as `(nrows, ncols, index)`: `(nrows, ncols)` specifies the size of an array of subplots, and `index` is the 1-based index of the subplot being created. Finally, `*args` can also directly be a *SubplotSpec* instance.

**sharex, sharey**

[*Axes*, optional] The x- or y-*axis* is shared with the x- or y-axis in the input *Axes*.

**frameon**

[bool, default: True] Whether the Axes frame is visible.

**box_aspect**

[float, optional] Set a fixed aspect for the Axes box, i.e. the ratio of height to width. See *set_box_aspect* for details.

**forward_navigation_events**

[bool or "auto", default: "auto"] Control whether pan/zoom events are passed through to Axes below this one. "auto" is *True* for axes with an invisible patch and *False* otherwise.

****kwargs**

Other optional keyword arguments:

| Property | Description |
|---|---|
| *adjustable* | {'box', 'datalim'} |
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi v |
| *alpha* | scalar or None |
| *anchor* | (float, float) or {'C', 'SW', 'S', 'SE', 'E', 'NE', ...} |
| *animated* | bool |
| *aspect* | {'auto', 'equal'} or float |
| *autoscale_on* | bool |
| *autoscalex_on* | unknown |
| *autoscaley_on* | unknown |
| *axes_locator* | Callable[[Axes, Renderer], Bbox] |
| *axisbelow* | bool or 'line' |
| *box_aspect* | float or None |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *facecolor* or fc | *color* |
| *figure* | *Figure* |
| *forward_navigation_events* | bool or "auto" |
| *frame_on* | bool |
| *gid* | str |
| *in_layout* | bool |
| *label* | object |
| *mouseover* | bool |
| *navigate* | bool |
| *navigate_mode* | unknown |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *position* | [left, bottom, width, height] or *Bbox* |
| *prop_cycle* | *Cycler* |
| *rasterization_zorder* | float or None |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *subplotspec* | unknown |
| *title* | str |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |

Table 1 – continued from

| Property | Description |
|---|---|
| *xbound* | (lower: float, upper: float) |
| *xlabel* | str |
| *xlim* | (left: float, right: float) |
| *xmargin* | float greater than -0.5 |
| *xscale* | unknown |
| *xticklabels* | unknown |
| *xticks* | unknown |
| *ybound* | (lower: float, upper: float) |
| *ylabel* | str |
| *ylim* | (bottom: float, top: float) |
| *ymargin* | float greater than -0.5 |
| *yscale* | unknown |
| *yticklabels* | unknown |
| *yticks* | unknown |
| *zorder* | float |

**Returns**

***Axes***

The new *Axes* object.

## Plotting

### Basic

| | |
|---|---|
| *Axes.plot* | Plot y versus x as lines and/or markers. |
| *Axes.errorbar* | Plot y versus x as lines and/or markers with attached errorbars. |
| *Axes.scatter* | A scatter plot of *y* vs. |
| *Axes.plot_date* | [*Deprecated*] Plot coercing the axis to treat floats as dates. |
| *Axes.step* | Make a step plot. |
| *Axes.loglog* | Make a plot with log scaling on both the x- and y-axis. |
| *Axes.semilogx* | Make a plot with log scaling on the x-axis. |
| *Axes.semilogy* | Make a plot with log scaling on the y-axis. |
| *Axes.fill_between* | Fill the area between two horizontal curves. |
| *Axes.fill_betweenx* | Fill the area between two vertical curves. |
| *Axes.bar* | Make a bar plot. |
| *Axes.barh* | Make a horizontal bar plot. |
| *Axes.bar_label* | Label a bar plot. |
| *Axes.stem* | Create a stem plot. |
| *Axes.eventplot* | Plot identical parallel lines at the given positions. |
| *Axes.pie* | Plot a pie chart. |
| *Axes.stackplot* | Draw a stacked area plot or a streamgraph. |
| *Axes.broken_barh* | Plot a horizontal sequence of rectangles. |
| *Axes.vlines* | Plot vertical lines at each *x* from *ymin* to *ymax*. |
| *Axes.hlines* | Plot horizontal lines at each *y* from *xmin* to *xmax*. |
| *Axes.fill* | Plot filled polygons. |

### matplotlib.axes.Axes.plot

Axes.**plot**(**args*, *scalex=True*, *scaley=True*, *data=None*, ***kwargs*)

Plot y versus x as lines and/or markers.

Call signatures:

```
plot([x], y, [fmt], *, data=None, **kwargs)
plot([x], y, [fmt], [x2], y2, [fmt2], ..., **kwargs)
```

The coordinates of the points or line nodes are given by *x*, *y*.

The optional parameter *fmt* is a convenient way for defining basic formatting like color, marker and linestyle. It's a shortcut string notation described in the *Notes* section below.

```
>>> plot(x, y)        # plot x and y using default line style and color
>>> plot(x, y, 'bo')  # plot x and y using blue circle markers
>>> plot(y)           # plot y using x as index array 0..N-1
>>> plot(y, 'r+')     # ditto, but with red plusses
```

You can use *Line2D* properties as keyword arguments for more control on the appearance. Line properties and *fmt* can be mixed. The following two calls yield identical results:

```
>>> plot(x, y, 'go--', linewidth=2, markersize=12)
>>> plot(x, y, color='green', marker='o', linestyle='dashed',
...      linewidth=2, markersize=12)
```

When conflicting with *fmt*, keyword arguments take precedence.

**Plotting labelled data**

There's a convenient way for plotting objects with labelled data (i.e. data that can be accessed by index `obj['y']`). Instead of giving the data in *x* and *y*, you can provide the object in the *data* parameter and just give the labels for *x* and *y*:

```
>>> plot('xlabel', 'ylabel', data=obj)
```

All indexable objects are supported. This could e.g. be a `dict`, a `pandas.DataFrame` or a structured numpy array.

**Plotting multiple sets of data**

There are various ways to plot multiple sets of data.

- The most straight forward way is just to call *plot* multiple times. Example:

  ```
  >>> plot(x1, y1, 'bo')
  >>> plot(x2, y2, 'go')
  ```

- If *x* and/or *y* are 2D arrays a separate data set will be drawn for every column. If both *x* and *y* are 2D, they must have the same shape. If only one of them is 2D with shape (N, m) the other must have length N and will be used for every data set m.

  Example:

  ```
  >>> x = [1, 2, 3]
  >>> y = np.array([[1, 2], [3, 4], [5, 6]])
  >>> plot(x, y)
  ```

  is equivalent to:

  ```
  >>> for col in range(y.shape[1]):
  ...     plot(x, y[:, col])
  ```

- The third way is to specify multiple sets of *[x]*, *y*, *[fmt]* groups:

  ```
  >>> plot(x1, y1, 'g^', x2, y2, 'g-')
  ```

  In this case, any additional keyword argument applies to all datasets. Also, this syntax cannot be combined with the *data* parameter.

By default, each line is assigned a different style specified by a 'style cycle'. The *fmt* and line property parameters are only necessary if you want explicit deviations from these defaults. Alternatively,

you can also change the style cycle using *rcParams["axes.prop_cycle"]* (default: `cycler('color', ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728', '#9467bd', '#8c564b', '#e377c2', '#7f7f7f', '#bcbd22', '#17becf'])`).

**Parameters**

**x, y**

[array-like or scalar] The horizontal / vertical coordinates of the data points. *x* values are optional and default to `range(len(y))`.

Commonly, these parameters are 1D arrays.

They can also be scalars, or two-dimensional (in that case, the columns represent separate data sets).

These arguments cannot be passed as keywords.

**fmt**

[str, optional] A format string, e.g. 'ro' for red circles. See the *Notes* section for a full description of the format strings.

Format strings are just an abbreviation for quickly setting basic line properties. All of these and more can also be controlled by keyword arguments.

This argument cannot be passed as keyword.

**data**

[indexable object, optional] An object with labelled data. If given, provide the label names to plot in *x* and *y*.

---

**Note:** Technically there's a slight ambiguity in calls where the second label is a valid *fmt*. `plot('n', 'o', data=obj)` could be `plt(x, y)` or `plt(y, fmt)`. In such cases, the former interpretation is chosen, but a warning is issued. You may suppress the warning by adding an empty format string `plot('n', 'o', '', data=obj)`.

---

**Returns**

**list of *Line2D***

A list of lines representing the plotted data.

**Other Parameters**

**scalex, scaley**

[bool, default: True] These parameters determine if the view limits are adapted to the data limits. The values are passed on to *autoscale_view*.

**kwargs

[*Line2D* properties, optional] *kwargs* are used to specify properties like a line label (for auto legends), linewidth, antialiasing, marker face color. Example:

```
>>> plot([1, 2, 3], [1, 2, 3], 'go-', label='line 1',␣
↪linewidth=2)
>>> plot([1, 2, 3], [1, 4, 9], 'rs', label='line 2')
```

If you specify multiple lines with one plot call, the kwargs apply to all those lines. In case the label object is iterable, each element is used as labels for each set of data.

Here is a list of available *Line2D* properties:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi va |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* or aa | bool |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* or c | *color* |
| *dash_capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *dash_joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *dashes* | sequence of floats (on/off ink in points) or (None, None) |
| *data* | (2, N) array or two 1D arrays |
| *drawstyle* or ds | {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: |
| *figure* | *Figure* |
| *fillstyle* | {'full', 'left', 'right', 'bottom', 'top', 'none'} |
| *gapcolor* | *color* or None |
| *gid* | str |
| *in_layout* | bool |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float |
| *marker* | marker style string, *Path* or *MarkerStyle* |
| *markeredgecolor* or mec | *color* |
| *markeredgewidth* or mew | float |
| *markerfacecolor* or mfc | *color* |
| *markerfacecoloralt* or mfcalt | *color* |
| *markersize* or ms | float |
| *markevery* | None or int or (int, int) or slice or list[int] or float or (float, floa |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | float or callable[[Artist, Event], tuple[bool, dict]] |

Table 2 – continued from

| Property | Description |
|---|---|
| *pickradius* | float |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *solid_capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *solid_joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *transform* | unknown |
| *url* | str |
| *visible* | bool |
| *xdata* | 1D array |
| *ydata* | 1D array |
| *zorder* | float |

**See also:**

***scatter***

XY scatter plot with markers of varying size and/or color ( sometimes also called bubble chart).

## Notes

### Format Strings

A format string consists of a part for color, marker and line:

```
fmt = '[marker][line][color]'
```

Each of them is optional. If not provided, the value from the style cycle is used. Exception: If `line` is given, but no `marker`, the data will be a line without markers.

Other combinations such as `[color][marker][line]` are also supported, but note that their parsing may be ambiguous.

### Markers

| character | description |
|---|---|
| `'.'` | point marker |
| `','` | pixel marker |
| `'o'` | circle marker |
| `'v'` | triangle_down marker |
| `'^'` | triangle_up marker |
| `'<'` | triangle_left marker |
| `'>'` | triangle_right marker |
| `'1'` | tri_down marker |
| `'2'` | tri_up marker |
| `'3'` | tri_left marker |
| `'4'` | tri_right marker |
| `'8'` | octagon marker |
| `'s'` | square marker |
| `'p'` | pentagon marker |
| `'P'` | plus (filled) marker |
| `'*'` | star marker |
| `'h'` | hexagon1 marker |
| `'H'` | hexagon2 marker |
| `'+'` | plus marker |
| `'x'` | x marker |
| `'X'` | x (filled) marker |
| `'D'` | diamond marker |
| `'d'` | thin_diamond marker |
| `'\|'` | vline marker |
| `'_'` | hline marker |

**Line Styles**

| character | description |
|---|---|
| `'-'` | solid line style |
| `'--'` | dashed line style |
| `'-.'` | dash-dot line style |
| `':'` | dotted line style |

Example format strings:

```
'b'    # blue markers with default shape
'or'   # red circles
'-g'   # green solid line
'--'   # dashed line with default color
'^k:'  # black triangle_up markers connected by a dotted line
```

**Colors**

The supported color abbreviations are the single letter codes

| character | color |
|---|---|
| `'b'` | blue |
| `'g'` | green |
| `'r'` | red |
| `'c'` | cyan |
| `'m'` | magenta |
| `'y'` | yellow |
| `'k'` | black |
| `'w'` | white |

and the `'CN'` colors that index into the default property cycle.

If the color is the only part of the format string, you can additionally use any *matplotlib.colors* spec, e.g. full names (`'green'`) or hex strings (`'#008000'`).

### Examples using `matplotlib.axes.Axes.plot`

- *Plotting categorical variables*
- *Cross spectral density (CSD)*
- *Curve with error band*
- *EventCollection Demo*
- *Fill Between and Alpha*
- *Filling the area between lines*
- *Fill Betweenx Demo*
- *Customizing dashed line styles*
- *Lines with a ticked patheffect*
- *Marker reference*
- *Markevery Demo*
- *Mapping marker properties to multivariate data*
- *Power spectral density (PSD)*
- *Simple Plot*
- *Shade regions defined by a logical mask using fill_between*
- *Step Demo*
- *Creating a timeline with lines, dates, and text*
- *hlines and vlines*
- *Contour Corner Mask*

- *Contour plot of irregularly spaced data*
- *pcolormesh grids and shading*
- *Spectrogram*
- *Triinterp Demo*
- *Aligning Labels and Titles*
- *Programmatically controlling subplot adjustment*
- *Axes box aspect*
- *Axes Demo*
- *Controlling view limits using margins and sticky_edges*
- *Axes Props*
- *axhspan Demo*
- *Broken Axis*
- *Resizing Axes with constrained layout*
- *Resizing Axes with tight layout*
- *Figure labels: suptitle, supxlabel, supylabel*
- *Inverted axis*
- *Secondary Axis*
- *Sharing axis limits and views*
- *Figure subfigures*
- *Multiple subplots*
- *Creating multiple subplots using plt.subplots*
- *Plots with different scales*
- *Boxplots*
- *Polar plot*
- *Polar legend*
- *Accented text*
- *Align y-labels*
- *Scale invariant angle label*
- *Annotate Transform*
- *Annotating a plot*
- *Annotating Plots*
- *Annotation Polar*

- *Composing Custom Legends*
- *Date tick labels*
- *AnnotationBbox demo*
- *Labeling ticks using engineering notation*
- *Annotation arrow style reference*
- *Legend using pre-defined labels*
- *Legend Demo*
- *Mathtext*
- *Math fontfamily*
- *Multiline*
- *Rendering math equations using TeX*
- *Text Commands*
- *Text Rotation Relative To Line*
- *Title positioning*
- *Text watermark*
- *Color Demo*
- *Color by y-value*
- *Selecting individual colors from a colormap*
- *Ellipse with orientation arrow demo*
- *PathPatch object*
- *Bezier Curve*
- *Dark background style sheet*
- *FiveThirtyEight style sheet*
- *ggplot style sheet*
- *Multiple lines using pyplot*
- *Axes with a fixed physical size*
- *Parasite Simple*
- *Simple Axisline4*
- *Axis line styles*
- *Parasite Axes demo*
- *Parasite axis demo*
- *Custom spines with axisartist*

- *Simple Axisline*
- *Anatomy of a figure*
- *Integral as the area under a curve*
- *Stock prices over 32 years*
- *XKCD*
- *Decay*
- *The Bayes update*
- *The double pendulum problem*
- *Multiple Axes animation*
- *Animated 3D random walk*
- *Animated line plot*
- *MATPLOTLIB UNCHAINED*
- *Mouse move and click events*
- *Cross-hair cursor*
- *Data browser*
- *Keypress event*
- *Legend picking*
- *Looking Glass*
- *Path editor*
- *Pick event demo*
- *Pick event demo 2*
- *Resampling Data*
- *Timers*
- *Changing colors of lines intersecting a box*
- *Custom projection*
- *Patheffect Demo*
- *SVG Filter Line*
- *TickedStroke patheffect*
- *Zorder Demo*
- *Plot 2D data on 3D plot*
- *3D box surface plot*
- *Parametric curve*

- *Lorenz attractor*
- *2D and 3D Axes in same figure*
- *Asinh Demo*
- *Loglog Aspect*
- *Scales*
- *Symlog Demo*
- *Anscombe's quartet*
- *Ishikawa Diagram*
- *Radar chart (aka spider or star chart)*
- *Spines*
- *Spine placement*
- *Multiple y-axis with Spines*
- *Centered spines with arrows*
- *Centering labels between ticks*
- *Formatting date ticks using ConciseDateFormatter*
- *Date Demo Convert*
- *Custom tick formatter for time series*
- *Date Precision and Epochs*
- *Dollar ticks*
- *Major and minor ticks*
- *Multilevel (nested) ticks*
- *Set default y-axis tick labels on the right*
- *Setting tick labels from a list of values*
- *Move x-axis tick labels to the top*
- *Rotating custom tick labels*
- *Evans test*
- *CanvasAgg demo*
- *Annotated cursor*
- *Buttons*
- *Check buttons*
- *Cursor*
- *Multicursor*

- *Rectangle and ellipse selectors*
- *Slider*
- *Snapping Sliders to Discrete Values*
- *Span Selector*
- *Textbox*
- *Annotate Explain*
- *Connection styles for annotations*
- *Nested GridSpecs*
- *PGF fonts*
- *PGF texsystem*
- *Simple Annotate01*
- *Simple Legend01*
- *Simple Legend02*
- *Artist tutorial*
- *plot(x, y)*
- *fill_between(x, y1, y2)*
- *tricontour(x, y, z)*
- *tricontourf(x, y, z)*
- *tripcolor(x, y, z)*
- *Quick start guide*
- *Animations using Matplotlib*
- *Faster rendering by using blitting*
- *Styling with cycler*
- *Path Tutorial*
- *Transformations Tutorial*
- *Legend guide*
- *Constrained layout guide*
- *Tight layout guide*
- *Arranging multiple Axes in a Figure*
- *Autoscaling Axis*
- *Axis scales*
- *Axis ticks*

- *Plotting dates and strings*
- *Specifying colors*
- *Text in Matplotlib*
- *Annotations*

### matplotlib.axes.Axes.errorbar

Axes.**errorbar**(*x*, *y*, *yerr=None*, *xerr=None*, *fmt=''*, *ecolor=None*, *elinewidth=None*, *capsize=None*, *barsabove=False*, *lolims=False*, *uplims=False*, *xlolims=False*, *xuplims=False*, *errorevery=1*, *capthick=None*, ***, *data=None*, ***kwargs*)

Plot y versus x as lines and/or markers with attached errorbars.

*x*, *y* define the data locations, *xerr*, *yerr* define the errorbar sizes. By default, this draws the data markers/lines as well as the errorbars. Use fmt='none' to draw errorbars without any data markers.

New in version 3.7: Caps and error lines are drawn in polar coordinates on polar plots.

**Parameters**

**x, y**

[float or array-like] The data positions.

**xerr, yerr**

[float or array-like, shape(N,) or shape(2, N), optional] The errorbar sizes:

- scalar: Symmetric +/- values for all data points.
- shape(N,): Symmetric +/-values for each data point.
- shape(2, N): Separate - and + values for each bar. First row contains the lower errors, the second row contains the upper errors.
- *None*: No errorbar.

All values must be >= 0.

See *Different ways of specifying error bars* for an example on the usage of `xerr` and `yerr`.

**fmt**

[str, default: ''] The format for the data points / data lines. See *plot* for details.

Use 'none' (case-insensitive) to plot errorbars without any data markers.

**ecolor**

[*color*, default: None] The color of the errorbar lines. If None, use the color of the line connecting the markers.

**elinewidth**

[float, default: None] The linewidth of the errorbar lines. If None, the linewidth of the current style is used.

**capsize**

[float, default: *rcParams["errorbar.capsize"]* (default: 0.0)] The length of the error bar caps in points.

**capthick**

[float, default: None] An alias to the keyword argument *markeredgewidth* (a.k.a. *mew*). This setting is a more sensible name for the property that controls the thickness of the error bar cap in points. For backwards compatibility, if *mew* or *markeredgewidth* are given, then they will over-ride *capthick*. This may change in future releases.

**barsabove**

[bool, default: False] If True, will plot the errorbars above the plot symbols. Default is below.

**lolims, uplims, xlolims, xuplims**

[bool or array-like, default: False] These arguments can be used to indicate that a value gives only upper/lower limits. In that case a caret symbol is used to indicate this. *lims*-arguments may be scalars, or array-likes of the same length as *xerr* and *yerr*. To use limits with inverted axes, *set_xlim* or *set_ylim* must be called before *errorbar()*. Note the tricky parameter names: setting e.g. *lolims* to True means that the y-value is a *lower* limit of the True value, so, only an *upward*-pointing arrow will be drawn!

**errorevery**

[int or (int, int), default: 1] draws error bars on a subset of the data. *errorevery* =N draws error bars on the points (x[::N], y[::N]). *errorevery* =(start, N) draws error bars on the points (x[start::N], y[start::N]). e.g. errorevery=(6, 3) adds error bars to the data at (x[6], x[9], x[12], x[15], ...). Used to avoid overlapping error bars when two series share x-axis values.

**Returns**

***ErrorbarContainer***

The container contains:

- plotline: *Line2D* instance of x, y plot markers and/or line.
- caplines: A tuple of *Line2D* instances of the error bar caps.
- barlinecols: A tuple of *LineCollection* with the horizontal and vertical error ranges.

**Other Parameters**

**data**

[indexable object, optional] If given, the following parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception):

*x*, *y*, *xerr*, *yerr*

****kwargs**

All other keyword arguments are passed on to the *plot* call drawing the markers. For example, this code makes big red squares with thick green edges:

```
x, y, yerr = rand(3, 10)
errorbar(x, y, yerr, marker='s', mfc='red',
         mec='green', ms=20, mew=4)
```

where *mfc*, *mec*, *ms* and *mew* are aliases for the longer property names, *markerfacecolor*, *markeredgecolor*, *markersize* and *markeredgewidth*.

Valid kwargs for the marker properties are:

- *dashes*
- *dash_capstyle*
- *dash_joinstyle*
- *drawstyle*
- *fillstyle*
- *linestyle*
- *marker*
- *markeredgecolor*
- *markeredgewidth*
- *markerfacecolor*
- *markerfacecoloralt*
- *markersize*
- *markevery*
- *solid_capstyle*
- *solid_joinstyle*

Refer to the corresponding *Line2D* property for more details:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi va |
| *alpha* | scalar or None |
| *animated* | bool |

Table 3 – continued from

| Property | Description |
|---|---|
| *antialiased* or aa | bool |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* or c | *color* |
| *dash_capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *dash_joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *dashes* | sequence of floats (on/off ink in points) or (None, None) |
| *data* | (2, N) array or two 1D arrays |
| *drawstyle* or ds | {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: |
| *figure* | *Figure* |
| *fillstyle* | {'full', 'left', 'right', 'bottom', 'top', 'none'} |
| *gapcolor* | *color* or None |
| *gid* | str |
| *in_layout* | bool |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float |
| *marker* | marker style string, *Path* or *MarkerStyle* |
| *markeredgecolor* or mec | *color* |
| *markeredgewidth* or mew | float |
| *markerfacecolor* or mfc | *color* |
| *markerfacecoloralt* or mfcalt | *color* |
| *markersize* or ms | float |
| *markevery* | None or int or (int, int) or slice or list[int] or float or (float, floa |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | float or callable[[Artist, Event], tuple[bool, dict]] |
| *pickradius* | float |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *solid_capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *solid_joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *transform* | unknown |
| *url* | str |
| *visible* | bool |
| *xdata* | 1D array |
| *ydata* | 1D array |
| *zorder* | float |

### Examples using `matplotlib.axes.Axes.errorbar`

- *Errorbar limit selection*
- *Errorbar subsampling*
- *Errorbar function*
- *Different ways of specifying error bars*
- *Including upper and lower limits in error bars*
- *Creating boxes from error bars using PatchCollection*
- *Error bar rendering on polar axis*
- *Legend Demo*
- *Parasite Simple2*
- *3D errorbars*
- *Log Demo*
- *errorbar(x, y, yerr, xerr)*

## matplotlib.axes.Axes.scatter

`Axes.`**`scatter`**(*x*, *y*, *s=None*, *c=None*, *marker=None*, *cmap=None*, *norm=None*, *vmin=None*, *vmax=None*, *alpha=None*, *linewidths=None*, ***, *edgecolors=None*, *plotnonfinite=False*, *data=None*, ***kwargs*)

A scatter plot of *y* vs. *x* with varying marker size and/or color.

**Parameters**

**x, y**

[float or array-like, shape (n, )] The data positions.

**s**

[float or array-like, shape (n, ), optional] The marker size in points**2 (typographic points are 1/72 in.). Default is `rcParams['lines.markersize'] ** 2`.

The linewidth and edgecolor can visually interact with the marker size, and can lead to artifacts if the marker size is smaller than the linewidth.

If the linewidth is greater than 0 and the edgecolor is anything but *'none'*, then the effective size of the marker will be increased by half the linewidth because the stroke will be centered on the edge of the shape.

To eliminate the marker edge either set *linewidth=0* or *edgecolor='none'*.

**c**

[array-like or list of *color* or *color*, optional] The marker colors. Possible values:

- A scalar or sequence of n numbers to be mapped to colors using *cmap* and *norm*.
- A 2D array in which the rows are RGB or RGBA.
- A sequence of colors of length n.
- A single color format string.

Note that *c* should not be a single numeric RGB or RGBA sequence because that is indistinguishable from an array of values to be colormapped. If you want to specify the same RGB or RGBA value for all points, use a 2D array with a single row. Otherwise, value-matching will have precedence in case of a size matching with *x* and *y*.

If you wish to specify a single color for all points prefer the *color* keyword argument.

Defaults to `None`. In that case the marker color is determined by the value of *color*, *facecolor* or *facecolors*. In case those are not specified or `None`, the marker color is determined by the next color of the `Axes`' current "shape and fill" color cycle. This cycle defaults to *rcParams["axes.prop_cycle"]* (default: `cycler('color', ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728', '#9467bd', '#8c564b', '#e377c2', '#7f7f7f', '#bcbd22', '#17becf'])`).

**marker**

[*MarkerStyle*, default: *rcParams["scatter.marker"]* (default: `'o'`)] The marker style. *marker* can be either an instance of the class or the text shorthand for a particular marker. See *matplotlib.markers* for more information about marker styles.

**cmap**

[str or *Colormap*, default: *rcParams["image.cmap"]* (default: `'viridis'`)] The Colormap instance or registered colormap name used to map scalar data to colors.

This parameter is ignored if *c* is RGB(A).

**norm**

[str or *Normalize*, optional] The normalization method used to scale scalar data to the [0, 1] range before mapping to colors using *cmap*. By default, a linear scaling is used, mapping the lowest value to 0 and the highest to 1.

If given, this can be one of the following:

- An instance of *Normalize* or one of its subclasses (see *Colormap normalization*).
- A scale name, i.e. one of "linear", "log", "symlog", "logit", etc. For a list of available scales, call *matplotlib.scale.get_scale_names()*. In that case, a suitable *Normalize* subclass is dynamically generated and instantiated.

This parameter is ignored if *c* is RGB(A).

**vmin, vmax**

[float, optional] When using scalar data and no explicit *norm*, *vmin* and *vmax* define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data. It is an error to use *vmin*/*vmax* when a *norm* instance is given (but using a `str` *norm* name together with *vmin*/*vmax* is acceptable).

This parameter is ignored if *c* is RGB(A).

**alpha**

[float, default: None] The alpha blending value, between 0 (transparent) and 1 (opaque).

**linewidths**

[float or array-like, default: *rcParams["lines.linewidth"]* (default: `1.5`)] The linewidth of the marker edges. Note: The default *edgecolors* is 'face'. You may want to change this as well.

**edgecolors**

[{'face', 'none', *None*} or *color* or list of *color*, default: *rcParams["scatter.edgecolors"]* (default: `'face'`)] The edge color of the marker. Possible values:

- 'face': The edge color will always be the same as the face color.
- 'none': No patch boundary will be drawn.
- A color or sequence of colors.

For non-filled markers, *edgecolors* is ignored. Instead, the color is determined like with 'face', i.e. from *c*, *colors*, or *facecolors*.

**plotnonfinite**

[bool, default: False] Whether to plot points with nonfinite *c* (i.e. `inf`, `-inf` or `nan`). If `True` the points are drawn with the *bad* colormap color (see *Colormap.set_bad*).

**Returns**

***PathCollection***

**Other Parameters**

**data**

[indexable object, optional] If given, the following parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception):

*x*, *y*, *s*, *linewidths*, *edgecolors*, *c*, *facecolor*, *facecolors*, *color*

****kwargs**

[*Collection* properties]

**See also:**

***plot***

To plot scatter plots when markers are identical in size and color.

### Notes

- The *plot* function will be faster for scatterplots where markers don't vary in size or color.
- Any or all of *x*, *y*, *s*, and *c* may be masked arrays, in which case all masks will be combined and only unmasked points will be plotted.
- Fundamentally, scatter works with 1D arrays; *x*, *y*, *s*, and *c* may be input as N-D arrays, but within scatter they will be flattened. The exception is *c*, which will be flattened only if its size matches the size of *x* and *y*.

### Examples using `matplotlib.axes.Axes.scatter`

- *Scatter Demo2*
- *Scatter plot with histograms*
- *Scatter plots with a legend*
- *Advanced quiver and quiverkey functions*
- *Axes box aspect*
- *Axis Label Position*
- *Plot a confidence ellipse of a two-dimensional dataset*
- *Violin plot customization*
- *Scatter plot on polar axis*
- *Legend Demo*
- *Scatter plot*
- *Scatter Histogram (Locatable Axes)*
- *floating_axes features*
- *Rain simulation*
- *Animated scatter saved as GIF*
- *Pick event demo*
- *Zoom Window*

- *Plotting with keywords*
- *Zorder Demo*
- *Plot 2D data on 3D plot*
- *3D scatterplot*
- *Asinh Demo*
- *Automatically setting tick positions*
- *Unit handling*
- *Select indices from a collection using polygon selector*
- *Annotate Text Arrow*
- *scatter(x, y)*
- *scatter(xs, ys, zs)*
- *Quick start guide*
- *Animations using Matplotlib*
- *Plotting dates and strings*
- *Choosing Colormaps in Matplotlib*
- *Annotations*

### matplotlib.axes.Axes.plot_date

`Axes.`**`plot_date`**(*x*, *y*, *fmt='o'*, *tz=None*, *xdate=True*, *ydate=False*, ***, *data=None*, ***kwargs*)

[*Deprecated*] Plot coercing the axis to treat floats as dates.

Deprecated since version 3.9: This method exists for historic reasons and will be removed in version 3.11.

- `datetime`-like data should directly be plotted using *`plot`*.
- If you need to plot plain numeric data as *Matplotlib date format* or need to set a timezone, call `ax.xaxis.axis_date` / `ax.yaxis.axis_date` before *`plot`*. See *`Axis.axis_date`*.

Similar to *`plot`*, this plots *y* vs. *x* as lines or markers. However, the axis labels are formatted as dates depending on *xdate* and *ydate*. Note that *`plot`* will work with `datetime` and `numpy.datetime64` objects without resorting to this method.

**Parameters**

**x, y**

[array-like] The coordinates of the data points. If *xdate* or *ydate* is *True*, the respective values *x* or *y* are interpreted as *Matplotlib dates*.

**fmt**

[str, optional] The plot format string. For details, see the corresponding parameter in *plot*.

**tz**

[timezone string or `datetime.tzinfo`, default: *rcParams["timezone"]* (default: `'UTC'`)] The time zone to use in labeling dates.

**xdate**

[bool, default: True] If *True*, the *x*-axis will be interpreted as Matplotlib dates.

**ydate**

[bool, default: False] If *True*, the *y*-axis will be interpreted as Matplotlib dates.

### Returns

**list of *Line2D***

Objects representing the plotted data.

### Other Parameters

**data**

[indexable object, optional] If given, the following parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception):

*x*, *y*

****kwargs**

Keyword arguments control the *Line2D* properties:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi va |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* or aa | bool |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* or c | *color* |
| *dash_capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *dash_joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *dashes* | sequence of floats (on/off ink in points) or (None, None) |
| *data* | (2, N) array or two 1D arrays |
| *drawstyle* or ds | {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: |
| *figure* | *Figure* |

Table 4 – continued from

| Property | Description |
|---|---|
| *fillstyle* | {'full', 'left', 'right', 'bottom', 'top', 'none'} |
| *gapcolor* | *color* or None |
| *gid* | str |
| *in_layout* | bool |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float |
| *marker* | marker style string, *Path* or *MarkerStyle* |
| *markeredgecolor* or mec | *color* |
| *markeredgewidth* or mew | float |
| *markerfacecolor* or mfc | *color* |
| *markerfacecoloralt* or mfcalt | *color* |
| *markersize* or ms | float |
| *markevery* | None or int or (int, int) or slice or list[int] or float or (float, floa |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | float or callable[[Artist, Event], tuple[bool, dict]] |
| *pickradius* | float |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *solid_capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *solid_joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *transform* | unknown |
| *url* | str |
| *visible* | bool |
| *xdata* | 1D array |
| *ydata* | 1D array |
| *zorder* | float |

**See also:**

***matplotlib.dates***

Helper functions on dates.

***matplotlib.dates.date2num***

Convert dates to num.

***matplotlib.dates.num2date***

Convert num to dates.

***matplotlib.dates.drange***

Create an equally spaced sequence of dates.

### Notes

If you are using custom date tickers and formatters, it may be necessary to set the formatters/locators after the call to *plot_date*. *plot_date* will set the default tick locator to *AutoDateLocator* (if the tick locator is not already set to a *DateLocator* instance) and the default tick formatter to *AutoDateFormatter* (if the tick formatter is not already set to a *DateFormatter* instance).

Deprecated since version 3.9: Use plot instead.

## matplotlib.axes.Axes.step

Axes.**step**(*x*, *y*, **args*, *where='pre'*, *data=None*, ***kwargs*)

Make a step plot.

Call signatures:

```
step(x, y, [fmt], *, data=None, where='pre', **kwargs)
step(x, y, [fmt], x2, y2, [fmt2], ..., *, where='pre', **kwargs)
```

This is just a thin wrapper around *plot* which changes some formatting options. Most of the concepts and parameters of plot can be used here as well.

---

**Note:** This method uses a standard plot with a step drawstyle: The *x* values are the reference positions and steps extend left/right/both directions depending on *where*.

For the common case where you know the values and edges of the steps, use *stairs* instead.

---

#### Parameters

**x**

[array-like] 1D sequence of x positions. It is assumed, but not checked, that it is uniformly increasing.

**y**

[array-like] 1D sequence of y levels.

**fmt**

[str, optional] A format string, e.g. 'g' for a green line. See *plot* for a more detailed description.

Note: While full format strings are accepted, it is recommended to only specify the color. Line styles are currently ignored (use the keyword argument *linestyle* instead). Markers are accepted and plotted on the given positions, however, this is a rarely needed feature for step plots.

**where**

[{'pre', 'post', 'mid'}, default: 'pre'] Define where the steps should be placed:

- 'pre': The y value is continued constantly to the left from every *x* position, i.e. the interval `(x[i-1], x[i]]` has the value `y[i]`.
- 'post': The y value is continued constantly to the right from every *x* position, i.e. the interval `[x[i], x[i+1])` has the value `y[i]`.
- 'mid': Steps occur half-way between the *x* positions.

**data**

[indexable object, optional] An object with labelled data. If given, provide the label names to plot in *x* and *y*.

****kwargs**

Additional parameters are the same as those for *plot*.

**Returns**

**list of *Line2D***

Objects representing the plotted data.

## Examples using `matplotlib.axes.Axes.step`

- *Step Demo*
- *stairs(values)*

## matplotlib.axes.Axes.loglog

`Axes.`**`loglog`**`(`*args*, ***kwargs*`)`

Make a plot with log scaling on both the x- and y-axis.

Call signatures:

```
loglog([x], y, [fmt], data=None, **kwargs)
loglog([x], y, [fmt], [x2], y2, [fmt2], ..., **kwargs)
```

This is just a thin wrapper around *plot* which additionally changes both the x-axis and the y-axis to log scaling. All the concepts and parameters of plot can be used here as well.

The additional parameters *base*, *subs* and *nonpositive* control the x/y-axis properties. They are just forwarded to *Axes.set_xscale* and *Axes.set_yscale*. To use different properties on the x-axis and the y-axis, use e.g. `ax.set_xscale("log", base=10); ax.set_yscale("log", base=2)`.

**Parameters**

**base**

[float, default: 10] Base of the logarithm.

**subs**

[sequence, optional] The location of the minor ticks. If *None*, reasonable locations are automatically chosen depending on the number of decades in the plot. See *Axes.set_xscale*/*Axes.set_yscale* for details.

**nonpositive**

[{'mask', 'clip'}, default: 'clip'] Non-positive values can be masked as invalid, or clipped to a very small positive number.

****kwargs**

All parameters supported by *plot*.

**Returns**

**list of *Line2D***

Objects representing the plotted data.

### Examples using `matplotlib.axes.Axes.loglog`

- *Secondary Axis*
- *Log Demo*
- *Axis scales*

## matplotlib.axes.Axes.semilogx

Axes.**semilogx**(**args*, ***kwargs*)

Make a plot with log scaling on the x-axis.

Call signatures:

```
semilogx([x], y, [fmt], data=None, **kwargs)
semilogx([x], y, [fmt], [x2], y2, [fmt2], ..., **kwargs)
```

This is just a thin wrapper around *plot* which additionally changes the x-axis to log scaling. All the concepts and parameters of plot can be used here as well.

The additional parameters *base*, *subs*, and *nonpositive* control the x-axis properties. They are just forwarded to *Axes.set_xscale*.

**Parameters**

**base**

[float, default: 10] Base of the x logarithm.

**subs**

[array-like, optional] The location of the minor xticks. If *None*, reasonable locations are automatically chosen depending on the number of decades in the plot. See *Axes.set_xscale* for details.

**nonpositive**

[{'mask', 'clip'}, default: 'clip'] Non-positive values in x can be masked as invalid, or clipped to a very small positive number.

****kwargs**

All parameters supported by *plot*.

**Returns**

**list of *Line2D***

Objects representing the plotted data.

## Examples using `matplotlib.axes.Axes.semilogx`

- *Log Demo*
- *Log Axis*
- *Axis scales*

## matplotlib.axes.Axes.semilogy

`Axes.`**`semilogy`**(**args*, ***kwargs*)

Make a plot with log scaling on the y-axis.

Call signatures:

```
semilogy([x], y, [fmt], data=None, **kwargs)
semilogy([x], y, [fmt], [x2], y2, [fmt2], ..., **kwargs)
```

This is just a thin wrapper around *plot* which additionally changes the y-axis to log scaling. All the concepts and parameters of plot can be used here as well.

The additional parameters *base*, *subs*, and *nonpositive* control the y-axis properties. They are just forwarded to *Axes.set_yscale*.

**Parameters**

**base**

[float, default: 10] Base of the y logarithm.

**subs**

[array-like, optional] The location of the minor yticks. If *None*, reasonable locations are automatically chosen depending on the number of decades in the plot. See `Axes.set_yscale` for details.

**nonpositive**

[{'mask', 'clip'}, default: 'clip'] Non-positive values in y can be masked as invalid, or clipped to a very small positive number.

****kwargs**

All parameters supported by `plot`.

**Returns**

**list of *Line2D***

Objects representing the plotted data.

## Examples using `matplotlib.axes.Axes.semilogy`

- *Log Demo*
- *SkewT-logP diagram: using transforms and custom projections*
- *Axis scales*

## matplotlib.axes.Axes.fill_between

Axes.**fill_between**(*x*, *y1*, *y2=0*, *where=None*, *interpolate=False*, *step=None*, ***, *data=None*, ***kwargs*)

Fill the area between two horizontal curves.

The curves are defined by the points (*x*, *y1*) and (*x*, *y2*). This creates one or multiple polygons describing the filled area.

You may exclude some horizontal sections from filling using *where*.

By default, the edges connect the given points directly. Use *step* if the filling should be a step function, i.e. constant in between *x*.

**Parameters**

**x**

[array (length N)] The x coordinates of the nodes defining the curves.

**y1**

[array (length N) or scalar] The y coordinates of the nodes defining the first curve.

**y2**

[array (length N) or scalar, default: 0] The y coordinates of the nodes defining the second curve.

**where**

[array of bool (length N), optional] Define *where* to exclude some horizontal regions from being filled. The filled regions are defined by the coordinates `x[where]`. More precisely, fill between `x[i]` and `x[i+1]` if `where[i] and where[i+1]`. Note that this definition implies that an isolated *True* value between two *False* values in *where* will not result in filling. Both sides of the *True* position remain unfilled due to the adjacent *False* values.

**interpolate**

[bool, default: False] This option is only relevant if *where* is used and the two curves are crossing each other.

Semantically, *where* is often used for *y1 > y2* or similar. By default, the nodes of the polygon defining the filled region will only be placed at the positions in the *x* array. Such a polygon cannot describe the above semantics close to the intersection. The x-sections containing the intersection are simply clipped.

Setting *interpolate* to *True* will calculate the actual intersection point and extend the filled region up to this point.

**step**

[{'pre', 'post', 'mid'}, optional] Define *step* if the filling should be a step function, i.e. constant in between *x*. The value determines where the step will occur:

- 'pre': The y value is continued constantly to the left from every *x* position, i.e. the interval `(x[i-1], x[i]]` has the value `y[i]`.
- 'post': The y value is continued constantly to the right from every *x* position, i.e. the interval `[x[i], x[i+1])` has the value `y[i]`.
- 'mid': Steps occur half-way between the *x* positions.

**Returns**

***PolyCollection***

A *PolyCollection* containing the plotted polygons.

**Other Parameters**

**data**

[indexable object, optional] If given, the following parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception):

*x*, *y1*, *y2*, *where*

**kwargs

All other keyword arguments are passed on to *PolyCollection*. They control the *Polygon* properties:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a d |
| *alpha* | array-like or scalar or None |
| *animated* | bool |
| *antialiased* or aa or antialiaseds | bool or list of bools |
| *array* | array-like or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clim* | (vmin: float, vmax: float) |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *cmap* | *Colormap* or str or None |
| *color* | *color* or list of RGBA tuples |
| *edgecolor* or ec or edgecolors | *color* or list of *color* or 'face' |
| *facecolor* or facecolors or fc | *color* or list of *color* |
| *figure* | *Figure* |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or dashes or linestyles or ls | str or tuple or list thereof |
| *linewidth* or linewidths or lw | float or list of floats |
| *mouseover* | bool |
| *norm* | *Normalize* or str or None |
| *offset_transform* or transOffset | *Transform* |
| *offsets* | (N, 2) or (2,) array-like |
| *path_effects* | list of *AbstractPathEffect* |
| *paths* | list of array-like |
| *picker* | None or bool or float or callable |
| *pickradius* | float |
| *rasterized* | bool |
| sizes | numpy.ndarray or None |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *urls* | list of str or None |
| *verts* | list of array-like |
| *verts_and_codes* | unknown |
| *visible* | bool |

Table 5 – continued fro

| Property | Description |
|---|---|
| *zorder* | float |

**See also:**

***fill_between***

Fill between two sets of y-values.

***fill_betweenx***

Fill between two sets of x-values.

### Examples using `matplotlib.axes.Axes.fill_between`

- *Fill Between and Alpha*
- *Filling the area between lines*
- *Hatch-filled histograms*
- *Shade regions defined by a logical mask using fill_between*
- *fill_between(x, y1, y2)*

### matplotlib.axes.Axes.fill_betweenx

Axes.**fill_betweenx**(*y*, *x1*, *x2=0*, *where=None*, *step=None*, *interpolate=False*, ***, *data=None*, ***kwargs*)

Fill the area between two vertical curves.

The curves are defined by the points (*y*, *x1*) and (*y*, *x2*). This creates one or multiple polygons describing the filled area.

You may exclude some vertical sections from filling using *where*.

By default, the edges connect the given points directly. Use *step* if the filling should be a step function, i.e. constant in between *y*.

**Parameters**

**y**

[array (length N)] The y coordinates of the nodes defining the curves.

**x1**

[array (length N) or scalar] The x coordinates of the nodes defining the first curve.

**x2**

[array (length N) or scalar, default: 0] The x coordinates of the nodes defining the second curve.

**where**

[array of bool (length N), optional] Define *where* to exclude some vertical regions from being filled. The filled regions are defined by the coordinates `y[where]`. More precisely, fill between `y[i]` and `y[i+1]` if `where[i] and where[i+1]`. Note that this definition implies that an isolated *True* value between two *False* values in *where* will not result in filling. Both sides of the *True* position remain unfilled due to the adjacent *False* values.

**interpolate**

[bool, default: False] This option is only relevant if *where* is used and the two curves are crossing each other.

Semantically, *where* is often used for *x1 > x2* or similar. By default, the nodes of the polygon defining the filled region will only be placed at the positions in the *y* array. Such a polygon cannot describe the above semantics close to the intersection. The y-sections containing the intersection are simply clipped.

Setting *interpolate* to *True* will calculate the actual intersection point and extend the filled region up to this point.

**step**

[{'pre', 'post', 'mid'}, optional] Define *step* if the filling should be a step function, i.e. constant in between *y*. The value determines where the step will occur:

- 'pre': The y value is continued constantly to the left from every *x* position, i.e. the interval `(x[i-1], x[i]]` has the value `y[i]`.
- 'post': The y value is continued constantly to the right from every *x* position, i.e. the interval `[x[i], x[i+1])` has the value `y[i]`.
- 'mid': Steps occur half-way between the *x* positions.

**Returns**

***PolyCollection***

A *PolyCollection* containing the plotted polygons.

**Other Parameters**

**data**

[indexable object, optional] If given, the following parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception):

*y*, *x1*, *x2*, *where*

**kwargs

All other keyword arguments are passed on to *PolyCollection*. They control the *Polygon* properties:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a d |
| *alpha* | array-like or scalar or None |
| *animated* | bool |
| *antialiased* or aa or antialiaseds | bool or list of bools |
| *array* | array-like or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clim* | (vmin: float, vmax: float) |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *cmap* | *Colormap* or str or None |
| *color* | *color* or list of RGBA tuples |
| *edgecolor* or ec or edgecolors | *color* or list of *color* or 'face' |
| *facecolor* or facecolors or fc | *color* or list of *color* |
| *figure* | *Figure* |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or dashes or linestyles or ls | str or tuple or list thereof |
| *linewidth* or linewidths or lw | float or list of floats |
| *mouseover* | bool |
| *norm* | *Normalize* or str or None |
| *offset_transform* or transOffset | *Transform* |
| *offsets* | (N, 2) or (2,) array-like |
| *path_effects* | list of *AbstractPathEffect* |
| *paths* | list of array-like |
| *picker* | None or bool or float or callable |
| *pickradius* | float |
| *rasterized* | bool |
| sizes | numpy.ndarray or None |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *urls* | list of str or None |
| *verts* | list of array-like |
| *verts_and_codes* | unknown |
| *visible* | bool |

Table 6 – continued fro

| Property | Description |
|---|---|
| *zorder* | float |

**See also:**

***fill_between***

Fill between two sets of y-values.

***fill_betweenx***

Fill between two sets of x-values.

### Examples using `matplotlib.axes.Axes.fill_betweenx`

- *Fill Betweenx Demo*
- *Hatch-filled histograms*

### matplotlib.axes.Axes.bar

`Axes.`**`bar`**(*x*, *height*, *width=0.8*, *bottom=None*, *, *align='center'*, *data=None*, ***kwargs*)

Make a bar plot.

The bars are positioned at *x* with the given *align*ment. Their dimensions are given by *height* and *width*. The vertical baseline is *bottom* (default 0).

Many parameters can take either a single value applying to all bars or a sequence of values, one for each bar.

**Parameters**

**x**

[float or array-like] The x coordinates of the bars. See also *align* for the alignment of the bars to the coordinates.

**height**

[float or array-like] The height(s) of the bars.

Note that if *bottom* has units (e.g. datetime), *height* should be in units that are a difference from the value of *bottom* (e.g. timedelta).

**width**

[float or array-like, default: 0.8] The width(s) of the bars.

Note that if *x* has units (e.g. datetime), then *width* should be in units that are a difference (e.g. timedelta) around the *x* values.

**bottom**

[float or array-like, default: 0] The y coordinate(s) of the bottom side(s) of the bars.

Note that if *bottom* has units, then the y-axis will get a Locator and Formatter appropriate for the units (e.g. dates, or categorical).

**align**

[{'center', 'edge'}, default: 'center'] Alignment of the bars to the *x* coordinates:

- 'center': Center the base on the *x* positions.
- 'edge': Align the left edges of the bars with the *x* positions.

To align the bars on the right edge pass a negative *width* and `align='edge'`.

**Returns**

***BarContainer***

Container with all the bars and optionally errorbars.

**Other Parameters**

**color**

[*color* or list of *color*, optional] The colors of the bar faces.

**edgecolor**

[*color* or list of *color*, optional] The colors of the bar edges.

**linewidth**

[float or array-like, optional] Width of the bar edge(s). If 0, don't draw edges.

**tick_label**

[str or list of str, optional] The tick labels of the bars. Default: None (Use default numeric labels.)

**label**

[str or list of str, optional] A single label is attached to the resulting *BarContainer* as a label for the whole dataset. If a list is provided, it must be the same length as *x* and labels the individual bars. Repeated labels are not de-duplicated and will cause repeated label entries, so this is best used when bars also differ in style (e.g., by passing a list to *color*.)

**xerr, yerr**

[float or array-like of shape(N,) or shape(2, N), optional] If not *None*, add horizontal / vertical errorbars to the bar tips. The values are +/- sizes relative to the data:

- scalar: symmetric +/- values for all bars

- shape(N,): symmetric +/- values for each bar
- shape(2, N): Separate - and + values for each bar. First row contains the lower errors, the second row contains the upper errors.
- *None*: No errorbar. (Default)

See *Different ways of specifying error bars* for an example on the usage of *xerr* and *yerr*.

**ecolor**

[*color* or list of *color*, default: 'black'] The line color of the errorbars.

**capsize**

[float, default: `rcParams["errorbar.capsize"]` (default: `0.0`)] The length of the error bar caps in points.

**error_kw**

[dict, optional] Dictionary of keyword arguments to be passed to the `errorbar` method. Values of *ecolor* or *capsize* defined here take precedence over the independent keyword arguments.

**log**

[bool, default: False] If *True*, set the y-axis to be log scale.

**data**

[indexable object, optional] If given, all parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception).

****kwargs**

[`Rectangle` properties]

| Property | Description |
|---|---|
| `agg_filter` | a filter function, which takes a (m, n, 3) float array and a dpi value, and return |
| `alpha` | scalar or None |
| `angle` | unknown |
| `animated` | bool |
| `antialiased` or aa | bool or None |
| `bounds` | (left, bottom, width, height) |
| `capstyle` | `CapStyle` or {'butt', 'projecting', 'round'} |
| `clip_box` | `BboxBase` or None |
| `clip_on` | bool |
| `clip_path` | Patch or (Path, Transform) or None |
| `color` | *color* |
| `edgecolor` or ec | *color* or None |
| `facecolor` or fc | *color* or None |
| `figure` | `Figure` |

Table 7 – continued from previou

| Property | Description |
|---|---|
| *fill* | bool |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *height* | unknown |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float or None |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *width* | unknown |
| *x* | unknown |
| *xy* | (float, float) |
| *y* | unknown |
| *zorder* | float |

**See also:**

***barh***

Plot a horizontal bar plot.

### Notes

Stacked bars can be achieved by passing individual *bottom* values per bar. See *Stacked bar chart*.

## Examples using `matplotlib.axes.Axes.bar`

- *Bar color demo*
- *Bar Label Demo*
- *Stacked bar chart*
- *Grouped bar chart with labels*
- *Hat graph*

- *Watermark image*
- *Percentiles as horizontal bar chart*
- *Bar of pie*
- *Nested pie charts*
- *Bar chart on polar axis*
- *Legend Demo*
- *Ways to set a color's alpha value*
- *Hatch demo*
- *ggplot style sheet*
- *floating_axes features*
- *XKCD*
- *Pick event demo*
- *Create 2D bar graphs in different planes*
- *Log Bar*
- *Custom Ticker*
- *Group barchart with units*
- *bar(x, height)*
- *Quick start guide*
- *Path Tutorial*
- *Plotting dates and strings*

## matplotlib.axes.Axes.barh

`Axes.`**`barh`**(*y*, *width*, *height=0.8*, *left=None*, *, *align='center'*, *data=None*, ***kwargs*)

Make a horizontal bar plot.

The bars are positioned at *y* with the given *align*ment. Their dimensions are given by *width* and *height*. The horizontal baseline is *left* (default 0).

Many parameters can take either a single value applying to all bars or a sequence of values, one for each bar.

**Parameters**

**y**

[float or array-like] The y coordinates of the bars. See also *align* for the alignment of the bars to the coordinates.

**width**

[float or array-like] The width(s) of the bars.

Note that if *left* has units (e.g. datetime), *width* should be in units that are a difference from the value of *left* (e.g. timedelta).

**height**

[float or array-like, default: 0.8] The heights of the bars.

Note that if *y* has units (e.g. datetime), then *height* should be in units that are a difference (e.g. timedelta) around the *y* values.

**left**

[float or array-like, default: 0] The x coordinates of the left side(s) of the bars.

Note that if *left* has units, then the x-axis will get a Locator and Formatter appropriate for the units (e.g. dates, or categorical).

**align**

[{'center', 'edge'}, default: 'center'] Alignment of the base to the *y* coordinates*:

- 'center': Center the bars on the *y* positions.
- 'edge': Align the bottom edges of the bars with the *y* positions.

To align the bars on the top edge pass a negative *height* and `align='edge'`.

**Returns**

***BarContainer***

Container with all the bars and optionally errorbars.

**Other Parameters**

**color**

[*color* or list of *color*, optional] The colors of the bar faces.

**edgecolor**

[*color* or list of *color*, optional] The colors of the bar edges.

**linewidth**

[float or array-like, optional] Width of the bar edge(s). If 0, don't draw edges.

**tick_label**

[str or list of str, optional] The tick labels of the bars. Default: None (Use default numeric labels.)

**label**

[str or list of str, optional] A single label is attached to the resulting *BarContainer* as a label for the whole dataset. If a list is provided, it must be the same length as *y* and labels the individual bars. Repeated labels are not de-duplicated and will cause repeated label entries, so this is best used when bars also differ in style (e.g., by passing a list to *color*.)

**xerr, yerr**

[float or array-like of shape(N,) or shape(2, N), optional] If not *None*, add horizontal / vertical errorbars to the bar tips. The values are +/- sizes relative to the data:

- scalar: symmetric +/- values for all bars
- shape(N,): symmetric +/- values for each bar
- shape(2, N): Separate - and + values for each bar. First row contains the lower errors, the second row contains the upper errors.
- *None*: No errorbar. (default)

See *Different ways of specifying error bars* for an example on the usage of *xerr* and *yerr*.

**ecolor**

[*color* or list of *color*, default: 'black'] The line color of the errorbars.

**capsize**

[float, default: *rcParams["errorbar.capsize"]* (default: `0.0`)] The length of the error bar caps in points.

**error_kw**

[dict, optional] Dictionary of keyword arguments to be passed to the *errorbar* method. Values of *ecolor* or *capsize* defined here take precedence over the independent keyword arguments.

**log**

[bool, default: False] If `True`, set the x-axis to be log scale.

**data**

[indexable object, optional] If given, all parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception).

****kwargs**

[*Rectangle* properties]

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and return |
| *alpha* | scalar or None |
| *angle* | unknown |
| *animated* | bool |
| *antialiased* or aa | bool or None |
| *bounds* | (left, bottom, width, height) |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* | *color* |
| *edgecolor* or ec | *color* or None |
| *facecolor* or fc | *color* or None |
| *figure* | *Figure* |
| *fill* | bool |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *height* | unknown |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float or None |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *width* | unknown |
| *x* | unknown |
| *xy* | (float, float) |
| *y* | unknown |
| *zorder* | float |

**See also:**

***bar***

Plot a vertical bar plot.

### Notes

Stacked bars can be achieved by passing individual *left* values per bar. See *Discrete distribution as horizontal bar chart*.

### Examples using `matplotlib.axes.Axes.barh`

- *Bar Label Demo*
- *Horizontal bar chart*
- *Discrete distribution as horizontal bar chart*
- *Producing multiple histograms side by side*
- *The Lifecycle of a Plot*
- *Animations using Matplotlib*

## matplotlib.axes.Axes.bar_label

`Axes.`**`bar_label`**(*container*, *labels=None*, *, *fmt='%g'*, *label_type='edge'*, *padding=0*, ***kwargs*)

Label a bar plot.

Adds labels to bars in the given `BarContainer`. You may need to adjust the axis limits to fit the labels.

**Parameters**

**container**

[`BarContainer`] Container with all the bars and optionally errorbars, likely returned from `bar` or `barh`.

**labels**

[array-like, optional] A list of label texts, that should be displayed. If not given, the label texts will be the data values formatted with *fmt*.

**fmt**

[str or callable, default: '%g'] An unnamed %-style or {}-style format string for the label or a function to call with the value as the first argument. When *fmt* is a string and can be interpreted in both formats, %-style takes precedence over {}-style.

New in version 3.7: Support for {}-style format string and callables.

**label_type**

[{'edge', 'center'}, default: 'edge'] The label type. Possible values:

- 'edge': label placed at the end-point of the bar segment, and the value displayed will be the position of that end-point.

- 'center': label placed in the center of the bar segment, and the value displayed will be the length of that segment. (useful for stacked bars, i.e., *Bar Label Demo*)

**padding**

[float, default: 0] Distance of label from the end of the bar, in points.

****kwargs**

Any remaining keyword arguments are passed through to `Axes.annotate`. The alignment parameters ( *horizontalalignment* / *ha*, *verticalalignment* / *va*) are not supported because the labels are automatically aligned to the bars.

**Returns**

**list of *Annotation***

A list of `Annotation` instances for the labels.

### Examples using `matplotlib.axes.Axes.bar_label`

- *Bar Label Demo*
- *Grouped bar chart with labels*
- *Discrete distribution as horizontal bar chart*
- *Percentiles as horizontal bar chart*
- *Bar of pie*

## matplotlib.axes.Axes.stem

Axes.**stem**(**args*, *linefmt=None*, *markerfmt=None*, *basefmt=None*, *bottom=0*, *label=None*, *orientation='vertical'*, *data=None*)

Create a stem plot.

A stem plot draws lines perpendicular to a baseline at each location *locs* from the baseline to *heads*, and places a marker there. For vertical stem plots (the default), the *locs* are *x* positions, and the *heads* are *y* values. For horizontal stem plots, the *locs* are *y* positions, and the *heads* are *x* values.

Call signature:

```
stem([locs,] heads, linefmt=None, markerfmt=None, basefmt=None)
```

The *locs*-positions are optional. *linefmt* may be provided as positional, but all other formats must be provided as keyword arguments.

**Parameters**

**locs**

[array-like, default: (0, 1, ..., len(heads) - 1)] For vertical stem plots, the x-positions of the stems. For horizontal stem plots, the y-positions of the stems.

**heads**

[array-like] For vertical stem plots, the y-values of the stem heads. For horizontal stem plots, the x-values of the stem heads.

**linefmt**

[str, optional] A string defining the color and/or linestyle of the vertical lines:

| Character | Line Style |
|---|---|
| `'-'` | solid line |
| `'--'` | dashed line |
| `'-.'` | dash-dot line |
| `':'` | dotted line |

Default: 'C0-', i.e. solid line with the first color of the color cycle.

Note: Markers specified through this parameter (e.g. 'x') will be silently ignored. Instead, markers should be specified using *markerfmt*.

**markerfmt**

[str, optional] A string defining the color and/or shape of the markers at the stem heads. If the marker is not given, use the marker 'o', i.e. filled circles. If the color is not given, use the color from *linefmt*.

**basefmt**

[str, default: 'C3-' ('C2-' in classic mode)] A format string defining the properties of the baseline.

**orientation**

[{'vertical', 'horizontal'}, default: 'vertical'] The orientation of the stems.

**bottom**

[float, default: 0] The y/x-position of the baseline (depending on *orientation*).

**label**

[str, optional] The label to use for the stems in legends.

**data**

[indexable object, optional] If given, all parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception).

**Returns**

*StemContainer*

The container may be treated like a tuple (*markerline*, *stemlines*, *baseline*)

### Notes

**See also:**

The MATLAB function stem which inspired this method.

## Examples using `matplotlib.axes.Axes.stem`

- *Stem Plot*
- *Legend Demo*
- *3D stem*
- *stem(x, y)*

## matplotlib.axes.Axes.eventplot

`Axes.`**`eventplot`**(*positions*, *orientation='horizontal'*, *lineoffsets=1*, *linelengths=1*, *linewidths=None*, *colors=None*, *alpha=None*, *linestyles='solid'*, ***, *data=None*, ***kwargs*)

Plot identical parallel lines at the given positions.

This type of plot is commonly used in neuroscience for representing neural events, where it is usually called a spike raster, dot raster, or raster plot.

However, it is useful in any situation where you wish to show the timing or position of multiple sets of discrete events, such as the arrival times of people to a business on each day of the month or the date of hurricanes each year of the last century.

**Parameters**

**positions**

[array-like or list of array-like] A 1D array-like defines the positions of one sequence of events.

Multiple groups of events may be passed as a list of array-likes. Each group can be styled independently by passing lists of values to *lineoffsets*, *linelengths*, *linewidths*, *colors* and *linestyles*.

Note that *positions* can be a 2D array, but in practice different event groups usually have different counts so that one will use a list of different-length arrays rather than a 2D array.

**orientation**

[{'horizontal', 'vertical'}, default: 'horizontal'] The direction of the event sequence:

- 'horizontal': the events are arranged horizontally. The indicator lines are vertical.
- 'vertical': the events are arranged vertically. The indicator lines are horizontal.

**lineoffsets**

[float or array-like, default: 1] The offset of the center of the lines from the origin, in the direction orthogonal to *orientation*.

If *positions* is 2D, this can be a sequence with length matching the length of *positions*.

**linelengths**

[float or array-like, default: 1] The total height of the lines (i.e. the lines stretches from `lineoffset - linelength/2` to `lineoffset + linelength/2`).

If *positions* is 2D, this can be a sequence with length matching the length of *positions*.

**linewidths**

[float or array-like, default: *rcParams["lines.linewidth"]* (default: `1.5`)] The line width(s) of the event lines, in points.

If *positions* is 2D, this can be a sequence with length matching the length of *positions*.

**colors**

[*color* or list of color, default: *rcParams["lines.color"]* (default: `'C0'`)] The color(s) of the event lines.

If *positions* is 2D, this can be a sequence with length matching the length of *positions*.

**alpha**

[float or array-like, default: 1] The alpha blending value(s), between 0 (transparent) and 1 (opaque).

If *positions* is 2D, this can be a sequence with length matching the length of *positions*.

**linestyles**

[str or tuple or list of such values, default: 'solid'] Default is 'solid'. Valid strings are ['solid', 'dashed', 'dashdot', 'dotted', '-', '--', '-.', ':']. Dash tuples should be of the form:

```
(offset, onoffseq),
```

where *onoffseq* is an even length tuple of on and off ink in points.

If *positions* is 2D, this can be a sequence with length matching the length of *positions*.

**data**

[indexable object, optional] If given, the following parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception):

*positions*, *lineoffsets*, *linelengths*, *linewidths*, *colors*, *linestyles*

****kwargs**

Other keyword arguments are line collection properties. See `LineCollection` for a list of the valid properties.

**Returns**

**list of *EventCollection***

The `EventCollection` that were added.

### Notes

For *linelengths*, *linewidths*, *colors*, *alpha* and *linestyles*, if only a single value is given, that value is applied to all lines. If an array-like is given, it must have the same length as *positions*, and each value will be applied to the corresponding row of the array.

### Examples

## Examples using `matplotlib.axes.Axes.eventplot`

- *eventplot(D)*

## matplotlib.axes.Axes.pie

`Axes.`**`pie`**(*x*, *explode=None*, *labels=None*, *colors=None*, *autopct=None*, *pctdistance=0.6*, *shadow=False*, *labeldistance=1.1*, *startangle=0*, *radius=1*, *counterclock=True*, *wedgeprops=None*, *textprops=None*, *center=(0, 0)*, *frame=False*, *rotatelabels=False*, ***, *normalize=True*, *hatch=None*, *data=None*)

Plot a pie chart.

Make a pie chart of array *x*. The fractional area of each wedge is given by `x/sum(x)`.

The wedges are plotted counterclockwise, by default starting from the x-axis.

**Parameters**

**x**

[1D array-like] The wedge sizes.

**explode**

[array-like, default: None] If not *None*, is a `len(x)` array which specifies the fraction of the radius with which to offset each wedge.

**labels**

[list, default: None] A sequence of strings providing the labels for each wedge

**colors**

[*color* or list of *color*, default: None] A sequence of colors through which the pie chart will cycle. If *None*, will use the colors in the currently active cycle.

**hatch**

[str or list, default: None] Hatching pattern applied to all pie wedges or sequence of patterns through which the chart will cycle. For a list of valid patterns, see *Hatch style reference*.

New in version 3.7.

**autopct**

[None or str or callable, default: None] If not *None*, *autopct* is a string or function used to label the wedges with their numeric value. The label will be placed inside the wedge. If *autopct* is a format string, the label will be `fmt % pct`. If *autopct* is a function, then it will be called.

**pctdistance**

[float, default: 0.6] The relative distance along the radius at which the text generated by *autopct* is drawn. To draw the text outside the pie, set *pctdistance* > 1. This parameter is ignored if *autopct* is `None`.

**labeldistance**

[float or None, default: 1.1] The relative distance along the radius at which the labels are drawn. To draw the labels inside the pie, set *labeldistance* < 1. If set to `None`, labels are not drawn but are still stored for use in *legend*.

**shadow**

[bool or dict, default: False] If bool, whether to draw a shadow beneath the pie. If dict, draw a shadow passing the properties in the dict to *Shadow*.

New in version 3.8: *shadow* can be a dict.

**startangle**

[float, default: 0 degrees] The angle by which the start of the pie is rotated, counterclockwise from the x-axis.

**radius**

[float, default: 1] The radius of the pie.

**counterclock**

[bool, default: True] Specify fractions direction, clockwise or counterclockwise.

**wedgeprops**

[dict, default: None] Dict of arguments passed to each *patches.Wedge* of the pie. For example, `wedgeprops = {'linewidth': 3}` sets the width of the wedge border lines equal to 3. By default, `clip_on=False`. When there is a conflict between these properties and other keywords, properties passed to *wedgeprops* take precedence.

**textprops**

[dict, default: None] Dict of arguments to pass to the text objects.

**center**

[(float, float), default: (0, 0)] The coordinates of the center of the chart.

**frame**

[bool, default: False] Plot Axes frame with the chart if true.

**rotatelabels**

[bool, default: False] Rotate each label to the angle of the corresponding slice if true.

**normalize**

[bool, default: True] When *True*, always make a full pie by normalizing x so that `sum(x) == 1`. *False* makes a partial pie if `sum(x) <= 1` and raises a `ValueError` for `sum(x) > 1`.

**data**

[indexable object, optional] If given, the following parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception):

*x*, *explode*, *labels*, *colors*

**Returns**

**patches**

[list] A sequence of *matplotlib.patches.Wedge* instances

**texts**

[list] A list of the label *Text* instances.

**autotexts**

[list] A list of *Text* instances for the numeric labels. This will only be returned if the parameter *autopct* is not *None*.

### Notes

The pie chart will probably look best if the figure and Axes are square, or the Axes aspect is equal. This method sets the aspect ratio of the axis to "equal". The Axes aspect ratio can be controlled with *Axes.set_aspect*.

### Examples using `matplotlib.axes.Axes.pie`

- *Pie charts*
- *Bar of pie*
- *Nested pie charts*
- *Labeling a pie and a donut*
- *SVG filter pie*
- *pie(x)*

## matplotlib.axes.Axes.stackplot

Axes.**stackplot**(*x*, **args*, *labels=()*, *colors=None*, *hatch=None*, *baseline='zero'*, *data=None*, ***kwargs*)

Draw a stacked area plot or a streamgraph.

**Parameters**

**x**

[(N,) array-like]

**y**

[(M, N) array-like] The data is assumed to be unstacked. Each of the following calls is legal:

```
stackplot(x, y)           # where y has shape (M, N)
stackplot(x, y1, y2, y3)  # where y1, y2, y3, y4 have␣
 ↪length N
```

**baseline**

[{'zero', 'sym', 'wiggle', 'weighted_wiggle'}] Method used to calculate the baseline:

- `'zero'`: Constant zero baseline, i.e. a simple stacked plot.
- `'sym'`: Symmetric around zero and is sometimes called 'ThemeRiver'.
- `'wiggle'`: Minimizes the sum of the squared slopes.

- `'weighted_wiggle'`: Does the same but weights to account for size of each layer. It is also called 'Streamgraph'-layout. More details can be found at http://leebyron.com/streamgraph/.

**labels**

[list of str, optional] A sequence of labels to assign to each data series. If unspecified, then no labels will be applied to artists.

**colors**

[list of *color*, optional] A sequence of colors to be cycled through and used to color the stacked areas. The sequence need not be exactly the same length as the number of provided *y*, in which case the colors will repeat from the beginning.

If not specified, the colors from the Axes property cycle will be used.

**hatch**

[list of str, default: None] A sequence of hatching styles. See *Hatch style reference*. The sequence will be cycled through for filling the stacked areas from bottom to top. It need not be exactly the same length as the number of provided *y*, in which case the styles will repeat from the beginning.

**data**

[indexable object, optional] If given, all parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception).

****kwargs**

All other keyword arguments are passed to *`Axes.fill_between`*.

**Returns**

**list of *`PolyCollection`***

A list of *`PolyCollection`* instances, one for each element in the stacked area plot.

## Examples using `matplotlib.axes.Axes.stackplot`

- *Stackplots and streamgraphs*
- *stackplot(x, y)*

### matplotlib.axes.Axes.broken_barh

Axes.**broken_barh**(*xranges*, *yrange*, *, *data=None*, ***kwargs*)

Plot a horizontal sequence of rectangles.

A rectangle is drawn for each element of *xranges*. All rectangles have the same vertical position and size defined by *yrange*.

**Parameters**

**xranges**

[sequence of tuples (*xmin*, *xwidth*)] The x-positions and extents of the rectangles. For each tuple (*xmin*, *xwidth*) a rectangle is drawn from *xmin* to *xmin* + *xwidth*.

**yrange**

[(*ymin*, *yheight*)] The y-position and extent for all the rectangles.

**Returns**

*PolyCollection*

**Other Parameters**

**data**

[indexable object, optional] If given, all parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception).

****kwargs**

[*PolyCollection* properties] Each *kwarg* can be either a single argument applying to all rectangles, e.g.:

```
facecolors='black'
```

or a sequence of arguments over which is cycled, e.g.:

```
facecolors=('black', 'blue')
```

would create interleaving black and blue rectangles.

Supported keywords:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a d |
| *alpha* | array-like or scalar or None |
| *animated* | bool |
| *antialiased* or aa or antialiaseds | bool or list of bools |

Table 9 – continued fro

| Property | Description |
|---|---|
| *array* | array-like or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clim* | (vmin: float, vmax: float) |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *cmap* | *Colormap* or str or None |
| *color* | *color* or list of RGBA tuples |
| *edgecolor* or ec or edgecolors | *color* or list of *color* or 'face' |
| *facecolor* or facecolors or fc | *color* or list of *color* |
| *figure* | *Figure* |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or dashes or linestyles or ls | str or tuple or list thereof |
| *linewidth* or linewidths or lw | float or list of floats |
| *mouseover* | bool |
| *norm* | *Normalize* or str or None |
| *offset_transform* or transOffset | *Transform* |
| *offsets* | (N, 2) or (2,) array-like |
| *path_effects* | list of *AbstractPathEffect* |
| *paths* | list of array-like |
| *picker* | None or bool or float or callable |
| *pickradius* | float |
| *rasterized* | bool |
| sizes | numpy.ndarray or None |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *urls* | list of str or None |
| *verts* | list of array-like |
| *verts_and_codes* | unknown |
| *visible* | bool |
| *zorder* | float |

### Examples using `matplotlib.axes.Axes.broken_barh`

- *Broken Barh*

## matplotlib.axes.Axes.vlines

`Axes.`**`vlines`**(*x*, *ymin*, *ymax*, *colors=None*, *linestyles='solid'*, *label=''*, ***, *data=None*, ***kwargs*)

Plot vertical lines at each *x* from *ymin* to *ymax*.

**Parameters**

**x**

[float or array-like] x-indexes where to plot the lines.

**ymin, ymax**

[float or array-like] Respective beginning and end of each line. If scalars are provided, all lines will have the same length.

**colors**

[*color* or list of color, default: *rcParams["lines.color"]* (default: `'C0'`)]

**linestyles**

[{'solid', 'dashed', 'dashdot', 'dotted'}, default: 'solid']

**label**

[str, default: '']

**Returns**

***LineCollection***

**Other Parameters**

**data**

[indexable object, optional] If given, the following parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception):

*x*, *ymin*, *ymax*, *colors*

****kwargs**

[*LineCollection* properties.]

**See also:**

***hlines***

horizontal lines

***axvline***

vertical line across the Axes

### Examples using `matplotlib.axes.Axes.vlines`

- *Creating a timeline with lines, dates, and text*
- *hlines and vlines*
- *Violin plot customization*
- *Angle annotations on bracket arrows*

## matplotlib.axes.Axes.hlines

`Axes.`**`hlines`**(*y*, *xmin*, *xmax*, *colors=None*, *linestyles='solid'*, *label=''*, ***, *data=None*, ***kwargs*)

Plot horizontal lines at each *y* from *xmin* to *xmax*.

**Parameters**

**y**

[float or array-like] y-indexes where to plot the lines.

**xmin, xmax**

[float or array-like] Respective beginning and end of each line. If scalars are provided, all lines will have the same length.

**colors**

[*color* or list of color , default: *rcParams["lines.color"]* (default: `'C0'`)]

**linestyles**

[{'solid', 'dashed', 'dashdot', 'dotted'}, default: 'solid']

**label**

[str, default: '']

**Returns**

***LineCollection***

**Other Parameters**

**data**

[indexable object, optional] If given, the following parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception):

*y*, *xmin*, *xmax*, *colors*

****kwargs**

[*LineCollection* properties.]

**See also:**

***vlines***

vertical lines

***axhline***

horizontal line across the Axes

## Examples using `matplotlib.axes.Axes.hlines`

- *hlines and vlines*
- *Specifying colors*

## matplotlib.axes.Axes.fill

`Axes.`**`fill`**`(`**args*, *data=None*, ***kwargs*`)`

Plot filled polygons.

**Parameters**

***args**

[sequence of x, y, [color]] Each polygon is defined by the lists of *x* and *y* positions of its nodes, optionally followed by a *color* specifier. See *matplotlib.colors* for supported color specifiers. The standard color cycle is used for polygons without a color specifier.

You can plot multiple polygons by providing multiple *x*, *y*, *[color]* groups.

For example, each of the following is legal:

```
ax.fill(x, y)                    # a polygon with default␣
 ↪color
ax.fill(x, y, "b")               # a blue polygon
ax.fill(x, y, x2, y2)            # two polygons
ax.fill(x, y, "b", x2, y2, "r")  # a blue and a red polygon
```

**data**

[indexable object, optional] An object with labelled data. If given, provide the label names to plot in *x* and *y*, e.g.:

```
ax.fill("time", "signal",
        data={"time": [0, 1, 2], "signal": [0, 1, 0]})
```

**Returns**

**list of *Polygon***

**Other Parameters**

****kwargs**

[*Polygon* properties]

### Notes

Use *fill_between()* if you would like to fill the region between two curves.

### Examples using `matplotlib.axes.Axes.fill`

- *Filled polygon*
- *Radar chart (aka spider or star chart)*
- *Ellipse with units*

### Spans

| | |
|---|---|
| *Axes.axhline* | Add a horizontal line across the Axes. |
| *Axes.axhspan* | Add a horizontal span (rectangle) across the Axes. |
| *Axes.axvline* | Add a vertical line across the Axes. |
| *Axes.axvspan* | Add a vertical span (rectangle) across the Axes. |
| *Axes.axline* | Add an infinitely long straight line. |

### matplotlib.axes.Axes.axhline

Axes.**axhline**(*y=0*, *xmin=0*, *xmax=1*, ***kwargs*)

Add a horizontal line across the Axes.

**Parameters**

**y**

[float, default: 0] y position in data coordinates of the horizontal line.

**xmin**

[float, default: 0] Should be between 0 and 1, 0 being the far left of the plot, 1 the far right of the plot.

**xmax**

[float, default: 1] Should be between 0 and 1, 0 being the far left of the plot, 1 the far right of the plot.

**Returns**

*Line2D*

**Other Parameters**

****kwargs**

Valid keyword arguments are *Line2D* properties, except for 'transform':

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi va |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* or aa | bool |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* or c | *color* |
| *dash_capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *dash_joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *dashes* | sequence of floats (on/off ink in points) or (None, None) |
| *data* | (2, N) array or two 1D arrays |
| *drawstyle* or ds | {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: |
| *figure* | *Figure* |
| *fillstyle* | {'full', 'left', 'right', 'bottom', 'top', 'none'} |
| *gapcolor* | *color* or None |
| *gid* | str |

Table 10 – continued from

| Property | Description |
|---|---|
| *in_layout* | bool |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float |
| *marker* | marker style string, *Path* or *MarkerStyle* |
| *markeredgecolor* or mec | *color* |
| *markeredgewidth* or mew | float |
| *markerfacecolor* or mfc | *color* |
| *markerfacecoloralt* or mfcalt | *color* |
| *markersize* or ms | float |
| *markevery* | None or int or (int, int) or slice or list[int] or float or (float, floa |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | float or callable[[Artist, Event], tuple[bool, dict]] |
| *pickradius* | float |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *solid_capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *solid_joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *transform* | unknown |
| *url* | str |
| *visible* | bool |
| *xdata* | 1D array |
| *ydata* | 1D array |
| *zorder* | float |

**See also:**

***hlines***

Add horizontal lines in data coordinates.

***axhspan***

Add a horizontal span (rectangle) across the axis.

***axline***

Add a line with an arbitrary slope.

### Examples

- draw a thick red hline at 'y' = 0 that spans the xrange:

```
>>> axhline(linewidth=4, color='r')
```

- draw a default hline at 'y' = 1 that spans the xrange:

```
>>> axhline(y=1)
```

- draw a default hline at 'y' = .5 that spans the middle half of the xrange:

```
>>> axhline(y=.5, xmin=0.25, xmax=0.75)
```

## Examples using `matplotlib.axes.Axes.axhline`

- *Filling the area between lines*
- *Shade regions defined by a logical mask using fill_between*
- *Creating a timeline with lines, dates, and text*
- *axhspan Demo*
- *Plot a confidence ellipse of a two-dimensional dataset*
- *Bihistogram*
- *Multiline*
- *Usetex Baseline Test*
- *Colors in the default property cycle*
- *Infinite lines*
- *Cross-hair cursor*

## matplotlib.axes.Axes.axhspan

`Axes.`**`axhspan`**(*ymin*, *ymax*, *xmin=0*, *xmax=1*, ***kwargs*)

Add a horizontal span (rectangle) across the Axes.

The rectangle spans from *ymin* to *ymax* vertically, and, by default, the whole x-axis horizontally. The x-span can be set using *xmin* (default: 0) and *xmax* (default: 1) which are in axis units; e.g. `xmin = 0.5` always refers to the middle of the x-axis regardless of the limits set by *set_xlim*.

**Parameters**

**ymin**

[float] Lower y-coordinate of the span, in data units.

**ymax**

[float] Upper y-coordinate of the span, in data units.

**xmin**

[float, default: 0] Lower x-coordinate of the span, in x-axis (0-1) units.

**xmax**

[float, default: 1] Upper x-coordinate of the span, in x-axis (0-1) units.

### Returns

***Polygon***

Horizontal span (rectangle) from (xmin, ymin) to (xmax, ymax).

### Other Parameters

****kwargs**

[*Polygon* properties]

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and retur |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* or aa | bool or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *closed* | bool |
| *color* | *color* |
| *edgecolor* or ec | *color* or None |
| *facecolor* or fc | *color* or None |
| *figure* | *Figure* |
| *fill* | bool |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float or None |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |

Table 11 – continued from previc

| Property | Description |
|---|---|
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *xy* | (N, 2) array-like |
| *zorder* | float |

**See also:**

***axvspan***

Add a vertical span across the Axes.

### Examples using `matplotlib.axes.Axes.axhspan`

- *Fill Between and Alpha*
- *axhspan Demo*

## matplotlib.axes.Axes.axvline

Axes.**axvline**(*x=0*, *ymin=0*, *ymax=1*, ***kwargs*)

Add a vertical line across the Axes.

**Parameters**

**x**

[float, default: 0] x position in data coordinates of the vertical line.

**ymin**

[float, default: 0] Should be between 0 and 1, 0 being the bottom of the plot, 1 the top of the plot.

**ymax**

[float, default: 1] Should be between 0 and 1, 0 being the bottom of the plot, 1 the top of the plot.

**Returns**

***Line2D***

### Other Parameters

#### **kwargs

Valid keyword arguments are *Line2D* properties, except for 'transform':

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi va |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* or aa | bool |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* or c | *color* |
| *dash_capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *dash_joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *dashes* | sequence of floats (on/off ink in points) or (None, None) |
| *data* | (2, N) array or two 1D arrays |
| *drawstyle* or ds | {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: |
| *figure* | *Figure* |
| *fillstyle* | {'full', 'left', 'right', 'bottom', 'top', 'none'} |
| *gapcolor* | *color* or None |
| *gid* | str |
| *in_layout* | bool |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float |
| *marker* | marker style string, *Path* or *MarkerStyle* |
| *markeredgecolor* or mec | *color* |
| *markeredgewidth* or mew | float |
| *markerfacecolor* or mfc | *color* |
| *markerfacecoloralt* or mfcalt | *color* |
| *markersize* or ms | float |
| *markevery* | None or int or (int, int) or slice or list[int] or float or (float, floa |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | float or callable[[Artist, Event], tuple[bool, dict]] |
| *pickradius* | float |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *solid_capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *solid_joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *transform* | unknown |
| *url* | str |

Table 12 – continued from

| Property | Description |
|---|---|
| *visible* | bool |
| *xdata* | 1D array |
| *ydata* | 1D array |
| *zorder* | float |

**See also:**

***vlines***

Add vertical lines in data coordinates.

***axvspan***

Add a vertical span (rectangle) across the axis.

***axline***

Add a line with an arbitrary slope.

### Examples

- draw a thick red vline at $x = 0$ that spans the yrange:

```
>>> axvline(linewidth=4, color='r')
```

- draw a default vline at $x = 1$ that spans the yrange:

```
>>> axvline(x=1)
```

- draw a default vline at $x = .5$ that spans the middle half of the yrange:

```
>>> axvline(x=.5, ymin=0.25, ymax=0.75)
```

## Examples using `matplotlib.axes.Axes.axvline`

- *axhspan Demo*
- *Plot a confidence ellipse of a two-dimensional dataset*
- *Usetex Baseline Test*
- *Colors in the default property cycle*
- *Infinite lines*
- *Cross-hair cursor*
- *SkewT-logP diagram: using transforms and custom projections*

- *The Lifecycle of a Plot*
- *Annotations*

### matplotlib.axes.Axes.axvspan

Axes.**axvspan**(*xmin*, *xmax*, *ymin=0*, *ymax=1*, ***kwargs*)

Add a vertical span (rectangle) across the Axes.

The rectangle spans from *xmin* to *xmax* horizontally, and, by default, the whole y-axis vertically. The y-span can be set using *ymin* (default: 0) and *ymax* (default: 1) which are in axis units; e.g. `ymin = 0.5` always refers to the middle of the y-axis regardless of the limits set by *set_ylim*.

**Parameters**

**xmin**

[float] Lower x-coordinate of the span, in data units.

**xmax**

[float] Upper x-coordinate of the span, in data units.

**ymin**

[float, default: 0] Lower y-coordinate of the span, in y-axis units (0-1).

**ymax**

[float, default: 1] Upper y-coordinate of the span, in y-axis units (0-1).

**Returns**

***Polygon***

Vertical span (rectangle) from (xmin, ymin) to (xmax, ymax).

**Other Parameters**

****kwargs**

[*Polygon* properties]

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and retur |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* or aa | bool or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clip_box* | *BboxBase* or None |

Table 13 – continued from previc

| Property | Description |
|---|---|
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *closed* | bool |
| *color* | *color* |
| *edgecolor* or ec | *color* or None |
| *facecolor* or fc | *color* or None |
| *figure* | *Figure* |
| *fill* | bool |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float or None |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *xy* | (N, 2) array-like |
| *zorder* | float |

**See also:**

***axhspan***

Add a horizontal span across the Axes.

## Examples

Draw a vertical, green, translucent rectangle from x = 1.25 to x = 1.55 that spans the yrange of the Axes.

```
>>> axvspan(1.25, 1.55, facecolor='g', alpha=0.5)
```

### Examples using `matplotlib.axes.Axes.axvspan`

- *Fill Between and Alpha*
- *axhspan Demo*

## matplotlib.axes.Axes.axline

`Axes.`**`axline`**(*xy1*, *xy2=None*, *, *slope=None*, ***kwargs*)

Add an infinitely long straight line.

The line can be defined either by two points *xy1* and *xy2*, or by one point *xy1* and a *slope*.

This draws a straight line "on the screen", regardless of the x and y scales, and is thus also suitable for drawing exponential decays in semilog plots, power laws in loglog plots, etc. However, *slope* should only be used with linear scales; It has no clear meaning for all other scales, and thus the behavior is undefined. Please specify the line using the points *xy1*, *xy2* for non-linear scales.

The *transform* keyword argument only applies to the points *xy1*, *xy2*. The *slope* (if given) is always in data coordinates. This can be used e.g. with `ax.transAxes` for drawing grid lines with a fixed slope.

**Parameters**

**xy1, xy2**

[(float, float)] Points for the line to pass through. Either *xy2* or *slope* has to be given.

**slope**

[float, optional] The slope of the line. Either *xy2* or *slope* has to be given.

**Returns**

***AxLine***

**Other Parameters**

****kwargs**

Valid kwargs are *Line2D* properties

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi va |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* or aa | bool |
| *clip_box* | *BboxBase* or None |

Table 14 – continued from

| Property | Description |
|---|---|
| clip_on | bool |
| clip_path | Patch or (Path, Transform) or None |
| color or c | color |
| dash_capstyle | CapStyle or {'butt', 'projecting', 'round'} |
| dash_joinstyle | JoinStyle or {'miter', 'round', 'bevel'} |
| dashes | sequence of floats (on/off ink in points) or (None, None) |
| data | (2, N) array or two 1D arrays |
| drawstyle or ds | {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: |
| figure | Figure |
| fillstyle | {'full', 'left', 'right', 'bottom', 'top', 'none'} |
| gapcolor | color or None |
| gid | str |
| in_layout | bool |
| label | object |
| linestyle or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| linewidth or lw | float |
| marker | marker style string, Path or MarkerStyle |
| markeredgecolor or mec | color |
| markeredgewidth or mew | float |
| markerfacecolor or mfc | color |
| markerfacecoloralt or mfcalt | color |
| markersize or ms | float |
| markevery | None or int or (int, int) or slice or list[int] or float or (float, floa |
| mouseover | bool |
| path_effects | list of AbstractPathEffect |
| picker | float or callable[[Artist, Event], tuple[bool, dict]] |
| pickradius | float |
| rasterized | bool |
| sketch_params | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| solid_capstyle | CapStyle or {'butt', 'projecting', 'round'} |
| solid_joinstyle | JoinStyle or {'miter', 'round', 'bevel'} |
| transform | unknown |
| url | str |
| visible | bool |
| xdata | 1D array |
| ydata | 1D array |
| zorder | float |

**See also:**

**axhline**

for horizontal lines

***axvline***

for vertical lines

### Examples

Draw a thick red line passing through (0, 0) and (1, 1):

```
>>> axline((0, 0), (1, 1), linewidth=4, color='r')
```

### Examples using `matplotlib.axes.Axes.axline`

- *axhspan Demo*
- *Infinite lines*
- *Anscombe's quartet*

## Spectral

| | |
|---|---|
| *Axes.acorr* | Plot the autocorrelation of *x*. |
| *Axes.angle_spectrum* | Plot the angle spectrum. |
| *Axes.cohere* | Plot the coherence between *x* and *y*. |
| *Axes.csd* | Plot the cross-spectral density. |
| *Axes.magnitude_spectrum* | Plot the magnitude spectrum. |
| *Axes.phase_spectrum* | Plot the phase spectrum. |
| *Axes.psd* | Plot the power spectral density. |
| *Axes.specgram* | Plot a spectrogram. |
| *Axes.xcorr* | Plot the cross correlation between *x* and *y*. |

## matplotlib.axes.Axes.acorr

Axes.**acorr**(*x*, *, *data=None*, ***kwargs*)

Plot the autocorrelation of *x*.

**Parameters**

**x**

[array-like] Not run through Matplotlib's unit conversion, so this should be a unitless array.

**detrend**

[callable, default: *mlab.detrend_none* (no detrending)] A detrending function applied to *x*. It must have the signature

```
detrend(x: np.ndarray) -> np.ndarray
```

**normed**

[bool, default: True] If `True`, input vectors are normalised to unit length.

**usevlines**

[bool, default: True] Determines the plot style.

If `True`, vertical lines are plotted from 0 to the acorr value using *`Axes.vlines`*. Additionally, a horizontal line is plotted at y=0 using *`Axes.axhline`*.

If `False`, markers are plotted at the acorr values using *`Axes.plot`*.

**maxlags**

[int, default: 10] Number of lags to show. If `None`, will return all `2 * len(x) - 1` lags.

### Returns

**lags**

[array (length `2*maxlags+1`)] The lag vector.

**c**

[array (length `2*maxlags+1`)] The auto correlation vector.

**line**

[*`LineCollection`* or *`Line2D`*] *`Artist`* added to the Axes of the correlation:

- *`LineCollection`* if *usevlines* is True.
- *`Line2D`* if *usevlines* is False.

**b**

[*`Line2D`* or None] Horizontal line at 0 if *usevlines* is True None *usevlines* is False.

### Other Parameters

**linestyle**

[*`Line2D`* property, optional] The linestyle for plotting the data points. Only used if *usevlines* is `False`.

**marker**

[str, default: 'o'] The marker for plotting the data points. Only used if *usevlines* is `False`.

**data**

[indexable object, optional] If given, the following parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception):

*x*

****kwargs**

Additional parameters are passed to *Axes.vlines* and *Axes.axhline* if *usevlines* is `True`; otherwise they are passed to *Axes.plot*.

### Notes

The cross correlation is performed with `numpy.correlate` with `mode = "full"`.

## Examples using `matplotlib.axes.Axes.acorr`

- *Cross- and auto-correlation*

## matplotlib.axes.Axes.angle_spectrum

`Axes.`**`angle_spectrum`**(*x*, *Fs=None*, *Fc=None*, *window=None*, *pad_to=None*, *sides=None*, *, *data=None*, ***kwargs*)

Plot the angle spectrum.

Compute the angle spectrum (wrapped phase spectrum) of *x*. Data is padded to a length of *pad_to* and the windowing function *window* is applied to the signal.

**Parameters**

**x**

[1-D array or sequence] Array or sequence containing the data.

**Fs**

[float, default: 2] The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, *freqs*, in cycles per time unit.

**window**

[callable or ndarray, default: *window_hanning*] A function or a vector of length *NFFT*. To create window vectors see *window_hanning*, *window_none*, `numpy.blackman`, `numpy.hamming`, `numpy.bartlett`, `scipy.signal`, `scipy.signal.get_window`, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

**sides**

[{'default', 'onesided', 'twosided'}, optional] Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

**pad_to**

[int, optional] The number of points to which the data segment is padded when performing the FFT. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to `fft`. The default is None, which sets *pad_to* equal to the length of the input signal (i.e. no padding).

**Fc**

[int, default: 0] The center frequency of *x*, which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

**Returns**

**spectrum**

[1-D array] The values for the angle spectrum in radians (real valued).

**freqs**

[1-D array] The frequencies corresponding to the elements in *spectrum*.

**line**

[*Line2D*] The line created by this function.

**Other Parameters**

**data**

[indexable object, optional] If given, the following parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception):

*x*

****kwargs**

Keyword arguments control the *Line2D* properties:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi va |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* or aa | bool |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* or c | *color* |
| *dash_capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |

Table 15 – continued from

| Property | Description |
|---|---|
| *dash_joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *dashes* | sequence of floats (on/off ink in points) or (None, None) |
| *data* | (2, N) array or two 1D arrays |
| *drawstyle* or ds | {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: |
| *figure* | *Figure* |
| *fillstyle* | {'full', 'left', 'right', 'bottom', 'top', 'none'} |
| *gapcolor* | *color* or None |
| *gid* | str |
| *in_layout* | bool |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float |
| *marker* | marker style string, *Path* or *MarkerStyle* |
| *markeredgecolor* or mec | *color* |
| *markeredgewidth* or mew | float |
| *markerfacecolor* or mfc | *color* |
| *markerfacecoloralt* or mfcalt | *color* |
| *markersize* or ms | float |
| *markevery* | None or int or (int, int) or slice or list[int] or float or (float, floa |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | float or callable[[Artist, Event], tuple[bool, dict]] |
| *pickradius* | float |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *solid_capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *solid_joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *transform* | unknown |
| *url* | str |
| *visible* | bool |
| *xdata* | 1D array |
| *ydata* | 1D array |
| *zorder* | float |

**See also:**

***magnitude_spectrum***

Plots the magnitudes of the corresponding frequencies.

***phase_spectrum***

Plots the unwrapped version of this function.

***specgram***

Can plot the angle spectrum of segments within the signal in a colormap.

### matplotlib.axes.Axes.cohere

Axes.**cohere**(*x*, *y*, *NFFT=256*, *Fs=2*, *Fc=0*, *detrend=<function detrend_none>*, *window=<function window_hanning>*, *noverlap=0*, *pad_to=None*, *sides='default'*, *scale_by_freq=None*, ***, *data=None*, ***kwargs*)

Plot the coherence between *x* and *y*.

Coherence is the normalized cross spectral density:

$$C_{xy} = \frac{|P_{xy}|^2}{P_{xx}P_{yy}}$$

**Parameters**

**Fs**

[float, default: 2] The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, *freqs*, in cycles per time unit.

**window**

[callable or ndarray, default: *window_hanning*] A function or a vector of length *NFFT*. To create window vectors see *window_hanning*, *window_none*, numpy.blackman, numpy.hamming, numpy.bartlett, scipy.signal, scipy.signal.get_window, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

**sides**

[{'default', 'onesided', 'twosided'}, optional] Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

**pad_to**

[int, optional] The number of points to which the data segment is padded when performing the FFT. This can be different from *NFFT*, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to fft. The default is None, which sets *pad_to* equal to *NFFT*

**NFFT**

[int, default: 256] The number of data points used in each block for the FFT. A power 2 is most efficient. This should *NOT* be used to get zero padding, or the scaling of the result will be incorrect; use *pad_to* for this instead.

**detrend**

[{'none', 'mean', 'linear'} or callable, default: 'none'] The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the *detrend* parameter is a vector, in Matplotlib it is a function. The *mlab* module defines *detrend_none*, *detrend_mean*, and *detrend_linear*, but you can use a custom function as well. You can also use a string to choose one of the functions: 'none' calls *detrend_none*. 'mean' calls *detrend_mean*. 'linear' calls *detrend_linear*.

**scale_by_freq**

[bool, default: True] Whether the resulting density values should be scaled by the scaling frequency, which gives density in units of 1/Hz. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

**noverlap**

[int, default: 0 (no overlap)] The number of points of overlap between blocks.

**Fc**

[int, default: 0] The center frequency of *x*, which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

**Returns**

**Cxy**

[1-D array] The coherence vector.

**freqs**

[1-D array] The frequencies for the elements in *Cxy*.

**Other Parameters**

**data**

[indexable object, optional] If given, the following parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception):

*x*, *y*

****kwargs**

Keyword arguments control the *Line2D* properties:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi va |
| *alpha* | scalar or None |

Table 16 – continued from

| Property | Description |
|---|---|
| *animated* | bool |
| *antialiased* or aa | bool |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* or c | *color* |
| *dash_capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *dash_joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *dashes* | sequence of floats (on/off ink in points) or (None, None) |
| *data* | (2, N) array or two 1D arrays |
| *drawstyle* or ds | {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: |
| *figure* | *Figure* |
| *fillstyle* | {'full', 'left', 'right', 'bottom', 'top', 'none'} |
| *gapcolor* | *color* or None |
| *gid* | str |
| *in_layout* | bool |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float |
| *marker* | marker style string, *Path* or *MarkerStyle* |
| *markeredgecolor* or mec | *color* |
| *markeredgewidth* or mew | float |
| *markerfacecolor* or mfc | *color* |
| *markerfacecoloralt* or mfcalt | *color* |
| *markersize* or ms | float |
| *markevery* | None or int or (int, int) or slice or list[int] or float or (float, floa |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | float or callable[[Artist, Event], tuple[bool, dict]] |
| *pickradius* | float |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *solid_capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *solid_joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *transform* | unknown |
| *url* | str |
| *visible* | bool |
| *xdata* | 1D array |
| *ydata* | 1D array |
| *zorder* | float |

### References

Bendat & Piersol -- Random Data: Analysis and Measurement Procedures, John Wiley & Sons (1986)

## matplotlib.axes.Axes.csd

Axes.**csd**(*x*, *y*, *NFFT=None*, *Fs=None*, *Fc=None*, *detrend=None*, *window=None*, *noverlap=None*, *pad_to=None*, *sides=None*, *scale_by_freq=None*, *return_line=None*, *, *data=None*, ***kwargs*)

Plot the cross-spectral density.

The cross spectral density $P_{xy}$ by Welch's average periodogram method. The vectors $x$ and $y$ are divided into *NFFT* length segments. Each segment is detrended by function *detrend* and windowed by function *window*. *noverlap* gives the length of the overlap between segments. The product of the direct FFTs of $x$ and $y$ are averaged over each segment to compute $P_{xy}$, with a scaling to correct for power loss due to windowing.

If len($x$) < *NFFT* or len($y$) < *NFFT*, they will be zero padded to *NFFT*.

**Parameters**

**x, y**

[1-D arrays or sequences] Arrays or sequences containing the data.

**Fs**

[float, default: 2] The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, *freqs*, in cycles per time unit.

**window**

[callable or ndarray, default: *window_hanning*] A function or a vector of length *NFFT*. To create window vectors see *window_hanning*, *window_none*, numpy.blackman, numpy.hamming, numpy.bartlett, scipy.signal, scipy.signal.get_window, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

**sides**

[{'default', 'onesided', 'twosided'}, optional] Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

**pad_to**

[int, optional] The number of points to which the data segment is padded when performing the FFT. This can be different from *NFFT*, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot,

allowing for more detail. This corresponds to the *n* parameter in the call to `fft`. The default is None, which sets *pad_to* equal to *NFFT*

**NFFT**

[int, default: 256] The number of data points used in each block for the FFT. A power 2 is most efficient. This should *NOT* be used to get zero padding, or the scaling of the result will be incorrect; use *pad_to* for this instead.

**detrend**

[{'none', 'mean', 'linear'} or callable, default: 'none'] The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the *detrend* parameter is a vector, in Matplotlib it is a function. The `mlab` module defines `detrend_none`, `detrend_mean`, and `detrend_linear`, but you can use a custom function as well. You can also use a string to choose one of the functions: 'none' calls `detrend_none`. 'mean' calls `detrend_mean`. 'linear' calls `detrend_linear`.

**scale_by_freq**

[bool, default: True] Whether the resulting density values should be scaled by the scaling frequency, which gives density in units of 1/Hz. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

**noverlap**

[int, default: 0 (no overlap)] The number of points of overlap between segments.

**Fc**

[int, default: 0] The center frequency of *x*, which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

**return_line**

[bool, default: False] Whether to include the line object plotted in the returned values.

**Returns**

**Pxy**

[1-D array] The values for the cross spectrum $P_{xy}$ before scaling (complex valued).

**freqs**

[1-D array] The frequencies corresponding to the elements in *Pxy*.

**line**

[`Line2D`] The line created by this function. Only returned if *return_line* is True.

**Other Parameters**

**data**

[indexable object, optional] If given, the following parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception):

*x*, *y*

****kwargs**

Keyword arguments control the *Line2D* properties:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi va |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* or aa | bool |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* or c | *color* |
| *dash_capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *dash_joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *dashes* | sequence of floats (on/off ink in points) or (None, None) |
| *data* | (2, N) array or two 1D arrays |
| *drawstyle* or ds | {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: |
| *figure* | *Figure* |
| *fillstyle* | {'full', 'left', 'right', 'bottom', 'top', 'none'} |
| *gapcolor* | *color* or None |
| *gid* | str |
| *in_layout* | bool |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float |
| *marker* | marker style string, *Path* or *MarkerStyle* |
| *markeredgecolor* or mec | *color* |
| *markeredgewidth* or mew | float |
| *markerfacecolor* or mfc | *color* |
| *markerfacecoloralt* or mfcalt | *color* |
| *markersize* or ms | float |
| *markevery* | None or int or (int, int) or slice or list[int] or float or (float, floa |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | float or callable[[Artist, Event], tuple[bool, dict]] |
| *pickradius* | float |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *solid_capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |

Table 17 – continued from

| Property | Description |
|---|---|
| *solid_joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *transform* | unknown |
| *url* | str |
| *visible* | bool |
| *xdata* | 1D array |
| *ydata* | 1D array |
| *zorder* | float |

**See also:**

***psd***

is equivalent to setting `y = x`.

### Notes

For plotting, the power is plotted as $10\log_{10}(P_{xy})$ for decibels, though $P_{xy}$ itself is returned.

### References

Bendat & Piersol -- Random Data: Analysis and Measurement Procedures, John Wiley & Sons (1986)

### Examples using `matplotlib.axes.Axes.csd`

- *Cross spectral density (CSD)*

## matplotlib.axes.Axes.magnitude_spectrum

`Axes.`**`magnitude_spectrum`**(*x*, *Fs=None*, *Fc=None*, *window=None*, *pad_to=None*, *sides=None*, *scale=None*, ***, *data=None*, ***kwargs*)

Plot the magnitude spectrum.

Compute the magnitude spectrum of *x*. Data is padded to a length of *pad_to* and the windowing function *window* is applied to the signal.

**Parameters**

**x**

[1-D array or sequence] Array or sequence containing the data.

**Fs**

[float, default: 2] The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, *freqs*, in cycles per time unit.

**window**

[callable or ndarray, default: *window_hanning*] A function or a vector of length *NFFT*. To create window vectors see *window_hanning*, *window_none*, numpy.blackman, numpy.hamming, numpy.bartlett, scipy.signal, scipy.signal.get_window, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

**sides**

[{'default', 'onesided', 'twosided'}, optional] Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

**pad_to**

[int, optional] The number of points to which the data segment is padded when performing the FFT. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to fft. The default is None, which sets *pad_to* equal to the length of the input signal (i.e. no padding).

**scale**

[{'default', 'linear', 'dB'}] The scaling of the values in the *spec*. 'linear' is no scaling. 'dB' returns the values in dB scale, i.e., the dB amplitude (20 * log10). 'default' is 'linear'.

**Fc**

[int, default: 0] The center frequency of *x*, which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

**Returns**

**spectrum**

[1-D array] The values for the magnitude spectrum before scaling (real valued).

**freqs**

[1-D array] The frequencies corresponding to the elements in *spectrum*.

**line**

[*Line2D*] The line created by this function.

**Other Parameters**

**data**

[indexable object, optional] If given, the following parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception):

*x*

****kwargs**

Keyword arguments control the *Line2D* properties:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi va |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* or aa | bool |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* or c | *color* |
| *dash_capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *dash_joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *dashes* | sequence of floats (on/off ink in points) or (None, None) |
| *data* | (2, N) array or two 1D arrays |
| *drawstyle* or ds | {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: |
| *figure* | *Figure* |
| *fillstyle* | {'full', 'left', 'right', 'bottom', 'top', 'none'} |
| *gapcolor* | *color* or None |
| *gid* | str |
| *in_layout* | bool |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float |
| *marker* | marker style string, *Path* or *MarkerStyle* |
| *markeredgecolor* or mec | *color* |
| *markeredgewidth* or mew | float |
| *markerfacecolor* or mfc | *color* |
| *markerfacecoloralt* or mfcalt | *color* |
| *markersize* or ms | float |
| *markevery* | None or int or (int, int) or slice or list[int] or float or (float, floa |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | float or callable[[Artist, Event], tuple[bool, dict]] |
| *pickradius* | float |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *solid_capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |

Table 18 – continued from

| Property | Description |
|---|---|
| *solid_joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *transform* | unknown |
| *url* | str |
| *visible* | bool |
| *xdata* | 1D array |
| *ydata* | 1D array |
| *zorder* | float |

**See also:**

***psd***

Plots the power spectral density.

***angle_spectrum***

Plots the angles of the corresponding frequencies.

***phase_spectrum***

Plots the phase (unwrapped angle) of the corresponding frequencies.

***specgram***

Can plot the magnitude spectrum of segments within the signal in a colormap.

### matplotlib.axes.Axes.phase_spectrum

Axes.**phase_spectrum**(*x*, *Fs=None*, *Fc=None*, *window=None*, *pad_to=None*, *sides=None*, ***, *data=None*, ***kwargs*)

Plot the phase spectrum.

Compute the phase spectrum (unwrapped angle spectrum) of *x*. Data is padded to a length of *pad_to* and the windowing function *window* is applied to the signal.

**Parameters**

**x**

[1-D array or sequence] Array or sequence containing the data

**Fs**

[float, default: 2] The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, *freqs*, in cycles per time unit.

**window**

[callable or ndarray, default: *window_hanning*] A function or a vector of length *NFFT*. To create window vectors see *window_hanning*, *window_none*, numpy.blackman, numpy.hamming, numpy.bartlett, scipy.signal, scipy.signal.get_window, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

**sides**

[{'default', 'onesided', 'twosided'}, optional] Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

**pad_to**

[int, optional] The number of points to which the data segment is padded when performing the FFT. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to fft. The default is None, which sets *pad_to* equal to the length of the input signal (i.e. no padding).

**Fc**

[int, default: 0] The center frequency of *x*, which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

**Returns**

**spectrum**

[1-D array] The values for the phase spectrum in radians (real valued).

**freqs**

[1-D array] The frequencies corresponding to the elements in *spectrum*.

**line**

[*Line2D*] The line created by this function.

**Other Parameters**

**data**

[indexable object, optional] If given, the following parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception):

*x*

****kwargs**

Keyword arguments control the *Line2D* properties:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi va |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* or aa | bool |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* or c | *color* |
| *dash_capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *dash_joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *dashes* | sequence of floats (on/off ink in points) or (None, None) |
| *data* | (2, N) array or two 1D arrays |
| *drawstyle* or ds | {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: |
| *figure* | *Figure* |
| *fillstyle* | {'full', 'left', 'right', 'bottom', 'top', 'none'} |
| *gapcolor* | *color* or None |
| *gid* | str |
| *in_layout* | bool |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float |
| *marker* | marker style string, *Path* or *MarkerStyle* |
| *markeredgecolor* or mec | *color* |
| *markeredgewidth* or mew | float |
| *markerfacecolor* or mfc | *color* |
| *markerfacecoloralt* or mfcalt | *color* |
| *markersize* or ms | float |
| *markevery* | None or int or (int, int) or slice or list[int] or float or (float, floa |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | float or callable[[Artist, Event], tuple[bool, dict]] |
| *pickradius* | float |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *solid_capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *solid_joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *transform* | unknown |
| *url* | str |
| *visible* | bool |
| *xdata* | 1D array |
| *ydata* | 1D array |
| *zorder* | float |

**See also:**

***magnitude_spectrum***

Plots the magnitudes of the corresponding frequencies.

***angle_spectrum***

Plots the wrapped version of this function.

***specgram***

Can plot the phase spectrum of segments within the signal in a colormap.

## matplotlib.axes.Axes.psd

`Axes.`**`psd`**(*x*, *NFFT=None*, *Fs=None*, *Fc=None*, *detrend=None*, *window=None*, *noverlap=None*, *pad_to=None*, *sides=None*, *scale_by_freq=None*, *return_line=None*, *, *data=None*, ***kwargs*)

Plot the power spectral density.

The power spectral density $P_{xx}$ by Welch's average periodogram method. The vector $x$ is divided into *NFFT* length segments. Each segment is detrended by function *detrend* and windowed by function *window*. *noverlap* gives the length of the overlap between segments. The $|\mathrm{fft}(i)|^2$ of each segment $i$ are averaged to compute $P_{xx}$, with a scaling to correct for power loss due to windowing.

If len($x$) < *NFFT*, it will be zero padded to *NFFT*.

**Parameters**

**x**

[1-D array or sequence] Array or sequence containing the data

**Fs**

[float, default: 2] The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, *freqs*, in cycles per time unit.

**window**

[callable or ndarray, default: *window_hanning*] A function or a vector of length *NFFT*. To create window vectors see *window_hanning*, *window_none*, `numpy.blackman`, `numpy.hamming`, `numpy.bartlett`, `scipy.signal`, `scipy.signal.get_window`, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

**sides**

[{'default', 'onesided', 'twosided'}, optional] Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

**pad_to**

[int, optional] The number of points to which the data segment is padded when performing the FFT. This can be different from *NFFT*, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to `fft`. The default is None, which sets *pad_to* equal to *NFFT*

**NFFT**

[int, default: 256] The number of data points used in each block for the FFT. A power 2 is most efficient. This should *NOT* be used to get zero padding, or the scaling of the result will be incorrect; use *pad_to* for this instead.

**detrend**

[{'none', 'mean', 'linear'} or callable, default: 'none'] The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the *detrend* parameter is a vector, in Matplotlib it is a function. The `mlab` module defines `detrend_none`, `detrend_mean`, and `detrend_linear`, but you can use a custom function as well. You can also use a string to choose one of the functions: 'none' calls `detrend_none`. 'mean' calls `detrend_mean`. 'linear' calls `detrend_linear`.

**scale_by_freq**

[bool, default: True] Whether the resulting density values should be scaled by the scaling frequency, which gives density in units of 1/Hz. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

**noverlap**

[int, default: 0 (no overlap)] The number of points of overlap between segments.

**Fc**

[int, default: 0] The center frequency of *x*, which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

**return_line**

[bool, default: False] Whether to include the line object plotted in the returned values.

**Returns**

**Pxx**

[1-D array] The values for the power spectrum $P_{xx}$ before scaling (real valued).

**freqs**

[1-D array] The frequencies corresponding to the elements in *Pxx*.

**line**

[*Line2D*] The line created by this function. Only returned if *return_line* is True.

**Other Parameters**

**data**

[indexable object, optional] If given, the following parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception):

*x*

****kwargs**

Keyword arguments control the *Line2D* properties:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi va |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* or aa | bool |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* or c | *color* |
| *dash_capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *dash_joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *dashes* | sequence of floats (on/off ink in points) or (None, None) |
| *data* | (2, N) array or two 1D arrays |
| *drawstyle* or ds | {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: |
| *figure* | *Figure* |
| *fillstyle* | {'full', 'left', 'right', 'bottom', 'top', 'none'} |
| *gapcolor* | *color* or None |
| *gid* | str |
| *in_layout* | bool |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float |
| *marker* | marker style string, *Path* or *MarkerStyle* |
| *markeredgecolor* or mec | *color* |
| *markeredgewidth* or mew | float |
| *markerfacecolor* or mfc | *color* |
| *markerfacecoloralt* or mfcalt | *color* |
| *markersize* or ms | float |
| *markevery* | None or int or (int, int) or slice or list[int] or float or (float, floa |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |

Table 20 – continued from

| Property | Description |
|---|---|
| picker | float or callable[[Artist, Event], tuple[bool, dict]] |
| pickradius | float |
| rasterized | bool |
| sketch_params | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| solid_capstyle | CapStyle or {'butt', 'projecting', 'round'} |
| solid_joinstyle | JoinStyle or {'miter', 'round', 'bevel'} |
| transform | unknown |
| url | str |
| visible | bool |
| xdata | 1D array |
| ydata | 1D array |
| zorder | float |

**See also:**

**specgram**

Differs in the default overlap; in not returning the mean of the segment periodograms; in returning the times of the segments; and in plotting a colormap instead of a line.

**magnitude_spectrum**

Plots the magnitude spectrum.

**csd**

Plots the spectral density between two signals.

## Notes

For plotting, the power is plotted as $10\log_{10}(P_{xx})$ for decibels, though *Pxx* itself is returned.

## References

Bendat & Piersol -- Random Data: Analysis and Measurement Procedures, John Wiley & Sons (1986)

### Examples using `matplotlib.axes.Axes.psd`

- *Power spectral density (PSD)*

## matplotlib.axes.Axes.specgram

Axes.**specgram**(*x*, *NFFT=None*, *Fs=None*, *Fc=None*, *detrend=None*, *window=None*, *noverlap=None*, *cmap=None*, *xextent=None*, *pad_to=None*, *sides=None*, *scale_by_freq=None*, *mode=None*, *scale=None*, *vmin=None*, *vmax=None*, *, *data=None*, **kwargs*)

Plot a spectrogram.

Compute and plot a spectrogram of data in *x*. Data are split into *NFFT* length segments and the spectrum of each section is computed. The windowing function *window* is applied to each segment, and the amount of overlap of each segment is specified with *noverlap*. The spectrogram is plotted as a colormap (using imshow).

**Parameters**

**x**

[1-D array or sequence] Array or sequence containing the data.

**Fs**

[float, default: 2] The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, *freqs*, in cycles per time unit.

**window**

[callable or ndarray, default: *window_hanning*] A function or a vector of length *NFFT*. To create window vectors see *window_hanning*, *window_none*, `numpy.blackman`, `numpy.hamming`, `numpy.bartlett`, `scipy.signal`, `scipy.signal.get_window`, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

**sides**

[{'default', 'onesided', 'twosided'}, optional] Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

**pad_to**

[int, optional] The number of points to which the data segment is padded when performing the FFT. This can be different from *NFFT*, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to `fft`. The default is None, which sets *pad_to* equal to *NFFT*

**NFFT**

[int, default: 256] The number of data points used in each block for the FFT. A power 2 is most efficient. This should *NOT* be used to get zero padding, or the scaling of the result will be incorrect; use *pad_to* for this instead.

**detrend**

[{'none', 'mean', 'linear'} or callable, default: 'none'] The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the *detrend* parameter is a vector, in Matplotlib it is a function. The `mlab` module defines `detrend_none`, `detrend_mean`, and `detrend_linear`, but you can use a custom function as well. You can also use a string to choose one of the functions: 'none' calls `detrend_none`. 'mean' calls `detrend_mean`. 'linear' calls `detrend_linear`.

**scale_by_freq**

[bool, default: True] Whether the resulting density values should be scaled by the scaling frequency, which gives density in units of 1/Hz. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

**mode**

[{'default', 'psd', 'magnitude', 'angle', 'phase'}] What sort of spectrum to use. Default is 'psd', which takes the power spectral density. 'magnitude' returns the magnitude spectrum. 'angle' returns the phase spectrum without unwrapping. 'phase' returns the phase spectrum with unwrapping.

**noverlap**

[int, default: 128] The number of points of overlap between blocks.

**scale**

[{'default', 'linear', 'dB'}] The scaling of the values in the *spec*. 'linear' is no scaling. 'dB' returns the values in dB scale. When *mode* is 'psd', this is dB power (10 * log10). Otherwise, this is dB amplitude (20 * log10). 'default' is 'dB' if *mode* is 'psd' or 'magnitude' and 'linear' otherwise. This must be 'linear' if *mode* is 'angle' or 'phase'.

**Fc**

[int, default: 0] The center frequency of *x*, which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

**cmap**

[`Colormap`, default: `rcParams["image.cmap"]` (default: `'viridis'`)]

**xextent**

[*None* or (xmin, xmax)] The image extent along the x-axis. The default sets *xmin* to the left border of the first bin (*spectrum* column) and *xmax* to the right border

of the last bin. Note that for *noverlap>0* the width of the bins is smaller than those of the segments.

**data**

[indexable object, optional] If given, the following parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception):

*x*

****kwargs**

Additional keyword arguments are passed on to *imshow* which makes the specgram image. The origin keyword argument is not supported.

**Returns**

**spectrum**

[2D array] Columns are the periodograms of successive segments.

**freqs**

[1-D array] The frequencies corresponding to the rows in *spectrum*.

**t**

[1-D array] The times corresponding to midpoints of segments (i.e., the columns in *spectrum*).

**im**

[*AxesImage*] The image created by imshow containing the spectrogram.

**See also:**

***psd***

Differs in the default overlap; in returning the mean of the segment periodograms; in not returning times; and in generating a line plot instead of colormap.

***magnitude_spectrum***

A single spectrum, similar to having a single segment when *mode* is 'magnitude'. Plots a line instead of a colormap.

***angle_spectrum***

A single spectrum, similar to having a single segment when *mode* is 'angle'. Plots a line instead of a colormap.

***phase_spectrum***

A single spectrum, similar to having a single segment when *mode* is 'phase'. Plots a line instead of a colormap.

### Notes

The parameters *detrend* and *scale_by_freq* do only apply when *mode* is set to 'psd'.

## Examples using `matplotlib.axes.Axes.specgram`

- *Spectrogram*

## matplotlib.axes.Axes.xcorr

`Axes.`**`xcorr`**(*x*, *y*, *normed=True*, *detrend=<function detrend_none>*, *usevlines=True*, *maxlags=10*, ***, *data=None*, ***kwargs*)

Plot the cross correlation between *x* and *y*.

The correlation with lag k is defined as $\sum_n x[n+k] \cdot y^*[n]$, where $y^*$ is the complex conjugate of $y$.

**Parameters**

**x, y**

[array-like of length n] Neither *x* nor *y* are run through Matplotlib's unit conversion, so these should be unit-less arrays.

**detrend**

[callable, default: *mlab.detrend_none* (no detrending)] A detrending function applied to *x* and *y*. It must have the signature

```
detrend(x: np.ndarray) -> np.ndarray
```

**normed**

[bool, default: True] If `True`, input vectors are normalised to unit length.

**usevlines**

[bool, default: True] Determines the plot style.

If `True`, vertical lines are plotted from 0 to the xcorr value using *Axes.vlines*. Additionally, a horizontal line is plotted at y=0 using *Axes.axhline*.

If `False`, markers are plotted at the xcorr values using *Axes.plot*.

**maxlags**

[int, default: 10] Number of lags to show. If None, will return all `2 * len(x) - 1` lags.

**Returns**

**lags**

[array (length `2*maxlags+1`)] The lag vector.

**c**

[array (length `2*maxlags+1`)] The auto correlation vector.

**line**

[*LineCollection* or *Line2D*] *Artist* added to the Axes of the correlation:

- *LineCollection* if *usevlines* is True.
- *Line2D* if *usevlines* is False.

**b**

[*Line2D* or None] Horizontal line at 0 if *usevlines* is True None *usevlines* is False.

**Other Parameters**

**linestyle**

[*Line2D* property, optional] The linestyle for plotting the data points. Only used if *usevlines* is `False`.

**marker**

[str, default: 'o'] The marker for plotting the data points. Only used if *usevlines* is `False`.

**data**

[indexable object, optional] If given, the following parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception):

*x*, *y*

****kwargs**

Additional parameters are passed to *Axes.vlines* and *Axes.axhline* if *usevlines* is `True`; otherwise they are passed to *Axes.plot*.

## Notes

The cross correlation is performed with `numpy.correlate` with `mode = "full"`.

### Examples using `matplotlib.axes.Axes.xcorr`

- *Cross- and auto-correlation*

### Statistics

| | |
|---|---|
| *Axes.ecdf* | Compute and plot the empirical cumulative distribution function of *x*. |
| *Axes.boxplot* | Draw a box and whisker plot. |
| *Axes.violinplot* | Make a violin plot. |
| *Axes.bxp* | Draw a box and whisker plot from pre-computed statistics. |
| *Axes.violin* | Draw a violin plot from pre-computed statistics. |

## matplotlib.axes.Axes.ecdf

Axes.**ecdf**(*x*, *weights=None*, ***, *complementary=False*, *orientation='vertical'*, *compress=False*, *data=None*, ***kwargs*)

Compute and plot the empirical cumulative distribution function of *x*.

New in version 3.8.

**Parameters**

**x**

[1d array-like] The input data. Infinite entries are kept (and move the relevant end of the ecdf from 0/1), but NaNs and masked values are errors.

**weights**

[1d array-like or None, default: None] The weights of the entries; must have the same shape as *x*. Weights corresponding to NaN data points are dropped, and then the remaining weights are normalized to sum to 1. If unset, all entries have the same weight.

**complementary**

[bool, default: False] Whether to plot a cumulative distribution function, which increases from 0 to 1 (the default), or a complementary cumulative distribution function, which decreases from 1 to 0.

**orientation**

[{"vertical", "horizontal"}, default: "vertical"] Whether the entries are plotted along the x-axis ("vertical", the default) or the y-axis ("horizontal"). This parameter takes the same values as in *hist*.

**compress**

[bool, default: False] Whether multiple entries with the same values are grouped together (with a summed weight) before plotting. This is mainly useful if *x* contains many identical data points, to decrease the rendering complexity of the plot. If *x* contains no duplicate points, this has no effect and just uses some time and memory.

**Returns**

*Line2D*

**Other Parameters**

**data**

[indexable object, optional] If given, the following parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception):

*x*, *weights*

****kwargs**

Keyword arguments control the *Line2D* properties:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi va |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* or aa | bool |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* or c | *color* |
| *dash_capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *dash_joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *dashes* | sequence of floats (on/off ink in points) or (None, None) |
| *data* | (2, N) array or two 1D arrays |
| *drawstyle* or ds | {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: |
| *figure* | *Figure* |
| *fillstyle* | {'full', 'left', 'right', 'bottom', 'top', 'none'} |
| *gapcolor* | *color* or None |
| *gid* | str |
| *in_layout* | bool |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float |
| *marker* | marker style string, *Path* or *MarkerStyle* |

Table 21 – continued from

| Property | Description |
|---|---|
| *markeredgecolor* or mec | *color* |
| *markeredgewidth* or mew | float |
| *markerfacecolor* or mfc | *color* |
| *markerfacecoloralt* or mfcalt | *color* |
| *markersize* or ms | float |
| *markevery* | None or int or (int, int) or slice or list[int] or float or (float, floa |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | float or callable[[Artist, Event], tuple[bool, dict]] |
| *pickradius* | float |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *solid_capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *solid_joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *transform* | unknown |
| *url* | str |
| *visible* | bool |
| *xdata* | 1D array |
| *ydata* | 1D array |
| *zorder* | float |

### Notes

The ecdf plot can be thought of as a cumulative histogram with one bin per data entry; i.e. it reports on the entire dataset without any arbitrary binning.

If *x* contains NaNs or masked entries, either remove them first from the array (if they should not taken into account), or replace them by -inf or +inf (if they should be sorted at the beginning or the end of the array).

### Examples using `matplotlib.axes.Axes.ecdf`

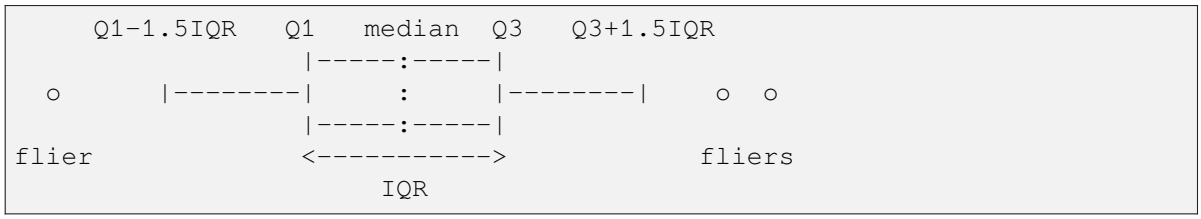
- *Plotting cumulative distributions*
- *ecdf(x)*

## matplotlib.axes.Axes.boxplot

Axes.**boxplot**(*x*, *notch=None*, *sym=None*, *vert=None*, *whis=None*, *positions=None*, *widths=None*, *patch_artist=None*, *bootstrap=None*, *usermedians=None*, *conf_intervals=None*, *meanline=None*, *showmeans=None*, *showcaps=None*, *showbox=None*, *showfliers=None*, *boxprops=None*, *tick_labels=None*, *flierprops=None*, *medianprops=None*, *meanprops=None*, *capprops=None*, *whiskerprops=None*, *manage_ticks=True*, *autorange=False*, *zorder=None*, *capwidths=None*, *label=None*, *, *data=None*)

Draw a box and whisker plot.

The box extends from the first quartile (Q1) to the third quartile (Q3) of the data, with a line at the median. The whiskers extend from the box to the farthest data point lying within 1.5x the inter-quartile range (IQR) from the box. Flier points are those past the end of the whiskers. See https://en.wikipedia.org/wiki/Box_plot for reference.




**Parameters**

**x**

[Array or a sequence of vectors.] The input data. If a 2D array, a boxplot is drawn for each column in *x*. If a sequence of 1D arrays, a boxplot is drawn for each array in *x*.

**notch**

[bool, default: *rcParams["boxplot.notch"]* (default: `False`)] Whether to draw a notched boxplot (`True`), or a rectangular boxplot (`False`). The notches represent the confidence interval (CI) around the median. The documentation for *bootstrap* describes how the locations of the notches are computed by default, but their locations may also be overridden by setting the *conf_intervals* parameter.

---

**Note:** In cases where the values of the CI are less than the lower quartile or greater than the upper quartile, the notches will extend beyond the box, giving it

a distinctive "flipped" appearance. This is expected behavior and consistent with other statistical visualization packages.

---

**sym**

[str, optional] The default symbol for flier points. An empty string ('') hides the fliers. If `None`, then the fliers default to 'b+'. More control is provided by the *flierprops* parameter.

**vert**

[bool, default: *rcParams["boxplot.vertical"]* (default: `True`)] If `True`, draws vertical boxes. If `False`, draw horizontal boxes.

**whis**

[float or (float, float), default: 1.5] The position of the whiskers.

If a float, the lower whisker is at the lowest datum above `Q1 - whis*(Q3-Q1)`, and the upper whisker at the highest datum below `Q3 + whis*(Q3-Q1)`, where Q1 and Q3 are the first and third quartiles. The default value of `whis = 1.5` corresponds to Tukey's original definition of boxplots.

If a pair of floats, they indicate the percentiles at which to draw the whiskers (e.g., (5, 95)). In particular, setting this to (0, 100) results in whiskers covering the whole range of the data.

In the edge case where `Q1 == Q3`, *whis* is automatically set to (0, 100) (cover the whole range of the data) if *autorange* is True.

Beyond the whiskers, data are considered outliers and are plotted as individual points.

**bootstrap**

[int, optional] Specifies whether to bootstrap the confidence intervals around the median for notched boxplots. If *bootstrap* is None, no bootstrapping is performed, and notches are calculated using a Gaussian-based asymptotic approximation (see McGill, R., Tukey, J.W., and Larsen, W.A., 1978, and Kendall and Stuart, 1967). Otherwise, bootstrap specifies the number of times to bootstrap the median to determine its 95% confidence intervals. Values between 1000 and 10000 are recommended.

**usermedians**

[1D array-like, optional] A 1D array-like of length `len(x)`. Each entry that is not `None` forces the value of the median for the corresponding dataset. For entries that are `None`, the medians are computed by Matplotlib as normal.

**conf_intervals**

[array-like, optional] A 2D array-like of shape `(len(x), 2)`. Each entry that is not None forces the location of the corresponding notch (which is only drawn if *notch* is `True`). For entries that are `None`, the notches are computed by the method specified by the other parameters (e.g., *bootstrap*).

**positions**

[array-like, optional] The positions of the boxes. The ticks and limits are automatically set to match the positions. Defaults to `range(1, N+1)` where N is the number of boxes to be drawn.

**widths**

[float or array-like] The widths of the boxes. The default is 0.5, or `0.15*(distance between extreme positions)`, if that is smaller.

**patch_artist**

[bool, default: *rcParams["boxplot.patchartist"]* (default: `False`)] If `False` produces boxes with the Line2D artist. Otherwise, boxes are drawn with Patch artists.

**tick_labels**

[list of str, optional] The tick labels of each boxplot. Ticks are always placed at the box *positions*. If *tick_labels* is given, the ticks are labelled accordingly. Otherwise, they keep their numeric values.

Changed in version 3.9: Renamed from *labels*, which is deprecated since 3.9 and will be removed in 3.11.

**manage_ticks**

[bool, default: True] If True, the tick locations and labels will be adjusted to match the boxplot positions.

**autorange**

[bool, default: False] When `True` and the data are distributed such that the 25th and 75th percentiles are equal, *whis* is set to (0, 100) such that the whisker ends are at the minimum and maximum of the data.

**meanline**

[bool, default: *rcParams["boxplot.meanline"]* (default: `False`)] If `True` (and *showmeans* is `True`), will try to render the mean as a line spanning the full width of the box according to *meanprops* (see below). Not recommended if *shownotches* is also True. Otherwise, means will be shown as points.

**zorder**

[float, default: `Line2D.zorder = 2`] The zorder of the boxplot.

**Returns**

**dict**

A dictionary mapping each component of the boxplot to a list of the *Line2D* instances created. That dictionary has the following keys (assuming vertical boxplots):

- `boxes`: the main body of the boxplot showing the quartiles and the median's confidence intervals if enabled.
- `medians`: horizontal lines at the median of each box.
- `whiskers`: the vertical lines extending to the most extreme, non-outlier data points.
- `caps`: the horizontal lines at the ends of the whiskers.
- `fliers`: points representing data that extend beyond the whiskers (fliers).
- `means`: points or lines representing the means.

**Other Parameters**

**showcaps**

[bool, default: *rcParams["boxplot.showcaps"]* (default: `True`)] Show the caps on the ends of whiskers.

**showbox**

[bool, default: *rcParams["boxplot.showbox"]* (default: `True`)] Show the central box.

**showfliers**

[bool, default: *rcParams["boxplot.showfliers"]* (default: `True`)] Show the outliers beyond the caps.

**showmeans**

[bool, default: *rcParams["boxplot.showmeans"]* (default: `False`)] Show the arithmetic means.

**capprops**

[dict, default: None] The style of the caps.

**capwidths**

[float or array, default: None] The widths of the caps.

**boxprops**

[dict, default: None] The style of the box.

**whiskerprops**

[dict, default: None] The style of the whiskers.

**flierprops**

[dict, default: None] The style of the fliers.

**medianprops**

[dict, default: None] The style of the median.

**meanprops**

[dict, default: None] The style of the mean.

**label**

[str or list of str, optional] Legend labels. Use a single string when all boxes have the same style and you only want a single legend entry for them. Use a list of strings to label all boxes individually. To be distinguishable, the boxes should be styled individually, which is currently only possible by modifying the returned artists, see e.g. *Boxplots*.

In the case of a single string, the legend entry will technically be associated with the first box only. By default, the legend will show the median line (`result["medians"]`); if *patch_artist* is True, the legend will show the box `Patch` artists (`result["boxes"]`) instead.

New in version 3.9.

**data**

[indexable object, optional] If given, all parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception).

**See also:**

***Axes.bxp***

Draw a boxplot from pre-computed statistics.

***violinplot***

Draw an estimate of the probability density function.

## Examples using `matplotlib.axes.Axes.boxplot`

- *Artist customization in box plots*
- *Box plots with custom fill colors*
- *Boxplots*
- *Box plot vs. violin plot comparison*
- *Separate calculation and plotting of boxplots*
- *boxplot(X)*

### matplotlib.axes.Axes.violinplot

Axes.**violinplot**(*dataset*, *positions=None*, *vert=True*, *widths=0.5*, *showmeans=False*, *showextrema=True*, *showmedians=False*, *quantiles=None*, *points=100*, *bw_method=None*, *side='both'*, ***, *data=None*)

Make a violin plot.

Make a violin plot for each column of *dataset* or each vector in sequence *dataset*. Each filled area extends to represent the entire data range, with optional lines at the mean, the median, the minimum, the maximum, and user-specified quantiles.

**Parameters**

**dataset**

[Array or a sequence of vectors.] The input data.

**positions**

[array-like, default: [1, 2, ..., n]] The positions of the violins; i.e. coordinates on the x-axis for vertical violins (or y-axis for horizontal violins).

**vert**

[bool, default: True.] If true, creates a vertical violin plot. Otherwise, creates a horizontal violin plot.

**widths**

[float or array-like, default: 0.5] The maximum width of each violin in units of the *positions* axis. The default is 0.5, which is half the available space when using default *positions*.

**showmeans**

[bool, default: False] Whether to show the mean with a line.

**showextrema**

[bool, default: True] Whether to show extrema with a line.

**showmedians**

[bool, default: False] Whether to show the median with a line.

**quantiles**

[array-like, default: None] If not None, set a list of floats in interval [0, 1] for each violin, which stands for the quantiles that will be rendered for that violin.

**points**

[int, default: 100] The number of points to evaluate each of the gaussian kernel density estimations at.

**bw_method**

[{'scott', 'silverman'} or float or callable, default: 'scott'] The method used to calculate the estimator bandwidth. If a float, this will be used directly as `kde.factor`. If a callable, it should take a *matplotlib.mlab.GaussianKDE* instance as its only parameter and return a float.

**side**

[{'both', 'low', 'high'}, default: 'both'] 'both' plots standard violins. 'low'/'high' only plots the side below/above the positions value.

**data**

[indexable object, optional] If given, the following parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception):

*dataset*

**Returns**

**dict**

A dictionary mapping each component of the violinplot to a list of the corresponding collection instances created. The dictionary has the following keys:

- `bodies`: A list of the *PolyCollection* instances containing the filled area of each violin.
- `cmeans`: A *LineCollection* instance that marks the mean values of each of the violin's distribution.
- `cmins`: A *LineCollection* instance that marks the bottom of each violin's distribution.
- `cmaxes`: A *LineCollection* instance that marks the top of each violin's distribution.
- `cbars`: A *LineCollection* instance that marks the centers of each violin's distribution.
- `cmedians`: A *LineCollection* instance that marks the median values of each of the violin's distribution.
- `cquantiles`: A *LineCollection* instance created to identify the quantile values of each of the violin's distribution.

**See also:**

***Axes.violin***

Draw a violin from pre-computed statistics.

***boxplot***

Draw a box and whisker plot.

### Examples using `matplotlib.axes.Axes.violinplot`

- *Box plot vs. violin plot comparison*
- *Violin plot customization*
- *Violin plot basics*
- *violinplot(D)*

## matplotlib.axes.Axes.bxp

`Axes.`**`bxp`**(*bxpstats*, *positions=None*, *widths=None*, *vert=True*, *patch_artist=False*, *shownotches=False*, *showmeans=False*, *showcaps=True*, *showbox=True*, *showfliers=True*, *boxprops=None*, *whiskerprops=None*, *flierprops=None*, *medianprops=None*, *capprops=None*, *meanprops=None*, *meanline=False*, *manage_ticks=True*, *zorder=None*, *capwidths=None*, *label=None*)

Draw a box and whisker plot from pre-computed statistics.

The box extends from the first quartile *q1* to the third quartile *q3* of the data, with a line at the median (*med*). The whiskers extend from *whislow* to *whishi*. Flier points are markers past the end of the whiskers. See https://en.wikipedia.org/wiki/Box_plot for reference.

```
      whislow    q1    med    q3    whishi
                   |-----:-----|
  o      |--------|     :     |--------|    o  o
                   |-----:-----|
flier                                    fliers
```

**Note:** This is a low-level drawing function for when you already have the statistical parameters. If you want a boxplot based on a dataset, use *`boxplot`* instead.

#### Parameters

**bxpstats**

[list of dicts] A list of dictionaries containing stats for each boxplot. Required keys are:

- `med`: Median (scalar).
- `q1`, `q3`: First & third quartiles (scalars).
- `whislo`, `whishi`: Lower & upper whisker positions (scalars).

Optional keys are:

- `mean`: Mean (scalar). Needed if `showmeans=True`.

- `fliers`: Data beyond the whiskers (array-like). Needed if `showfliers=True`.
- `cilo`, `cihi`: Lower & upper confidence intervals about the median. Needed if `shownotches=True`.
- `label`: Name of the dataset (str). If available, this will be used a tick label for the boxplot

**positions**

[array-like, default: [1, 2, ..., n]] The positions of the boxes. The ticks and limits are automatically set to match the positions.

**widths**

[float or array-like, default: None] The widths of the boxes. The default is `clip(0.15*(distance between extreme positions), 0.15, 0.5)`.

**capwidths**

[float or array-like, default: None] Either a scalar or a vector and sets the width of each cap. The default is `0.5*(width of the box)`, see *widths*.

**vert**

[bool, default: True] If `True` (default), makes the boxes vertical. If `False`, makes horizontal boxes.

**patch_artist**

[bool, default: False] If `False` produces boxes with the *Line2D* artist. If `True` produces boxes with the *Patch* artist.

**shownotches, showmeans, showcaps, showbox, showfliers**

[bool] Whether to draw the CI notches, the mean value (both default to False), the caps, the box, and the fliers (all three default to True).

**boxprops, whiskerprops, capprops, flierprops, medianprops, meanprops**

[dict, optional] Artist properties for the boxes, whiskers, caps, fliers, medians, and means.

**meanline**

[bool, default: False] If `True` (and *showmeans* is `True`), will try to render the mean as a line spanning the full width of the box according to *meanprops*. Not recommended if *shownotches* is also True. Otherwise, means will be shown as points.

**manage_ticks**

[bool, default: True] If True, the tick locations and labels will be adjusted to match the boxplot positions.

**label**

[str or list of str, optional] Legend labels. Use a single string when all boxes have the same style and you only want a single legend entry for them. Use a list of strings to label all boxes individually. To be distinguishable, the boxes should be styled individually, which is currently only possible by modifying the returned artists, see e.g. *Boxplots*.

In the case of a single string, the legend entry will technically be associated with the first box only. By default, the legend will show the median line (`result["medians"]`); if *patch_artist* is True, the legend will show the box *Patch* artists (`result["boxes"]`) instead.

New in version 3.9.

**zorder**

[float, default: `Line2D.zorder = 2`] The zorder of the resulting boxplot.

**Returns**

**dict**

A dictionary mapping each component of the boxplot to a list of the *Line2D* instances created. That dictionary has the following keys (assuming vertical boxplots):

- `boxes`: main bodies of the boxplot showing the quartiles, and the median's confidence intervals if enabled.
- `medians`: horizontal lines at the median of each box.
- `whiskers`: vertical lines up to the last non-outlier data.
- `caps`: horizontal lines at the ends of the whiskers.
- `fliers`: points representing data beyond the whiskers (fliers).
- `means`: points or lines representing the means.

**See also:**

***boxplot***

Draw a boxplot from data instead of pre-computed statistics.

### Examples using `matplotlib.axes.Axes.bxp`

- *Separate calculation and plotting of boxplots*

## matplotlib.axes.Axes.violin

`Axes.`**`violin`**(*vpstats*, *positions=None*, *vert=True*, *widths=0.5*, *showmeans=False*, *showextrema=True*, *showmedians=False*, *side='both'*)

Draw a violin plot from pre-computed statistics.

Draw a violin plot for each column of *vpstats*. Each filled area extends to represent the entire data range, with optional lines at the mean, the median, the minimum, the maximum, and the quantiles values.

**Parameters**

**vpstats**

[list of dicts] A list of dictionaries containing stats for each violin plot. Required keys are:

- `coords`: A list of scalars containing the coordinates that the violin's kernel density estimate were evaluated at.
- `vals`: A list of scalars containing the values of the kernel density estimate at each of the coordinates given in *coords*.
- `mean`: The mean value for this violin's dataset.
- `median`: The median value for this violin's dataset.
- `min`: The minimum value for this violin's dataset.
- `max`: The maximum value for this violin's dataset.

Optional keys are:

- `quantiles`: A list of scalars containing the quantile values for this violin's dataset.

**positions**

[array-like, default: [1, 2, ..., n]] The positions of the violins; i.e. coordinates on the x-axis for vertical violins (or y-axis for horizontal violins).

**vert**

[bool, default: True.] If true, plots the violins vertically. Otherwise, plots the violins horizontally.

**widths**

[float or array-like, default: 0.5] The maximum width of each violin in units of the *positions* axis. The default is 0.5, which is half available space when using default *positions*.

**showmeans**

[bool, default: False] Whether to show the mean with a line.

**showextrema**

[bool, default: True] Whether to show extrema with a line.

**showmedians**

[bool, default: False] Whether to show the median with a line.

**side**

[{'both', 'low', 'high'}, default: 'both'] 'both' plots standard violins. 'low'/'high' only plots the side below/above the positions value.

**Returns**

**dict**

A dictionary mapping each component of the violinplot to a list of the corresponding collection instances created. The dictionary has the following keys:

- `bodies`: A list of the *`PolyCollection`* instances containing the filled area of each violin.
- `cmeans`: A *`LineCollection`* instance that marks the mean values of each of the violin's distribution.
- `cmins`: A *`LineCollection`* instance that marks the bottom of each violin's distribution.
- `cmaxes`: A *`LineCollection`* instance that marks the top of each violin's distribution.
- `cbars`: A *`LineCollection`* instance that marks the centers of each violin's distribution.
- `cmedians`: A *`LineCollection`* instance that marks the median values of each of the violin's distribution.
- `cquantiles`: A *`LineCollection`* instance created to identify the quantiles values of each of the violin's distribution.

**See also:**

***`violin`***

Draw a violin plot from data instead of pre-computed statistics.

### Binned

| | |
|---|---|
| *Axes.hexbin* | Make a 2D hexagonal binning plot of points *x*, *y*. |
| *Axes.hist* | Compute and plot a histogram. |
| *Axes.hist2d* | Make a 2D histogram plot. |
| *Axes.stairs* | Draw a stepwise constant function as a line or a filled plot. |

### matplotlib.axes.Axes.hexbin

Axes.**hexbin**(*x*, *y*, *C=None*, *gridsize=100*, *bins=None*, *xscale='linear'*, *yscale='linear'*, *extent=None*, *cmap=None*, *norm=None*, *vmin=None*, *vmax=None*, *alpha=None*, *linewidths=None*, *edgecolors='face'*, *reduce_C_function=<function mean>*, *mincnt=None*, *marginals=False*, ***, *data=None*, ***kwargs*)

Make a 2D hexagonal binning plot of points *x*, *y*.

If *C* is *None*, the value of the hexagon is determined by the number of points in the hexagon. Otherwise, *C* specifies values at the coordinate (x[i], y[i]). For each hexagon, these values are reduced using *reduce_C_function*.

**Parameters**

**x, y**

[array-like] The data positions. *x* and *y* must be of the same length.

**C**

[array-like, optional] If given, these values are accumulated in the bins. Otherwise, every point has a value of 1. Must be of the same length as *x* and *y*.

**gridsize**

[int or (int, int), default: 100] If a single int, the number of hexagons in the *x*-direction. The number of hexagons in the *y*-direction is chosen such that the hexagons are approximately regular.

Alternatively, if a tuple (*nx*, *ny*), the number of hexagons in the *x*-direction and the *y*-direction. In the *y*-direction, counting is done along vertically aligned hexagons, not along the zig-zag chains of hexagons; see the following illustration.

To get approximately regular hexagons, choose $n_x = \sqrt{3}\,n_y$.

**bins**

['log' or int or sequence, default: None] Discretization of the hexagon values.

- If *None*, no binning is applied; the color of each hexagon directly corresponds to its count value.

- If 'log', use a logarithmic scale for the colormap. Internally, $log_{10}(i+1)$ is used to determine the hexagon color. This is equivalent to `norm=LogNorm()`.
- If an integer, divide the counts in the specified number of bins, and color the hexagons accordingly.
- If a sequence of values, the values of the lower bound of the bins to be used.

**xscale**

[{'linear', 'log'}, default: 'linear'] Use a linear or log10 scale on the horizontal axis.

**yscale**

[{'linear', 'log'}, default: 'linear'] Use a linear or log10 scale on the vertical axis.

**mincnt**

[int >= 0, default: *None*] If not *None*, only display cells with at least *mincnt* number of points in the cell.

**marginals**

[bool, default: *False*] If marginals is *True*, plot the marginal density as colormapped rectangles along the bottom of the x-axis and left of the y-axis.

**extent**

[4-tuple of float, default: *None*] The limits of the bins (xmin, xmax, ymin, ymax). The default assigns the limits based on *gridsize*, *x*, *y*, *xscale* and *yscale*.

If *xscale* or *yscale* is set to 'log', the limits are expected to be the exponent for a power of 10. E.g. for x-limits of 1 and 50 in 'linear' scale and y-limits of 10 and 1000 in 'log' scale, enter (1, 50, 1, 3).

**Returns**

***PolyCollection***

A *PolyCollection* defining the hexagonal bins.

- *PolyCollection.get_offsets* contains a Mx2 array containing the x, y positions of the M hexagon centers.
- *PolyCollection.get_array* contains the values of the M hexagons.

If *marginals* is *True*, horizontal bar and vertical bar (both PolyCollections) will be attached to the return collection as attributes *hbar* and *vbar*.

**Other Parameters**

**cmap**

[str or *Colormap*, default: *rcParams["image.cmap"]* (default: `'viridis'`)] The Colormap instance or registered colormap name used to map scalar data to colors.

**norm**

[str or *Normalize*, optional] The normalization method used to scale scalar data to the [0, 1] range before mapping to colors using *cmap*. By default, a linear scaling is used, mapping the lowest value to 0 and the highest to 1.

If given, this can be one of the following:

- An instance of *Normalize* or one of its subclasses (see *Colormap normalization*).
- A scale name, i.e. one of "linear", "log", "symlog", "logit", etc. For a list of available scales, call *matplotlib.scale.get_scale_names()*. In that case, a suitable *Normalize* subclass is dynamically generated and instantiated.

**vmin, vmax**

[float, optional] When using scalar data and no explicit *norm*, *vmin* and *vmax* define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data. It is an error to use *vmin*/*vmax* when a *norm* instance is given (but using a `str` *norm* name together with *vmin*/*vmax* is acceptable).

**alpha**

[float between 0 and 1, optional] The alpha blending value, between 0 (transparent) and 1 (opaque).

**linewidths**

[float, default: *None*] If *None*, defaults to *rcParams["patch.linewidth"]* (default: `1.0`).

**edgecolors**

[{'face', 'none', *None*} or color, default: 'face'] The color of the hexagon edges. Possible values are:

- 'face': Draw the edges in the same color as the fill color.
- 'none': No edges are drawn. This can sometimes lead to unsightly unpainted pixels between the hexagons.
- *None*: Draw outlines in the default color.
- An explicit color.

**reduce_C_function**

[callable, default: `numpy.mean`] The function to aggregate *C* within the bins. It is ignored if *C* is not given. This must have the signature:

```
def reduce_C_function(C: array) -> float
```

Commonly used functions are:

- `numpy.mean`: average of the points
- `numpy.sum`: integral of the point values
- `numpy.amax`: value taken from the largest point

By default will only reduce cells with at least 1 point because some reduction functions (such as `numpy.amax`) will error/warn with empty input. Changing *mincnt* will adjust the cutoff, and if set to 0 will pass empty input to the reduction function.

**data**

[indexable object, optional] If given, the following parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception):

*x*, *y*, *C*

****kwargs**

[*PolyCollection* properties] All other keyword arguments are passed on to *PolyCollection*:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a d |
| *alpha* | array-like or scalar or None |
| *animated* | bool |
| *antialiased* or aa or antialiaseds | bool or list of bools |
| *array* | array-like or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clim* | (vmin: float, vmax: float) |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *cmap* | *Colormap* or str or None |
| *color* | *color* or list of RGBA tuples |
| *edgecolor* or ec or edgecolors | *color* or list of *color* or 'face' |
| *facecolor* or facecolors or fc | *color* or list of *color* |
| *figure* | *Figure* |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or dashes or linestyles or ls | str or tuple or list thereof |
| *linewidth* or linewidths or lw | float or list of floats |
| *mouseover* | bool |
| *norm* | *Normalize* or str or None |
| *offset_transform* or transOffset | *Transform* |
| *offsets* | (N, 2) or (2,) array-like |

Table 22 – continued fro

| Property | Description |
|---|---|
| *path_effects* | list of *AbstractPathEffect* |
| *paths* | list of array-like |
| *picker* | None or bool or float or callable |
| *pickradius* | float |
| *rasterized* | bool |
| sizes | numpy.ndarray or None |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *urls* | list of str or None |
| *verts* | list of array-like |
| *verts_and_codes* | unknown |
| *visible* | bool |
| *zorder* | float |

**See also:**

***hist2d***

2D histogram rectangular bins

### Examples using `matplotlib.axes.Axes.hexbin`

- *Hexagonal binned plot*
- *hexbin(x, y, C)*

### matplotlib.axes.Axes.hist

Axes.**hist**(*x*, *bins=None*, *range=None*, *density=False*, *weights=None*, *cumulative=False*, *bottom=None*, *histtype='bar'*, *align='mid'*, *orientation='vertical'*, *rwidth=None*, *log=False*, *color=None*, *label=None*, *stacked=False*, ***, *data=None*, ***kwargs*)

Compute and plot a histogram.

This method uses numpy.histogram to bin the data in *x* and count the number of values in each bin, then draws the distribution either as a *BarContainer* or *Polygon*. The *bins*, *range*, *density*, and *weights* parameters are forwarded to numpy.histogram.

If the data has already been binned and counted, use *bar* or *stairs* to plot the distribution:

```
counts, bins = np.histogram(x)
plt.stairs(counts, bins)
```

Alternatively, plot pre-computed bins and counts using `hist()` by treating each bin as a single point with a weight equal to its count:

```
plt.hist(bins[:-1], bins, weights=counts)
```

The data input *x* can be a singular array, a list of datasets of potentially different lengths ([*x0*, *x1*, ...]), or a 2D ndarray in which each column is a dataset. Note that the ndarray form is transposed relative to the list form. If the input is an array, then the return value is a tuple (*n*, *bins*, *patches*); if the input is a sequence of arrays, then the return value is a tuple ([*n0*, *n1*, ...], *bins*, [*patches0*, *patches1*, ...]).

Masked arrays are not supported.

**Parameters**

**x**

[(n,) array or sequence of (n,) arrays] Input values, this takes either a single array or a sequence of arrays which are not required to be of the same length.

**bins**

[int or sequence or str, default: *rcParams["hist.bins"]* (default: `10`)] If *bins* is an integer, it defines the number of equal-width bins in the range.

If *bins* is a sequence, it defines the bin edges, including the left edge of the first bin and the right edge of the last bin; in this case, bins may be unequally spaced. All but the last (righthand-most) bin is half-open. In other words, if *bins* is:

```
[1, 2, 3, 4]
```

then the first bin is `[1, 2)` (including 1, but excluding 2) and the second `[2, 3)`. The last bin, however, is `[3, 4]`, which *includes* 4.

If *bins* is a string, it is one of the binning strategies supported by `numpy.histogram_bin_edges`: 'auto', 'fd', 'doane', 'scott', 'stone', 'rice', 'sturges', or 'sqrt'.

**range**

[tuple or None, default: None] The lower and upper range of the bins. Lower and upper outliers are ignored. If not provided, *range* is `(x.min(), x.max())`. Range has no effect if *bins* is a sequence.

If *bins* is a sequence or *range* is specified, autoscaling is based on the specified bin range instead of the range of x.

**density**

[bool, default: False] If `True`, draw and return a probability density: each bin will display the bin's raw count divided by the total number of counts *and the bin width* (`density = counts / (sum(counts) * np.diff(bins))`), so that the area under the histogram integrates to 1 (`np.sum(density * np.diff(bins)) == 1`).

If *stacked* is also `True`, the sum of the histograms is normalized to 1.

**weights**

[(n,) array-like or None, default: None] An array of weights, of the same shape as *x*. Each value in *x* only contributes its associated weight towards the bin count (instead of 1). If *density* is `True`, the weights are normalized, so that the integral of the density over the range remains 1.

**cumulative**

[bool or -1, default: False] If `True`, then a histogram is computed where each bin gives the counts in that bin plus all bins for smaller values. The last bin gives the total number of datapoints.

If *density* is also `True` then the histogram is normalized such that the last bin equals 1.

If *cumulative* is a number less than 0 (e.g., -1), the direction of accumulation is reversed. In this case, if *density* is also `True`, then the histogram is normalized such that the first bin equals 1.

**bottom**

[array-like, scalar, or None, default: None] Location of the bottom of each bin, i.e. bins are drawn from `bottom` to `bottom + hist(x, bins)` If a scalar, the bottom of each bin is shifted by the same amount. If an array, each bin is shifted independently and the length of bottom must match the number of bins. If None, defaults to 0.

**histtype**

[{'bar', 'barstacked', 'step', 'stepfilled'}, default: 'bar'] The type of histogram to draw.

- 'bar' is a traditional bar-type histogram. If multiple data are given the bars are arranged side by side.
- 'barstacked' is a bar-type histogram where multiple data are stacked on top of each other.
- 'step' generates a lineplot that is by default unfilled.
- 'stepfilled' generates a lineplot that is by default filled.

**align**

[{'left', 'mid', 'right'}, default: 'mid'] The horizontal alignment of the histogram bars.

- 'left': bars are centered on the left bin edges.
- 'mid': bars are centered between the bin edges.
- 'right': bars are centered on the right bin edges.

**orientation**

[{'vertical', 'horizontal'}, default: 'vertical'] If 'horizontal', *barh* will be used for bar-type histograms and the *bottom* kwarg will be the left edges.

**rwidth**

[float or None, default: None] The relative width of the bars as a fraction of the bin width. If `None`, automatically compute the width.

Ignored if *histtype* is 'step' or 'stepfilled'.

**log**

[bool, default: False] If `True`, the histogram axis will be set to a log scale.

**color**

[*color* or list of *color* or None, default: None] Color or sequence of colors, one per dataset. Default (`None`) uses the standard line color sequence.

**label**

[str or list of str, optional] String, or sequence of strings to match multiple datasets. Bar charts yield multiple patches per dataset, but only the first gets the label, so that *legend* will work as expected.

**stacked**

[bool, default: False] If `True`, multiple data are stacked on top of each other If `False` multiple data are arranged side by side if histtype is 'bar' or on top of each other if histtype is 'step'

**Returns**

**n**

[array or list of arrays] The values of the histogram bins. See *density* and *weights* for a description of the possible semantics. If input *x* is an array, then this is an array of length *nbins*. If input is a sequence of arrays `[data1, data2, ...]`, then this is a list of arrays with the values of the histograms for each of the arrays in the same order. The dtype of the array *n* (or of its element arrays) will always be float even if no weighting or normalization is used.

**bins**

[array] The edges of the bins. Length nbins + 1 (nbins left edges and right edge of last bin). Always a single array even when multiple data sets are passed in.

**patches**

[*BarContainer* or list of a single *Polygon* or list of such objects] Container of individual artists used to create the histogram or list of such containers if there are multiple input datasets.

**Other Parameters**

**data**

[indexable object, optional] If given, the following parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception):

*x*, *weights*

****kwargs**

*Patch* properties

**See also:**

***hist2d***

2D histogram with rectangular bins

***hexbin***

2D histogram with hexagonal bins

***stairs***

Plot a pre-computed histogram

***bar***

Plot a pre-computed histogram

### Notes

For large numbers of bins (>1000), plotting can be significantly accelerated by using *stairs* to plot a pre-computed histogram (`plt.stairs(*np.histogram(data))`), or by setting *histtype* to 'step' or 'stepfilled' rather than 'bar' or 'barstacked'.

### Examples using `matplotlib.axes.Axes.hist`

- *Scatter plot with histograms*
- *Axes Demo*
- *Histograms*
- *Bihistogram*
- *Plotting cumulative distributions*
- *Demo of the histogram function's different histtype settings*
- *The histogram (hist) function with multiple data sets*
- *Histogram bins, density, and weight*
- *Placing text boxes*
- *Bayesian Methods for Hackers style sheet*
- *Scatter Histogram (Locatable Axes)*
- *Animated histogram*

- *Building histograms using Rectangles and PolyCollections*
- *Artist tutorial*
- *hist(x)*
- *Quick start guide*
- *Path Tutorial*
- *Transformations Tutorial*

### matplotlib.axes.Axes.hist2d

`Axes.`**`hist2d`**(*x*, *y*, *bins=10*, *range=None*, *density=False*, *weights=None*, *cmin=None*, *cmax=None*, ***, *data=None*, ***kwargs*)

Make a 2D histogram plot.

**Parameters**

**x, y**

[array-like, shape (n, )] Input values

**bins**

[None or int or [int, int] or array-like or [array, array]] The bin specification:

- If int, the number of bins for the two dimensions (`nx = ny = bins`).
- If `[int, int]`, the number of bins in each dimension (`nx, ny = bins`).
- If array-like, the bin edges for the two dimensions (`x_edges = y_edges = bins`).
- If `[array, array]`, the bin edges in each dimension (`x_edges, y_edges = bins`).

The default value is 10.

**range**

[array-like shape(2, 2), optional] The leftmost and rightmost edges of the bins along each dimension (if not specified explicitly in the bins parameters): `[[xmin, xmax], [ymin, ymax]]`. All values outside of this range will be considered outliers and not tallied in the histogram.

**density**

[bool, default: False] Normalize histogram. See the documentation for the *density* parameter of *hist* for more details.

**weights**

[array-like, shape (n, ), optional] An array of values w_i weighing each sample (x_i, y_i).

**cmin, cmax**

[float, default: None] All bins that has count less than *cmin* or more than *cmax* will not be displayed (set to NaN before passing to *pcolormesh*) and these count values in the return value count histogram will also be set to nan upon return.

**Returns**

**h**

[2D array] The bi-dimensional histogram of samples x and y. Values in x are histogrammed along the first dimension and values in y are histogrammed along the second dimension.

**xedges**

[1D array] The bin edges along the x-axis.

**yedges**

[1D array] The bin edges along the y-axis.

**image**

[*QuadMesh*]

**Other Parameters**

**cmap**

[str or *Colormap*, default: *rcParams["image.cmap"]* (default: `'viridis'`)] The Colormap instance or registered colormap name used to map scalar data to colors.

**norm**

[str or *Normalize*, optional] The normalization method used to scale scalar data to the [0, 1] range before mapping to colors using *cmap*. By default, a linear scaling is used, mapping the lowest value to 0 and the highest to 1.

If given, this can be one of the following:

- An instance of *Normalize* or one of its subclasses (see *Colormap normalization*).
- A scale name, i.e. one of "linear", "log", "symlog", "logit", etc. For a list of available scales, call *matplotlib.scale.get_scale_names()*. In that case, a suitable *Normalize* subclass is dynamically generated and instantiated.

**vmin, vmax**

[float, optional] When using scalar data and no explicit *norm*, *vmin* and *vmax* define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data. It is an error to use *vmin*/*vmax* when a

*norm* instance is given (but using a `str` *norm* name together with *vmin*/*vmax* is acceptable).

**alpha**

[`0 <= scalar <= 1` or `None`, optional] The alpha blending value.

**data**

[indexable object, optional] If given, the following parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception):

*x*, *y*, *weights*

****kwargs**

Additional parameters are passed along to the *pcolormesh* method and *QuadMesh* constructor.

**See also:**

***hist***

1D histogram plotting

***hexbin***

2D histogram with hexagonal bins

### Notes

- Currently `hist2d` calculates its own axis limits, and any limits previously set are ignored.
- Rendering the histogram with a logarithmic color scale is accomplished by passing a *colors.LogNorm* instance to the *norm* keyword argument. Likewise, power-law normalization (similar in effect to gamma correction) can be accomplished with *colors.PowerNorm*.

### Examples using `matplotlib.axes.Axes.hist2d`

- *Histograms*
- *Exploring normalizations*
- *hist2d(x, y)*

### matplotlib.axes.Axes.stairs

Axes.**stairs**(*values*, *edges=None*, *, *orientation='vertical'*, *baseline=0*, *fill=False*, *data=None*, ***kwargs*)

Draw a stepwise constant function as a line or a filled plot.

*edges* define the x-axis positions of the steps. *values* the function values between these steps. Depending on *fill*, the function is drawn either as a continuous line with vertical segments at the edges, or as a filled area.

**Parameters**

**values**

[array-like] The step heights.

**edges**

[array-like] The step positions, with `len(edges) == len(vals) + 1`, between which the curve takes on vals values.

**orientation**

[{'vertical', 'horizontal'}, default: 'vertical'] The direction of the steps. Vertical means that *values* are along the y-axis, and edges are along the x-axis.

**baseline**

[float, array-like or None, default: 0] The bottom value of the bounding edges or when `fill=True`, position of lower edge. If *fill* is True or an array is passed to *baseline*, a closed path is drawn.

**fill**

[bool, default: False] Whether the area under the step curve should be filled.

**Returns**

**StepPatch**

[*StepPatch*]

**Other Parameters**

**data**

[indexable object, optional] If given, all parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception).

****kwargs**

*StepPatch* properties

### Examples using `matplotlib.axes.Axes.stairs`

- *Stairs Demo*
- *stairs(values)*

### Contours

| | |
|---|---|
| *Axes.clabel* | Label a contour plot. |
| *Axes.contour* | Plot contour lines. |
| *Axes.contourf* | Plot filled contours. |

## matplotlib.axes.Axes.clabel

Axes.**clabel**(*CS*, *levels=None*, ***kwargs*)

Label a contour plot.

Adds labels to line contours in given *ContourSet*.

**Parameters**

**CS**

[*ContourSet* instance] Line contours to label.

**levels**

[array-like, optional] A list of level values, that should be labeled. The list must be a subset of `CS.levels`. If not given, all levels are labeled.

****kwargs**

All other parameters are documented in *clabel*.

### Examples using `matplotlib.axes.Axes.clabel`

- *Contour Demo*
- *Contour Label Demo*
- *Contourf demo*
- *Contouring the solution space of optimizations*
- *Patheffect Demo*
- *TickedStroke patheffect*

## matplotlib.axes.Axes.contour

Axes.**contour**(**args*, *data=None*, ***kwargs*)

Plot contour lines.

Call signature:

```
contour([X, Y,] Z, [levels], **kwargs)
```

*contour* and *contourf* draw contour lines and filled contours, respectively. Except as noted, function signatures and return values are the same for both versions.

**Parameters**

**X, Y**

[array-like, optional] The coordinates of the values in *Z*.

*X* and *Y* must both be 2D with the same shape as *Z* (e.g. created via numpy.meshgrid), or they must both be 1-D such that `len(X) == N` is the number of columns in *Z* and `len(Y) == M` is the number of rows in *Z*.

*X* and *Y* must both be ordered monotonically.

If not given, they are assumed to be integer indices, i.e. `X = range(N), Y = range(M)`.

**Z**

[(M, N) array-like] The height values over which the contour is drawn. Color-mapping is controlled by *cmap*, *norm*, *vmin*, and *vmax*.

**levels**

[int or array-like, optional] Determines the number and positions of the contour lines / regions.

If an int *n*, use *MaxNLocator*, which tries to automatically choose no more than *n+1* "nice" contour levels between minimum and maximum numeric values of *Z*.

If array-like, draw contour lines at the specified levels. The values must be in increasing order.

**Returns**

***QuadContourSet***

**Other Parameters**

**corner_mask**

[bool, default: *rcParams["contour.corner_mask"]* (default: `True`)] Enable/disable corner masking, which only has an effect if *Z* is a masked array. If

`False`, any quad touching a masked point is masked out. If `True`, only the triangular corners of quads nearest those points are always masked out, other triangular corners comprising three unmasked points are contoured as usual.

**colors**

[*color* or list of *color*, optional] The colors of the levels, i.e. the lines for *contour* and the areas for *contourf*.

The sequence is cycled for the levels in ascending order. If the sequence is shorter than the number of levels, it's repeated.

As a shortcut, single color strings may be used in place of one-element lists, i.e. `'red'` instead of `['red']` to color all levels with the same color. This shortcut does only work for color strings, not for other ways of specifying colors.

By default (value *None*), the colormap specified by *cmap* will be used.

**alpha**

[float, default: 1] The alpha blending value, between 0 (transparent) and 1 (opaque).

**cmap**

[str or *Colormap*, default: *rcParams["image.cmap"]* (default: `'viridis'`)] The Colormap instance or registered colormap name used to map scalar data to colors.

This parameter is ignored if *colors* is set.

**norm**

[str or *Normalize*, optional] The normalization method used to scale scalar data to the [0, 1] range before mapping to colors using *cmap*. By default, a linear scaling is used, mapping the lowest value to 0 and the highest to 1.

If given, this can be one of the following:

- An instance of *Normalize* or one of its subclasses (see *Colormap normalization*).
- A scale name, i.e. one of "linear", "log", "symlog", "logit", etc. For a list of available scales, call *matplotlib.scale.get_scale_names()*. In that case, a suitable *Normalize* subclass is dynamically generated and instantiated.

This parameter is ignored if *colors* is set.

**vmin, vmax**

[float, optional] When using scalar data and no explicit *norm*, *vmin* and *vmax* define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data. It is an error to use *vmin*/*vmax* when a *norm* instance is given (but using a `str` *norm* name together with *vmin*/*vmax* is acceptable).

If *vmin* or *vmax* are not given, the default color scaling is based on *levels*.

This parameter is ignored if *colors* is set.

**origin**

[{*None*, 'upper', 'lower', 'image'}, default: None] Determines the orientation and exact position of *Z* by specifying the position of `Z[0, 0]`. This is only relevant, if *X*, *Y* are not given.

- *None*: `Z[0, 0]` is at X=0, Y=0 in the lower left corner.
- 'lower': `Z[0, 0]` is at X=0.5, Y=0.5 in the lower left corner.
- 'upper': `Z[0, 0]` is at X=N+0.5, Y=0.5 in the upper left corner.
- 'image': Use the value from *rcParams["image.origin"]* (default: `'upper'`).

**extent**

[(x0, x1, y0, y1), optional] If *origin* is not *None*, then *extent* is interpreted as in *imshow*: it gives the outer pixel boundaries. In this case, the position of Z[0, 0] is the center of the pixel, not a corner. If *origin* is *None*, then (*x0*, *y0*) is the position of Z[0, 0], and (*x1*, *y1*) is the position of Z[-1, -1].

This argument is ignored if *X* and *Y* are specified in the call to contour.

**locator**

[ticker.Locator subclass, optional] The locator is used to determine the contour levels if they are not given explicitly via *levels*. Defaults to *MaxNLocator*.

**extend**

[{'neither', 'both', 'min', 'max'}, default: 'neither'] Determines the `contourf`-coloring of values that are outside the *levels* range.

If 'neither', values outside the *levels* range are not colored. If 'min', 'max' or 'both', color the values below, above or below and above the *levels* range.

Values below `min(levels)` and above `max(levels)` are mapped to the under/over values of the *Colormap*. Note that most colormaps do not have dedicated colors for these by default, so that the over and under values are the edge values of the colormap. You may want to set these values explicitly using *Colormap.set_under* and *Colormap.set_over*.

---

**Note:** An existing *QuadContourSet* does not get notified if properties of its colormap are changed. Therefore, an explicit call `QuadContourSet.changed()` is needed after modifying the colormap. The explicit call can be left out, if a colorbar is assigned to the *QuadContourSet* because it internally calls `QuadContourSet.changed()`.

---

Example:

```
x = np.arange(1, 10)
y = x.reshape(-1, 1)
h = x * y

cs = plt.contourf(h, levels=[10, 30, 50],
    colors=['#808080', '#A0A0A0', '#C0C0C0'], extend='both
 ↪')
cs.cmap.set_over('red')
cs.cmap.set_under('blue')
cs.changed()
```

**xunits, yunits**

[registered units, optional] Override axis units by specifying an instance of a *matplotlib.units.ConversionInterface*.

**antialiased**

[bool, optional] Enable antialiasing, overriding the defaults. For filled contours, the default is *False*. For line contours, it is taken from *rcParams["lines.antialiased"]* (default: `True`).

**nchunk**

[int >= 0, optional] If 0, no subdivision of the domain. Specify a positive integer to divide the domain into subdomains of *nchunk* by *nchunk* quads. Chunking reduces the maximum length of polygons generated by the contouring algorithm which reduces the rendering workload passed on to the backend and also requires slightly less RAM. It can however introduce rendering artifacts at chunk boundaries depending on the backend, the *antialiased* flag and value of *alpha*.

**linewidths**

[float or array-like, default: *rcParams["contour.linewidth"]* (default: `None`)] *Only applies to* *contour*.

The line width of the contour lines.

If a number, all levels will be plotted with this linewidth.

If a sequence, the levels in ascending order will be plotted with the linewidths in the order specified.

If None, this falls back to *rcParams["lines.linewidth"]* (default: `1.5`).

**linestyles**

[{*None*, 'solid', 'dashed', 'dashdot', 'dotted'}, optional] *Only applies to* *contour*.

If *linestyles* is *None*, the default is 'solid' unless the lines are monochrome. In that case, negative contours will instead take their linestyle from the *negative_linestyles* argument.

*linestyles* can also be an iterable of the above strings specifying a set of linestyles to be used. If this iterable is shorter than the number of contour levels it will be repeated as necessary.

**negative_linestyles**

[{*None*, 'solid', 'dashed', 'dashdot', 'dotted'}, optional] *Only applies to* `contour`.

If *linestyles* is *None* and the lines are monochrome, this argument specifies the line style for negative contours.

If *negative_linestyles* is *None*, the default is taken from `rcParams["contour.negative_linestyles"]`.

*negative_linestyles* can also be an iterable of the above strings specifying a set of linestyles to be used. If this iterable is shorter than the number of contour levels it will be repeated as necessary.

**hatches**

[list[str], optional] *Only applies to* `contourf`.

A list of cross hatch patterns to use on the filled areas. If None, no hatching will be added to the contour.

**algorithm**

[{'mpl2005', 'mpl2014', 'serial', 'threaded'}, optional] Which contouring algorithm to use to calculate the contour lines and polygons. The algorithms are implemented in ContourPy, consult the ContourPy documentation for further information.

The default is taken from `rcParams["contour.algorithm"]` (default: `'mpl2014'`).

**clip_path**

[`Patch` or `Path` or `TransformedPath`] Set the clip path. See `set_clip_path`.

New in version 3.8.

**data**

[indexable object, optional] If given, all parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception).

## Notes

1. `contourf` differs from the MATLAB version in that it does not draw the polygon edges. To draw edges, add line contours with calls to `contour`.
2. `contourf` fills intervals that are closed at the top; that is, for boundaries *z1* and *z2*, the filled region is:

```
z1 < Z <= z2
```

   except for the lowest interval, which is closed on both sides (i.e. it includes the lowest value).
3. `contour` and `contourf` use a marching squares algorithm to compute contour locations. More information can be found in ContourPy documentation.

### Examples using `matplotlib.axes.Axes.contour`

- *Contour Corner Mask*
- *Contour Demo*
- *Contour Image*
- *Contour Label Demo*
- *Contourf demo*
- *Contourf Hatching*
- *Contouring the solution space of optimizations*
- *Blend transparency with color in 2D images*
- *Contour plot of irregularly spaced data*
- *Patheffect Demo*
- *TickedStroke patheffect*
- *Plot contour (level) curves in 3D*
- *Plot contour (level) curves in 3D using the extend3d option*
- *Project contour profiles onto a graph*
- *contour(X, Y, Z)*

## matplotlib.axes.Axes.contourf

Axes.**contourf**(**args*, *data=None*, ***kwargs*)

Plot filled contours.

Call signature:

```
contourf([X, Y,] Z, [levels], **kwargs)
```

*contour* and *contourf* draw contour lines and filled contours, respectively. Except as noted, function signatures and return values are the same for both versions.

**Parameters**

**X, Y**

[array-like, optional] The coordinates of the values in *Z*.

*X* and *Y* must both be 2D with the same shape as *Z* (e.g. created via `numpy.meshgrid`), or they must both be 1-D such that `len(X) == N` is the number of columns in *Z* and `len(Y) == M` is the number of rows in *Z*.

*X* and *Y* must both be ordered monotonically.

If not given, they are assumed to be integer indices, i.e. `X = range(N), Y = range(M)`.

**Z**

[(M, N) array-like] The height values over which the contour is drawn. Color-mapping is controlled by *cmap*, *norm*, *vmin*, and *vmax*.

**levels**

[int or array-like, optional] Determines the number and positions of the contour lines / regions.

If an int *n*, use *MaxNLocator*, which tries to automatically choose no more than *n+1* "nice" contour levels between minimum and maximum numeric values of *Z*.

If array-like, draw contour lines at the specified levels. The values must be in increasing order.

### Returns

*QuadContourSet*

### Other Parameters

**corner_mask**

[bool, default: *rcParams["contour.corner_mask"]* (default: `True`)] Enable/disable corner masking, which only has an effect if *Z* is a masked array. If `False`, any quad touching a masked point is masked out. If `True`, only the triangular corners of quads nearest those points are always masked out, other triangular corners comprising three unmasked points are contoured as usual.

**colors**

[*color* or list of *color*, optional] The colors of the levels, i.e. the lines for *contour* and the areas for *contourf*.

The sequence is cycled for the levels in ascending order. If the sequence is shorter than the number of levels, it's repeated.

As a shortcut, single color strings may be used in place of one-element lists, i.e. `'red'` instead of `['red']` to color all levels with the same color. This shortcut does only work for color strings, not for other ways of specifying colors.

By default (value *None*), the colormap specified by *cmap* will be used.

**alpha**

[float, default: 1] The alpha blending value, between 0 (transparent) and 1 (opaque).

**cmap**

[str or *Colormap*, default: *rcParams["image.cmap"]* (default: 'viridis')] The Colormap instance or registered colormap name used to map scalar data to colors.

This parameter is ignored if *colors* is set.

**norm**

[str or *Normalize*, optional] The normalization method used to scale scalar data to the [0, 1] range before mapping to colors using *cmap*. By default, a linear scaling is used, mapping the lowest value to 0 and the highest to 1.

If given, this can be one of the following:

- An instance of *Normalize* or one of its subclasses (see *Colormap normalization*).
- A scale name, i.e. one of "linear", "log", "symlog", "logit", etc. For a list of available scales, call *matplotlib.scale.get_scale_names()*. In that case, a suitable *Normalize* subclass is dynamically generated and instantiated.

This parameter is ignored if *colors* is set.

**vmin, vmax**

[float, optional] When using scalar data and no explicit *norm*, *vmin* and *vmax* define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data. It is an error to use *vmin*/*vmax* when a *norm* instance is given (but using a `str` *norm* name together with *vmin*/*vmax* is acceptable).

If *vmin* or *vmax* are not given, the default color scaling is based on *levels*.

This parameter is ignored if *colors* is set.

**origin**

[{*None*, 'upper', 'lower', 'image'}, default: None] Determines the orientation and exact position of *Z* by specifying the position of `Z[0, 0]`. This is only relevant, if *X*, *Y* are not given.

- *None*: `Z[0, 0]` is at X=0, Y=0 in the lower left corner.
- 'lower': `Z[0, 0]` is at X=0.5, Y=0.5 in the lower left corner.
- 'upper': `Z[0, 0]` is at X=N+0.5, Y=0.5 in the upper left corner.
- 'image': Use the value from *rcParams["image.origin"]* (default: 'upper').

**extent**

[(x0, x1, y0, y1), optional] If *origin* is not *None*, then *extent* is interpreted as in *imshow*: it gives the outer pixel boundaries. In this case, the position of Z[0, 0] is the center of the pixel, not a corner. If *origin* is *None*, then (*x0*, *y0*) is the position of Z[0, 0], and (*x1*, *y1*) is the position of Z[-1, -1].

This argument is ignored if *X* and *Y* are specified in the call to contour.

**locator**

[ticker.Locator subclass, optional] The locator is used to determine the contour levels if they are not given explicitly via *levels*. Defaults to *MaxNLocator*.

**extend**

[{'neither', 'both', 'min', 'max'}, default: 'neither'] Determines the `contourf`-coloring of values that are outside the *levels* range.

If 'neither', values outside the *levels* range are not colored. If 'min', 'max' or 'both', color the values below, above or below and above the *levels* range.

Values below `min(levels)` and above `max(levels)` are mapped to the under/over values of the *Colormap*. Note that most colormaps do not have dedicated colors for these by default, so that the over and under values are the edge values of the colormap. You may want to set these values explicitly using *Colormap.set_under* and *Colormap.set_over*.

---

**Note:** An existing *QuadContourSet* does not get notified if properties of its colormap are changed. Therefore, an explicit call `QuadContourSet.changed()` is needed after modifying the colormap. The explicit call can be left out, if a colorbar is assigned to the *QuadContourSet* because it internally calls `QuadContourSet.changed()`.

---

Example:

```
x = np.arange(1, 10)
y = x.reshape(-1, 1)
h = x * y

cs = plt.contourf(h, levels=[10, 30, 50],
    colors=['#808080', '#A0A0A0', '#C0C0C0'], extend='both
 ↪')
cs.cmap.set_over('red')
cs.cmap.set_under('blue')
cs.changed()
```

**xunits, yunits**

[registered units, optional] Override axis units by specifying an instance of a *matplotlib.units.ConversionInterface*.

**antialiased**

[bool, optional] Enable antialiasing, overriding the defaults. For filled contours, the default is *False*. For line contours, it is taken from *rcParams["lines.antialiased"]* (default: `True`).

**nchunk**

[int >= 0, optional] If 0, no subdivision of the domain. Specify a positive integer to divide the domain into subdomains of *nchunk* by *nchunk* quads. Chunking reduces the maximum length of polygons generated by the contouring algorithm which reduces the rendering workload passed on to the backend and also requires slightly less RAM. It can however introduce rendering artifacts at chunk boundaries depending on the backend, the *antialiased* flag and value of *alpha*.

**linewidths**

[float or array-like, default: `rcParams["contour.linewidth"]` (default: `None`)] *Only applies to* `contour`.

The line width of the contour lines.

If a number, all levels will be plotted with this linewidth.

If a sequence, the levels in ascending order will be plotted with the linewidths in the order specified.

If None, this falls back to `rcParams["lines.linewidth"]` (default: `1.5`).

**linestyles**

[{*None*, 'solid', 'dashed', 'dashdot', 'dotted'}, optional] *Only applies to* `contour`.

If *linestyles* is *None*, the default is 'solid' unless the lines are monochrome. In that case, negative contours will instead take their linestyle from the *negative_linestyles* argument.

*linestyles* can also be an iterable of the above strings specifying a set of linestyles to be used. If this iterable is shorter than the number of contour levels it will be repeated as necessary.

**negative_linestyles**

[{*None*, 'solid', 'dashed', 'dashdot', 'dotted'}, optional] *Only applies to* `contour`.

If *linestyles* is *None* and the lines are monochrome, this argument specifies the line style for negative contours.

If *negative_linestyles* is *None*, the default is taken from `rcParams["contour.negative_linestyles"]`.

*negative_linestyles* can also be an iterable of the above strings specifying a set of linestyles to be used. If this iterable is shorter than the number of contour levels it will be repeated as necessary.

**hatches**

[list[str], optional] *Only applies to* `contourf`.

A list of cross hatch patterns to use on the filled areas. If None, no hatching will be added to the contour.

**algorithm**

[{'mpl2005', 'mpl2014', 'serial', 'threaded'}, optional] Which contouring algorithm to use to calculate the contour lines and polygons. The algorithms are implemented in ContourPy, consult the ContourPy documentation for further information.

The default is taken from *rcParams["contour.algorithm"]* (default: `'mpl2014'`).

**clip_path**

[*Patch* or *Path* or *TransformedPath*] Set the clip path. See *set_clip_path*.

New in version 3.8.

**data**

[indexable object, optional] If given, all parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception).

## Notes

1. *contourf* differs from the MATLAB version in that it does not draw the polygon edges. To draw edges, add line contours with calls to *contour*.
2. *contourf* fills intervals that are closed at the top; that is, for boundaries *z1* and *z2*, the filled region is:

```
z1 < Z <= z2
```

   except for the lowest interval, which is closed on both sides (i.e. it includes the lowest value).

3. *contour* and *contourf* use a marching squares algorithm to compute contour locations. More information can be found in ContourPy documentation.

## Examples using `matplotlib.axes.Axes.contourf`

- *Contour Corner Mask*
- *Contourf demo*
- *Contourf Hatching*
- *Contourf and log color scale*
- *Contour plot of irregularly spaced data*
- *pcolormesh*
- *Triinterp Demo*
- *3D box surface plot*
- *Filled contours*
- *Project filled contour onto a graph*

- *contourf(X, Y, Z)*

### 2D arrays

| | |
|---|---|
| *Axes.imshow* | Display data as an image, i.e., on a 2D regular raster. |
| *Axes.matshow* | Plot the values of a 2D matrix or array as color-coded image. |
| *Axes.pcolor* | Create a pseudocolor plot with a non-regular rectangular grid. |
| *Axes.pcolorfast* | Create a pseudocolor plot with a non-regular rectangular grid. |
| *Axes.pcolormesh* | Create a pseudocolor plot with a non-regular rectangular grid. |
| *Axes.spy* | Plot the sparsity pattern of a 2D array. |

### matplotlib.axes.Axes.imshow

Axes.**imshow**(*X*, *cmap=None*, *norm=None*, ***, *aspect=None*, *interpolation=None*, *alpha=None*, *vmin=None*, *vmax=None*, *origin=None*, *extent=None*, *interpolation_stage=None*, *filternorm=True*, *filterrad=4.0*, *resample=None*, *url=None*, *data=None*, ***kwargs*)

Display data as an image, i.e., on a 2D regular raster.

The input may either be actual RGB(A) data, or 2D scalar data, which will be rendered as a pseudocolor image. For displaying a grayscale image, set up the colormapping using the parameters `cmap='gray', vmin=0, vmax=255`.

The number of pixels used to render an image is set by the Axes size and the figure *dpi*. This can lead to aliasing artifacts when the image is resampled, because the displayed image size will usually not match the size of *X* (see *Image antialiasing*). The resampling can be controlled via the *interpolation* parameter and/or *rcParams["image.interpolation"]* (default: `'antialiased'`).

**Parameters**

**X**

[array-like or PIL image] The image data. Supported array shapes are:

- (M, N): an image with scalar data. The values are mapped to colors using normalization and a colormap. See parameters *norm*, *cmap*, *vmin*, *vmax*.
- (M, N, 3): an image with RGB values (0-1 float or 0-255 int).
- (M, N, 4): an image with RGBA values (0-1 float or 0-255 int), i.e. including transparency.

The first two dimensions (M, N) define the rows and columns of the image.

Out-of-range RGB(A) values are clipped.

**cmap**

[str or *Colormap*, default: *rcParams["image.cmap"]* (default: `'viridis'`)] The Colormap instance or registered colormap name used to map scalar data to colors.

This parameter is ignored if *X* is RGB(A).

**norm**

[str or *Normalize*, optional] The normalization method used to scale scalar data to the [0, 1] range before mapping to colors using *cmap*. By default, a linear scaling is used, mapping the lowest value to 0 and the highest to 1.

If given, this can be one of the following:

- An instance of *Normalize* or one of its subclasses (see *Colormap normalization*).
- A scale name, i.e. one of "linear", "log", "symlog", "logit", etc. For a list of available scales, call *matplotlib.scale.get_scale_names()*. In that case, a suitable *Normalize* subclass is dynamically generated and instantiated.

This parameter is ignored if *X* is RGB(A).

**vmin, vmax**

[float, optional] When using scalar data and no explicit *norm*, *vmin* and *vmax* define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data. It is an error to use *vmin*/*vmax* when a *norm* instance is given (but using a `str` *norm* name together with *vmin*/*vmax* is acceptable).

This parameter is ignored if *X* is RGB(A).

**aspect**

[{'equal', 'auto'} or float or None, default: None] The aspect ratio of the Axes. This parameter is particularly relevant for images since it determines whether data pixels are square.

This parameter is a shortcut for explicitly calling *Axes.set_aspect*. See there for further details.

- 'equal': Ensures an aspect ratio of 1. Pixels will be square (unless pixel sizes are explicitly made non-square in data coordinates using *extent*).
- 'auto': The Axes is kept fixed and the aspect is adjusted so that the data fit in the Axes. In general, this will result in non-square pixels.

Normally, None (the default) means to use *rcParams["image.aspect"]* (default: `'equal'`). However, if the image uses a transform that does not contain the axes data transform, then None means to not modify the axes aspect at all (in that case, directly call *Axes.set_aspect* if desired).

**interpolation**

[str, default: *rcParams["image.interpolation"]* (default: 'antialiased')] The interpolation method used.

Supported values are 'none', 'antialiased', 'nearest', 'bilinear', 'bicubic', 'spline16', 'spline36', 'hanning', 'hamming', 'hermite', 'kaiser', 'quadric', 'catrom', 'gaussian', 'bessel', 'mitchell', 'sinc', 'lanczos', 'blackman'.

The data *X* is resampled to the pixel size of the image on the figure canvas, using the interpolation method to either up- or downsample the data.

If *interpolation* is 'none', then for the ps, pdf, and svg backends no down- or upsampling occurs, and the image data is passed to the backend as a native image. Note that different ps, pdf, and svg viewers may display these raw pixels differently. On other backends, 'none' is the same as 'nearest'.

If *interpolation* is the default 'antialiased', then 'nearest' interpolation is used if the image is upsampled by more than a factor of three (i.e. the number of display pixels is at least three times the size of the data array). If the upsampling rate is smaller than 3, or the image is downsampled, then 'hanning' interpolation is used to act as an anti-aliasing filter, unless the image happens to be upsampled by exactly a factor of two or one.

See *Interpolations for imshow* for an overview of the supported interpolation methods, and *Image antialiasing* for a discussion of image antialiasing.

Some interpolation methods require an additional radius parameter, which can be set by *filterrad*. Additionally, the antigrain image resize filter is controlled by the parameter *filternorm*.

**interpolation_stage**

[{'data', 'rgba'}, default: 'data'] If 'data', interpolation is carried out on the data provided by the user. If 'rgba', the interpolation is carried out after the colormapping has been applied (visual interpolation).

**alpha**

[float or array-like, optional] The alpha blending value, between 0 (transparent) and 1 (opaque). If *alpha* is an array, the alpha blending values are applied pixel by pixel, and *alpha* must have the same shape as *X*.

**origin**

[{'upper', 'lower'}, default: *rcParams["image.origin"]* (default: 'upper')] Place the [0, 0] index of the array in the upper left or lower left corner of the Axes. The convention (the default) 'upper' is typically used for matrices and images.

Note that the vertical axis points upward for 'lower' but downward for 'upper'.

See the *origin and extent in imshow* tutorial for examples and a more detailed description.

**extent**

[floats (left, right, bottom, top), optional] The bounding box in data coordinates that the image will fill. These values may be unitful and match the units of the Axes. The image is stretched individually along x and y to fill the box.

The default extent is determined by the following conditions. Pixels have unit size in data coordinates. Their centers are on integer coordinates, and their center coordinates range from 0 to columns-1 horizontally and from 0 to rows-1 vertically.

Note that the direction of the vertical axis and thus the default values for top and bottom depend on *origin*:

- For `origin == 'upper'` the default is `(-0.5, numcols-0.5, numrows-0.5, -0.5)`.
- For `origin == 'lower'` the default is `(-0.5, numcols-0.5, -0.5, numrows-0.5)`.

See the *origin and extent in imshow* tutorial for examples and a more detailed description.

**filternorm**

[bool, default: True] A parameter for the antigrain image resize filter (see the antigrain documentation). If *filternorm* is set, the filter normalizes integer values and corrects the rounding errors. It doesn't do anything with the source floating point values, it corrects only integers according to the rule of 1.0 which means that any sum of pixel weights must be equal to 1.0. So, the filter function must produce a graph of the proper shape.

**filterrad**

[float > 0, default: 4.0] The filter radius for filters that have a radius parameter, i.e. when interpolation is one of: 'sinc', 'lanczos' or 'blackman'.

**resample**

[bool, default: *rcParams["image.resample"]* (default: `True`)] When *True*, use a full resampling method. When *False*, only resample when the output image is larger than the input image.

**url**

[str, optional] Set the url of the created *AxesImage*. See *Artist.set_url*.

**Returns**

***AxesImage***

**Other Parameters**

**data**

[indexable object, optional] If given, all parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception).

****kwargs**

[*Artist* properties] These parameters are passed on to the constructor of the *AxesImage* artist.

**See also:**

***matshow***

Plot a matrix or an array as an image.

### Notes

Unless *extent* is used, pixel centers will be located at integer coordinates. In other words: the origin will coincide with the center of pixel (0, 0).

There are two common representations for RGB images with an alpha channel:

- Straight (unassociated) alpha: R, G, and B channels represent the color of the pixel, disregarding its opacity.
- Premultiplied (associated) alpha: R, G, and B channels represent the color of the pixel, adjusted for its opacity by multiplication.

*imshow* expects RGB images adopting the straight (unassociated) alpha representation.

### Examples using `matplotlib.axes.Axes.imshow`

- *Bar chart with gradients*
- *Affine transform of an image*
- *Barcode*
- *Interactive Adjustment of Colormap Range*
- *Contour Demo*
- *Contour Image*
- *Creating annotated heatmaps*
- *Image antialiasing*
- *Clipping images with patches*
- *Many ways to plot images*
- *Image Masked*
- *Blend transparency with color in 2D images*

- *Modifying the coordinate formatter*
- *Interpolations for imshow*
- *Layer Images*
- *Visualize matrices with matshow*
- *Multiple images*
- *pcolor images*
- *Shading example*
- *Axes box aspect*
- *Zoom region inset Axes*
- *Using a text as a Path*
- *Colorbar*
- *Colormap reference*
- *Creating a colormap from a list of colors*
- *Anchored Direction Arrow*
- *Demo Axes Grid*
- *Axes Grid2*
- *HBoxDivider and VBoxDivider demo*
- *Colorbar with AxesDivider*
- *Controlling the position and size of colorbars with Inset Axes*
- *Inset locator demo 2*
- *Simple ImageGrid*
- *Simple ImageGrid 2*
- *Shaded & power normalized rendering*
- *pyplot animation*
- *Animated image using a precomputed list of images*
- *Scroll event*
- *Pick event demo*
- *Viewlims*
- *Patheffect Demo*
- *Topographic hillshading*
- *Colorbar Tick Labelling*
- *imshow(Z)*

- *Constrained layout guide*
- *Tight layout guide*
- *Creating Colormaps in Matplotlib*
- *Choosing Colormaps in Matplotlib*

### matplotlib.axes.Axes.matshow

Axes.**matshow**(*Z*, ***kwargs*)

Plot the values of a 2D matrix or array as color-coded image.

The matrix will be shown the way it would be printed, with the first row at the top. Row and column numbering is zero-based.

**Parameters**

**Z**

[(M, N) array-like] The matrix to be displayed.

**Returns**

***AxesImage***

**Other Parameters**

****kwargs**

[*imshow* arguments]

**See also:**

***imshow***

More general function to plot data on a 2D regular raster.

#### Notes

This is just a convenience function wrapping *imshow* to set useful defaults for displaying a matrix. In particular:

- Set `origin='upper'`.
- Set `interpolation='nearest'`.
- Set `aspect='equal'`.
- Ticks are placed to the left and above.
- Ticks are formatted to show integer indices.

## matplotlib.axes.Axes.pcolor

Axes.**pcolor**(**args*, *shading=None*, *alpha=None*, *norm=None*, *cmap=None*, *vmin=None*, *vmax=None*, *data=None*, ***kwargs*)

Create a pseudocolor plot with a non-regular rectangular grid.

Call signature:

```
pcolor([X, Y,] C, **kwargs)
```

*X* and *Y* can be used to specify the corners of the quadrilaterals.

---

**Hint:** `pcolor()` can be very slow for large arrays. In most cases you should use the similar but much faster `pcolormesh` instead. See *Differences between pcolor() and pcolormesh()* for a discussion of the differences.

---

### Parameters

#### C

[2D array-like] The color-mapped values. Color-mapping is controlled by *cmap*, *norm*, *vmin*, and *vmax*.

#### X, Y

[array-like, optional] The coordinates of the corners of quadrilaterals of a pcolormesh:

```
(X[i+1, j], Y[i+1, j])         (X[i+1, j+1], Y[i+1, j+1])
                       ●------●
                       |      |
                       ●------●
    (X[i, j], Y[i, j])         (X[i, j+1], Y[i, j+1])
```

Note that the column index corresponds to the x-coordinate, and the row index corresponds to y. For details, see the *Notes* section below.

If `shading='flat'` the dimensions of *X* and *Y* should be one greater than those of *C*, and the quadrilateral is colored due to the value at `C[i, j]`. If *X*, *Y* and *C* have equal dimensions, a warning will be raised and the last row and column of *C* will be ignored.

If `shading='nearest'`, the dimensions of *X* and *Y* should be the same as those of *C* (if not, a ValueError will be raised). The color `C[i, j]` will be centered on `(X[i, j], Y[i, j])`.

If *X* and/or *Y* are 1-D arrays or column vectors they will be expanded as needed into the appropriate 2D arrays, making a rectangular grid.

#### shading

[{'flat', 'nearest', 'auto'}, default: *rcParams["pcolor.shading"]* (default: `'auto'`)] The fill style for the quadrilateral. Possible values:

- 'flat': A solid color is used for each quad. The color of the quad (i, j), (i+1, j), (i, j+1), (i+1, j+1) is given by `C[i, j]`. The dimensions of *X* and *Y* should be one greater than those of *C*; if they are the same as *C*, then a deprecation warning is raised, and the last row and column of *C* are dropped.
- 'nearest': Each grid point will have a color centered on it, extending halfway between the adjacent grid centers. The dimensions of *X* and *Y* must be the same as *C*.
- 'auto': Choose 'flat' if dimensions of *X* and *Y* are one larger than *C*. Choose 'nearest' if dimensions are the same.

See *pcolormesh grids and shading* for more description.

**cmap**

[str or *Colormap*, default: *rcParams["image.cmap"]* (default: `'viridis'`)] The Colormap instance or registered colormap name used to map scalar data to colors.

**norm**

[str or *Normalize*, optional] The normalization method used to scale scalar data to the [0, 1] range before mapping to colors using *cmap*. By default, a linear scaling is used, mapping the lowest value to 0 and the highest to 1.

If given, this can be one of the following:

- An instance of *Normalize* or one of its subclasses (see *Colormap normalization*).
- A scale name, i.e. one of "linear", "log", "symlog", "logit", etc. For a list of available scales, call *matplotlib.scale.get_scale_names()*. In that case, a suitable *Normalize* subclass is dynamically generated and instantiated.

**vmin, vmax**

[float, optional] When using scalar data and no explicit *norm*, *vmin* and *vmax* define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data. It is an error to use *vmin*/*vmax* when a *norm* instance is given (but using a `str` *norm* name together with *vmin*/*vmax* is acceptable).

**edgecolors**

[{'none', None, 'face', color, color sequence}, optional] The color of the edges. Defaults to 'none'. Possible values:

- 'none' or '': No edge.
- *None*: *rcParams["patch.edgecolor"]* (default: `'black'`) will be used. Note that currently *rcParams["patch.force_edgecolor"]*

(default: `False`) has to be True for this to work.

- 'face': Use the adjacent face color.
- A color or sequence of colors will set the edge color.

The singular form *edgecolor* works as an alias.

**alpha**

[float, default: None] The alpha blending value of the face color, between 0 (transparent) and 1 (opaque). Note: The edgecolor is currently not affected by this.

**snap**

[bool, default: False] Whether to snap the mesh to pixel boundaries.

**Returns**

*matplotlib.collections.PolyQuadMesh*

**Other Parameters**

**antialiaseds**

[bool, default: False] The default *antialiaseds* is False if the default *edgecolors*="none" is used. This eliminates artificial lines at patch boundaries, and works regardless of the value of alpha. If *edgecolors* is not "none", then the default *antialiaseds* is taken from *rcParams["patch.antialiased"]* (default: `True`). Stroking the edges may be preferred if *alpha* is 1, but will cause artifacts otherwise.

**data**

[indexable object, optional] If given, all parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception).

****kwargs**

Additionally, the following arguments are allowed. They are passed along to the *PolyQuadMesh* constructor:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a d |
| *alpha* | array-like or scalar or None |
| *animated* | bool |
| *antialiased* or aa or antialiaseds | bool or list of bools |
| *array* | array-like or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clim* | (vmin: float, vmax: float) |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |

Table 23 – continued fro

| Property | Description |
|---|---|
| *clip_path* | Patch or (Path, Transform) or None |
| *cmap* | *Colormap* or str or None |
| *color* | *color* or list of RGBA tuples |
| *edgecolor* or ec or edgecolors | *color* or list of *color* or 'face' |
| *facecolor* or facecolors or fc | *color* or list of *color* |
| *figure* | *Figure* |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or dashes or linestyles or ls | str or tuple or list thereof |
| *linewidth* or linewidths or lw | float or list of floats |
| *mouseover* | bool |
| *norm* | *Normalize* or str or None |
| *offset_transform* or transOffset | *Transform* |
| *offsets* | (N, 2) or (2,) array-like |
| *path_effects* | list of *AbstractPathEffect* |
| *paths* | list of array-like |
| *picker* | None or bool or float or callable |
| *pickradius* | float |
| *rasterized* | bool |
| sizes | numpy.ndarray or None |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *urls* | list of str or None |
| *verts* | list of array-like |
| *verts_and_codes* | unknown |
| *visible* | bool |
| *zorder* | float |

**See also:**

***pcolormesh***

for an explanation of the differences between pcolor and pcolormesh.

***imshow***

If *X* and *Y* are each equidistant, *imshow* can be a faster alternative.

### Notes

#### Masked arrays

*X*, *Y* and *C* may be masked arrays. If either `C[i, j]`, or one of the vertices surrounding `C[i, j]` (*X* or *Y* at `[i, j], [i+1, j], [i, j+1], [i+1, j+1]`) is masked, nothing is plotted.

#### Grid orientation

The grid orientation follows the standard matrix convention: An array *C* with shape (nrows, ncolumns) is plotted with the column number as *X* and the row number as *Y*.

## Examples using `matplotlib.axes.Axes.pcolor`

- *pcolor images*
- *Controlling view limits using margins and sticky_edges*

## matplotlib.axes.Axes.pcolorfast

`Axes.`**`pcolorfast`**(**args*, *alpha=None*, *norm=None*, *cmap=None*, *vmin=None*, *vmax=None*, *data=None*, ***kwargs*)

Create a pseudocolor plot with a non-regular rectangular grid.

Call signature:

```
ax.pcolorfast([X, Y], C, /, **kwargs)
```

This method is similar to *pcolor* and *pcolormesh*. It's designed to provide the fastest pcolor-type plotting with the Agg backend. To achieve this, it uses different algorithms internally depending on the complexity of the input grid (regular rectangular, non-regular rectangular or arbitrary quadrilateral).

**Warning:** This method is experimental. Compared to *pcolor* or *pcolormesh* it has some limitations:

- It supports only flat shading (no outlines)
- It lacks support for log scaling of the axes.
- It does not have a pyplot wrapper.

### Parameters

**C**

[array-like] The image data. Supported array shapes are:

- (M, N): an image with scalar data. Color-mapping is controlled by *cmap*, *norm*, *vmin*, and *vmax*.

- (M, N, 3): an image with RGB values (0-1 float or 0-255 int).
- (M, N, 4): an image with RGBA values (0-1 float or 0-255 int), i.e. including transparency.

The first two dimensions (M, N) define the rows and columns of the image.

This parameter can only be passed positionally.

**X, Y**

[tuple or array-like, default: `(0, N), (0, M)`] *X* and *Y* are used to specify the coordinates of the quadrilaterals. There are different ways to do this:

- Use tuples `X=(xmin, xmax)` and `Y=(ymin, ymax)` to define a *uniform rectangular grid*.

  The tuples define the outer edges of the grid. All individual quadrilaterals will be of the same size. This is the fastest version.

- Use 1D arrays *X*, *Y* to specify a *non-uniform rectangular grid*.

  In this case *X* and *Y* have to be monotonic 1D arrays of length *N+1* and *M+1*, specifying the x and y boundaries of the cells.

  The speed is intermediate. Note: The grid is checked, and if found to be uniform the fast version is used.

- Use 2D arrays *X*, *Y* if you need an *arbitrary quadrilateral grid* (i.e. if the quadrilaterals are not rectangular).

  In this case *X* and *Y* are 2D arrays with shape (M + 1, N + 1), specifying the x and y coordinates of the corners of the colored quadrilaterals.

  This is the most general, but the slowest to render. It may produce faster and more compact output using ps, pdf, and svg backends, however.

These arguments can only be passed positionally.

**cmap**

[str or *Colormap*, default: *rcParams["image.cmap"]* (default: `'viridis'`)] The Colormap instance or registered colormap name used to map scalar data to colors.

This parameter is ignored if *C* is RGB(A).

**norm**

[str or *Normalize*, optional] The normalization method used to scale scalar data to the [0, 1] range before mapping to colors using *cmap*. By default, a linear scaling is used, mapping the lowest value to 0 and the highest to 1.

If given, this can be one of the following:

- An instance of *Normalize* or one of its subclasses (see *Colormap normalization*).

- A scale name, i.e. one of "linear", "log", "symlog", "logit", etc. For a list of available scales, call *matplotlib.scale.get_scale_names()*. In that case, a suitable *Normalize* subclass is dynamically generated and instantiated.

This parameter is ignored if *C* is RGB(A).

**vmin, vmax**

[float, optional] When using scalar data and no explicit *norm*, *vmin* and *vmax* define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data. It is an error to use *vmin*/*vmax* when a *norm* instance is given (but using a `str` *norm* name together with *vmin*/*vmax* is acceptable).

This parameter is ignored if *C* is RGB(A).

**alpha**

[float, default: None] The alpha blending value, between 0 (transparent) and 1 (opaque).

**snap**

[bool, default: False] Whether to snap the mesh to pixel boundaries.

### Returns

***AxesImage* or *PcolorImage* or *QuadMesh***

The return type depends on the type of grid:

- *AxesImage* for a regular rectangular grid.
- *PcolorImage* for a non-regular rectangular grid.
- *QuadMesh* for a non-rectangular grid.

### Other Parameters

**data**

[indexable object, optional] If given, all parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception).

****kwargs**

Supported additional parameters depend on the type of grid. See return types of *image* for further description.

### Examples using `matplotlib.axes.Axes.pcolorfast`

- *pcolor images*

## matplotlib.axes.Axes.pcolormesh

Axes.**pcolormesh**(**args*, *alpha=None*, *norm=None*, *cmap=None*, *vmin=None*, *vmax=None*, *shading=None*, *antialiased=False*, *data=None*, ***kwargs*)

Create a pseudocolor plot with a non-regular rectangular grid.

Call signature:

```
pcolormesh([X, Y,] C, **kwargs)
```

*X* and *Y* can be used to specify the corners of the quadrilaterals.

---

**Hint:** `pcolormesh` is similar to `pcolor`. It is much faster and preferred in most cases. For a detailed discussion on the differences see *Differences between pcolor() and pcolormesh()*.

---

#### Parameters

**C**

[array-like] The mesh data. Supported array shapes are:

- (M, N) or M*N: a mesh with scalar data. The values are mapped to colors using normalization and a colormap. See parameters *norm*, *cmap*, *vmin*, *vmax*.
- (M, N, 3): an image with RGB values (0-1 float or 0-255 int).
- (M, N, 4): an image with RGBA values (0-1 float or 0-255 int), i.e. including transparency.

The first two dimensions (M, N) define the rows and columns of the mesh data.

**X, Y**

[array-like, optional] The coordinates of the corners of quadrilaterals of a pcolormesh:

```
(X[i+1, j], Y[i+1, j])         (X[i+1, j+1], Y[i+1, j+1])
                       ●━━━━━●
                       ┃     ┃
                       ●━━━━━●
    (X[i, j], Y[i, j])         (X[i, j+1], Y[i, j+1])
```

Note that the column index corresponds to the x-coordinate, and the row index corresponds to y. For details, see the *Notes* section below.

If `shading='flat'` the dimensions of *X* and *Y* should be one greater than those of *C*, and the quadrilateral is colored due to the value at `C[i, j]`. If *X*, *Y* and *C* have equal dimensions, a warning will be raised and the last row and column of *C* will be ignored.

If `shading='nearest'` or `'gouraud'`, the dimensions of *X* and *Y* should be the same as those of *C* (if not, a ValueError will be raised). For `'nearest'` the color `C[i, j]` is centered on `(X[i, j], Y[i, j])`. For `'gouraud'`, a smooth interpolation is carried out between the quadrilateral corners.

If *X* and/or *Y* are 1-D arrays or column vectors they will be expanded as needed into the appropriate 2D arrays, making a rectangular grid.

**cmap**

[str or *Colormap*, default: *rcParams["image.cmap"]* (default: `'viridis'`)] The Colormap instance or registered colormap name used to map scalar data to colors.

**norm**

[str or *Normalize*, optional] The normalization method used to scale scalar data to the [0, 1] range before mapping to colors using *cmap*. By default, a linear scaling is used, mapping the lowest value to 0 and the highest to 1.

If given, this can be one of the following:

- An instance of *Normalize* or one of its subclasses (see *Colormap normalization*).
- A scale name, i.e. one of "linear", "log", "symlog", "logit", etc. For a list of available scales, call *matplotlib.scale.get_scale_names()*. In that case, a suitable *Normalize* subclass is dynamically generated and instantiated.

**vmin, vmax**

[float, optional] When using scalar data and no explicit *norm*, *vmin* and *vmax* define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data. It is an error to use *vmin*/*vmax* when a *norm* instance is given (but using a `str` *norm* name together with *vmin*/*vmax* is acceptable).

**edgecolors**

[{'none', None, 'face', color, color sequence}, optional] The color of the edges. Defaults to 'none'. Possible values:

- 'none' or '': No edge.
- *None*: *rcParams["patch.edgecolor"]* (default: `'black'`) will be used. Note that currently *rcParams["patch.force_edgecolor"]* (default: `False`) has to be True for this to work.
- 'face': Use the adjacent face color.

- A color or sequence of colors will set the edge color.

The singular form *edgecolor* works as an alias.

**alpha**

[float, default: None] The alpha blending value, between 0 (transparent) and 1 (opaque).

**shading**

[{'flat', 'nearest', 'gouraud', 'auto'}, optional] The fill style for the quadrilateral; defaults to `rcParams["pcolor.shading"]` (default: `'auto'`). Possible values:

- 'flat': A solid color is used for each quad. The color of the quad (i, j), (i+1, j), (i, j+1), (i+1, j+1) is given by `C[i, j]`. The dimensions of *X* and *Y* should be one greater than those of *C*; if they are the same as *C*, then a deprecation warning is raised, and the last row and column of *C* are dropped.
- 'nearest': Each grid point will have a color centered on it, extending halfway between the adjacent grid centers. The dimensions of *X* and *Y* must be the same as *C*.
- 'gouraud': Each quad will be Gouraud shaded: The color of the corners (i', j') are given by `C[i', j']`. The color values of the area in between is interpolated from the corner values. The dimensions of *X* and *Y* must be the same as *C*. When Gouraud shading is used, *edgecolors* is ignored.
- 'auto': Choose 'flat' if dimensions of *X* and *Y* are one larger than *C*. Choose 'nearest' if dimensions are the same.

See *pcolormesh grids and shading* for more description.

**snap**

[bool, default: False] Whether to snap the mesh to pixel boundaries.

**rasterized**

[bool, optional] Rasterize the pcolormesh when drawing vector graphics. This can speed up rendering and produce smaller files for large data sets. See also *Rasterization for vector graphics*.

**Returns**

***matplotlib.collections.QuadMesh***

**Other Parameters**

**data**

[indexable object, optional] If given, all parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception).

**kwargs

Additionally, the following arguments are allowed. They are passed along to the *QuadMesh* constructor:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a d |
| *alpha* | array-like or scalar or None |
| *animated* | bool |
| *antialiased* or aa or antialiaseds | bool or list of bools |
| array | array-like |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clim* | (vmin: float, vmax: float) |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *cmap* | *Colormap* or str or None |
| *color* | *color* or list of RGBA tuples |
| *edgecolor* or ec or edgecolors | *color* or list of *color* or 'face' |
| *facecolor* or facecolors or fc | *color* or list of *color* |
| *figure* | *Figure* |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or dashes or linestyles or ls | str or tuple or list thereof |
| *linewidth* or linewidths or lw | float or list of floats |
| *mouseover* | bool |
| *norm* | *Normalize* or str or None |
| *offset_transform* or transOffset | *Transform* |
| *offsets* | (N, 2) or (2,) array-like |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *pickradius* | float |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *urls* | list of str or None |
| *visible* | bool |
| *zorder* | float |

**See also:**

***pcolor***

An alternative implementation with slightly different features. For a detailed discussion on the differences see *Differences between pcolor() and pcolormesh()*.

***imshow***

If *X* and *Y* are each equidistant, *imshow* can be a faster alternative.

## Notes

### Masked arrays

*C* may be a masked array. If `C[i, j]` is masked, the corresponding quadrilateral will be transparent. Masking of *X* and *Y* is not supported. Use *pcolor* if you need this functionality.

### Grid orientation

The grid orientation follows the standard matrix convention: An array *C* with shape (nrows, ncolumns) is plotted with the column number as *X* and the row number as *Y*.

### Differences between pcolor() and pcolormesh()

Both methods are used to create a pseudocolor plot of a 2D array using quadrilaterals.

The main difference lies in the created object and internal data handling: While *pcolor* returns a *PolyQuadMesh*, *pcolormesh* returns a *QuadMesh*. The latter is more specialized for the given purpose and thus is faster. It should almost always be preferred.

There is also a slight difference in the handling of masked arrays. Both *pcolor* and *pcolormesh* support masked arrays for *C*. However, only *pcolor* supports masked arrays for *X* and *Y*. The reason lies in the internal handling of the masked values. *pcolor* leaves out the respective polygons from the PolyQuadMesh. *pcolormesh* sets the facecolor of the masked elements to transparent. You can see the difference when using edgecolors. While all edges are drawn irrespective of masking in a QuadMesh, the edge between two adjacent masked quadrilaterals in *pcolor* is not drawn as the corresponding polygons do not exist in the PolyQuadMesh. Because PolyQuadMesh draws each individual polygon, it also supports applying hatches and linestyles to the collection.

Another difference is the support of Gouraud shading in *pcolormesh*, which is not available with *pcolor*.

## Examples using `matplotlib.axes.Axes.pcolormesh`

- *pcolor images*
- *pcolormesh grids and shading*
- *pcolormesh*
- *QuadMesh Demo*
- *Figure subfigures*
- *Time Series Histogram*

- *Rasterization for vector graphics*
- *pcolormesh(X, Y, Z)*
- *Constrained layout guide*
- *Placing colorbars*
- *Creating Colormaps in Matplotlib*
- *Colormap normalization*

## matplotlib.axes.Axes.spy

Axes.**spy**(*Z*, *precision=0*, *marker=None*, *markersize=None*, *aspect='equal'*, *origin='upper'*, ***kwargs*)

Plot the sparsity pattern of a 2D array.

This visualizes the non-zero values of the array.

Two plotting styles are available: image and marker. Both are available for full arrays, but only the marker style works for `scipy.sparse.spmatrix` instances.

**Image style**

If *marker* and *markersize* are *None*, `imshow` is used. Any extra remaining keyword arguments are passed to this method.

**Marker style**

If *Z* is a `scipy.sparse.spmatrix` or *marker* or *markersize* are *None*, a `Line2D` object will be returned with the value of marker determining the marker type, and any remaining keyword arguments passed to `plot`.

**Parameters**

**Z**

[(M, N) array-like] The array to be plotted.

**precision**

[float or 'present', default: 0] If *precision* is 0, any non-zero value will be plotted. Otherwise, values of $|Z| > precision$ will be plotted.

For `scipy.sparse.spmatrix` instances, you can also pass 'present'. In this case any value present in the array will be plotted, even if it is identically zero.

**aspect**

[{'equal', 'auto', None} or float, default: 'equal'] The aspect ratio of the Axes. This parameter is particularly relevant for images since it determines whether data pixels are square.

This parameter is a shortcut for explicitly calling `Axes.set_aspect`. See there for further details.

- 'equal': Ensures an aspect ratio of 1. Pixels will be square.
- 'auto': The Axes is kept fixed and the aspect is adjusted so that the data fit in the Axes. In general, this will result in non-square pixels.
- *None*: Use `rcParams["image.aspect"]` (default: `'equal'`).

**origin**

[{'upper', 'lower'}, default: `rcParams["image.origin"]` (default: `'upper'`)] Place the [0, 0] index of the array in the upper left or lower left corner of the Axes. The convention 'upper' is typically used for matrices and images.

**Returns**

***AxesImage* or *Line2D***

The return type depends on the plotting style (see above).

**Other Parameters**

****kwargs**

The supported additional parameters depend on the plotting style.

For the image style, you can pass the following additional parameters of `imshow`:

- *cmap*
- *alpha*
- *url*
- any `Artist` properties (passed on to the `AxesImage`)

For the marker style, you can pass any `Line2D` property except for *linestyle*:

| Property | Description |
|---|---|
| `agg_filter` | a filter function, which takes a (m, n, 3) float array and a dpi va |
| `alpha` | scalar or None |
| `animated` | bool |
| `antialiased` or aa | bool |
| `clip_box` | `BboxBase` or None |
| `clip_on` | bool |
| `clip_path` | Patch or (Path, Transform) or None |
| `color` or c | *color* |
| `dash_capstyle` | `CapStyle` or {'butt', 'projecting', 'round'} |
| `dash_joinstyle` | `JoinStyle` or {'miter', 'round', 'bevel'} |
| `dashes` | sequence of floats (on/off ink in points) or (None, None) |
| `data` | (2, N) array or two 1D arrays |
| `drawstyle` or ds | {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: |
| `figure` | `Figure` |

Table 25 – continued from

| Property | Description |
|---|---|
| *fillstyle* | {'full', 'left', 'right', 'bottom', 'top', 'none'} |
| *gapcolor* | *color* or None |
| *gid* | str |
| *in_layout* | bool |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float |
| *marker* | marker style string, *Path* or *MarkerStyle* |
| *markeredgecolor* or mec | *color* |
| *markeredgewidth* or mew | float |
| *markerfacecolor* or mfc | *color* |
| *markerfacecoloralt* or mfcalt | *color* |
| *markersize* or ms | float |
| *markevery* | None or int or (int, int) or slice or list[int] or float or (float, floa |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | float or callable[[Artist, Event], tuple[bool, dict]] |
| *pickradius* | float |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *solid_capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *solid_joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *transform* | unknown |
| *url* | str |
| *visible* | bool |
| *xdata* | 1D array |
| *ydata* | 1D array |
| *zorder* | float |

### Examples using `matplotlib.axes.Axes.spy`

- *Spy Demos*

### Unstructured triangles

| | |
|---|---|
| *Axes.tripcolor* | Create a pseudocolor plot of an unstructured triangular grid. |
| *Axes.triplot* | Draw an unstructured triangular grid as lines and/or markers. |
| *Axes.tricontour* | Draw contour lines on an unstructured triangular grid. |
| *Axes.tricontourf* | Draw contour regions on an unstructured triangular grid. |

### matplotlib.axes.Axes.tripcolor

Axes.**tripcolor**(**args*, *alpha=1.0*, *norm=None*, *cmap=None*, *vmin=None*, *vmax=None*, *shading='flat'*, *facecolors=None*, ***kwargs*)

Create a pseudocolor plot of an unstructured triangular grid.

Call signatures:

```
tripcolor(triangulation, c, *, ...)
tripcolor(x, y, c, *, [triangles=triangles], [mask=mask], ...)
```

The triangular grid can be specified either by passing a *Triangulation* object as the first parameter, or by passing the points *x*, *y* and optionally the *triangles* and a *mask*. See *Triangulation* for an explanation of these parameters.

It is possible to pass the triangles positionally, i.e. `tripcolor(x, y, triangles, c, ...)`. However, this is discouraged. For more clarity, pass *triangles* via keyword argument.

If neither of *triangulation* or *triangles* are given, the triangulation is calculated on the fly. In this case, it does not make sense to provide colors at the triangle faces via *c* or *facecolors* because there are multiple possible triangulations for a group of points and you don't know which triangles will be constructed.

**Parameters**

**triangulation**

[*Triangulation*] An already created triangular grid.

**x, y, triangles, mask**

Parameters defining the triangular grid. See *Triangulation*. This is mutually exclusive with specifying *triangulation*.

**c**

[array-like] The color values, either for the points or for the triangles. Which one is automatically inferred from the length of *c*, i.e. does it match the number of points

or the number of triangles. If there are the same number of points and triangles in the triangulation it is assumed that color values are defined at points; to force the use of color values at triangles use the keyword argument `facecolors=c` instead of just `c`. This parameter is position-only.

**facecolors**

[array-like, optional] Can be used alternatively to *c* to specify colors at the triangle faces. This parameter takes precedence over *c*.

**shading**

[{'flat', 'gouraud'}, default: 'flat'] If 'flat' and the color values *c* are defined at points, the color values used for each triangle are from the mean c of the triangle's three points. If *shading* is 'gouraud' then color values must be defined at points.

**other_parameters**

All other parameters are the same as for *pcolor*.

### Examples using `matplotlib.axes.Axes.tripcolor`

- *Tripcolor Demo*
- *tripcolor(x, y, z)*

## matplotlib.axes.Axes.triplot

Axes.**triplot**(**args*, ***kwargs*)

Draw an unstructured triangular grid as lines and/or markers.

Call signatures:

```
triplot(triangulation, ...)
triplot(x, y, [triangles], *, [mask=mask], ...)
```

The triangular grid can be specified either by passing a *Triangulation* object as the first parameter, or by passing the points *x*, *y* and optionally the *triangles* and a *mask*. If neither of *triangulation* or *triangles* are given, the triangulation is calculated on the fly.

**Parameters**

**triangulation**

[*Triangulation*] An already created triangular grid.

**x, y, triangles, mask**

Parameters defining the triangular grid. See *Triangulation*. This is mutually exclusive with specifying *triangulation*.

**other_parameters**

All other args and kwargs are forwarded to *plot*.

**Returns**

**lines**

[*Line2D*] The drawn triangles edges.

**markers**

[*Line2D*] The drawn marker nodes.

### Examples using `matplotlib.axes.Axes.triplot`

- *Tricontour Smooth Delaunay*
- *Tricontour Smooth User*
- *Trigradient Demo*
- *Triinterp Demo*
- *Triplot Demo*
- *Trifinder Event Demo*
- *triplot(x, y)*

## matplotlib.axes.Axes.tricontour

Axes.**tricontour**(**args*, ***kwargs*)

Draw contour lines on an unstructured triangular grid.

Call signatures:

```
tricontour(triangulation, z, [levels], ...)
tricontour(x, y, z, [levels], *, [triangles=triangles], [mask=mask], ...)
```

The triangular grid can be specified either by passing a *Triangulation* object as the first parameter, or by passing the points *x*, *y* and optionally the *triangles* and a *mask*. See *Triangulation* for an explanation of these parameters. If neither of *triangulation* or *triangles* are given, the triangulation is calculated on the fly.

It is possible to pass *triangles* positionally, i.e. `tricontour(x, y, triangles, z, ...)`. However, this is discouraged. For more clarity, pass *triangles* via keyword argument.

**Parameters**

**triangulation**

[*Triangulation*, optional] An already created triangular grid.

**x, y, triangles, mask**

Parameters defining the triangular grid. See *Triangulation*. This is mutually exclusive with specifying *triangulation*.

**z**

[array-like] The height values over which the contour is drawn. Color-mapping is controlled by *cmap*, *norm*, *vmin*, and *vmax*.

---

**Note:** All values in *z* must be finite. Hence, nan and inf values must either be removed or *set_mask* be used.

---

**levels**

[int or array-like, optional] Determines the number and positions of the contour lines / regions.

If an int *n*, use *MaxNLocator*, which tries to automatically choose no more than *n+1* "nice" contour levels between between minimum and maximum numeric values of *Z*.

If array-like, draw contour lines at the specified levels. The values must be in increasing order.

### Returns

***TriContourSet***

### Other Parameters

**colors**

[*color* or list of *color*, optional] The colors of the levels, i.e., the contour lines.

The sequence is cycled for the levels in ascending order. If the sequence is shorter than the number of levels, it is repeated.

As a shortcut, single color strings may be used in place of one-element lists, i.e. `'red'` instead of `['red']` to color all levels with the same color. This shortcut does only work for color strings, not for other ways of specifying colors.

By default (value *None*), the colormap specified by *cmap* will be used.

**alpha**

[float, default: 1] The alpha blending value, between 0 (transparent) and 1 (opaque).

**cmap**

[str or *Colormap*, default: *rcParams["image.cmap"]* (default: `'viridis'`)] The Colormap instance or registered colormap name used to map scalar data to colors.

This parameter is ignored if *colors* is set.

**norm**

[str or `Normalize`, optional] The normalization method used to scale scalar data to the [0, 1] range before mapping to colors using *cmap*. By default, a linear scaling is used, mapping the lowest value to 0 and the highest to 1.

If given, this can be one of the following:

- An instance of `Normalize` or one of its subclasses (see *Colormap normalization*).
- A scale name, i.e. one of "linear", "log", "symlog", "logit", etc. For a list of available scales, call `matplotlib.scale.get_scale_names()`. In that case, a suitable `Normalize` subclass is dynamically generated and instantiated.

This parameter is ignored if *colors* is set.

**vmin, vmax**

[float, optional] When using scalar data and no explicit *norm*, *vmin* and *vmax* define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data. It is an error to use *vmin*/*vmax* when a *norm* instance is given (but using a `str` *norm* name together with *vmin*/*vmax* is acceptable).

If *vmin* or *vmax* are not given, the default color scaling is based on *levels*.

This parameter is ignored if *colors* is set.

**origin**

[{*None*, 'upper', 'lower', 'image'}, default: None] Determines the orientation and exact position of *z* by specifying the position of `z[0, 0]`. This is only relevant, if *X*, *Y* are not given.

- *None*: `z[0, 0]` is at X=0, Y=0 in the lower left corner.
- 'lower': `z[0, 0]` is at X=0.5, Y=0.5 in the lower left corner.
- 'upper': `z[0, 0]` is at X=N+0.5, Y=0.5 in the upper left corner.
- 'image': Use the value from `rcParams["image.origin"]` (default: `'upper'`).

**extent**

[(x0, x1, y0, y1), optional] If *origin* is not *None*, then *extent* is interpreted as in `imshow`: it gives the outer pixel boundaries. In this case, the position of z[0, 0] is the center of the pixel, not a corner. If *origin* is *None*, then (*x0*, *y0*) is the position of z[0, 0], and (*x1*, *y1*) is the position of z[-1, -1].

This argument is ignored if *X* and *Y* are specified in the call to contour.

**locator**

[ticker.Locator subclass, optional] The locator is used to determine the contour levels if they are not given explicitly via *levels*. Defaults to *MaxNLocator*.

**extend**

[{'neither', 'both', 'min', 'max'}, default: 'neither'] Determines the tricontour-coloring of values that are outside the *levels* range.

If 'neither', values outside the *levels* range are not colored. If 'min', 'max' or 'both', color the values below, above or below and above the *levels* range.

Values below `min(levels)` and above `max(levels)` are mapped to the under/over values of the *Colormap*. Note that most colormaps do not have dedicated colors for these by default, so that the over and under values are the edge values of the colormap. You may want to set these values explicitly using *Colormap.set_under* and *Colormap.set_over*.

---

**Note:** An existing *TriContourSet* does not get notified if properties of its colormap are changed. Therefore, an explicit call to *ContourSet.changed()* is needed after modifying the colormap. The explicit call can be left out, if a colorbar is assigned to the *TriContourSet* because it internally calls *ContourSet.changed()*.

---

**xunits, yunits**

[registered units, optional] Override axis units by specifying an instance of a *matplotlib.units.ConversionInterface*.

**antialiased**

[bool, optional] Enable antialiasing, overriding the defaults. For filled contours, the default is *True*. For line contours, it is taken from *rcParams["lines.antialiased"]* (default: `True`).

**linewidths**

[float or array-like, default: *rcParams["contour.linewidth"]* (default: `None`)] The line width of the contour lines.

If a number, all levels will be plotted with this linewidth.

If a sequence, the levels in ascending order will be plotted with the linewidths in the order specified.

If None, this falls back to *rcParams["lines.linewidth"]* (default: `1.5`).

**linestyles**

[{*None*, 'solid', 'dashed', 'dashdot', 'dotted'}, optional] If *linestyles* is *None*, the default is 'solid' unless the lines are monochrome. In that case, negative contours will take their linestyle from *rcParams["contour.negative_linestyle"]* (default: `'dashed'`) setting.

*linestyles* can also be an iterable of the above strings specifying a set of linestyles to be used. If this iterable is shorter than the number of contour levels it will be repeated as necessary.

### Examples using `matplotlib.axes.Axes.tricontour`

- *Contour plot of irregularly spaced data*
- *Tricontour Demo*
- *Tricontour Smooth Delaunay*
- *Tricontour Smooth User*
- *Trigradient Demo*
- *Triangular 3D contour plot*
- *tricontour(x, y, z)*

## matplotlib.axes.Axes.tricontourf

`Axes.`**`tricontourf`**`(*args, **kwargs)`

Draw contour regions on an unstructured triangular grid.

Call signatures:

```
tricontourf(triangulation, z, [levels], ...)
tricontourf(x, y, z, [levels], *, [triangles=triangles], [mask=mask], ...
 ↪)
```

The triangular grid can be specified either by passing a *Triangulation* object as the first parameter, or by passing the points *x*, *y* and optionally the *triangles* and a *mask*. See *Triangulation* for an explanation of these parameters. If neither of *triangulation* or *triangles* are given, the triangulation is calculated on the fly.

It is possible to pass *triangles* positionally, i.e. `tricontourf(x, y, triangles, z, ...)`. However, this is discouraged. For more clarity, pass *triangles* via keyword argument.

**Parameters**

**triangulation**

[*Triangulation*, optional] An already created triangular grid.

**x, y, triangles, mask**

Parameters defining the triangular grid. See *Triangulation*. This is mutually exclusive with specifying *triangulation*.

**z**

[array-like] The height values over which the contour is drawn. Color-mapping is controlled by *cmap*, *norm*, *vmin*, and *vmax*.

---

**Note:** All values in $z$ must be finite. Hence, nan and inf values must either be removed or `set_mask` be used.

---

**levels**

[int or array-like, optional] Determines the number and positions of the contour lines / regions.

If an int *n*, use `MaxNLocator`, which tries to automatically choose no more than *n+1* "nice" contour levels between between minimum and maximum numeric values of *Z*.

If array-like, draw contour lines at the specified levels. The values must be in increasing order.

**Returns**

***TriContourSet***

**Other Parameters**

**colors**

[*color* or list of *color*, optional] The colors of the levels, i.e., the contour regions.

The sequence is cycled for the levels in ascending order. If the sequence is shorter than the number of levels, it is repeated.

As a shortcut, single color strings may be used in place of one-element lists, i.e. `'red'` instead of `['red']` to color all levels with the same color. This shortcut does only work for color strings, not for other ways of specifying colors.

By default (value *None*), the colormap specified by *cmap* will be used.

**alpha**

[float, default: 1] The alpha blending value, between 0 (transparent) and 1 (opaque).

**cmap**

[str or `Colormap`, default: *rcParams["image.cmap"]* (default: `'viridis'`)] The Colormap instance or registered colormap name used to map scalar data to colors.

This parameter is ignored if *colors* is set.

**norm**

[str or *Normalize*, optional] The normalization method used to scale scalar data to the [0, 1] range before mapping to colors using *cmap*. By default, a linear scaling is used, mapping the lowest value to 0 and the highest to 1.

If given, this can be one of the following:

- An instance of *Normalize* or one of its subclasses (see *Colormap normalization*).
- A scale name, i.e. one of "linear", "log", "symlog", "logit", etc. For a list of available scales, call *matplotlib.scale.get_scale_names()*. In that case, a suitable *Normalize* subclass is dynamically generated and instantiated.

This parameter is ignored if *colors* is set.

**vmin, vmax**

[float, optional] When using scalar data and no explicit *norm*, *vmin* and *vmax* define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data. It is an error to use *vmin*/*vmax* when a *norm* instance is given (but using a `str` *norm* name together with *vmin*/*vmax* is acceptable).

If *vmin* or *vmax* are not given, the default color scaling is based on *levels*.

This parameter is ignored if *colors* is set.

**origin**

[{*None*, 'upper', 'lower', 'image'}, default: None] Determines the orientation and exact position of *z* by specifying the position of `z[0, 0]`. This is only relevant, if *X*, *Y* are not given.

- *None*: `z[0, 0]` is at X=0, Y=0 in the lower left corner.
- 'lower': `z[0, 0]` is at X=0.5, Y=0.5 in the lower left corner.
- 'upper': `z[0, 0]` is at X=N+0.5, Y=0.5 in the upper left corner.
- 'image': Use the value from *rcParams["image.origin"]* (default: `'upper'`).

**extent**

[(x0, x1, y0, y1), optional] If *origin* is not *None*, then *extent* is interpreted as in *imshow*: it gives the outer pixel boundaries. In this case, the position of z[0, 0] is the center of the pixel, not a corner. If *origin* is *None*, then (*x0*, *y0*) is the position of z[0, 0], and (*x1*, *y1*) is the position of z[-1, -1].

This argument is ignored if *X* and *Y* are specified in the call to contour.

**locator**

[ticker.Locator subclass, optional] The locator is used to determine the contour levels if they are not given explicitly via *levels*. Defaults to *MaxNLocator*.

**extend**

[{'neither', 'both', 'min', 'max'}, default: 'neither'] Determines the `tricontourf`-coloring of values that are outside the *levels* range.

If 'neither', values outside the *levels* range are not colored. If 'min', 'max' or 'both', color the values below, above or below and above the *levels* range.

Values below `min(levels)` and above `max(levels)` are mapped to the under/over values of the *Colormap*. Note that most colormaps do not have dedicated colors for these by default, so that the over and under values are the edge values of the colormap. You may want to set these values explicitly using *Colormap.set_under* and *Colormap.set_over*.

---

**Note:** An existing *TriContourSet* does not get notified if properties of its colormap are changed. Therefore, an explicit call to *ContourSet.changed()* is needed after modifying the colormap. The explicit call can be left out, if a colorbar is assigned to the *TriContourSet* because it internally calls *ContourSet.changed()*.

---

**xunits, yunits**

[registered units, optional] Override axis units by specifying an instance of a *matplotlib.units.ConversionInterface*.

**antialiased**

[bool, optional] Enable antialiasing, overriding the defaults. For filled contours, the default is *True*. For line contours, it is taken from *rcParams["lines.antialiased"]* (default: `True`).

**hatches**

[list[str], optional] A list of crosshatch patterns to use on the filled areas. If None, no hatching will be added to the contour.

## Notes

*tricontourf* fills intervals that are closed at the top; that is, for boundaries *z1* and *z2*, the filled region is:

```
z1 < Z <= z2
```

except for the lowest interval, which is closed on both sides (i.e. it includes the lowest value).

### Examples using `matplotlib.axes.Axes.tricontourf`

- *Contour plot of irregularly spaced data*
- *Tricontour Demo*
- *Tricontour Smooth Delaunay*
- *Tricontour Smooth User*
- *Triinterp Demo*
- *Triangular 3D filled contour plot*
- *tricontourf(x, y, z)*

### Text and annotations

| | |
|---|---|
| `Axes.annotate` | Annotate the point *xy* with text *text*. |
| `Axes.text` | Add text to the Axes. |
| `Axes.table` | Add a table to an `Axes`. |
| `Axes.arrow` | Add an arrow to the Axes. |
| `Axes.inset_axes` | Add a child inset Axes to this existing Axes. |
| `Axes.indicate_inset` | Add an inset indicator to the Axes. |
| `Axes.indicate_inset_zoom` | Add an inset indicator rectangle to the Axes based on the axis limits for an *inset_ax* and draw connectors between *inset_ax* and the rectangle. |
| `Axes.secondary_xaxis` | Add a second x-axis to this `Axes`. |
| `Axes.secondary_yaxis` | Add a second y-axis to this `Axes`. |

## matplotlib.axes.Axes.annotate

Axes.**annotate**(*text*, *xy*, *xytext=None*, *xycoords='data'*, *textcoords=None*, *arrowprops=None*, *annotation_clip=None*, ***kwargs*)

Annotate the point *xy* with text *text*.

In the simplest form, the text is placed at *xy*.

Optionally, the text can be displayed in another position *xytext*. An arrow pointing from the text to the annotated point *xy* can then be added by defining *arrowprops*.

**Parameters**

**text**

[str] The text of the annotation.

**xy**

[(float, float)] The point *(x, y)* to annotate. The coordinate system is determined by *xycoords*.

**xytext**

[(float, float), default: *xy*] The position *(x, y)* to place the text at. The coordinate system is determined by *textcoords*.

**xycoords**

[single or two-tuple of str or *Artist* or *Transform* or callable, default: 'data'] The coordinate system that *xy* is given in. The following types of values are supported:

- One of the following strings:

| Value | Description |
|---|---|
| 'figure points' | Points from the lower left of the figure |
| 'figure pixels' | Pixels from the lower left of the figure |
| 'figure fraction' | Fraction of figure from lower left |
| 'subfigure points' | Points from the lower left of the subfigure |
| 'subfigure pixels' | Pixels from the lower left of the subfigure |
| 'subfigure fraction' | Fraction of subfigure from lower left |
| 'axes points' | Points from lower left corner of the Axes |
| 'axes pixels' | Pixels from lower left corner of the Axes |
| 'axes fraction' | Fraction of Axes from lower left |
| 'data' | Use the coordinate system of the object being annotated (default) |
| 'polar' | *(theta, r)* if not native 'data' coordinates |

  Note that 'subfigure pixels' and 'figure pixels' are the same for the parent figure, so users who want code that is usable in a subfigure can use 'subfigure pixels'.

- An *Artist*: *xy* is interpreted as a fraction of the artist's *Bbox*. E.g. *(0, 0)* would be the lower left corner of the bounding box and *(0.5, 1)* would be the center top of the bounding box.

- A *Transform* to transform *xy* to screen coordinates.

- A function with one of the following signatures:

```
def transform(renderer) -> Bbox
def transform(renderer) -> Transform
```

  where *renderer* is a *RendererBase* subclass.

  The result of the function is interpreted like the *Artist* and *Transform* cases above.

- A tuple *(xcoords, ycoords)* specifying separate coordinate systems for *x* and *y*. *xcoords* and *ycoords* must each be of one of the above described types.

See *Advanced annotation* for more details.

**textcoords**

[single or two-tuple of str or `Artist` or `Transform` or callable, default: value of *xycoords*] The coordinate system that *xytext* is given in.

All *xycoords* values are valid as well as the following strings:

| Value | Description |
|---|---|
| 'offset points' | Offset, in points, from the *xy* value |
| 'offset pixels' | Offset, in pixels, from the *xy* value |
| 'offset fontsize' | Offset, relative to fontsize, from the *xy* value |

**arrowprops**

[dict, optional] The properties used to draw a `FancyArrowPatch` arrow between the positions *xy* and *xytext*. Defaults to None, i.e. no arrow is drawn.

For historical reasons there are two different ways to specify arrows, "simple" and "fancy":

**Simple arrow:**

If *arrowprops* does not contain the key 'arrowstyle' the allowed keys are:

| Key | Description |
|---|---|
| width | The width of the arrow in points |
| headwidth | The width of the base of the arrow head in points |
| headlength | The length of the arrow head in points |
| shrink | Fraction of total length to shrink from both ends |
| ? | Any `FancyArrowPatch` property |

The arrow is attached to the edge of the text box, the exact position (corners or centers) depending on where it's pointing to.

**Fancy arrow:**

This is used if 'arrowstyle' is provided in the *arrowprops*.

Valid keys are the following `FancyArrowPatch` parameters:

| Key | Description |
|---|---|
| arrowstyle | The arrow style |
| connectionstyle | The connection style |
| relpos | See below; default is (0.5, 0.5) |
| patchA | Default is bounding box of the text |
| patchB | Default is None |
| shrinkA | In points. Default is 2 points |
| shrinkB | In points. Default is 2 points |
| mutation_scale | Default is text size (in points) |
| mutation_aspect | Default is 1 |
| ? | Any `FancyArrowPatch` property |

The exact starting point position of the arrow is defined by *relpos*. It's a tuple of relative coordinates of the text box, where (0, 0) is the lower left corner and (1, 1) is the upper right corner. Values <0 and >1 are supported and specify points outside the text box. By default (0.5, 0.5), so the starting point is centered in the text box.

**annotation_clip**

[bool or None, default: None] Whether to clip (i.e. not draw) the annotation when the annotation point *xy* is outside the Axes area.

- If *True*, the annotation will be clipped when *xy* is outside the Axes.
- If *False*, the annotation will always be drawn.
- If *None*, the annotation will be clipped when *xy* is outside the Axes and *xycoords* is 'data'.

****kwargs**

Additional kwargs are passed to `Text`.

**Returns**

`Annotation`

**See also:**

*Annotations*

### Examples using `matplotlib.axes.Axes.annotate`

- *Broken Barh*
- *Hat graph*
- *Creating a timeline with lines, dates, and text*
- *Combining two subplots using subplots and GridSpec*
- *Labeling a pie and a donut*
- *Scale invariant angle label*
- *Annotate Transform*
- *Annotating a plot*
- *Annotating Plots*
- *Annotation Polar*
- *Annotation arrow style reference*
- *Labelling subplots*
- *Concatenating text objects with different properties*
- *Rendering math equations using TeX*
- *Text Commands*
- *Mmh Donuts!!!*
- *Simple Axis Pad*
- *XKCD*
- *Patheffect Demo*
- *Ishikawa Diagram*
- *Annotation with units*
- *Annotate Explain*
- *Connection styles for annotations*
- *Simple Annotate01*
- *Quick start guide*
- *Faster rendering by using blitting*
- *Transformations Tutorial*
- *Text in Matplotlib*
- *Annotations*

## matplotlib.axes.Axes.text

Axes.**text**(*x*, *y*, *s*, *fontdict=None*, ***kwargs*)

Add text to the Axes.

Add the text *s* to the Axes at location *x*, *y* in data coordinates, with a default `horizontalalignment` on the `left` and `verticalalignment` at the `baseline`. See *Text alignment*.

**Parameters**

**x, y**

[float] The position to place the text. By default, this is in data coordinates. The coordinate system can be changed using the *transform* parameter.

**s**

[str] The text.

**fontdict**

[dict, default: None]

**Discouraged**

The use of *fontdict* is discouraged. Parameters should be passed as individual keyword arguments or using dictionary-unpacking `text(..., **fontdict)`.

A dictionary to override the default text properties. If fontdict is None, the defaults are determined by *rcParams*.

**Returns**

***Text***

The created *Text* instance.

**Other Parameters**

****kwargs**

[*Text* properties.] Other miscellaneous text parameters.

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array a |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* | bool |
| *backgroundcolor* | *color* |

Table 26 –

| Property | Description |
|---|---|
| bbox | dict with properties for patches.FancyBboxPa |
| clip_box | unknown |
| clip_on | unknown |
| clip_path | unknown |
| color or c | color |
| figure | Figure |
| fontfamily or family or fontname | {FONTNAME, 'serif', 'sans-serif', 'cursive', 'fantasy' |
| fontproperties or font or font_properties | font_manager.FontProperties or str or |
| fontsize or size | float or {'xx-small', 'x-small', 'small', 'medium', 'large |
| fontstretch or stretch | {a numeric value in range 0-1000, 'ultra-condensed', |
| fontstyle or style | {'normal', 'italic', 'oblique'} |
| fontvariant or variant | {'normal', 'small-caps'} |
| fontweight or weight | {a numeric value in range 0-1000, 'ultralight', 'light', |
| gid | str |
| horizontalalignment or ha | {'left', 'center', 'right'} |
| in_layout | bool |
| label | object |
| linespacing | float (multiple of font size) |
| math_fontfamily | str |
| mouseover | bool |
| multialignment or ma | {'left', 'right', 'center'} |
| parse_math | bool |
| path_effects | list of AbstractPathEffect |
| picker | None or bool or float or callable |
| position | (float, float) |
| rasterized | bool |
| rotation | float or {'vertical', 'horizontal'} |
| rotation_mode | {None, 'default', 'anchor'} |
| sketch_params | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| text | object |
| transform | Transform |
| transform_rotates_text | bool |
| url | str |
| usetex | bool, default: rcParams["text.usetex"] (de |
| verticalalignment or va | {'baseline', 'bottom', 'center', 'center_baseline', 'top'} |
| visible | bool |
| wrap | bool |
| x | float |
| y | float |
| zorder | float |

### Examples

Individual keyword arguments can be used to override any given parameter:

```
>>> text(x, y, s, fontsize=12)
```

The default transform specifies that text is in data coords, alternatively, you can specify text in axis coords ((0, 0) is lower-left and (1, 1) is upper-right). The example below places text in the center of the Axes:

```
>>> text(0.5, 0.5, 'matplotlib', horizontalalignment='center',
...      verticalalignment='center', transform=ax.transAxes)
```

You can put a rectangular box around the text instance (e.g., to set a background color) by using the keyword *bbox*. *bbox* is a dictionary of `Rectangle` properties. For example:

```
>>> text(x, y, s, bbox=dict(facecolor='red', alpha=0.5))
```

## Examples using `matplotlib.axes.Axes.text`

- *Marker reference*
- *BboxImage Demo*
- *Creating annotated heatmaps*
- *Boxplots*
- *Accented text*
- *Angle annotations on bracket arrows*
- *Arrow Demo*
- *Text Rotation Mode*
- *Annotation arrow style reference*
- *Labelling subplots*
- *Mathtext*
- *Math fontfamily*
- *Multiline*
- *Placing text boxes*
- *Concatenating text objects with different properties*
- *Rendering math equations using TeX*
- *Text alignment*
- *Text Commands*

- *Text Rotation Relative To Line*
- *Usetex Baseline Test*
- *Text watermark*
- *List of named colors*
- *Drawing fancy boxes*
- *Hatch style reference*
- *Anatomy of a figure*
- *Integral as the area under a curve*
- *Shaded & power normalized rendering*
- *Pan/zoom events of overlapping axes*
- *Stock prices over 32 years*
- *The double pendulum problem*
- *MATPLOTLIB UNCHAINED*
- *Cross-hair cursor*
- *Data browser*
- *Pick event demo 2*
- *Packed-bubble chart*
- *Rasterization for vector graphics*
- *Text annotations in 3D*
- *Anscombe's quartet*
- *Ishikawa Diagram*
- *Date tick locators and formatters*
- *Mouse Cursor*
- *Annotate Explain*
- *Annotate Text Arrow*
- *Connection styles for annotations*
- *Custom box styles*
- *PGF fonts*
- *PGF texsystem*
- *Simple Annotate01*
- *The Lifecycle of a Plot*
- *Quick start guide*

- *Path Tutorial*
- *Transformations Tutorial*
- *Arranging multiple Axes in a Figure*
- *Axis ticks*
- *Plotting dates and strings*
- *Specifying colors*
- *Choosing Colormaps in Matplotlib*
- *Text in Matplotlib*
- *Text properties and layout*
- *Annotations*

## matplotlib.axes.Axes.table

Axes.**table**(*cellText=None*, *cellColours=None*, *cellLoc='right'*, *colWidths=None*, *rowLabels=None*, *rowColours=None*, *rowLoc='left'*, *colLabels=None*, *colColours=None*, *colLoc='center'*, *loc='bottom'*, *bbox=None*, *edges='closed'*, ***kwargs*)

Add a table to an `Axes`.

At least one of *cellText* or *cellColours* must be specified. These parameters must be 2D lists, in which the outer lists define the rows and the inner list define the column values per row. Each row must have the same number of elements.

The table can optionally have row and column headers, which are configured using *rowLabels*, *rowColours*, *rowLoc* and *colLabels*, *colColours*, *colLoc* respectively.

For finer grained control over tables, use the `Table` class and add it to the Axes with `Axes.add_table`.

**Parameters**

**cellText**

[2D list of str, optional] The texts to place into the table cells.

*Note*: Line breaks in the strings are currently not accounted for and will result in the text exceeding the cell boundaries.

**cellColours**

[2D list of *color*, optional] The background colors of the cells.

**cellLoc**

[{'right', 'center', 'left'}] The alignment of the text within the cells.

**colWidths**

[list of float, optional] The column widths in units of the axes. If not given, all columns will have a width of *1 / ncols*.

**rowLabels**

[list of str, optional] The text of the row header cells.

**rowColours**

[list of *color*, optional] The colors of the row header cells.

**rowLoc**

[{'left', 'center', 'right'}] The text alignment of the row header cells.

**colLabels**

[list of str, optional] The text of the column header cells.

**colColours**

[list of *color*, optional] The colors of the column header cells.

**colLoc**

[{'center', 'left', 'right'}] The text alignment of the column header cells.

**loc**

[str, default: 'bottom'] The position of the cell with respect to *ax*. This must be one of the `codes`.

**bbox**

[`Bbox` or [xmin, ymin, width, height], optional] A bounding box to draw the table into. If this is not *None*, this overrides *loc*.

**edges**

[{'closed', 'open', 'horizontal', 'vertical'} or substring of 'BRTL'] The cell edges to be drawn with a line. See also `visible_edges`.

**Returns**

***Table***

The created table.

**Other Parameters**

****kwargs**

`Table` properties.

| Property | Description |
|---|---|
| agg_fil | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image |
| alpha | scalar or None |
| animated | bool |
| clip_bo | BboxBase or None |
| clip_on | bool |
| clip_pa | Patch or (Path, Transform) or None |
| figure | Figure |
| fontsize | float |
| gid | str |
| in_layo | bool |
| label | object |
| mouseov | bool |
| path_ef | list of AbstractPathEffect |
| picker | None or bool or float or callable |
| rasterized | bool |
| sketch_ | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| transform | Transform |
| url | str |
| visible | bool |
| zorder | float |

## matplotlib.axes.Axes.arrow

Axes.**arrow**(*x*, *y*, *dx*, *dy*, ***kwargs*)

Add an arrow to the Axes.

This draws an arrow from `(x, y)` to `(x+dx, y+dy)`.

**Parameters**

**x, y**

[float] The x and y coordinates of the arrow base.

**dx, dy**

[float] The length of the arrow along x and y direction.

**width**

[float, default: 0.001] Width of full arrow tail.

**length_includes_head**

[bool, default: False] True if head is to be counted in calculating the length.

**head_width**

[float or None, default: 3*width] Total width of the full arrow head.

**head_length**

[float or None, default: 1.5*head_width] Length of arrow head.

**shape**

[{'full', 'left', 'right'}, default: 'full'] Draw the left-half, right-half, or full arrow.

**overhang**

[float, default: 0] Fraction that the arrow is swept back (0 overhang means triangular shape). Can be negative or greater than one.

**head_starts_at_zero**

[bool, default: False] If True, the head starts being drawn at coordinate 0 instead of ending at coordinate 0.

****kwargs**

*Patch* properties:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and return |
| *alpha* | unknown |
| *animated* | bool |
| *antialiased* or aa | bool or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* | *color* |
| *edgecolor* or ec | *color* or None |
| *facecolor* or fc | *color* or None |
| *figure* | *Figure* |
| *fill* | bool |
| *gid* | str |
| *hatch* | {'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'} |

Table 27 – continued from previc

| Property | Description |
|---|---|
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float or None |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *zorder* | float |

**Returns**

***FancyArrow***

The created *FancyArrow* object.

## Notes

The resulting arrow is affected by the Axes aspect ratio and limits. This may produce an arrow whose head is not square with its stem. To create an arrow whose head is square with its stem, use *annotate()* for example:

```
>>> ax.annotate("", xy=(0.5, 0.5), xytext=(0, 0),
...             arrowprops=dict(arrowstyle="->"))
```

### Examples using `matplotlib.axes.Axes.arrow`

- *Arrow Demo*

## matplotlib.axes.Axes.inset_axes

`Axes.`**`inset_axes`**(*bounds*, *, *transform=None*, *zorder=5*, ***kwargs*)

Add a child inset Axes to this existing Axes.

**Parameters**

**bounds**

[[x0, y0, width, height]] Lower-left corner of inset Axes, and its width and height.

**transform**

[*Transform*] Defaults to `ax.transAxes`, i.e. the units of *rect* are in Axes-relative coordinates.

**projection**

[{None, 'aitoff', 'hammer', 'lambert', 'mollweide', 'polar', 'rectilinear', str}, optional] The projection type of the inset *Axes*. *str* is the name of a custom projection, see *projections*. The default None results in a 'rectilinear' projection.

**polar**

[bool, default: False] If True, equivalent to projection='polar'.

**axes_class**

[subclass type of *Axes*, optional] The *axes.Axes* subclass that is instantiated. This parameter is incompatible with *projection* and *polar*. See *axisartist* for examples.

**zorder**

[number] Defaults to 5 (same as *Axes.legend*). Adjust higher or lower to change whether it is above or below data plotted on the parent Axes.

****kwargs**

Other keyword arguments are passed on to the inset Axes class.

**Returns**

**ax**

The created *Axes* instance.

### Examples

This example makes two inset Axes, the first is in Axes-relative coordinates, and the second in data-coordinates:

```
fig, ax = plt.subplots()
ax.plot(range(10))
axin1 = ax.inset_axes([0.8, 0.1, 0.15, 0.15])
axin2 = ax.inset_axes(
        [5, 7, 2.3, 2.3], transform=ax.transData)
```

## Examples using `matplotlib.axes.Axes.inset_axes`

- *Scatter plot with histograms*
- *Zoom region inset Axes*
- *Check buttons*
- *Placing colorbars*

## matplotlib.axes.Axes.indicate_inset

Axes.**indicate_inset**(*bounds*, *inset_ax=None*, *, *transform=None*, *facecolor='none'*, *edgecolor='0.5'*, *alpha=0.5*, *zorder=4.99*, ***kwargs*)

Add an inset indicator to the Axes. This is a rectangle on the plot at the position indicated by *bounds* that optionally has lines that connect the rectangle to an inset Axes (*Axes.inset_axes*).

**Parameters**

**bounds**

[[x0, y0, width, height]] Lower-left corner of rectangle to be marked, and its width and height.

**inset_ax**

[*Axes*] An optional inset Axes to draw connecting lines to. Two lines are drawn connecting the indicator box to the inset Axes on corners chosen so as to not overlap with the indicator box.

**transform**

[*Transform*] Transform for the rectangle coordinates. Defaults to `ax.transAxes`, i.e. the units of *rect* are in Axes-relative coordinates.

**facecolor**

[*color*, default: 'none'] Facecolor of the rectangle.

**edgecolor**

[*color*, default: '0.5'] Color of the rectangle and color of the connecting lines.

**alpha**

[float, default: 0.5] Transparency of the rectangle and connector lines.

**zorder**

[float, default: 4.99] Drawing order of the rectangle and connector lines. The default, 4.99, is just below the default level of inset Axes.

****kwargs**

Other keyword arguments are passed on to the *Rectangle* patch:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and retur |
| *alpha* | scalar or None |
| *angle* | unknown |
| *animated* | bool |
| *antialiased* or aa | bool or None |
| *bounds* | (left, bottom, width, height) |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* | *color* |
| *edgecolor* or ec | *color* or None |
| *facecolor* or fc | *color* or None |
| *figure* | *Figure* |
| *fill* | bool |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *height* | unknown |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float or None |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |

Table 28 – continued from previc

| Property | Description |
|---|---|
| visible | bool |
| width | unknown |
| x | unknown |
| xy | (float, float) |
| y | unknown |
| zorder | float |

**Returns**

**rectangle_patch**

[*patches.Rectangle*] The indicator frame.

**connector_lines**

[4-tuple of *patches.ConnectionPatch*] The four connector lines connecting to (lower_left, upper_left, lower_right upper_right) corners of *inset_ax*. Two lines are set with visibility to *False*, but the user can set the visibility to True if the automatic choice is not deemed correct.

**Warning:** This method is experimental as of 3.0, and the API may change.

## matplotlib.axes.Axes.indicate_inset_zoom

Axes.**indicate_inset_zoom**(*inset_ax*, ***kwargs*)

Add an inset indicator rectangle to the Axes based on the axis limits for an *inset_ax* and draw connectors between *inset_ax* and the rectangle.

**Parameters**

**inset_ax**

[*Axes*] Inset Axes to draw connecting lines to. Two lines are drawn connecting the indicator box to the inset Axes on corners chosen so as to not overlap with the indicator box.

****kwargs**

Other keyword arguments are passed on to *Axes.indicate_inset*

**Returns**

**rectangle_patch**

[*patches.Rectangle*] Rectangle artist.

**connector_lines**

[4-tuple of *patches.ConnectionPatch*] Each of four connector lines coming from the rectangle drawn on this axis, in the order lower left, upper left, lower right, upper right. Two are set with visibility to *False*, but the user can set the visibility to *True* if the automatic choice is not deemed correct.

**Warning:** This method is experimental as of 3.0, and the API may change.

### Examples using `matplotlib.axes.Axes.indicate_inset_zoom`

- *Zoom region inset Axes*

## matplotlib.axes.Axes.secondary_xaxis

`Axes.`**`secondary_xaxis`**(*location*, *, *functions=None*, *transform=None*, ***kwargs*)

Add a second x-axis to this *Axes*.

For example if we want to have a second scale for the data plotted on the xaxis.

**Parameters**

**location**

[{'top', 'bottom', 'left', 'right'} or float] The position to put the secondary axis. Strings can be 'top' or 'bottom' for orientation='x' and 'right' or 'left' for orientation='y'. A float indicates the relative position on the parent Axes to put the new Axes, 0.0 being the bottom (or left) and 1.0 being the top (or right).

**functions**

[2-tuple of func, or Transform with an inverse] If a 2-tuple of functions, the user specifies the transform function and its inverse. i.e. `functions=(lambda x: 2 / x, lambda x: 2 / x)` would be an reciprocal transform with a factor of 2. Both functions must accept numpy arrays as input.

The user can also directly supply a subclass of *transforms.Transform* so long as it has an inverse.

See *Secondary Axis* for examples of making these conversions.

**transform**

[*Transform*, optional] If specified, *location* will be placed relative to this transform (in the direction of the axis) rather than the parent's axis. i.e. a secondary x-axis will use the provided y transform and the x transform of the parent.

New in version 3.9.

**Returns**

**ax**

[axes._secondary_axes.SecondaryAxis]

**Other Parameters**

****kwargs**

[*Axes* properties.] Other miscellaneous Axes parameters.

**Warning:** This method is experimental as of 3.1, and the API may change.

### Examples

The main axis shows frequency, and the secondary axis shows period.

To add a secondary axis relative to your data, you can pass a transform to the new axis.

### Examples using `matplotlib.axes.Axes.secondary_xaxis`

- *Secondary Axis*
- *Multilevel (nested) ticks*
- *Quick start guide*

## matplotlib.axes.Axes.secondary_yaxis

Axes.**secondary_yaxis**(*location*, *, *functions=None*, *transform=None*, ***kwargs*)

Add a second y-axis to this *Axes*.

For example if we want to have a second scale for the data plotted on the yaxis.

**Parameters**

**location**

[{'top', 'bottom', 'left', 'right'} or float] The position to put the secondary axis. Strings can be 'top' or 'bottom' for orientation='x' and 'right' or 'left' for orientation='y'. A float indicates the relative position on the parent Axes to put the new Axes, 0.0 being the bottom (or left) and 1.0 being the top (or right).

**functions**

[2-tuple of func, or Transform with an inverse] If a 2-tuple of functions, the user specifies the transform function and its inverse. i.e. `functions=(lambda x: 2 / x, lambda x: 2 / x)` would be an reciprocal transform with a factor of 2. Both functions must accept numpy arrays as input.

The user can also directly supply a subclass of *transforms.Transform* so long as it has an inverse.

See *Secondary Axis* for examples of making these conversions.

**transform**

[*Transform*, optional] If specified, *location* will be placed relative to this transform (in the direction of the axis) rather than the parent's axis. i.e. a secondary x-axis will use the provided y transform and the x transform of the parent.

New in version 3.9.

**Returns**

**ax**

[axes._secondary_axes.SecondaryAxis]

**Other Parameters**

****kwargs**

[*Axes* properties.] Other miscellaneous Axes parameters.

**Warning:** This method is experimental as of 3.1, and the API may change.

## Examples

Add a secondary Axes that converts from radians to degrees

To add a secondary axis relative to your data, you can pass a transform to the new axis.

### Examples using `matplotlib.axes.Axes.secondary_yaxis`

- *Secondary Axis*

### Vector fields

| | |
|---|---|
| *Axes.barbs* | Plot a 2D field of wind barbs. |
| *Axes.quiver* | Plot a 2D field of arrows. |
| *Axes.quiverkey* | Add a key to a quiver plot. |
| *Axes.streamplot* | Draw streamlines of a vector flow. |

## matplotlib.axes.Axes.barbs

Axes.**barbs**(**args*, *data=None*, ***kwargs*)

Plot a 2D field of wind barbs.

Call signature:

```
barbs([X, Y], U, V, [C], **kwargs)
```

Where *X*, *Y* define the barb locations, *U*, *V* define the barb directions, and *C* optionally sets the color.

All arguments may be 1D or 2D. *U*, *V*, *C* may be masked arrays, but masked *X*, *Y* are not supported at present.

Barbs are traditionally used in meteorology as a way to plot the speed and direction of wind observations, but can technically be used to plot any two dimensional vector quantity. As opposed to arrows, which give vector magnitude by the length of the arrow, the barbs give more quantitative information about the vector magnitude by putting slanted lines or a triangle for various increments in magnitude, as show schematically below:

```
:                    /\    \
:                   /  \    \
:                  /    \    \    \
:                 /      \    \    \
:                ------------------------------
```

The largest increment is given by a triangle (or "flag"). After those come full lines (barbs). The smallest increment is a half line. There is only, of course, ever at most 1 half line. If the magnitude is small and only needs a single half-line and no full lines or triangles, the half-line is offset from the end of the barb so that it can be easily distinguished from barbs with a single full line. The magnitude for the barb shown above would nominally be 65, using the standard increments of 50, 10, and 5.

See also https://en.wikipedia.org/wiki/Wind_barb.

**Parameters**

**X, Y**

[1D or 2D array-like, optional] The x and y coordinates of the barb locations. See *pivot* for how the barbs are drawn to the x, y positions.

If not given, they will be generated as a uniform integer meshgrid based on the dimensions of *U* and *V*.

If *X* and *Y* are 1D but *U*, *V* are 2D, *X*, *Y* are expanded to 2D using `X, Y = np.meshgrid(X, Y)`. In this case `len(X)` and `len(Y)` must match the column and row dimensions of *U* and *V*.

**U, V**

[1D or 2D array-like] The x and y components of the barb shaft.

**C**

[1D or 2D array-like, optional] Numeric data that defines the barb colors by colormapping via *norm* and *cmap*.

This does not support explicit colors. If you want to set colors directly, use *barbcolor* instead.

**length**

[float, default: 7] Length of the barb in points; the other parts of the barb are scaled against this.

**pivot**

[{'tip', 'middle'} or float, default: 'tip'] The part of the arrow that is anchored to the *X*, *Y* grid. The barb rotates about this point. This can also be a number, which shifts the start of the barb that many points away from grid point.

**barbcolor**

[*color* or color sequence] The color of all parts of the barb except for the flags. This parameter is analogous to the *edgecolor* parameter for polygons, which can be used instead. However this parameter will override facecolor.

**flagcolor**

[*color* or color sequence] The color of any flags on the barb. This parameter is analogous to the *facecolor* parameter for polygons, which can be used instead. However, this parameter will override facecolor. If this is not set (and *C* has not either) then *flagcolor* will be set to match *barbcolor* so that the barb has a uniform color. If *C* has been set, *flagcolor* has no effect.

**sizes**

[dict, optional] A dictionary of coefficients specifying the ratio of a given feature to the length of the barb. Only those values one wishes to override need to be included. These features include:

- 'spacing' - space between features (flags, full/half barbs)
- 'height' - height (distance from shaft to top) of a flag or full barb

- 'width' - width of a flag, twice the width of a full barb
- 'emptybarb' - radius of the circle used for low magnitudes

**fill_empty**

[bool, default: False] Whether the empty barbs (circles) that are drawn should be filled with the flag color. If they are not filled, the center is transparent.

**rounding**

[bool, default: True] Whether the vector magnitude should be rounded when allocating barb components. If True, the magnitude is rounded to the nearest multiple of the half-barb increment. If False, the magnitude is simply truncated to the next lowest multiple.

**barb_increments**

[dict, optional] A dictionary of increments specifying values to associate with different parts of the barb. Only those values one wishes to override need to be included.

- 'half' - half barbs (Default is 5)
- 'full' - full barbs (Default is 10)
- 'flag' - flags (default is 50)

**flip_barb**

[bool or array-like of bool, default: False] Whether the lines and flags should point opposite to normal. Normal behavior is for the barbs and lines to point right (comes from wind barbs having these features point towards low pressure in the Northern Hemisphere).

A single value is applied to all barbs. Individual barbs can be flipped by passing a bool array of the same size as *U* and *V*.

**Returns**

**barbs**

[*Barbs*]

**Other Parameters**

**data**

[indexable object, optional] If given, all parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception).

****kwargs**

The barbs can further be customized using *PolyCollection* keyword arguments:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a d |
| *alpha* | array-like or scalar or None |
| *animated* | bool |
| *antialiased* or aa or antialiaseds | bool or list of bools |
| *array* | array-like or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clim* | (vmin: float, vmax: float) |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *cmap* | *Colormap* or str or None |
| *color* | *color* or list of RGBA tuples |
| *edgecolor* or ec or edgecolors | *color* or list of *color* or 'face' |
| *facecolor* or facecolors or fc | *color* or list of *color* |
| *figure* | *Figure* |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or dashes or linestyles or ls | str or tuple or list thereof |
| *linewidth* or linewidths or lw | float or list of floats |
| *mouseover* | bool |
| *norm* | *Normalize* or str or None |
| *offset_transform* or transOffset | *Transform* |
| *offsets* | (N, 2) or (2,) array-like |
| *path_effects* | list of *AbstractPathEffect* |
| *paths* | list of array-like |
| *picker* | None or bool or float or callable |
| *pickradius* | float |
| *rasterized* | bool |
| sizes | numpy.ndarray or None |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *urls* | list of str or None |
| *verts* | list of array-like |
| *verts_and_codes* | unknown |
| *visible* | bool |
| *zorder* | float |

### Examples using `matplotlib.axes.Axes.barbs`

- *Wind Barbs*
- *barbs(X, Y, U, V)*

## matplotlib.axes.Axes.quiver

`Axes.`**`quiver`**(**args*, *data=None*, ***kwargs*)

Plot a 2D field of arrows.

Call signature:

```
quiver([X, Y], U, V, [C], **kwargs)
```

*X*, *Y* define the arrow locations, *U*, *V* define the arrow directions, and *C* optionally sets the color.

**Arrow length**

The default settings auto-scales the length of the arrows to a reasonable size. To change this behavior see the *scale* and *scale_units* parameters.

**Arrow shape**

The arrow shape is determined by *width*, *headwidth*, *headlength* and *headaxislength*. See the notes below.

**Arrow styling**

Each arrow is internally represented by a filled polygon with a default edge linewidth of 0. As a result, an arrow is rather a filled area, not a line with a head, and *PolyCollection* properties like *linewidth*, *edgecolor*, *facecolor*, etc. act accordingly.

**Parameters**

**X, Y**

[1D or 2D array-like, optional] The x and y coordinates of the arrow locations.

If not given, they will be generated as a uniform integer meshgrid based on the dimensions of *U* and *V*.

If *X* and *Y* are 1D but *U*, *V* are 2D, *X*, *Y* are expanded to 2D using `X, Y = np.meshgrid(X, Y)`. In this case `len(X)` and `len(Y)` must match the column and row dimensions of *U* and *V*.

**U, V**

[1D or 2D array-like] The x and y direction components of the arrow vectors. The interpretation of these components (in data or in screen space) depends on *angles*.

*U* and *V* must have the same number of elements, matching the number of arrow locations in *X*, *Y*. *U* and *V* may be masked. Locations masked in any of *U*, *V*, and *C* will not be drawn.

**C**

[1D or 2D array-like, optional] Numeric data that defines the arrow colors by colormapping via *norm* and *cmap*.

This does not support explicit colors. If you want to set colors directly, use *color* instead. The size of *C* must match the number of arrow locations.

**angles**

[{'uv', 'xy'} or array-like, default: 'uv'] Method for determining the angle of the arrows.

- 'uv': Arrow direction in screen coordinates. Use this if the arrows symbolize a quantity that is not based on *X*, *Y* data coordinates.

  If $U == V$ the orientation of the arrow on the plot is 45 degrees counter-clockwise from the horizontal axis (positive to the right).

- 'xy': Arrow direction in data coordinates, i.e. the arrows point from (x, y) to (x+u, y+v). Use this e.g. for plotting a gradient field.

- Arbitrary angles may be specified explicitly as an array of values in degrees, counter-clockwise from the horizontal axis.

  In this case *U*, *V* is only used to determine the length of the arrows.

Note: inverting a data axis will correspondingly invert the arrows only with `angles='xy'`.

**pivot**

[{'tail', 'mid', 'middle', 'tip'}, default: 'tail'] The part of the arrow that is anchored to the *X*, *Y* grid. The arrow rotates about this point.

'mid' is a synonym for 'middle'.

**scale**

[float, optional] Scales the length of the arrow inversely.

Number of data units per arrow length unit, e.g., m/s per plot width; a smaller scale parameter makes the arrow longer. Default is *None*.

If *None*, a simple autoscaling algorithm is used, based on the average vector length and the number of vectors. The arrow length unit is given by the *scale_units* parameter.

**scale_units**

[{'width', 'height', 'dots', 'inches', 'x', 'y', 'xy'}, optional] If the *scale* kwarg is *None*, the arrow length unit. Default is *None*.

e.g. *scale_units* is 'inches', *scale* is 2.0, and `(u, v) = (1, 0)`, then the vector will be 0.5 inches long.

If *scale_units* is 'width' or 'height', then the vector will be half the width/height of the axes.

If *scale_units* is 'x' then the vector will be 0.5 x-axis units. To plot vectors in the x-y plane, with u and v having the same units as x and y, use `angles='xy', scale_units='xy', scale=1`.

**units**

[{'width', 'height', 'dots', 'inches', 'x', 'y', 'xy'}, default: 'width'] Affects the arrow size (except for the length). In particular, the shaft *width* is measured in multiples of this unit.

Supported values are:

- 'width', 'height': The width or height of the Axes.
- 'dots', 'inches': Pixels or inches based on the figure dpi.
- 'x', 'y', 'xy': $X$, $Y$ or $\sqrt{X^2 + Y^2}$ in data units.

The following table summarizes how these values affect the visible arrow size under zooming and figure size changes:

| units | zoom | figure size change |
|---|---|---|
| 'x', 'y', 'xy' | arrow size scales | — |
| 'width', 'height' | — | arrow size scales |
| 'dots', 'inches' | — | — |

**width**

[float, optional] Shaft width in arrow units. All head parameters are relative to *width*.

The default depends on choice of *units* above, and number of vectors; a typical starting value is about 0.005 times the width of the plot.

**headwidth**

[float, default: 3] Head width as multiple of shaft *width*. See the notes below.

**headlength**

[float, default: 5] Head length as multiple of shaft *width*. See the notes below.

**headaxislength**

[float, default: 4.5] Head length at shaft intersection as multiple of shaft *width*. See the notes below.

**minshaft**

[float, default: 1] Length below which arrow scales, in units of head length. Do not set this to less than 1, or small arrows will look terrible!

**minlength**

[float, default: 1] Minimum length as a multiple of shaft width; if an arrow length is less than this, plot a dot (hexagon) of this diameter instead.

**color**

[*color* or list *color*, optional] Explicit color(s) for the arrows. If *C* has been set, *color* has no effect.

This is a synonym for the `PolyCollection` *facecolor* parameter.

**Returns**

***Quiver***

**Other Parameters**

**data**

[indexable object, optional] If given, all parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception).

****kwargs**

[`PolyCollection` properties, optional] All other keyword arguments are passed on to `PolyCollection`:

| Property | Description |
|---|---|
| `agg_filter` | a filter function, which takes a (m, n, 3) float array and a d |
| `alpha` | array-like or scalar or None |
| `animated` | bool |
| `antialiased` or aa or antialiaseds | bool or list of bools |
| `array` | array-like or None |
| `capstyle` | `CapStyle` or {'butt', 'projecting', 'round'} |
| `clim` | (vmin: float, vmax: float) |
| `clip_box` | `BboxBase` or None |
| `clip_on` | bool |
| `clip_path` | Patch or (Path, Transform) or None |
| `cmap` | `Colormap` or str or None |
| `color` | *color* or list of RGBA tuples |
| `edgecolor` or ec or edgecolors | *color* or list of *color* or 'face' |
| `facecolor` or facecolors or fc | *color* or list of *color* |
| `figure` | `Figure` |
| `gid` | str |
| `hatch` | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| `in_layout` | bool |
| `joinstyle` | `JoinStyle` or {'miter', 'round', 'bevel'} |
| `label` | object |
| `linestyle` or dashes or linestyles or ls | str or tuple or list thereof |
| `linewidth` or linewidths or lw | float or list of floats |
| `mouseover` | bool |
| `norm` | `Normalize` or str or None |

Table 30 – continued fro

| Property | Description |
|---|---|
| *offset_transform* or transOffset | *Transform* |
| *offsets* | (N, 2) or (2,) array-like |
| *path_effects* | list of *AbstractPathEffect* |
| *paths* | list of array-like |
| *picker* | None or bool or float or callable |
| *pickradius* | float |
| *rasterized* | bool |
| sizes | numpy.ndarray or None |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *urls* | list of str or None |
| *verts* | list of array-like |
| *verts_and_codes* | unknown |
| *visible* | bool |
| *zorder* | float |

**See also:**

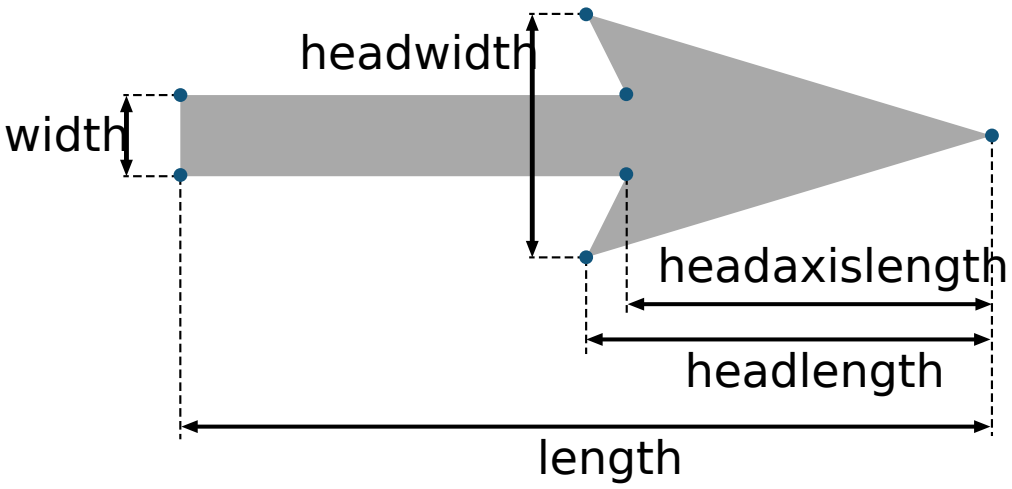
***Axes.quiverkey***

Add a key to a quiver plot.

## Notes

### Arrow shape

The arrow is drawn as a polygon using the nodes as shown below. The values *headwidth*, *headlength*, and *headaxislength* are in units of *width*.



The defaults give a slightly swept-back arrow. Here are some guidelines how to get other head shapes:

- To make the head a triangle, make *headaxislength* the same as *headlength*.
- To make the arrow more pointed, reduce *headwidth* or increase *headlength* and *headaxislength*.
- To make the head smaller relative to the shaft, scale down all the head parameters proportionally.
- To remove the head completely, set all *head* parameters to 0.
- To get a diamond-shaped head, make *headaxislength* larger than *headlength*.
- Warning: For *headaxislength* < (*headlength* / *headwidth*), the "headaxis" nodes (i.e. the ones connecting the head with the shaft) will protrude out of the head in forward direction so that the arrow head looks broken.

### Examples using `matplotlib.axes.Axes.quiver`

- *Advanced quiver and quiverkey functions*
- *Quiver Simple Demo*
- *Trigradient Demo*
- *3D quiver plot*
- *quiver(X, Y, U, V)*

### matplotlib.axes.Axes.quiverkey

`Axes.`**`quiverkey`**(*Q*, *X*, *Y*, *U*, *label*, ***kwargs*)

Add a key to a quiver plot.

The positioning of the key depends on *X*, *Y*, *coordinates*, and *labelpos*. If *labelpos* is 'N' or 'S', *X*, *Y* give the position of the middle of the key arrow. If *labelpos* is 'E', *X*, *Y* positions the head, and if *labelpos* is 'W', *X*, *Y* positions the tail; in either of these two cases, *X*, *Y* is somewhere in the middle of the arrow+label key object.

**Parameters**

**Q**

[*Quiver*] A *Quiver* object as returned by a call to *quiver()*.

**X, Y**

[float] The location of the key.

**U**

[float] The length of the key.

**label**

[str] The key label (e.g., length and units of the key).

**angle**

[float, default: 0] The angle of the key arrow, in degrees anti-clockwise from the horizontal axis.

**coordinates**

[{'axes', 'figure', 'data', 'inches'}, default: 'axes'] Coordinate system and units for *X*, *Y*: 'axes' and 'figure' are normalized coordinate systems with (0, 0) in the lower left and (1, 1) in the upper right; 'data' are the axes data coordinates (used for the locations of the vectors in the quiver plot itself); 'inches' is position in the figure in inches, with (0, 0) at the lower left corner.

**color**

[*color*] Overrides face and edge colors from *Q*.

**labelpos**

[{'N', 'S', 'E', 'W'}] Position the label above, below, to the right, to the left of the arrow, respectively.

**labelsep**

[float, default: 0.1] Distance in inches between the arrow and the label.

**labelcolor**

[*color*, default: `rcParams["text.color"]` (default: `'black'`)] Label color.

**fontproperties**

[dict, optional] A dictionary with keyword arguments accepted by the `FontProperties` initializer: *family*, *style*, *variant*, *size*, *weight*.

****kwargs**

Any additional keyword arguments are used to override vector properties taken from *Q*.

## Examples using `matplotlib.axes.Axes.quiverkey`

- *Advanced quiver and quiverkey functions*
- *Quiver Simple Demo*

### matplotlib.axes.Axes.streamplot

Axes.**streamplot**(*x*, *y*, *u*, *v*, *density=1*, *linewidth=None*, *color=None*, *cmap=None*, *norm=None*, *arrowsize=1*, *arrowstyle='-|>'*, *minlength=0.1*, *transform=None*, *zorder=None*, *start_points=None*, *maxlength=4.0*, *integration_direction='both'*, *broken_streamlines=True*, ***, *data=None*)

Draw streamlines of a vector flow.

**Parameters**

**x, y**

[1D/2D arrays] Evenly spaced strictly increasing arrays to make a grid. If 2D, all rows of *x* must be equal and all columns of *y* must be equal; i.e., they must be as if generated by `np.meshgrid(x_1d, y_1d)`.

**u, v**

[2D arrays] *x* and *y*-velocities. The number of rows and columns must match the length of *y* and *x*, respectively.

**density**

[float or (float, float)] Controls the closeness of streamlines. When `density = 1`, the domain is divided into a 30x30 grid. *density* linearly scales this grid. Each cell in the grid can have, at most, one traversing streamline. For different densities in each direction, use a tuple (density_x, density_y).

**linewidth**

[float or 2D array] The width of the streamlines. With a 2D array the line width can be varied across the grid. The array must have the same shape as *u* and *v*.

**color**

[*color* or 2D array] The streamline color. If given an array, its values are converted to colors using *cmap* and *norm*. The array must have the same shape as *u* and *v*.

**cmap, norm**

Data normalization and colormapping parameters for *color*; only used if *color* is an array of floats. See *imshow* for a detailed description.

**arrowsize**

[float] Scaling factor for the arrow size.

**arrowstyle**

[str] Arrow style specification. See *FancyArrowPatch*.

**minlength**

[float] Minimum length of streamline in axes coordinates.

**start_points**

[(N, 2) array] Coordinates of starting points for the streamlines in data coordinates (the same coordinates as the *x* and *y* arrays).

**zorder**

[float] The zorder of the streamlines and arrows. Artists with lower zorder values are drawn first.

**maxlength**

[float] Maximum length of streamline in axes coordinates.

**integration_direction**

[{'forward', 'backward', 'both'}, default: 'both'] Integrate the streamline in forward, backward or both directions.

**data**

[indexable object, optional] If given, the following parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception):

*x*, *y*, *u*, *v*, *start_points*

**broken_streamlines**

[boolean, default: True] If False, forces streamlines to continue until they leave the plot domain. If True, they may be terminated if they come too close to another streamline.

**Returns**

**StreamplotSet**

Container object with attributes

- `lines`: *`LineCollection`* of streamlines
- `arrows`: *`PatchCollection`* containing *`FancyArrowPatch`* objects representing the arrows half-way along streamlines.

This container will probably change in the future to allow changes to the colormap, alpha, etc. for both lines and arrows, but these changes should be backward compatible.

### Examples using `matplotlib.axes.Axes.streamplot`

- *Streamplot*
- *streamplot(X, Y, U, V)*

### Clearing

| | |
|---|---|
| *Axes.cla* | Clear the Axes. |
| *Axes.clear* | Clear the Axes. |

### matplotlib.axes.Axes.cla

Axes.**cla**()
Clear the Axes.

### matplotlib.axes.Axes.clear

Axes.**clear**()
Clear the Axes.

### Examples using `matplotlib.axes.Axes.clear`

- *pyplot animation*
- *Data browser*

### Appearance

| | |
|---|---|
| *Axes.axis* | Convenience method to get or set some axis properties. |
| *Axes.set_axis_off* | Hide all visual components of the x- and y-axis. |
| *Axes.set_axis_on* | Do not hide all visual components of the x- and y-axis. |
| *Axes.set_frame_on* | Set whether the Axes rectangle patch is drawn. |
| *Axes.get_frame_on* | Get whether the Axes rectangle patch is drawn. |
| *Axes.set_axisbelow* | Set whether axis ticks and gridlines are above or below most artists. |
| *Axes.get_axisbelow* | Get whether axis ticks and gridlines are above or below most artists. |
| *Axes.grid* | Configure the grid lines. |
| *Axes.get_facecolor* | Get the facecolor of the Axes. |
| *Axes.set_facecolor* | Set the facecolor of the Axes. |

### matplotlib.axes.Axes.axis

Axes.**axis**(*arg=None*, */*, *, *emit=True*, ***kwargs*)

Convenience method to get or set some axis properties.

Call signatures:

```
xmin, xmax, ymin, ymax = axis()
xmin, xmax, ymin, ymax = axis([xmin, xmax, ymin, ymax])
xmin, xmax, ymin, ymax = axis(option)
xmin, xmax, ymin, ymax = axis(**kwargs)
```

**Parameters**

**xmin, xmax, ymin, ymax**

[float, optional] The axis limits to be set. This can also be achieved using

```
ax.set(xlim=(xmin, xmax), ylim=(ymin, ymax))
```

**option**

[bool or str] If a bool, turns axis lines and labels on or off. If a string, possible values are:

| Value | Description |
|---|---|
| 'off' or Fals | Hide all axis decorations, i.e. axis labels, spines, tick marks, tick labels, and grid lines. This is the same as *set_axis_off()*. |
| 'on' or True | Do not hide all axis decorations, i.e. axis labels, spines, tick marks, tick labels, and grid lines. This is the same as *set_axis_on()*. |
| 'equal | Set equal scaling (i.e., make circles circular) by changing the axis limits. This is the same as `ax.set_aspect('equal', adjustable='datalim')`. Explicit data limits may not be respected in this case. |
| 'scale | Set equal scaling (i.e., make circles circular) by changing dimensions of the plot box. This is the same as `ax.set_aspect('equal', adjustable='box', anchor='C')`. Additionally, further autoscaling will be disabled. |
| 'tight' | Set limits just large enough to show all data, then disable further autoscaling. |
| 'auto' | Automatic scaling (fill plot box with data). |
| 'image' | 'scaled' with axis limits equal to data limits. |
| 'squar | Square plot; similar to 'scaled', but initially forcing `xmax-xmin == ymax-ymin`. |

**emit**

[bool, default: True] Whether observers are notified of the axis limit change. This option is passed on to *set_xlim* and *set_ylim*.

**Returns**

**xmin, xmax, ymin, ymax**

[float] The axis limits.

**See also:**

***matplotlib.axes.Axes.set_xlim***
***matplotlib.axes.Axes.set_ylim***

### Notes

For 3D Axes, this method additionally takes *zmin*, *zmax* as parameters and likewise returns them.

## Examples using `matplotlib.axes.Axes.axis`

- *Filled polygon*
- *Clipping images with patches*
- *Bar of pie*
- *Hatch style reference*
- *PathPatch object*
- *ggplot style sheet*
- *Parasite Simple2*
- *Simple Axisline4*
- *Axis Direction*
- *axis_direction demo*
- *Axis line styles*
- *floating_axes features*
- *Parasite Axes demo*
- *Parasite axis demo*
- *Ticklabel alignment*
- *Ticklabel direction*
- *Simple axis direction*
- *Simple axis tick label and tick directions*
- *Simple Axis Pad*
- *Custom spines with axisartist*
- *Simple Axisline*
- *Simple Axisline3*
- *Packed-bubble chart*
- *TickedStroke patheffect*
- *Ishikawa Diagram*
- *Quick start guide*
- *Specifying colors*

- *Text in Matplotlib*

### matplotlib.axes.Axes.set_axis_off

Axes.**set_axis_off**()

Hide all visual components of the x- and y-axis.

This sets a flag to suppress drawing of all axis decorations, i.e. axis labels, axis spines, and the axis tick component (tick markers, tick labels, and grid lines). Individual visibility settings of these components are ignored as long as *set_axis_off()* is in effect.

### Examples using `matplotlib.axes.Axes.set_axis_off`

- *Marker reference*
- *Barcode*
- *Blend transparency with color in 2D images*
- *Nested pie charts*
- *Annotation arrow style reference*
- *Text alignment*
- *Colormap reference*
- *Reference for Matplotlib artists*
- *Drawing fancy boxes*
- *Choosing Colormaps in Matplotlib*
- *Text properties and layout*

### matplotlib.axes.Axes.set_axis_on

Axes.**set_axis_on**()

Do not hide all visual components of the x- and y-axis.

This reverts the effect of a prior *set_axis_off()* call. Whether the individual axis decorations are drawn is controlled by their respective visibility settings.

This is on by default.

### matplotlib.axes.Axes.set_frame_on

Axes.**set_frame_on**(*b*)

Set whether the Axes rectangle patch is drawn.

**Parameters**

**b**

[bool]

### matplotlib.axes.Axes.get_frame_on

Axes.**get_frame_on**()

Get whether the Axes rectangle patch is drawn.

### matplotlib.axes.Axes.set_axisbelow

Axes.**set_axisbelow**(*b*)

Set whether axis ticks and gridlines are above or below most artists.

This controls the zorder of the ticks and gridlines. For more information on the zorder see *Zorder Demo*.

**Parameters**

**b**

[bool or 'line'] Possible values:

- *True* (zorder = 0.5): Ticks and gridlines are below patches and lines, though still above images.
- 'line' (zorder = 1.5): Ticks and gridlines are above patches (e.g. rectangles, with default zorder = 1) but still below lines and markers (with their default zorder = 2).
- *False* (zorder = 2.5): Ticks and gridlines are above patches and lines / markers.

**See also:**

***get_axisbelow***

### Notes

For more control, call the *set_zorder* method of each axis.

## matplotlib.axes.Axes.get_axisbelow

Axes.**get_axisbelow**()

Get whether axis ticks and gridlines are above or below most artists.

**Returns**

**bool or 'line'**

**See also:**

***set_axisbelow***

## matplotlib.axes.Axes.grid

Axes.**grid**(*visible=None*, *which='major'*, *axis='both'*, ***kwargs*)

Configure the grid lines.

**Parameters**

**visible**

[bool or None, optional] Whether to show the grid lines. If any *kwargs* are supplied, it is assumed you want the grid on and *visible* will be set to True.

If *visible* is *None* and there are no *kwargs*, this toggles the visibility of the lines.

**which**

[{'major', 'minor', 'both'}, optional] The grid lines to apply the changes on.

**axis**

[{'both', 'x', 'y'}, optional] The axis to apply the changes on.

****kwargs**

[*Line2D* properties] Define the line properties of the grid, e.g.:

```
grid(color='r', linestyle='-', linewidth=2)
```

Valid keyword arguments are:

| Property | Description |
|---|---|
| agg_filter | a filter function, which takes a (m, n, 3) float array and a dpi va |
| alpha | scalar or None |
| animated | bool |
| antialiased or aa | bool |
| clip_box | BboxBase or None |
| clip_on | bool |
| clip_path | Patch or (Path, Transform) or None |
| color or c | color |
| dash_capstyle | CapStyle or {'butt', 'projecting', 'round'} |
| dash_joinstyle | JoinStyle or {'miter', 'round', 'bevel'} |
| dashes | sequence of floats (on/off ink in points) or (None, None) |
| data | (2, N) array or two 1D arrays |
| drawstyle or ds | {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: |
| figure | Figure |
| fillstyle | {'full', 'left', 'right', 'bottom', 'top', 'none'} |
| gapcolor | color or None |
| gid | str |
| in_layout | bool |
| label | object |
| linestyle or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| linewidth or lw | float |
| marker | marker style string, Path or MarkerStyle |
| markeredgecolor or mec | color |
| markeredgewidth or mew | float |
| markerfacecolor or mfc | color |
| markerfacecoloralt or mfcalt | color |
| markersize or ms | float |
| markevery | None or int or (int, int) or slice or list[int] or float or (float, floa |
| mouseover | bool |
| path_effects | list of AbstractPathEffect |
| picker | float or callable[[Artist, Event], tuple[bool, dict]] |
| pickradius | float |
| rasterized | bool |
| sketch_params | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| solid_capstyle | CapStyle or {'butt', 'projecting', 'round'} |
| solid_joinstyle | JoinStyle or {'miter', 'round', 'bevel'} |
| transform | unknown |
| url | str |
| visible | bool |
| xdata | 1D array |
| ydata | 1D array |
| zorder | float |

### Notes

The axis is drawn as a unit, so the effective zorder for drawing the grid is determined by the zorder of each axis, not by the zorder of the `Line2D` objects comprising the grid. Therefore, to set grid zorder, use `set_axisbelow` or, for more control, call the `set_zorder` method of each axis.

### Examples using `matplotlib.axes.Axes.grid`

- *Broken Barh*
- *Cross spectral density (CSD)*
- *Fill Between and Alpha*
- *Power spectral density (PSD)*
- *Scatter Demo2*
- *Scatter plots with a legend*
- *Simple Plot*
- *Cross- and auto-correlation*
- *Contour Corner Mask*
- *Creating annotated heatmaps*
- *Many ways to plot images*
- *Watermark image*
- *Axes Props*
- *Inverted axis*
- *Plotting cumulative distributions*
- *Polar plot*
- *Date tick labels*
- *Multiline*
- *Text watermark*
- *PathPatch object*
- *axis_direction demo*
- *floating_axes features*
- *Anatomy of a figure*
- *Stock prices over 32 years*
- *Decay*
- *The double pendulum problem*

- *Custom projection*
- *Patheffect Demo*
- *2D and 3D Axes in same figure*
- *Asinh Demo*
- *Log Demo*
- *Scales*
- *Log Axis*
- *Symlog Demo*
- *Artist tests*
- *Quick start guide*
- *Axis scales*
- *Axis ticks*

### matplotlib.axes.Axes.get_facecolor

Axes.**get_facecolor**()

Get the facecolor of the Axes.

### matplotlib.axes.Axes.set_facecolor

Axes.**set_facecolor**(*color*)

Set the facecolor of the Axes.

**Parameters**

**color**

[*color*]

### Examples using `matplotlib.axes.Axes.set_facecolor`

- *Color Demo*
- *Colors in the default property cycle*

### Property cycle

| | |
|---|---|
| *Axes.set_prop_cycle* | Set the property cycle of the Axes. |

## matplotlib.axes.Axes.set_prop_cycle

Axes.**set_prop_cycle**(**args*, ***kwargs*)

Set the property cycle of the Axes.

The property cycle controls the style properties such as color, marker and linestyle of future plot commands. The style properties of data already added to the Axes are not modified.

Call signatures:

```
set_prop_cycle(cycler)
set_prop_cycle(label=values[, label2=values2[, ...]])
set_prop_cycle(label, values)
```

Form 1 sets given `Cycler` object.

Form 2 creates a `Cycler` which cycles over one or more properties simultaneously and set it as the property cycle of the Axes. If multiple properties are given, their value lists must have the same length. This is just a shortcut for explicitly creating a cycler and passing it to the function, i.e. it's short for `set_prop_cycle(cycler(label=values label2=values2, ...))`.

Form 3 creates a `Cycler` for a single property and set it as the property cycle of the Axes. This form exists for compatibility with the original `cycler.cycler` interface. Its use is discouraged in favor of the kwarg form, i.e. `set_prop_cycle(label=values)`.

**Parameters**

**cycler**

[`Cycler`] Set the given Cycler. *None* resets to the cycle defined by the current style.

**label**

[str] The property key. Must be a valid *Artist* property. For example, 'color' or 'linestyle'. Aliases are allowed, such as 'c' for 'color' and 'lw' for 'linewidth'.

**values**

[iterable] Finite-length iterable of the property values. These values are validated and will raise a ValueError if invalid.

**See also:**

***matplotlib.rcsetup.cycler***

Convenience function for creating validated cyclers for properties.

**cycler.cycler**

The original function for creating unvalidated cyclers.

### Examples

Setting the property cycle for a single property:

```
>>> ax.set_prop_cycle(color=['red', 'green', 'blue'])
```

Setting the property cycle for simultaneously cycling over multiple properties (e.g. red circle, green plus, blue cross):

```
>>> ax.set_prop_cycle(color=['red', 'green', 'blue'],
...                   marker=['o', '+', 'x'])
```

## Examples using `matplotlib.axes.Axes.set_prop_cycle`

- *Stock prices over 32 years*
- *Styling with cycler*

## Axis / limits

| | |
|---|---|
| *Axes.get_xaxis* | [*Discouraged*] Return the XAxis instance. |
| *Axes.get_yaxis* | [*Discouraged*] Return the YAxis instance. |

## matplotlib.axes.Axes.get_xaxis

Axes.**get_xaxis**()

[*Discouraged*] Return the XAxis instance.

**Discouraged**

The use of this function is discouraged. You should instead directly access the attribute `ax.xaxis`.

### matplotlib.axes.Axes.get_yaxis

Axes.**get_yaxis**()

[*Discouraged*] Return the YAxis instance.

**Discouraged**

The use of this function is discouraged. You should instead directly access the attribute `ax.yaxis`.

### Axis limits and direction

| | |
|---|---|
| *Axes.invert_xaxis* | Invert the x-axis. |
| *Axes.xaxis_inverted* | Return whether the xaxis is oriented in the "inverse" direction. |
| *Axes.invert_yaxis* | Invert the y-axis. |
| *Axes.yaxis_inverted* | Return whether the yaxis is oriented in the "inverse" direction. |
| *Axes.set_xlim* | Set the x-axis view limits. |
| *Axes.get_xlim* | Return the x-axis view limits. |
| *Axes.set_ylim* | Set the y-axis view limits. |
| *Axes.get_ylim* | Return the y-axis view limits. |
| *Axes.update_datalim* | Extend the `dataLim` Bbox to include the given points. |
| *Axes.set_xbound* | Set the lower and upper numerical bounds of the x-axis. |
| *Axes.get_xbound* | Return the lower and upper x-axis bounds, in increasing order. |
| *Axes.set_ybound* | Set the lower and upper numerical bounds of the y-axis. |
| *Axes.get_ybound* | Return the lower and upper y-axis bounds, in increasing order. |

### matplotlib.axes.Axes.invert_xaxis

Axes.**invert_xaxis**()

Invert the x-axis.

**See also:**

***xaxis_inverted***
***get_xlim*, *set_xlim***
***get_xbound*, *set_xbound***

### matplotlib.axes.Axes.xaxis_inverted

Axes.**xaxis_inverted**()

Return whether the xaxis is oriented in the "inverse" direction.

The "normal" direction is increasing to the right for the x-axis and to the top for the y-axis; the "inverse" direction is increasing to the left for the x-axis and to the bottom for the y-axis.

### matplotlib.axes.Axes.invert_yaxis

Axes.**invert_yaxis**()

Invert the y-axis.

**See also:**

***yaxis_inverted***
***get_ylim***, ***set_ylim***
***get_ybound***, ***set_ybound***

### Examples using `matplotlib.axes.Axes.invert_yaxis`

- *Bar Label Demo*
- *Horizontal bar chart*
- *Marker reference*

### matplotlib.axes.Axes.yaxis_inverted

Axes.**yaxis_inverted**()

Return whether the yaxis is oriented in the "inverse" direction.

The "normal" direction is increasing to the right for the x-axis and to the top for the y-axis; the "inverse" direction is increasing to the left for the x-axis and to the bottom for the y-axis.

### matplotlib.axes.Axes.set_xlim

Axes.**set_xlim**(*left=None*, *right=None*, *, *emit=True*, *auto=False*, *xmin=None*, *xmax=None*)

Set the x-axis view limits.

**Parameters**

**left**

[float, optional] The left xlim in data coordinates. Passing *None* leaves the limit unchanged.

The left and right xlims may also be passed as the tuple (*left*, *right*) as the first positional argument (or as the *left* keyword argument).

**right**

[float, optional] The right xlim in data coordinates. Passing *None* leaves the limit unchanged.

**emit**

[bool, default: True] Whether to notify observers of limit change.

**auto**

[bool or None, default: False] Whether to turn on autoscaling of the x-axis. True turns on, False turns off, None leaves unchanged.

**xmin, xmax**

[float, optional] They are equivalent to left and right respectively, and it is an error to pass both *xmin* and *left* or *xmax* and *right*.

**Returns**

**left, right**

[(float, float)] The new x-axis limits in data coordinates.

**See also:**

***get_xlim***
***set_xbound***, ***get_xbound***
***invert_xaxis***, ***xaxis_inverted***

## Notes

The *left* value may be greater than the *right* value, in which case the x-axis values will decrease from left to right.

## Examples

```
>>> set_xlim(left, right)
>>> set_xlim((left, right))
>>> left, right = set_xlim(left, right)
```

One limit may be left unchanged.

```
>>> set_xlim(right=right_lim)
```

Limits may be passed in reverse order to flip the direction of the x-axis. For example, suppose $x$ represents the number of years before present. The x-axis limits might be set like the following so 5000 years ago is on the left of the plot and the present is on the right.

```
>>> set_xlim(5000, 0)
```

### Examples using `matplotlib.axes.Axes.set_xlim`

- *Bar Label Demo*
- *Broken Barh*
- *Cross spectral density (CSD)*
- *EventCollection Demo*
- *Markevery Demo*
- *Contouring the solution space of optimizations*
- *Image nonuniform*
- *pcolormesh grids and shading*
- *Spectrogram*
- *Axes box aspect*
- *Axes Demo*
- *Inverted axis*
- *Boxplots*
- *Violin plot customization*
- *Including upper and lower limits in error bars*
- *Bar of pie*
- *Annotate Transform*
- *AnnotationBbox demo*
- *Using a text as a Path*
- *Text Rotation Relative To Line*
- *Mmh Donuts!!!*
- *Plotting multiple lines with a LineCollection*
- *Inset locator demo 2*
- *Parasite Simple2*
- *Simple Axis Pad*
- *Anatomy of a figure*

- *Stock prices over 32 years*
- *XKCD*
- *Decay*
- *Rain simulation*
- *Animated scatter saved as GIF*
- *Path editor*
- *Poly Editor*
- *Resampling Data*
- *Zoom Window*
- *Custom projection*
- *SVG Filter Line*
- *TickedStroke patheffect*
- *Plot 2D data on 3D plot*
- *Draw flat objects in 3D plot*
- *Text annotations in 3D*
- *Asinh Demo*
- *Loglog Aspect*
- *Scales*
- *Ishikawa Diagram*
- *SkewT-logP diagram: using transforms and custom projections*
- *Formatting date ticks using ConciseDateFormatter*
- *Date Demo Convert*
- *Date tick locators and formatters*
- *Multilevel (nested) ticks*
- *Annotation with units*
- *Artist tests*
- *Annotated cursor*
- *Cursor*
- *Span Selector*
- *Annotate Text Arrow*
- *Path Tutorial*
- *Transformations Tutorial*

- *Axis scales*
- *Axis ticks*
- *Plotting dates and strings*
- *Specifying colors*
- *Choosing Colormaps in Matplotlib*

### matplotlib.axes.Axes.get_xlim

Axes.**get_xlim**()

Return the x-axis view limits.

**Returns**

**left, right**

[(float, float)] The current x-axis limits in data coordinates.

**See also:**

***Axes.set_xlim***
***Axes.set_xbound***, ***Axes.get_xbound***
***Axes.invert_xaxis***, ***Axes.xaxis_inverted***

#### Notes

The x-axis may be inverted, in which case the *left* value will be greater than the *right* value.

### Examples using `matplotlib.axes.Axes.get_xlim`

- *Decay*

### matplotlib.axes.Axes.set_ylim

Axes.**set_ylim**(*bottom=None*, *top=None*, *, *emit=True*, *auto=False*, *ymin=None*, *ymax=None*)

Set the y-axis view limits.

**Parameters**

**bottom**

[float, optional] The bottom ylim in data coordinates. Passing *None* leaves the limit unchanged.

The bottom and top ylims may also be passed as the tuple (*bottom*, *top*) as the first positional argument (or as the *bottom* keyword argument).

**top**

[float, optional] The top ylim in data coordinates. Passing *None* leaves the limit unchanged.

**emit**

[bool, default: True] Whether to notify observers of limit change.

**auto**

[bool or None, default: False] Whether to turn on autoscaling of the y-axis. *True* turns on, *False* turns off, *None* leaves unchanged.

**ymin, ymax**

[float, optional] They are equivalent to bottom and top respectively, and it is an error to pass both *ymin* and *bottom* or *ymax* and *top*.

**Returns**

**bottom, top**

[(float, float)] The new y-axis limits in data coordinates.

**See also:**

***get_ylim***
***set_ybound***, ***get_ybound***
***invert_yaxis***, ***yaxis_inverted***

## Notes

The *bottom* value may be greater than the *top* value, in which case the y-axis values will decrease from *bottom* to *top*.

## Examples

```
>>> set_ylim(bottom, top)
>>> set_ylim((bottom, top))
>>> bottom, top = set_ylim(bottom, top)
```

One limit may be left unchanged.

```
>>> set_ylim(top=top_lim)
```

Limits may be passed in reverse order to flip the direction of the y-axis. For example, suppose `y` represents depth of the ocean in m. The y-axis limits might be set like the following so 5000 m depth is at the bottom of the plot and the surface, 0 m, is at the top.

```
>>> set_ylim(5000, 0)
```

### Examples using `matplotlib.axes.Axes.set_ylim`

- *Grouped bar chart with labels*
- *Broken Barh*
- *EventCollection Demo*
- *Hat graph*
- *Markevery Demo*
- *Power spectral density (PSD)*
- *Contouring the solution space of optimizations*
- *Image nonuniform*
- *pcolormesh grids and shading*
- *Axes Demo*
- *Broken Axis*
- *Boxplots*
- *Align y-labels*
- *Annotate Transform*
- *Annotating a plot*
- *AnnotationBbox demo*
- *Using a text as a Path*
- *Line, Poly and RegularPoly Collection with autoscaling*
- *Mmh Donuts!!!*
- *Plotting multiple lines with a LineCollection*
- *Inset locator demo 2*
- *Parasite Simple2*
- *Simple Axis Pad*
- *Simple Axisline*
- *Anatomy of a figure*
- *Integral as the area under a curve*

- *XKCD*
- *Decay*
- *Animated histogram*
- *Rain simulation*
- *MATPLOTLIB UNCHAINED*
- *Data browser*
- *Path editor*
- *Pick event demo 2*
- *Poly Editor*
- *Zoom Window*
- *Custom projection*
- *SVG Filter Line*
- *TickedStroke patheffect*
- *Plot 2D data on 3D plot*
- *Draw flat objects in 3D plot*
- *Text annotations in 3D*
- *Asinh Demo*
- *Loglog Aspect*
- *Log Demo*
- *Ishikawa Diagram*
- *SkewT-logP diagram: using transforms and custom projections*
- *Annotation with units*
- *Artist tests*
- *Annotated cursor*
- *Cursor*
- *Span Selector*
- *Annotate Text Arrow*
- *Quick start guide*
- *Path Tutorial*
- *Transformations Tutorial*
- *Axis ticks*
- *Placing colorbars*

- *Specifying colors*
- *Choosing Colormaps in Matplotlib*
- *Annotations*

### matplotlib.axes.Axes.get_ylim

Axes.**get_ylim**()

Return the y-axis view limits.

**Returns**

**bottom, top**

[(float, float)] The current y-axis limits in data coordinates.

**See also:**

***Axes.set_ylim***
***Axes.set_ybound***, ***Axes.get_ybound***
***Axes.invert_yaxis***, ***Axes.yaxis_inverted***

#### Notes

The y-axis may be inverted, in which case the *bottom* value will be greater than the *top* value.

### Examples using `matplotlib.axes.Axes.get_ylim`

- *Line, Poly and RegularPoly Collection with autoscaling*

### matplotlib.axes.Axes.update_datalim

Axes.**update_datalim**(*xys*, *updatex=True*, *updatey=True*)

Extend the `dataLim` Bbox to include the given points.

If no data is set currently, the Bbox will ignore its limits and set the bound to be the bounds of the xydata (*xys*). Otherwise, it will compute the bounds of the union of its current data and the data in *xys*.

**Parameters**

**xys**

[2D array-like] The points to include in the data limits Bbox. This can be either a list of (x, y) tuples or a (N, 2) array.

**updatex, updatey**

[bool, default: True] Whether to update the x/y limits.

## matplotlib.axes.Axes.set_xbound

Axes.**set_xbound**(*lower=None*, *upper=None*)

Set the lower and upper numerical bounds of the x-axis.

This method will honor axis inversion regardless of parameter order. It will not change the autoscaling setting (*get_autoscalex_on()*).

**Parameters**

**lower, upper**

[float or None] The lower and upper bounds. If *None*, the respective axis bound is not modified.

**See also:**

***get_xbound***
***get_xlim***, ***set_xlim***
***invert_xaxis***, ***xaxis_inverted***

## matplotlib.axes.Axes.get_xbound

Axes.**get_xbound**()

Return the lower and upper x-axis bounds, in increasing order.

**See also:**

***set_xbound***
***get_xlim***, ***set_xlim***
***invert_xaxis***, ***xaxis_inverted***

## matplotlib.axes.Axes.set_ybound

Axes.**set_ybound**(*lower=None*, *upper=None*)

Set the lower and upper numerical bounds of the y-axis.

This method will honor axis inversion regardless of parameter order. It will not change the autoscaling setting (*get_autoscaley_on()*).

**Parameters**

**lower, upper**

[float or None]

The lower and upper bounds. If *None*, the respective axis bound is not modified.

**See also:**

***get_ybound***
***get_ylim*, *set_ylim***
***invert_yaxis*, *yaxis_inverted***

### matplotlib.axes.Axes.get_ybound

`Axes.`**`get_ybound`**`()`

Return the lower and upper y-axis bounds, in increasing order.

**See also:**

***set_ybound***
***get_ylim*, *set_ylim***
***invert_yaxis*, *yaxis_inverted***

### Axis labels, title, and legend

| | |
|---|---|
| *Axes.set_xlabel* | Set the label for the x-axis. |
| *Axes.get_xlabel* | Get the xlabel text string. |
| *Axes.set_ylabel* | Set the label for the y-axis. |
| *Axes.get_ylabel* | Get the ylabel text string. |
| *Axes.label_outer* | Only show "outer" labels and tick labels. |
| *Axes.set_title* | Set a title for the Axes. |
| *Axes.get_title* | Get an Axes title. |
| *Axes.legend* | Place a legend on the Axes. |
| *Axes.get_legend* | Return the *Legend* instance, or None if no legend is defined. |
| *Axes.get_legend_handles_labels* | Return handles and labels for legend |

### matplotlib.axes.Axes.set_xlabel

Axes.**set_xlabel**(*xlabel*, *fontdict=None*, *labelpad=None*, *, *loc=None*, ***kwargs*)

Set the label for the x-axis.

**Parameters**

**xlabel**

[str] The label text.

**labelpad**

[float, default: *rcParams["axes.labelpad"]* (default: 4.0)] Spacing in points from the Axes bounding box including ticks and tick labels. If None, the previous value is left as is.

**loc**

[{'left', 'center', 'right'}, default: *rcParams["xaxis.labellocation"]* (default: 'center')] The label position. This is a high-level alternative for passing parameters *x* and *horizontalalignment*.

**Other Parameters**

****kwargs**

[*Text* properties] *Text* properties control the appearance of the label.

**See also:**

***text***

Documents the properties supported by *Text*.

### Examples using matplotlib.axes.Axes.set_xlabel

- *Bar Label Demo*
- *Horizontal bar chart*
- *Broken Barh*
- *Cross spectral density (CSD)*
- *Fill Between and Alpha*
- *Filling the area between lines*
- *Fill Betweenx Demo*
- *Hatch-filled histograms*
- *Hat graph*

- *Mapping marker properties to multivariate data*
- *Power spectral density (PSD)*
- *Scatter Demo2*
- *Stackplots and streamgraphs*
- *hlines and vlines*
- *Contourf demo*
- *Spectrogram*
- *Tricontour Demo*
- *Tripcolor Demo*
- *Triplot Demo*
- *Aligning Labels and Titles*
- *Axes Demo*
- *Axis Label Position*
- *Resizing Axes with constrained layout*
- *Resizing Axes with tight layout*
- *Figure labels: suptitle, supxlabel, supylabel*
- *Inverted axis*
- *Secondary Axis*
- *Figure subfigures*
- *Multiple subplots*
- *Plots with different scales*
- *Boxplots*
- *Box plot vs. violin plot comparison*
- *Violin plot customization*
- *Plotting cumulative distributions*
- *Histogram bins, density, and weight*
- *Producing multiple histograms side by side*
- *Accented text*
- *Labeling ticks using engineering notation*
- *Using ttf font files*
- *Legend Demo*
- *Mathtext*

- *Multiline*
- *Rendering math equations using TeX*
- *Text Commands*
- *Title positioning*
- *Color Demo*
- *Line, Poly and RegularPoly Collection with autoscaling*
- *Ellipse Collection*
- *Dark background style sheet*
- *Make room for ylabel using axes_grid*
- *Parasite Simple*
- *Ticklabel alignment*
- *Simple axis tick label and tick directions*
- *Simple Axisline*
- *Anatomy of a figure*
- *XKCD*
- *Keypress event*
- *Plot 2D data on 3D plot*
- *Create 2D bar graphs in different planes*
- *3D errorbars*
- *Lorenz attractor*
- *Automatic text offsetting*
- *3D scatterplot*
- *3D surface with polar coordinates*
- *Text annotations in 3D*
- *Asinh Demo*
- *Log Bar*
- *Centering labels between ticks*
- *Slider*
- *PGF fonts*
- *PGF texsystem*
- *Artist tutorial*
- *Quick start guide*

- *Constrained layout guide*
- *Tight layout guide*
- *Arranging multiple Axes in a Figure*
- *Axis scales*
- *Plotting dates and strings*
- *Choosing Colormaps in Matplotlib*
- *Text in Matplotlib*

### matplotlib.axes.Axes.get_xlabel

`Axes.get_xlabel()`

Get the xlabel text string.

### matplotlib.axes.Axes.set_ylabel

`Axes.set_ylabel`(*ylabel*, *fontdict=None*, *labelpad=None*, *, *loc=None*, ***kwargs*)

Set the label for the y-axis.

**Parameters**

**ylabel**

[str] The label text.

**labelpad**

[float, default: *rcParams["axes.labelpad"]* (default: `4.0`)] Spacing in points from the Axes bounding box including ticks and tick labels. If None, the previous value is left as is.

**loc**

[{'bottom', 'center', 'top'}, default: *rcParams["yaxis.labellocation"]* (default: `'center'`)] The label position. This is a high-level alternative for passing parameters *y* and *horizontalalignment*.

**Other Parameters**

****kwargs**

[*Text* properties] *Text* properties control the appearance of the label.

**See also:**

**text**

Documents the properties supported by `Text`.

### Examples using `matplotlib.axes.Axes.set_ylabel`

- *Bar color demo*
- *Grouped bar chart with labels*
- *Cross spectral density (CSD)*
- *Fill Between and Alpha*
- *Hatch-filled histograms*
- *Hat graph*
- *Mapping marker properties to multivariate data*
- *Power spectral density (PSD)*
- *Scatter Demo2*
- *Stackplots and streamgraphs*
- *Contourf demo*
- *Creating annotated heatmaps*
- *Spectrogram*
- *Tricontour Demo*
- *Tripcolor Demo*
- *Triplot Demo*
- *Aligning Labels and Titles*
- *Axes Demo*
- *Axis Label Position*
- *Resizing Axes with constrained layout*
- *Resizing Axes with tight layout*
- *Figure labels: suptitle, supxlabel, supylabel*
- *Secondary Axis*
- *Figure subfigures*
- *Multiple subplots*
- *Plots with different scales*
- *Box plots with custom fill colors*
- *Boxplots*

- *Box plot vs. violin plot comparison*
- *Violin plot customization*
- *Plotting cumulative distributions*
- *Histogram bins, density, and weight*
- *Producing multiple histograms side by side*
- *Accented text*
- *Align y-labels*
- *Date tick labels*
- *Legend Demo*
- *Mathtext*
- *Multiline*
- *Rendering math equations using TeX*
- *Text Commands*
- *Color Demo*
- *Line, Poly and RegularPoly Collection with autoscaling*
- *Ellipse Collection*
- *Dark background style sheet*
- *Make room for ylabel using axes_grid*
- *Parasite Simple*
- *Ticklabel alignment*
- *Simple axis tick label and tick directions*
- *Simple Axisline*
- *Anatomy of a figure*
- *XKCD*
- *Pick event demo*
- *Plot 2D data on 3D plot*
- *Create 2D bar graphs in different planes*
- *3D errorbars*
- *Lorenz attractor*
- *2D and 3D Axes in same figure*
- *Automatic text offsetting*
- *3D scatterplot*

- *3D surface with polar coordinates*
- *Text annotations in 3D*
- *Asinh Demo*
- *Log Bar*
- *Symlog Demo*
- *Topographic hillshading*
- *Artist tutorial*
- *Quick start guide*
- *Constrained layout guide*
- *Tight layout guide*
- *Arranging multiple Axes in a Figure*
- *Axis scales*
- *Choosing Colormaps in Matplotlib*
- *Text in Matplotlib*

### matplotlib.axes.Axes.get_ylabel

`Axes.`**`get_ylabel`**`()`

Get the ylabel text string.

### matplotlib.axes.Axes.label_outer

`Axes.`**`label_outer`**`(`*remove_inner_ticks=False*`)`

Only show "outer" labels and tick labels.

x-labels are only kept for subplots on the last row (or first row, if labels are on the top side); y-labels only for subplots on the first column (or last column, if labels are on the right side).

**Parameters**

**remove_inner_ticks**

[bool, default: False] If True, remove the inner ticks as well (not only tick labels).

New in version 3.8.

### Examples using `matplotlib.axes.Axes.label_outer`

- *Fill Between and Alpha*
- *Creating multiple subplots using plt.subplots*
- *Plotting cumulative distributions*

## matplotlib.axes.Axes.set_title

`Axes.`**`set_title`**(*label*, *fontdict=None*, *loc=None*, *pad=None*, *, *y=None*, ***kwargs*)

Set a title for the Axes.

Set one of the three available Axes titles. The available titles are positioned above the Axes in the center, flush with the left edge, and flush with the right edge.

**Parameters**

**label**

[str] Text to use for the title

**fontdict**

[dict]

---

**Discouraged**

The use of *fontdict* is discouraged. Parameters should be passed as individual keyword arguments or using dictionary-unpacking `set_title(..., **fontdict)`.

---

A dictionary controlling the appearance of the title text, the default *fontdict* is:

```
{'fontsize': rcParams['axes.titlesize'],
 'fontweight': rcParams['axes.titleweight'],
 'color': rcParams['axes.titlecolor'],
 'verticalalignment': 'baseline',
 'horizontalalignment': loc}
```

**loc**

[{'center', 'left', 'right'}, default: *rcParams["axes.titlelocation"]* (default: `'center'`)] Which title to set.

**y**

[float, default: *rcParams["axes.titley"]* (default: `None`)] Vertical Axes location for the title (1.0 is the top). If None (the default) and *rcParams["axes.titley"]* (default: `None`) is also None, y is determined automatically to avoid decorators on the Axes.

**pad**

[float, default: *rcParams["axes.titlepad"]* (default: 6.0)] The offset of the title from the top of the Axes, in points.

**Returns**

***Text***

The matplotlib text instance representing the title

**Other Parameters**

****kwargs**

[*Text* properties] Other keyword arguments are text properties, see *Text* for a list of valid text properties.

## Examples using `matplotlib.axes.Axes.set_title`

- *Bar color demo*
- *Bar Label Demo*
- *Stacked bar chart*
- *Grouped bar chart with labels*
- *Horizontal bar chart*
- *Errorbar subsampling*
- *EventCollection Demo*
- *Fill Between and Alpha*
- *Filling the area between lines*
- *Fill Betweenx Demo*
- *Hat graph*
- *Markevery Demo*
- *Power spectral density (PSD)*
- *Scatter Demo2*
- *Stackplots and streamgraphs*
- *hlines and vlines*
- *Cross- and auto-correlation*
- *Interactive Adjustment of Colormap Range*
- *Contour Corner Mask*

- *Contour Demo*
- *Contour Label Demo*
- *Contourf demo*
- *Creating annotated heatmaps*
- *Image antialiasing*
- *Many ways to plot images*
- *Image Masked*
- *Image nonuniform*
- *Interpolations for imshow*
- *Contour plot of irregularly spaced data*
- *pcolor images*
- *pcolormesh grids and shading*
- *pcolormesh*
- *Advanced quiver and quiverkey functions*
- *Tricontour Demo*
- *Tricontour Smooth Delaunay*
- *Tricontour Smooth User*
- *Trigradient Demo*
- *Tripcolor Demo*
- *Triplot Demo*
- *Aligning Labels and Titles*
- *Axes Demo*
- *Controlling view limits using margins and sticky_edges*
- *Resizing Axes with constrained layout*
- *Resizing Axes with tight layout*
- *Figure labels: suptitle, supxlabel, supylabel*
- *Inverted axis*
- *Secondary Axis*
- *Figure subfigures*
- *Creating multiple subplots using plt.subplots*
- *Plot a confidence ellipse of a two-dimensional dataset*
- *Violin plot customization*

- *Different ways of specifying error bars*
- *Including upper and lower limits in error bars*
- *Hexagonal binned plot*
- *The histogram (hist) function with multiple data sets*
- *Histogram bins, density, and weight*
- *Bar of pie*
- *Labeling a pie and a donut*
- *Polar plot*
- *Error bar rendering on polar axis*
- *Accented text*
- *Align y-labels*
- *Scale invariant angle label*
- *Angle annotations on bracket arrows*
- *Date tick labels*
- *Labeling ticks using engineering notation*
- *Using ttf font files*
- *Labelling subplots*
- *Legend Demo*
- *Mathtext*
- *Math fontfamily*
- *Multiline*
- *Rendering math equations using TeX*
- *Text Commands*
- *Title positioning*
- *Color Demo*
- *Creating a colormap from a list of colors*
- *Ways to set a color's alpha value*
- *Line, Poly and RegularPoly Collection with autoscaling*
- *Compound path*
- *Mmh Donuts!!!*
- *Plotting multiple lines with a LineCollection*
- *Bezier Curve*

- *Bayesian Methods for Hackers style sheet*
- *Dark background style sheet*
- *FiveThirtyEight style sheet*
- *Make room for ylabel using axes_grid*
- *Axis Direction*
- *Anatomy of a figure*
- *XKCD*
- *pyplot animation*
- *Cross-hair cursor*
- *Data browser*
- *Keypress event*
- *Legend picking*
- *Looking Glass*
- *Path editor*
- *Pick event demo*
- *Pick event demo 2*
- *Poly Editor*
- *Trifinder Event Demo*
- *Viewlims*
- *Packed-bubble chart*
- *Rasterization for vector graphics*
- *Zorder Demo*
- *Demo of 3D bar charts*
- *Lorenz attractor*
- *3D wireframe plots in one direction*
- *Asinh Demo*
- *Loglog Aspect*
- *Exploring normalizations*
- *Scales*
- *Radar chart (aka spider or star chart)*
- *Topographic hillshading*
- *Spines*

- *Spine placement*
- *Colorbar Tick Labelling*
- *Custom tick formatter for time series*
- *Date Precision and Epochs*
- *Move x-axis tick labels to the top*
- *Artist tests*
- *Group barchart with units*
- *Evans test*
- *Annotated cursor*
- *Rectangle and ellipse selectors*
- *Span Selector*
- *Artist tutorial*
- *Quick start guide*
- *Styling with cycler*
- *Transformations Tutorial*
- *Constrained layout guide*
- *Tight layout guide*
- *Axis scales*
- *Axis ticks*
- *Specifying colors*
- *Colormap normalization*
- *Text in Matplotlib*

### matplotlib.axes.Axes.get_title

`Axes.`**`get_title`**(*loc='center'*)

Get an Axes title.

Get one of the three available Axes titles. The available titles are positioned above the Axes in the center, flush with the left edge, and flush with the right edge.

**Parameters**

**loc**

[{'center', 'left', 'right'}, str, default: 'center'] Which title to return.

**Returns**

**str**

The title text string.

## matplotlib.axes.Axes.legend

Axes.**legend**(**args*, ***kwargs*)

Place a legend on the Axes.

Call signatures:

```
legend()
legend(handles, labels)
legend(handles=handles)
legend(labels)
```

The call signatures correspond to the following different ways to use this method:

**1. Automatic detection of elements to be shown in the legend**

The elements to be added to the legend are automatically determined, when you do not pass in any extra arguments.

In this case, the labels are taken from the artist. You can specify them either at artist creation or by calling the *set_label()* method on the artist:

```
ax.plot([1, 2, 3], label='Inline label')
ax.legend()
```

or:

```
line, = ax.plot([1, 2, 3])
line.set_label('Label via method')
ax.legend()
```

---

**Note:** Specific artists can be excluded from the automatic legend element selection by using a label starting with an underscore, "_". A string starting with an underscore is the default label for all artists, so calling *Axes.legend* without any arguments and without setting the labels manually will result in a UserWarning and an empty legend being drawn.

---

**2. Explicitly listing the artists and labels in the legend**

For full control of which artists have a legend entry, it is possible to pass an iterable of legend artists followed by an iterable of legend labels respectively:

```
ax.legend([line1, line2, line3], ['label1', 'label2', 'label3'])
```

**3. Explicitly listing the artists in the legend**

This is similar to 2, but the labels are taken from the artists' label properties. Example:

```
line1, = ax.plot([1, 2, 3], label='label1')
line2, = ax.plot([1, 2, 3], label='label2')
ax.legend(handles=[line1, line2])
```

**4. Labeling existing plot elements**

---

**Discouraged**

This call signature is discouraged, because the relation between plot elements and labels is only implicit by their order and can easily be mixed up.

---

To make a legend for all artists on an Axes, call this function with an iterable of strings, one for each legend item. For example:

```
ax.plot([1, 2, 3])
ax.plot([5, 6, 7])
ax.legend(['First line', 'Second line'])
```

**Parameters**

**handles**

[list of (*Artist* or tuple of *Artist*), optional] A list of Artists (lines, patches) to be added to the legend. Use this together with *labels*, if you need full control on what is shown in the legend and the automatic mechanism described above is not sufficient.

The length of handles and labels should be the same in this case. If they are not, they are truncated to the smaller length.

If an entry contains a tuple, then the legend handler for all Artists in the tuple will be placed alongside a single label.

**labels**

[list of str, optional] A list of labels to show next to the artists. Use this together with *handles*, if you need full control on what is shown in the legend and the automatic mechanism described above is not sufficient.

**Returns**

*Legend*

**Other Parameters**

**loc**

[str or pair of floats, default: *rcParams["legend.loc"]* (default: 'best')]
The location of the legend.

The strings 'upper left', 'upper right', 'lower left', 'lower right' place the legend at the corresponding corner of the axes.

The strings 'upper center', 'lower center', 'center left', 'center right' place the legend at the center of the corresponding edge of the axes.

The string 'center' places the legend at the center of the axes.

The string 'best' places the legend at the location, among the nine locations defined so far, with the minimum overlap with other drawn artists. This option can be quite slow for plots with large amounts of data; your plotting speed may benefit from providing a specific location.

The location can also be a 2-tuple giving the coordinates of the lower-left corner of the legend in axes coordinates (in which case *bbox_to_anchor* will be ignored).

For back-compatibility, 'center right' (but no other location) can also be spelled 'right', and each "string" location can also be given as a numeric value:

| Location String | Location Code |
|---|---|
| 'best' (Axes only) | 0 |
| 'upper right' | 1 |
| 'upper left' | 2 |
| 'lower left' | 3 |
| 'lower right' | 4 |
| 'right' | 5 |
| 'center left' | 6 |
| 'center right' | 7 |
| 'lower center' | 8 |
| 'upper center' | 9 |
| 'center' | 10 |

**bbox_to_anchor**

[*BboxBase*, 2-tuple, or 4-tuple of floats] Box that is used to position the legend in conjunction with *loc*. Defaults to axes.bbox (if called as a method to *Axes.legend*) or figure.bbox (if *Figure.legend*). This argument allows arbitrary placement of the legend.

Bbox coordinates are interpreted in the coordinate system given by *bbox_transform*, with the default transform Axes or Figure coordinates, depending on which legend is called.

If a 4-tuple or *BboxBase* is given, then it specifies the bbox (x, y, width, height) that the legend is placed in. To put the legend in the best location in the bottom right quadrant of the Axes (or figure):

```
loc='best', bbox_to_anchor=(0.5, 0., 0.5, 0.5)
```

A 2-tuple `(x, y)` places the corner of the legend specified by *loc* at x, y. For example, to put the legend's upper right-hand corner in the center of the Axes (or figure) the following keywords can be used:

```
loc='upper right', bbox_to_anchor=(0.5, 0.5)
```

**ncols**

[int, default: 1] The number of columns that the legend has.

For backward compatibility, the spelling *ncol* is also supported but it is discouraged. If both are given, *ncols* takes precedence.

**prop**

[None or *FontProperties* or dict] The font properties of the legend. If None (default), the current *matplotlib.rcParams* will be used.

**fontsize**

[int or {'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large'}] The font size of the legend. If the value is numeric the size will be the absolute font size in points. String values are relative to the current default font size. This argument is only used if *prop* is not specified.

**labelcolor**

[str or list, default: *rcParams["legend.labelcolor"]* (default: `'None'`)] The color of the text in the legend. Either a valid color string (for example, 'red'), or a list of color strings. The labelcolor can also be made to match the color of the line or marker using 'linecolor', 'markerfacecolor' (or 'mfc'), or 'markeredgecolor' (or 'mec').

Labelcolor can be set globally using *rcParams["legend.labelcolor"]* (default: `'None'`). If None, use *rcParams["text.color"]* (default: `'black'`).

**numpoints**

[int, default: *rcParams["legend.numpoints"]* (default: `1`)] The number of marker points in the legend when creating a legend entry for a *Line2D* (line).

**scatterpoints**

[int, default: *rcParams["legend.scatterpoints"]* (default: `1`)] The number of marker points in the legend when creating a legend entry for a *PathCollection* (scatter plot).

**scatteryoffsets**

[iterable of floats, default: `[0.375, 0.5, 0.3125]`] The vertical offset (relative to the font size) for the markers created for a scatter plot legend entry. 0.0 is

at the base the legend text, and 1.0 is at the top. To draw all markers at the same height, set to `[0.5]`.

**markerscale**

[float, default: *rcParams["legend.markerscale"]* (default: `1.0`)] The relative size of legend markers compared to the originally drawn ones.

**markerfirst**

[bool, default: True] If *True*, legend marker is placed to the left of the legend label. If *False*, legend marker is placed to the right of the legend label.

**reverse**

[bool, default: False] If *True*, the legend labels are displayed in reverse order from the input. If *False*, the legend labels are displayed in the same order as the input.

New in version 3.7.

**frameon**

[bool, default: *rcParams["legend.frameon"]* (default: `True`)] Whether the legend should be drawn on a patch (frame).

**fancybox**

[bool, default: *rcParams["legend.fancybox"]* (default: `True`)] Whether round edges should be enabled around the *FancyBboxPatch* which makes up the legend's background.

**shadow**

[None, bool or dict, default: *rcParams["legend.shadow"]* (default: `False`)] Whether to draw a shadow behind the legend. The shadow can be configured using *Patch* keywords. Customization via *rcParams["legend.shadow"]* (default: `False`) is currently not supported.

**framealpha**

[float, default: *rcParams["legend.framealpha"]* (default: `0.8`)] The alpha transparency of the legend's background. If *shadow* is activated and *framealpha* is `None`, the default value is ignored.

**facecolor**

["inherit" or color, default: *rcParams["legend.facecolor"]* (default: `'inherit'`)] The legend's background color. If `"inherit"`, use *rcParams["axes.facecolor"]* (default: `'white'`).

**edgecolor**

["inherit" or color, default: *rcParams["legend.edgecolor"]* (default: `'0.8'`)] The legend's background patch edge color. If `"inherit"`, use *rcParams["axes.edgecolor"]* (default: `'black'`).

**mode**

[{"expand", None}] If *mode* is set to `"expand"` the legend will be horizontally expanded to fill the Axes area (or *bbox_to_anchor* if defines the legend's size).

**bbox_transform**

[None or *Transform*] The transform for the bounding box (*bbox_to_anchor*). For a value of `None` (default) the Axes' `transAxes` transform will be used.

**title**

[str or None] The legend's title. Default is no title (`None`).

**title_fontproperties**

[None or *FontProperties* or dict] The font properties of the legend's title. If None (default), the *title_fontsize* argument will be used if present; if *title_fontsize* is also None, the current *rcParams["legend.title_fontsize"]* (default: `None`) will be used.

**title_fontsize**

[int or {'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large'}, default: *rcParams["legend.title_fontsize"]* (default: `None`)] The font size of the legend's title. Note: This cannot be combined with *title_fontproperties*. If you want to set the fontsize alongside other font properties, use the *size* parameter in *title_fontproperties*.

**alignment**

[{'center', 'left', 'right'}, default: 'center'] The alignment of the legend title and the box of entries. The entries are aligned as a single block, so that markers always lined up.

**borderpad**

[float, default: *rcParams["legend.borderpad"]* (default: `0.4`)] The fractional whitespace inside the legend border, in font-size units.

**labelspacing**

[float, default: *rcParams["legend.labelspacing"]* (default: `0.5`)] The vertical space between the legend entries, in font-size units.

**handlelength**

[float, default: *rcParams["legend.handlelength"]* (default: `2.0`)] The length of the legend handles, in font-size units.

**handleheight**

[float, default: *rcParams["legend.handleheight"]* (default: `0.7`)] The height of the legend handles, in font-size units.

**handletextpad**

[float, default: *rcParams["legend.handletextpad"]* (default: `0.8`)] The pad between the legend handle and text, in font-size units.

**borderaxespad**

[float, default: `rcParams["legend.borderaxespad"]` (default: `0.5`)] The pad between the Axes and legend border, in font-size units.

**columnspacing**

[float, default: `rcParams["legend.columnspacing"]` (default: `2.0`)] The spacing between columns, in font-size units.

**handler_map**

[dict or None] The custom dictionary mapping instances or types to a legend handler. This *handler_map* updates the default handler map found at `matplotlib.legend.Legend.get_legend_handler_map`.

**draggable**

[bool, default: False] Whether the legend can be dragged with the mouse.

**See also:**

***`Figure.legend`***

### Notes

Some artists are not supported by this function. See *Legend guide* for details.

### Examples

## Examples using `matplotlib.axes.Axes.legend`

- *Bar color demo*
- *Bar Label Demo*
- *Stacked bar chart*
- *Grouped bar chart with labels*
- *Plotting categorical variables*
- *Fill Between and Alpha*
- *Hat graph*
- *Discrete distribution as horizontal bar chart*
- *Customizing dashed line styles*
- *Lines with a ticked patheffect*

- *Scatter plots with a legend*
- *Stackplots and streamgraphs*
- *Stairs Demo*
- *Contourf Hatching*
- *Contourf and log color scale*
- *Tricontour Demo*
- *Secondary Axis*
- *Plot a confidence ellipse of a two-dimensional dataset*
- *Bihistogram*
- *Plotting cumulative distributions*
- *The histogram (hist) function with multiple data sets*
- *Histogram bins, density, and weight*
- *Bar of pie*
- *Labeling a pie and a donut*
- *Polar legend*
- *Composing Custom Legends*
- *Legend using pre-defined labels*
- *Legend Demo*
- *Mathtext*
- *Rendering math equations using TeX*
- *Inset locator demo*
- *Parasite Simple*
- *Parasite Axes demo*
- *Parasite axis demo*
- *Anatomy of a figure*
- *Legend picking*
- *Patheffect Demo*
- *TickedStroke patheffect*
- *Plot 2D data on 3D plot*
- *Parametric curve*
- *Asinh Demo*
- *Multiple y-axis with Spines*

- *Custom tick formatter for time series*
- *Group barchart with units*
- *Simple Legend01*
- *Simple Legend02*
- *Quick start guide*
- *Animations using Matplotlib*
- *Legend guide*
- *Constrained layout guide*
- *Tight layout guide*
- *Specifying colors*

### matplotlib.axes.Axes.get_legend

Axes.**get_legend**()

Return the *Legend* instance, or None if no legend is defined.

### matplotlib.axes.Axes.get_legend_handles_labels

Axes.**get_legend_handles_labels**(*legend_handler_map=None*)

Return handles and labels for legend

`ax.legend()` is equivalent to

```
h, l = ax.get_legend_handles_labels()
ax.legend(h, l)
```

### Examples using `matplotlib.axes.Axes.get_legend_handles_labels`

- *Legend guide*

### Axis scales

| | |
|---|---|
| *Axes.set_xscale* | Set the xaxis' scale. |
| *Axes.get_xscale* | Return the xaxis' scale (as a str). |
| *Axes.set_yscale* | Set the yaxis' scale. |
| *Axes.get_yscale* | Return the yaxis' scale (as a str). |

### matplotlib.axes.Axes.set_xscale

Axes.**set_xscale**(*value*, ***kwargs*)

Set the xaxis' scale.

**Parameters**

**value**

[{"linear", "log", "symlog", "logit", ...} or *ScaleBase*] The axis scale type to apply.

****kwargs**

Different keyword arguments are accepted, depending on the scale. See the respective class keyword arguments:

- *matplotlib.scale.LinearScale*
- *matplotlib.scale.LogScale*
- *matplotlib.scale.SymmetricalLogScale*
- *matplotlib.scale.LogitScale*
- *matplotlib.scale.FuncScale*
- *matplotlib.scale.AsinhScale*

#### Notes

By default, Matplotlib supports the above-mentioned scales. Additionally, custom scales may be registered using *matplotlib.scale.register_scale*. These scales can then also be used here.

### Examples using `matplotlib.axes.Axes.set_xscale`

- *Markevery Demo*
- *Labeling ticks using engineering notation*
- *Inset locator demo*
- *Asinh Demo*
- *Loglog Aspect*
- *Log Demo*
- *Scales*
- *Symlog Demo*
- *Axis scales*

### matplotlib.axes.Axes.get_xscale

Axes.**get_xscale**()

Return the xaxis' scale (as a str).

### matplotlib.axes.Axes.set_yscale

Axes.**set_yscale**(*value*, ***kwargs*)

Set the yaxis' scale.

**Parameters**

**value**

[{"linear", "log", "symlog", "logit", ...} or *ScaleBase*] The axis scale type to apply.

****kwargs**

Different keyword arguments are accepted, depending on the scale. See the respective class keyword arguments:

- *matplotlib.scale.LinearScale*
- *matplotlib.scale.LogScale*
- *matplotlib.scale.SymmetricalLogScale*
- *matplotlib.scale.LogitScale*
- *matplotlib.scale.FuncScale*
- *matplotlib.scale.AsinhScale*

#### Notes

By default, Matplotlib supports the above-mentioned scales. Additionally, custom scales may be registered using *matplotlib.scale.register_scale*. These scales can then also be used here.

### Examples using `matplotlib.axes.Axes.set_yscale`

- *Markevery Demo*
- *Artist customization in box plots*
- *Different ways of specifying error bars*
- *Stock prices over 32 years*
- *Asinh Demo*
- *Loglog Aspect*

- *Log Bar*
- *Log Demo*
- *Scales*
- *Symlog Demo*
- *Axis scales*

### matplotlib.axes.Axes.get_yscale

Axes.**get_yscale**()

Return the yaxis' scale (as a str).

### Autoscaling and margins

| | |
|---|---|
| *Axes.use_sticky_edges* | When autoscaling, whether to obey all `Artist.sticky_edges`. |
| *Axes.margins* | Set or retrieve autoscaling margins. |
| *Axes.get_xmargin* | Retrieve autoscaling margin of the x-axis. |
| *Axes.get_ymargin* | Retrieve autoscaling margin of the y-axis. |
| *Axes.set_xmargin* | Set padding of X data limits prior to autoscaling. |
| *Axes.set_ymargin* | Set padding of Y data limits prior to autoscaling. |
| *Axes.relim* | Recompute the data limits based on current artists. |
| *Axes.autoscale* | Autoscale the axis view to the data (toggle). |
| *Axes.autoscale_view* | Autoscale the view limits using the data limits. |
| *Axes.set_autoscale_on* | Set whether autoscaling is applied to each axis on the next draw or call to *Axes.autoscale_view*. |
| *Axes.get_autoscale_on* | Return True if each axis is autoscaled, False otherwise. |
| *Axes.set_autoscalex_on* | Set whether the xaxis is autoscaled when drawing or by *Axes.autoscale_view*. |
| *Axes.get_autoscalex_on* | Return whether the xaxis is autoscaled. |
| *Axes.set_autoscaley_on* | Set whether the yaxis is autoscaled when drawing or by *Axes.autoscale_view*. |
| *Axes.get_autoscaley_on* | Return whether the yaxis is autoscaled. |

## matplotlib.axes.Axes.use_sticky_edges

**property** Axes.**use_sticky_edges**

When autoscaling, whether to obey all `Artist.sticky_edges`.

Default is `True`.

Setting this to `False` ensures that the specified margins will be applied, even if the plot includes an image, for example, which would otherwise force a view limit to coincide with its data limit.

The changing this property does not change the plot until *autoscale* or *autoscale_view* is called.

## matplotlib.axes.Axes.margins

Axes.**margins**(**margins*, *x=None*, *y=None*, *tight=True*)

Set or retrieve autoscaling margins.

The padding added to each limit of the Axes is the *margin* times the data interval. All input parameters must be floats greater than -0.5. Passing both positional and keyword arguments is invalid and will raise a TypeError. If no arguments (positional or otherwise) are provided, the current margins will remain unchanged and simply be returned.

Specifying any margin changes only the autoscaling; for example, if *xmargin* is not None, then *xmargin* times the X data interval will be added to each end of that interval before it is used in autoscaling.

**Parameters**

***margins**

[float, optional] If a single positional argument is provided, it specifies both margins of the x-axis and y-axis limits. If two positional arguments are provided, they will be interpreted as *xmargin*, *ymargin*. If setting the margin on a single axis is desired, use the keyword arguments described below.

**x, y**

[float, optional] Specific margin values for the x-axis and y-axis, respectively. These cannot be used with positional arguments, but can be used individually to alter on e.g., only the y-axis.

**tight**

[bool or None, default: True] The *tight* parameter is passed to *autoscale_view*, which is executed after a margin is changed; the default here is *True*, on the assumption that when margins are specified, no additional padding to match tick marks is usually desired. Setting *tight* to *None* preserves the previous setting.

**Returns**

**xmargin, ymargin**

[float]

### Notes

If a previously used Axes method such as *pcolor()* has set *use_sticky_edges* to True, only the limits not set by the "sticky artists" will be modified. To force all of the margins to be set, set *use_sticky_edges* to False before calling *margins()*.

## Examples using `matplotlib.axes.Axes.margins`

- *Marker reference*
- *Creating a timeline with lines, dates, and text*
- *Trigradient Demo*
- *Controlling view limits using margins and sticky_edges*
- *Scale invariant angle label*
- *ggplot style sheet*
- *Autoscaling Axis*

## matplotlib.axes.Axes.get_xmargin

Axes.**get_xmargin**()

Retrieve autoscaling margin of the x-axis.

New in version 3.9.

**Returns**

**xmargin**

[float]

**See also:**

***matplotlib.axes.Axes.set_xmargin***

### matplotlib.axes.Axes.get_ymargin

Axes.**get_ymargin**()

Retrieve autoscaling margin of the y-axis.

New in version 3.9.

**Returns**

**ymargin**

[float]

**See also:**

*matplotlib.axes.Axes.set_ymargin*

### matplotlib.axes.Axes.set_xmargin

Axes.**set_xmargin**(*m*)

Set padding of X data limits prior to autoscaling.

*m* times the data interval will be added to each end of that interval before it is used in autoscaling. If *m* is negative, this will clip the data range instead of expanding it.

For example, if your data is in the range [0, 2], a margin of 0.1 will result in a range [-0.2, 2.2]; a margin of -0.1 will result in a range of [0.2, 1.8].

**Parameters**

**m**

[float greater than -0.5]

### Examples using `matplotlib.axes.Axes.set_xmargin`

- *Automatically setting tick positions*

### matplotlib.axes.Axes.set_ymargin

Axes.**set_ymargin**(*m*)

Set padding of Y data limits prior to autoscaling.

*m* times the data interval will be added to each end of that interval before it is used in autoscaling. If *m* is negative, this will clip the data range instead of expanding it.

For example, if your data is in the range [0, 2], a margin of 0.1 will result in a range [-0.2, 2.2]; a margin of -0.1 will result in a range of [0.2, 1.8].

**Parameters**

**m**

[float greater than -0.5]

## matplotlib.axes.Axes.relim

Axes.**relim**(*visible_only=False*)

Recompute the data limits based on current artists.

At present, *Collection* instances are not supported.

**Parameters**

**visible_only**

[bool, default: False] Whether to exclude invisible artists.

## Examples using matplotlib.axes.Axes.relim

- *Packed-bubble chart*
- *Textbox*

## matplotlib.axes.Axes.autoscale

Axes.**autoscale**(*enable=True*, *axis='both'*, *tight=None*)

Autoscale the axis view to the data (toggle).

Convenience method for simple axis view autoscaling. It turns autoscaling on or off, and then, if autoscaling for either axis is on, it performs the autoscaling on the specified axis or Axes.

**Parameters**

**enable**

[bool or None, default: True] True turns autoscaling on, False turns it off. None leaves the autoscaling state unchanged.

**axis**

[{'both', 'x', 'y'}, default: 'both'] The axis on which to operate. (For 3D Axes, *axis* can also be set to 'z', and 'both' refers to all three Axes.)

**tight**

[bool or None, default: None] If True, first set the margins to zero. Then, this argument is forwarded to *autoscale_view* (regardless of its value); see the description of its behavior there.

### Examples using `matplotlib.axes.Axes.autoscale`

- *Axes box aspect*
- *Autoscaling Axis*

## matplotlib.axes.Axes.autoscale_view

`Axes.`**`autoscale_view`**(*tight=None*, *scalex=True*, *scaley=True*)

Autoscale the view limits using the data limits.

**Parameters**

**tight**

[bool or None] If *True*, only expand the axis limits using the margins. Note that unlike for *autoscale*, `tight=True` does *not* set the margins to zero.

If *False* and *rcParams["axes.autolimit_mode"]* (default: `'data'`) is 'round_numbers', then after expansion by the margins, further expand the axis limits using the axis major locator.

If None (the default), reuse the value set in the previous call to *autoscale_view* (the initial value is False, but the default style sets *rcParams["axes.autolimit_mode"]* (default: `'data'`) to 'data', in which case this behaves like True).

**scalex**

[bool, default: True] Whether to autoscale the x-axis.

**scaley**

[bool, default: True] Whether to autoscale the y-axis.

### Notes

The autoscaling preserves any preexisting axis direction reversal.

The data limits are not updated automatically when artist data are changed after the artist has been added to an Axes instance. In that case, use *matplotlib.axes.Axes.relim()* prior to calling autoscale_view.

If the views of the Axes are fixed, e.g. via *set_xlim*, they will not be changed by autoscale_view(). See *matplotlib.axes.Axes.autoscale()* for an alternative.

### Examples using `matplotlib.axes.Axes.autoscale_view`

- *Line, Poly and RegularPoly Collection with autoscaling*
- *Compound path*
- *Ellipse Collection*
- *Building histograms using Rectangles and PolyCollections*
- *Packed-bubble chart*
- *Group barchart with units*
- *Textbox*
- *Autoscaling Axis*

## matplotlib.axes.Axes.set_autoscale_on

Axes.**set_autoscale_on**(*b*)

Set whether autoscaling is applied to each axis on the next draw or call to *Axes.autoscale_view*.

**Parameters**

**b**

[bool]

### Examples using `matplotlib.axes.Axes.set_autoscale_on`

- *Resampling Data*

## matplotlib.axes.Axes.get_autoscale_on

Axes.**get_autoscale_on**()

Return True if each axis is autoscaled, False otherwise.

## matplotlib.axes.Axes.set_autoscalex_on

Axes.**set_autoscalex_on**(*b*)

Set whether the xaxis is autoscaled when drawing or by *Axes.autoscale_view*. If b is None, then the value is not changed.

**Parameters**

**b**

[bool]

### matplotlib.axes.Axes.get_autoscalex_on

Axes.**get_autoscalex_on**()

Return whether the xaxis is autoscaled.

### matplotlib.axes.Axes.set_autoscaley_on

Axes.**set_autoscaley_on**(*b*)

Set whether the yaxis is autoscaled when drawing or by *Axes.autoscale_view*. If b is None, then the value is not changed.

**Parameters**

**b**

[bool]

### matplotlib.axes.Axes.get_autoscaley_on

Axes.**get_autoscaley_on**()

Return whether the yaxis is autoscaled.

### Aspect ratio

| | |
|---|---|
| *Axes.apply_aspect* | Adjust the Axes for a specified data aspect ratio. |
| *Axes.set_aspect* | Set the aspect ratio of the Axes scaling, i.e. y/x-scale. |
| *Axes.get_aspect* | Return the aspect ratio of the Axes scaling. |
| *Axes.set_box_aspect* | Set the Axes box aspect, i.e. the ratio of height to width. |
| *Axes.get_box_aspect* | Return the Axes box aspect, i.e. the ratio of height to width. |
| *Axes.set_adjustable* | Set how the Axes adjusts to achieve the required aspect ratio. |
| *Axes.get_adjustable* | Return whether the Axes will adjust its physical dimension ('box') or its data limits ('datalim') to achieve the desired aspect ratio. |

### matplotlib.axes.Axes.apply_aspect

Axes.**apply_aspect**(*position=None*)

Adjust the Axes for a specified data aspect ratio.

Depending on *get_adjustable* this will modify either the Axes box (position) or the view limits. In the former case, *get_anchor* will affect the position.

**Parameters**

**position**

[None or .Bbox] If not None, this defines the position of the Axes within the figure as a Bbox. See *get_position* for further details.

**See also:**

***matplotlib.axes.Axes.set_aspect***

For a description of aspect ratio handling.

***matplotlib.axes.Axes.set_adjustable***

Set how the Axes adjusts to achieve the required aspect ratio.

***matplotlib.axes.Axes.set_anchor***

Set the position in case of extra space.

#### Notes

This is called automatically when each Axes is drawn. You may need to call it yourself if you need to update the Axes position and/or view limits before the Figure is drawn.

### matplotlib.axes.Axes.set_aspect

Axes.**set_aspect**(*aspect*, *adjustable=None*, *anchor=None*, *share=False*)

Set the aspect ratio of the Axes scaling, i.e. y/x-scale.

**Parameters**

**aspect**

[{'auto', 'equal'} or float] Possible values:

- 'auto': fill the position rectangle with data.
- 'equal': same as `aspect=1`, i.e. same scaling for x and y.
- *float*: The displayed size of 1 unit in y-data coordinates will be *aspect* times the displayed size of 1 unit in x-data coordinates; e.g. for `aspect=2` a square in data coordinates will be rendered with a height of twice its width.

**adjustable**

[None or {'box', 'datalim'}, optional] If not `None`, this defines which parameter will be adjusted to meet the required aspect. See *`set_adjustable`* for further details.

**anchor**

[None or str or (float, float), optional] If not `None`, this defines where the Axes will be drawn if there is extra space due to aspect constraints. The most common way to specify the anchor are abbreviations of cardinal directions:

| value | description |
|---|---|
| 'C' | centered |
| 'SW' | lower left corner |
| 'S' | middle of bottom edge |
| 'SE' | lower right corner |
| etc. | |

See *`set_anchor`* for further details.

**share**

[bool, default: False] If `True`, apply the settings to all shared Axes.

**See also:**

***matplotlib.axes.Axes.set_adjustable***

Set how the Axes adjusts to achieve the required aspect ratio.

***matplotlib.axes.Axes.set_anchor***

Set the position in case of extra space.

## Examples using `matplotlib.axes.Axes.set_aspect`

- *Tricontour Demo*
- *Tricontour Smooth Delaunay*
- *Tricontour Smooth User*
- *Trigradient Demo*
- *Tripcolor Demo*
- *Triplot Demo*
- *Axes box aspect*
- *Controlling view limits using margins and sticky_edges*

- *Multiline*
- *Mmh Donuts!!!*
- *Plotting multiple lines with a LineCollection*
- *Inset locator demo 2*
- *Scatter Histogram (Locatable Axes)*
- *Simple Anchored Artists*
- *Simple Axis Pad*
- *The double pendulum problem*
- *Multiple Axes animation*
- *Anchored Artists*
- *Rasterization for vector graphics*
- *3D surface (solid color)*
- *3D voxel plot of the NumPy logo*
- *3D voxel / volumetric plot with RGB colors*
- *Loglog Aspect*
- *Annotate Text Arrow*
- *Arranging multiple Axes in a Figure*
- *Placing colorbars*
- *Colormap normalization*

### matplotlib.axes.Axes.get_aspect

Axes.**get_aspect**()

Return the aspect ratio of the Axes scaling.

This is either "auto" or a float giving the ratio of y/x-scale.

### matplotlib.axes.Axes.set_box_aspect

Axes.**set_box_aspect**(*aspect=None*)

Set the Axes box aspect, i.e. the ratio of height to width.

This defines the aspect of the Axes in figure space and is not to be confused with the data aspect (see *set_aspect*).

**Parameters**

**aspect**

[float or None] Changes the physical dimensions of the Axes, such that the ratio of the Axes height to the Axes width in physical units is equal to *aspect*. Defining a box aspect will change the *adjustable* property to 'datalim' (see `set_adjustable`).

*None* will disable a fixed box aspect so that height and width of the Axes are chosen independently.

**See also:**

***matplotlib.axes.Axes.set_aspect***

for a description of aspect handling.

## Examples using `matplotlib.axes.Axes.set_box_aspect`

- *Axes box aspect*
- *Multiple Axes animation*
- *3D box surface plot*

## matplotlib.axes.Axes.get_box_aspect

`Axes.`**`get_box_aspect`**`()`

Return the Axes box aspect, i.e. the ratio of height to width.

The box aspect is `None` (i.e. chosen depending on the available figure space) unless explicitly specified.

**See also:**

***matplotlib.axes.Axes.set_box_aspect***

for a description of box aspect.

***matplotlib.axes.Axes.set_aspect***

for a description of aspect handling.

### matplotlib.axes.Axes.set_adjustable

Axes.**set_adjustable**(*adjustable*, *share=False*)

Set how the Axes adjusts to achieve the required aspect ratio.

**Parameters**

**adjustable**

[{'box', 'datalim'}] If 'box', change the physical dimensions of the Axes. If 'datalim', change the `x` or `y` data limits.

**share**

[bool, default: False] If `True`, apply the settings to all shared Axes.

**See also:**

***matplotlib.axes.Axes.set_aspect***

For a description of aspect handling.

#### Notes

Shared Axes (of which twinned Axes are a special case) impose restrictions on how aspect ratios can be imposed. For twinned Axes, use 'datalim'. For Axes that share both x and y, use 'box'. Otherwise, either 'datalim' or 'box' may be used. These limitations are partly a requirement to avoid over-specification, and partly a result of the particular implementation we are currently using, in which the adjustments for aspect ratios are done sequentially and independently on each Axes as it is drawn.

### Examples using `matplotlib.axes.Axes.set_adjustable`

- *Loglog Aspect*

### matplotlib.axes.Axes.get_adjustable

Axes.**get_adjustable**()

Return whether the Axes will adjust its physical dimension ('box') or its data limits ('datalim') to achieve the desired aspect ratio.

**See also:**

***matplotlib.axes.Axes.set_adjustable***

Set how the Axes adjusts to achieve the required aspect ratio.

***matplotlib.axes.Axes.set_aspect***

For a description of aspect handling.

### Ticks and tick labels

| | |
|---|---|
| *Axes.set_xticks* | Set the xaxis' tick locations and optionally tick labels. |
| *Axes.get_xticks* | Return the xaxis' tick locations in data coordinates. |
| *Axes.set_xticklabels* | [*Discouraged*] Set the xaxis' tick labels with list of string labels. |
| *Axes.get_xticklabels* | Get the xaxis' tick labels. |
| *Axes.get_xmajorticklabels* | Return the xaxis' major tick labels, as a list of *Text*. |
| *Axes.get_xminorticklabels* | Return the xaxis' minor tick labels, as a list of *Text*. |
| *Axes.get_xgridlines* | Return the xaxis' grid lines as a list of *Line2D*s. |
| *Axes.get_xticklines* | Return the xaxis' tick lines as a list of *Line2D*s. |
| *Axes.xaxis_date* | Set up axis ticks and labels to treat data along the xaxis as dates. |
| *Axes.set_yticks* | Set the yaxis' tick locations and optionally tick labels. |
| *Axes.get_yticks* | Return the yaxis' tick locations in data coordinates. |
| *Axes.set_yticklabels* | [*Discouraged*] Set the yaxis' tick labels with list of string labels. |
| *Axes.get_yticklabels* | Get the yaxis' tick labels. |
| *Axes.get_ymajorticklabels* | Return the yaxis' major tick labels, as a list of *Text*. |
| *Axes.get_yminorticklabels* | Return the yaxis' minor tick labels, as a list of *Text*. |
| *Axes.get_ygridlines* | Return the yaxis' grid lines as a list of *Line2D*s. |
| *Axes.get_yticklines* | Return the yaxis' tick lines as a list of *Line2D*s. |
| *Axes.yaxis_date* | Set up axis ticks and labels to treat data along the yaxis as dates. |
| *Axes.minorticks_off* | Remove minor ticks from the Axes. |
| *Axes.minorticks_on* | Display minor ticks on the Axes. |
| *Axes.ticklabel_format* | Configure the *ScalarFormatter* used by default for linear Axes. |
| *Axes.tick_params* | Change the appearance of ticks, tick labels, and gridlines. |
| *Axes.locator_params* | Control behavior of major tick locators. |

### matplotlib.axes.Axes.set_xticks

Axes.**set_xticks**(*ticks*, *labels=None*, *, *minor=False*, ***kwargs*)

Set the xaxis' tick locations and optionally tick labels.

If necessary, the view limits of the Axis are expanded so that all given ticks are visible.

**Parameters**

**ticks**

[1D array-like] Array of tick locations (either floats or in axis units). The axis *Locator* is replaced by a *FixedLocator*.

Pass an empty list (`set_xticks([])`) to remove all ticks.

Some tick formatters will not label arbitrary tick positions; e.g. log formatters only label decade ticks by default. In such a case you can set a formatter explicitly on the axis using *Axis.set_major_formatter* or provide formatted *labels* yourself.

**labels**

[list of str, optional] Tick labels for each location in *ticks*; must have the same length as *ticks*. If set, the labels are used as is, via a *FixedFormatter*. If not set, the labels are generated using the axis tick *Formatter*.

**minor**

[bool, default: False] If `False`, set only the major ticks; if `True`, only the minor ticks.

****kwargs**

*Text* properties for the labels. Using these is only allowed if you pass *labels*. In other cases, please use *tick_params*.

### Notes

The mandatory expansion of the view limits is an intentional design choice to prevent the surprise of a non-visible tick. If you need other limits, you should set the limits explicitly after setting the ticks.

### Examples using `matplotlib.axes.Axes.set_xticks`

- *Grouped bar chart with labels*
- *Hat graph*
- *Power spectral density (PSD)*
- *Creating annotated heatmaps*
- *Box plot vs. violin plot comparison*
- *Violin plot customization*
- *Producing multiple histograms side by side*
- *Multiline*
- *Rendering math equations using TeX*
- *ggplot style sheet*
- *Scatter Histogram (Locatable Axes)*

- *Simple Axisline4*
- *Ticklabel alignment*
- *Ticklabel direction*
- *Integral as the area under a curve*
- *Shaded & power normalized rendering*
- *XKCD*
- *Rain simulation*
- *MATPLOTLIB UNCHAINED*
- *Log Bar*
- *Rotating custom tick labels*
- *Group barchart with units*
- *The Lifecycle of a Plot*
- *Axis ticks*

### matplotlib.axes.Axes.get_xticks

Axes.**get_xticks**(*, *minor=False*)

Return the xaxis' tick locations in data coordinates.

The locations are not clipped to the current axis limits and hence may contain locations that are not visible in the output.

**Parameters**

**minor**

[bool, default: False] True to return the minor tick directions, False to return the major tick directions.

**Returns**

**array of tick locations**

### matplotlib.axes.Axes.set_xticklabels

Axes.**set_xticklabels**(*labels*, *, *minor=False*, *fontdict=None*, ***kwargs*)

[*Discouraged*] Set the xaxis' tick labels with list of string labels.

---

**Discouraged**

The use of this method is discouraged, because of the dependency on tick positions. In most cases, you'll want to use `Axes.set_[x/y/z]ticks(positions, labels)` or `Axes.set_xticks` instead.

If you are using this method, you should always fix the tick positions before, e.g. by using *Axes.set_xticks* or by explicitly setting a *FixedLocator*. Otherwise, ticks are free to move and the labels may end up in unexpected positions.

---

**Parameters**

**labels**

[sequence of str or of *Text*s] Texts for labeling each tick location in the sequence set by *Axes.set_xticks*; the number of labels must match the number of locations. The labels are used as is, via a *FixedFormatter* (without further formatting).

**minor**

[bool] If True, set minor ticks instead of major ticks.

**fontdict**

[dict, optional]

---

**Discouraged**

The use of *fontdict* is discouraged. Parameters should be passed as individual keyword arguments or using dictionary-unpacking `set_ticklabels(..., **fontdict)`.

---

A dictionary controlling the appearance of the ticklabels. The default *fontdict* is:

```
{'fontsize': rcParams['axes.titlesize'],
 'fontweight': rcParams['axes.titleweight'],
 'verticalalignment': 'baseline',
 'horizontalalignment': loc}
```

****kwargs**

Text properties.

**Warning:** This only sets the properties of the current ticks. Ticks are not guaranteed to be persistent. Various operations can create, delete and modify the Tick instances. There is an imminent risk that these settings can get lost if you work on the figure further (including also panning/zooming on a displayed figure).

Use `set_tick_params` instead if possible.

**Returns**

**list of *Text*s**

For each tick, includes `tick.label1` if it is visible, then `tick.label2` if it is visible, in that order.

### Examples using `matplotlib.axes.Axes.set_xticklabels`

- *Creating annotated heatmaps*
- *Managing multiple figures in pyplot*
- *Boxplots*
- *Rendering math equations using TeX*
- *XKCD*
- *Constrained layout guide*

## matplotlib.axes.Axes.get_xticklabels

`Axes.`**`get_xticklabels`**(*minor=False*, *which=None*)

Get the xaxis' tick labels.

**Parameters**

**minor**

[bool] Whether to return the minor or the major ticklabels.

**which**

[None, ('minor', 'major', 'both')] Overrides *minor*.

Selects which ticklabels to return

**Returns**

**list of *Text***

### Examples using `matplotlib.axes.Axes.get_xticklabels`

- *Creating annotated heatmaps*
- *Boxplots*
- *Date tick labels*
- *Pick event demo*
- *Centering labels between ticks*
- *Formatting date ticks using ConciseDateFormatter*
- *Evans test*
- *The Lifecycle of a Plot*

## matplotlib.axes.Axes.get_xmajorticklabels

Axes.**get_xmajorticklabels**()

Return the xaxis' major tick labels, as a list of *Text*.

## matplotlib.axes.Axes.get_xminorticklabels

Axes.**get_xminorticklabels**()

Return the xaxis' minor tick labels, as a list of *Text*.

## matplotlib.axes.Axes.get_xgridlines

Axes.**get_xgridlines**()

Return the xaxis' grid lines as a list of *Line2D*s.

## matplotlib.axes.Axes.get_xticklines

Axes.**get_xticklines**(*minor=False*)

Return the xaxis' tick lines as a list of *Line2D*s.

### matplotlib.axes.Axes.xaxis_date

Axes.**xaxis_date**(*tz=None*)

Set up axis ticks and labels to treat data along the xaxis as dates.

**Parameters**

**tz**

[str or datetime.tzinfo, default: *rcParams["timezone"]* (default: 'UTC')] The timezone used to create date labels.

### matplotlib.axes.Axes.set_yticks

Axes.**set_yticks**(*ticks*, *labels=None*, *, *minor=False*, ***kwargs*)

Set the yaxis' tick locations and optionally tick labels.

If necessary, the view limits of the Axis are expanded so that all given ticks are visible.

**Parameters**

**ticks**

[1D array-like] Array of tick locations (either floats or in axis units). The axis *Locator* is replaced by a *FixedLocator*.

Pass an empty list (set_yticks([])) to remove all ticks.

Some tick formatters will not label arbitrary tick positions; e.g. log formatters only label decade ticks by default. In such a case you can set a formatter explicitly on the axis using *Axis.set_major_formatter* or provide formatted *labels* yourself.

**labels**

[list of str, optional] Tick labels for each location in *ticks*; must have the same length as *ticks*. If set, the labels are used as is, via a *FixedFormatter*. If not set, the labels are generated using the axis tick *Formatter*.

**minor**

[bool, default: False] If False, set only the major ticks; if True, only the minor ticks.

****kwargs**

*Text* properties for the labels. Using these is only allowed if you pass *labels*. In other cases, please use *tick_params*.

### Notes

The mandatory expansion of the view limits is an intentional design choice to prevent the surprise of a non-visible tick. If you need other limits, you should set the limits explicitly after setting the ticks.

### Examples using `matplotlib.axes.Axes.set_yticks`

- *Bar Label Demo*
- *Horizontal bar chart*
- *Broken Barh*
- *Power spectral density (PSD)*
- *Creating annotated heatmaps*
- *Programmatically controlling subplot adjustment*
- *Rendering math equations using TeX*
- *Make room for ylabel using axes_grid*
- *Scatter Histogram (Locatable Axes)*
- *Ticklabel alignment*
- *Ticklabel direction*
- *Integral as the area under a curve*
- *Shaded & power normalized rendering*
- *XKCD*
- *Rain simulation*
- *MATPLOTLIB UNCHAINED*
- *Create 2D bar graphs in different planes*
- *SkewT-logP diagram: using transforms and custom projections*
- *Axis ticks*

## matplotlib.axes.Axes.get_yticks

Axes.**get_yticks**(*, *minor=False*)

Return the yaxis' tick locations in data coordinates.

The locations are not clipped to the current axis limits and hence may contain locations that are not visible in the output.

**Parameters**

**minor**

[bool, default: False] True to return the minor tick directions, False to return the major tick directions.

**Returns**

**array of tick locations**

## matplotlib.axes.Axes.set_yticklabels

Axes.**set_yticklabels**(*labels*, *, *minor=False*, *fontdict=None*, ***kwargs*)

[*Discouraged*] Set the yaxis' tick labels with list of string labels.

---

**Discouraged**

The use of this method is discouraged, because of the dependency on tick positions. In most cases, you'll want to use `Axes.set_[x/y/z]ticks(positions, labels)` or `Axes.set_yticks` instead.

If you are using this method, you should always fix the tick positions before, e.g. by using *Axes.set_yticks* or by explicitly setting a *FixedLocator*. Otherwise, ticks are free to move and the labels may end up in unexpected positions.

---

**Parameters**

**labels**

[sequence of str or of *Text*s] Texts for labeling each tick location in the sequence set by *Axes.set_yticks*; the number of labels must match the number of locations. The labels are used as is, via a *FixedFormatter* (without further formatting).

**minor**

[bool] If True, set minor ticks instead of major ticks.

**fontdict**

[dict, optional]

---

**Discouraged**

The use of *fontdict* is discouraged. Parameters should be passed as individual keyword arguments or using dictionary-unpacking `set_ticklabels(..., **fontdict)`.

---

A dictionary controlling the appearance of the ticklabels. The default *fontdict* is:

```
{'fontsize': rcParams['axes.titlesize'],
 'fontweight': rcParams['axes.titleweight'],
 'verticalalignment': 'baseline',
 'horizontalalignment': loc}
```

****kwargs**

Text properties.

> **Warning:** This only sets the properties of the current ticks. Ticks are not guaranteed to be persistent. Various operations can create, delete and modify the Tick instances. There is an imminent risk that these settings can get lost if you work on the figure further (including also panning/zooming on a displayed figure).
>
> Use *set_tick_params* instead if possible.

**Returns**

**list of *Text*s**

For each tick, includes `tick.label1` if it is visible, then `tick.label2` if it is visible, in that order.

### Examples using `matplotlib.axes.Axes.set_yticklabels`

- *Artist customization in box plots*
- *Violin plot basics*
- *Rendering math equations using TeX*
- *Constrained layout guide*

## matplotlib.axes.Axes.get_yticklabels

`Axes.`**`get_yticklabels`**(*minor=False*, *which=None*)

Get the yaxis' tick labels.

**Parameters**

**minor**

[bool] Whether to return the minor or the major ticklabels.

**which**

[None, ('minor', 'major', 'both')] Overrides *minor*.

Selects which ticklabels to return

**Returns**

**list of *Text***

### Examples using `matplotlib.axes.Axes.get_yticklabels`

- *Programmatically controlling subplot adjustment*
- *Colorbar Tick Labelling*

## matplotlib.axes.Axes.get_ymajorticklabels

`Axes.`**`get_ymajorticklabels`**`()`

Return the yaxis' major tick labels, as a list of *Text*.

## matplotlib.axes.Axes.get_yminorticklabels

`Axes.`**`get_yminorticklabels`**`()`

Return the yaxis' minor tick labels, as a list of *Text*.

## matplotlib.axes.Axes.get_ygridlines

`Axes.`**`get_ygridlines`**`()`

Return the yaxis' grid lines as a list of *Line2D*s.

## matplotlib.axes.Axes.get_yticklines

`Axes.`**`get_yticklines`**`(`*minor=False*`)`

Return the yaxis' tick lines as a list of *Line2D*s.

## matplotlib.axes.Axes.yaxis_date

`Axes.`**`yaxis_date`**`(`*tz=None*`)`

Set up axis ticks and labels to treat data along the yaxis as dates.

**Parameters**

**tz**

[str or `datetime.tzinfo`, default: *rcParams["timezone"]* (default: `'UTC'`)] The timezone used to create date labels.

### matplotlib.axes.Axes.minorticks_off

Axes.**minorticks_off**()

Remove minor ticks from the Axes.

### matplotlib.axes.Axes.minorticks_on

Axes.**minorticks_on**()

Display minor ticks on the Axes.

Displaying minor ticks may reduce performance; you may turn them off using *minorticks_off()* if drawing speed is a problem.

### matplotlib.axes.Axes.ticklabel_format

Axes.**ticklabel_format**(*, *axis='both'*, *style=None*, *scilimits=None*, *useOffset=None*, *useLocale=None*, *useMathText=None*)

Configure the *ScalarFormatter* used by default for linear Axes.

If a parameter is not set, the corresponding property of the formatter is left unchanged.

**Parameters**

**axis**

[{'x', 'y', 'both'}, default: 'both'] The axis to configure. Only major ticks are affected.

**style**

[{'sci', 'scientific', 'plain'}] Whether to use scientific notation. The formatter default is to use scientific notation. 'sci' is equivalent to 'scientific'.

**scilimits**

[pair of ints (m, n)] Scientific notation is used only for numbers outside the range $10^m$ to $10^n$ (and only if the formatter is configured to use scientific notation at all). Use (0, 0) to include all numbers. Use (m, m) where m != 0 to fix the order of magnitude to $10^m$. The formatter default is *rcParams["axes.formatter.limits"]* (default: `[-5, 6]`).

**useOffset**

[bool or float] If True, the offset is calculated as needed. If False, no offset is used. If a numeric value, it sets the offset. The formatter default is *rcParams["axes.formatter.useoffset"]* (default: `True`).

**useLocale**

[bool] Whether to format the number using the current locale or using the C (English) locale. This affects e.g. the decimal separator. The formatter default is *rcParams["axes.formatter.use_locale"]* (default: False).

**useMathText**

[bool] Render the offset and scientific notation in mathtext. The formatter default is *rcParams["axes.formatter.use_mathtext"]* (default: False).

**Raises**

**AttributeError**

If the current formatter is not a *ScalarFormatter*.

## Examples using `matplotlib.axes.Axes.ticklabel_format`

- *The default tick formatter*

## matplotlib.axes.Axes.tick_params

Axes.**tick_params**(*axis='both'*, ***kwargs*)

Change the appearance of ticks, tick labels, and gridlines.

Tick properties that are not explicitly set using the keyword arguments remain unchanged unless *reset* is True. For the current style settings, see *Axis.get_tick_params*.

**Parameters**

**axis**

[{'x', 'y', 'both'}, default: 'both'] The axis to which the parameters are applied.

**which**

[{'major', 'minor', 'both'}, default: 'major'] The group of ticks to which the parameters are applied.

**reset**

[bool, default: False] Whether to reset the ticks to defaults before updating them.

**Other Parameters**

**direction**

[{'in', 'out', 'inout'}] Puts ticks inside the Axes, outside the Axes, or both.

**length**

[float] Tick length in points.

**width**

[float] Tick width in points.

**color**

[*color*] Tick color.

**pad**

[float] Distance in points between tick and label.

**labelsize**

[float or str] Tick label font size in points or as a string (e.g., 'large').

**labelcolor**

[*color*] Tick label color.

**labelfontfamily**

[str] Tick label font.

**colors**

[*color*] Tick color and label color.

**zorder**

[float] Tick and label zorder.

**bottom, top, left, right**

[bool] Whether to draw the respective ticks.

**labelbottom, labeltop, labelleft, labelright**

[bool] Whether to draw the respective tick labels.

**labelrotation**

[float] Tick label rotation

**grid_color**

[*color*] Gridline color.

**grid_alpha**

[float] Transparency of gridlines: 0 (transparent) to 1 (opaque).

**grid_linewidth**

[float] Width of gridlines in points.

**grid_linestyle**

[str] Any valid `Line2D` line style spec.

### Examples

```
ax.tick_params(direction='out', length=6, width=2, colors='r',
               grid_color='r', grid_alpha=0.5)
```

This will make all major ticks be red, pointing out of the box, and with dimensions 6 points by 2 points. Tick labels will also be red. Gridlines will be red and translucent.

## Examples using `matplotlib.axes.Axes.tick_params`

- *Scatter plot with histograms*
- *Creating annotated heatmaps*
- *Image Masked*
- *Aligning Labels and Titles*
- *Axes Props*
- *Broken Axis*
- *Plots with different scales*
- *Polar legend*
- *Color Demo*
- *Demo Axes Grid*
- *Inset locator demo*
- *Inset locator demo 2*
- *Make room for ylabel using axes_grid*
- *Simple axes divider 3*
- *Anatomy of a figure*
- *Stock prices over 32 years*
- *Anscombe's quartet*
- *Multiple y-axis with Spines*
- *Centering labels between ticks*
- *Date tick locators and formatters*
- *Fig Axes Customize Simple*
- *Major and minor ticks*
- *Move x-axis tick labels to the top*
- *Axis ticks*

- *Text in Matplotlib*

### matplotlib.axes.Axes.locator_params

Axes.**locator_params**(*axis='both'*, *tight=None*, ***kwargs*)

Control behavior of major tick locators.

Because the locator is involved in autoscaling, *autoscale_view* is called automatically after the parameters are changed.

**Parameters**

**axis**

[{'both', 'x', 'y'}, default: 'both'] The axis on which to operate. (For 3D Axes, *axis* can also be set to 'z', and 'both' refers to all three axes.)

**tight**

[bool or None, optional] Parameter passed to *autoscale_view*. Default is None, for no change.

**Other Parameters**

****kwargs**

Remaining keyword arguments are passed to directly to the `set_params()` method of the locator. Supported keywords depend on the type of the locator. See for example *set_params* for the *ticker.MaxNLocator* used by default for linear.

#### Examples

When plotting small subplots, one might want to reduce the maximum number of ticks and use tight bounds, for example:

```
ax.locator_params(tight=True, nbins=4)
```

### Examples using `matplotlib.axes.Axes.locator_params`

- *Contourf demo*
- *Constrained layout guide*
- *Tight layout guide*

### Units

| | |
|---|---|
| *Axes.convert_xunits* | Convert *x* using the unit type of the xaxis. |
| *Axes.convert_yunits* | Convert *y* using the unit type of the yaxis. |
| *Axes.have_units* | Return whether units are set on any axis. |

#### matplotlib.axes.Axes.convert_xunits

Axes.**convert_xunits**(*x*)

Convert *x* using the unit type of the xaxis.

If the artist is not contained in an Axes or if the xaxis does not have units, *x* itself is returned.

#### matplotlib.axes.Axes.convert_yunits

Axes.**convert_yunits**(*y*)

Convert *y* using the unit type of the yaxis.

If the artist is not contained in an Axes or if the yaxis does not have units, *y* itself is returned.

#### matplotlib.axes.Axes.have_units

Axes.**have_units**()

Return whether units are set on any axis.

### Adding artists

| | |
|---|---|
| *Axes.add_artist* | Add an *Artist* to the Axes; return the artist. |
| *Axes.add_child_axes* | Add an AxesBase to the Axes' children; return the child Axes. |
| *Axes.add_collection* | Add a *Collection* to the Axes; return the collection. |
| *Axes.add_container* | Add a *Container* to the Axes' containers; return the container. |
| *Axes.add_image* | Add an *AxesImage* to the Axes; return the image. |
| *Axes.add_line* | Add a *Line2D* to the Axes; return the line. |
| *Axes.add_patch* | Add a *Patch* to the Axes; return the patch. |
| *Axes.add_table* | Add a *Table* to the Axes; return the table. |

### matplotlib.axes.Axes.add_artist

Axes.**add_artist**(*a*)

Add an *Artist* to the Axes; return the artist.

Use *add_artist* only for artists for which there is no dedicated "add" method; and if necessary, use a method such as *update_datalim* to manually update the dataLim if the artist is to be included in autoscaling.

If no `transform` has been specified when creating the artist (e.g. `artist.get_transform() == None`) then the transform is set to `ax.transData`.

### Examples using `matplotlib.axes.Axes.add_artist`

- *Scatter plots with a legend*
- *BboxImage Demo*
- *Bar of pie*
- *Annotating Plots*
- *AnnotationBbox demo*
- *Using a text as a Path*
- *Reference for Matplotlib artists*
- *Ellipse Demo*
- *Anchored Direction Arrow*
- *Axes Grid2*
- *Inset locator demo 2*
- *Simple Anchored Artists*
- *Anatomy of a figure*
- *Anchored Artists*
- *Artist tests*
- *Annotate Explain*
- *Simple Annotate01*
- *Simple Legend02*
- *Legend guide*
- *Annotations*

### matplotlib.axes.Axes.add_child_axes

Axes.**add_child_axes**(*ax*)

Add an AxesBase to the Axes' children; return the child Axes.

This is the lowlevel version. See *axes.Axes.inset_axes*.

### matplotlib.axes.Axes.add_collection

Axes.**add_collection**(*collection*, *autolim=True*)

Add a *Collection* to the Axes; return the collection.

### Examples using matplotlib.axes.Axes.add_collection

- *EventCollection Demo*
- *Creating boxes from error bars using PatchCollection*
- *Line, Poly and RegularPoly Collection with autoscaling*
- *Ellipse Collection*
- *Plotting multiple lines with a LineCollection*
- *Circles, Wedges and Polygons*
- *Lasso Demo*
- *Artist tests*
- *Autoscaling Axis*

### matplotlib.axes.Axes.add_container

Axes.**add_container**(*container*)

Add a *Container* to the Axes' containers; return the container.

### matplotlib.axes.Axes.add_image

Axes.**add_image**(*image*)

Add an *AxesImage* to the Axes; return the image.

### Examples using `matplotlib.axes.Axes.add_image`

- *Image nonuniform*

### matplotlib.axes.Axes.add_line

`Axes.`**`add_line`**`(`*line*`)`

Add a `Line2D` to the Axes; return the line.

### Examples using `matplotlib.axes.Axes.add_line`

- *Artist within an artist*
- *Artist tests*
- *Artist tutorial*

### matplotlib.axes.Axes.add_patch

`Axes.`**`add_patch`**`(`*p*`)`

Add a `Patch` to the Axes; return the patch.

### Examples using `matplotlib.axes.Axes.add_patch`

- *Curve with error band*
- *Many ways to plot images*
- *Axes box aspect*
- *Controlling view limits using margins and sticky_edges*
- *Boxplots*
- *Plot a confidence ellipse of a two-dimensional dataset*
- *Angle annotations on bracket arrows*
- *Annotating Plots*
- *Text alignment*
- *Compound path*
- *Dolphins*
- *Mmh Donuts!!!*
- *Ellipse with orientation arrow demo*

- *Drawing fancy boxes*
- *Hatch demo*
- *Hatch style reference*
- *PathPatch object*
- *Bezier Curve*
- *ggplot style sheet*
- *Inset locator demo*
- *Firefox*
- *Integral as the area under a curve*
- *Looking Glass*
- *Path editor*
- *Poly Editor*
- *Trifinder Event Demo*
- *Viewlims*
- *Changing colors of lines intersecting a box*
- *Building histograms using Rectangles and PolyCollections*
- *Packed-bubble chart*
- *SVG filter pie*
- *TickedStroke patheffect*
- *Draw flat objects in 3D plot*
- *Ishikawa Diagram*
- *Artist tests*
- *Ellipse with units*
- *Artist tutorial*
- *Path Tutorial*
- *Transformations Tutorial*
- *Legend guide*
- *Specifying colors*
- *Text properties and layout*
- *Annotations*

### matplotlib.axes.Axes.add_table

Axes.**add_table**(*tab*)

Add a *Table* to the Axes; return the table.

### Twinning and sharing

| | |
|---|---|
| *Axes.twinx* | Create a twin Axes sharing the xaxis. |
| *Axes.twiny* | Create a twin Axes sharing the yaxis. |
| *Axes.sharex* | Share the x-axis with *other*. |
| *Axes.sharey* | Share the y-axis with *other*. |
| *Axes.get_shared_x_axes* | Return an immutable view on the shared x-axes Grouper. |
| *Axes.get_shared_y_axes* | Return an immutable view on the shared y-axes Grouper. |

### matplotlib.axes.Axes.twinx

Axes.**twinx**()

Create a twin Axes sharing the xaxis.

Create a new Axes with an invisible x-axis and an independent y-axis positioned opposite to the original one (i.e. at right). The x-axis autoscale setting will be inherited from the original Axes. To ensure that the tick marks of both y-axes align, see *LinearLocator*.

**Returns**

**Axes**

The newly created Axes instance

#### Notes

For those who are 'picking' artists while using twinx, pick events are only called for the artists in the top-most Axes.

### Examples using `matplotlib.axes.Axes.twinx`

- *Axes box aspect*
- *Plots with different scales*
- *Percentiles as horizontal bar chart*
- *Parasite Simple*
- *Parasite axis demo*
- *Pan/zoom events of overlapping axes*
- *Multiple y-axis with Spines*
- *Quick start guide*

## matplotlib.axes.Axes.twiny

Axes.**twiny**()

Create a twin Axes sharing the yaxis.

Create a new Axes with an invisible y-axis and an independent x-axis positioned opposite to the original one (i.e. at top). The y-axis autoscale setting will be inherited from the original Axes. To ensure that the tick marks of both x-axes align, see *LinearLocator*.

**Returns**

**Axes**

The newly created Axes instance

### Notes

For those who are 'picking' artists while using twiny, pick events are only called for the artists in the top-most Axes.

### Examples using `matplotlib.axes.Axes.twiny`

- *Plots with different scales*

### matplotlib.axes.Axes.sharex

Axes.**sharex**(*other*)

Share the x-axis with *other*.

This is equivalent to passing `sharex=other` when constructing the Axes, and cannot be used if the x-axis is already being shared with another Axes.

### Examples using `matplotlib.axes.Axes.sharex`

- *Power spectral density (PSD)*

### matplotlib.axes.Axes.sharey

Axes.**sharey**(*other*)

Share the y-axis with *other*.

This is equivalent to passing `sharey=other` when constructing the Axes, and cannot be used if the y-axis is already being shared with another Axes.

### Examples using `matplotlib.axes.Axes.sharey`

- *Power spectral density (PSD)*

### matplotlib.axes.Axes.get_shared_x_axes

Axes.**get_shared_x_axes**()

Return an immutable view on the shared x-axes Grouper.

### matplotlib.axes.Axes.get_shared_y_axes

Axes.**get_shared_y_axes**()

Return an immutable view on the shared y-axes Grouper.

### Axes position

| | |
|---|---|
| *Axes.get_anchor* | Get the anchor location. |
| *Axes.set_anchor* | Define the anchor location. |
| *Axes.get_axes_locator* | Return the axes_locator. |
| *Axes.set_axes_locator* | Set the Axes locator. |
| *Axes.get_subplotspec* | Return the *SubplotSpec* associated with the subplot, or None. |
| *Axes.set_subplotspec* | Set the *SubplotSpec*. |
| *Axes.reset_position* | Reset the active position to the original position. |
| *Axes.get_position* | Return the position of the Axes within the figure as a *Bbox*. |
| *Axes.set_position* | Set the Axes position. |

## matplotlib.axes.Axes.get_anchor

Axes.**get_anchor**()

Get the anchor location.

**See also:**

***matplotlib.axes.Axes.set_anchor***

for a description of the anchor.

***matplotlib.axes.Axes.set_aspect***

for a description of aspect handling.

## matplotlib.axes.Axes.set_anchor

Axes.**set_anchor**(*anchor*, *share=False*)

Define the anchor location.

The actual drawing area (active position) of the Axes may be smaller than the Bbox (original position) when a fixed aspect is required. The anchor defines where the drawing area will be located within the available space.

**Parameters**

**anchor**

[(float, float) or {'C', 'SW', 'S', 'SE', 'E', 'NE', ...}] Either an (*x*, *y*) pair of relative coordinates (0 is left or bottom, 1 is right or top), 'C' (center), or a cardinal direction ('SW', southwest, is bottom left, etc.). str inputs are shorthands for (*x*, *y*) coordinates, as shown in the following diagram:

```
┌──────────────────┬──────────────────┬──────────────────┐
│ 'NW' (0.0, 1.0)  │ 'N' (0.5, 1.0)   │ 'NE' (1.0, 1.0)  │
├──────────────────┼──────────────────┼──────────────────┤
│ 'W'  (0.0, 0.5)  │ 'C' (0.5, 0.5)   │ 'E'  (1.0, 0.5)  │
├──────────────────┼──────────────────┼──────────────────┤
│ 'SW' (0.0, 0.0)  │ 'S' (0.5, 0.0)   │ 'SE' (1.0, 0.0)  │
└──────────────────┴──────────────────┴──────────────────┘
```

**share**

[bool, default: False] If `True`, apply the settings to all shared Axes.

**See also:**

***matplotlib.axes.Axes.set_aspect***

for a description of aspect handling.

## matplotlib.axes.Axes.get_axes_locator

`Axes.`**`get_axes_locator`**`()`

Return the axes_locator.

## matplotlib.axes.Axes.set_axes_locator

`Axes.`**`set_axes_locator`**`(`*`locator`*`)`

Set the Axes locator.

**Parameters**

**locator**

[Callable[[Axes, Renderer], Bbox]]

## Examples using `matplotlib.axes.Axes.set_axes_locator`

- *HBoxDivider and VBoxDivider demo*

### matplotlib.axes.Axes.get_subplotspec

Axes.**get_subplotspec**()

Return the *SubplotSpec* associated with the subplot, or None.

### Examples using `matplotlib.axes.Axes.get_subplotspec`

- *Nested GridSpecs*
- *Arranging multiple Axes in a Figure*

### matplotlib.axes.Axes.set_subplotspec

Axes.**set_subplotspec**(*subplotspec*)

Set the *SubplotSpec*. associated with the subplot.

### matplotlib.axes.Axes.reset_position

Axes.**reset_position**()

Reset the active position to the original position.

This undoes changes to the active position (as defined in *set_position*) which may have been performed to satisfy fixed-aspect constraints.

### matplotlib.axes.Axes.get_position

Axes.**get_position**(*original=False*)

Return the position of the Axes within the figure as a *Bbox*.

**Parameters**

**original**

[bool] If `True`, return the original position. Otherwise, return the active position. For an explanation of the positions see *set_position*.

**Returns**

***Bbox***

### Examples using `matplotlib.axes.Axes.get_position`

- *Contour Demo*

## matplotlib.axes.Axes.set_position

`Axes.`**`set_position`**(*pos*, *which='both'*)

Set the Axes position.

Axes have two position attributes. The 'original' position is the position allocated for the Axes. The 'active' position is the position the Axes is actually drawn at. These positions are usually the same unless a fixed aspect is set to the Axes. See *`Axes.set_aspect`* for details.

**Parameters**

**pos**

[[left, bottom, width, height] or *`Bbox`*] The new position of the Axes in *`Figure`* coordinates.

**which**

[{'both', 'active', 'original'}, default: 'both'] Determines which position variables to change.

**See also:**

***`matplotlib.transforms.Bbox.from_bounds`***
***`matplotlib.transforms.Bbox.from_extents`***

### Examples using `matplotlib.axes.Axes.set_position`

- *Contour Demo*

### Async/event based

| | |
|---|---|
| *`Axes.stale`* | Whether the artist is 'stale' and needs to be re-drawn for the output to match the internal state of the artist. |
| *`Axes.pchanged`* | Call all of the registered callbacks. |
| *`Axes.add_callback`* | Add a callback function that will be called whenever one of the *`Artist`*'s properties changes. |
| *`Axes.remove_callback`* | Remove a callback based on its observer id. |

### matplotlib.axes.Axes.stale

**property** Axes.**stale**

Whether the artist is 'stale' and needs to be re-drawn for the output to match the internal state of the artist.

### matplotlib.axes.Axes.pchanged

Axes.**pchanged**()

Call all of the registered callbacks.

This function is triggered internally when a property is changed.

**See also:**

*add_callback*
*remove_callback*

### matplotlib.axes.Axes.add_callback

Axes.**add_callback**(*func*)

Add a callback function that will be called whenever one of the *Artist*'s properties changes.

**Parameters**

**func**

[callable] The callback function. It must have the signature:

```
def func(artist: Artist) -> Any
```

where *artist* is the calling *Artist*. Return values may exist but are ignored.

**Returns**

**int**

The observer id associated with the callback. This id can be used for removing the callback with *remove_callback* later.

**See also:**

*remove_callback*

### matplotlib.axes.Axes.remove_callback

Axes.**remove_callback**(*oid*)

Remove a callback based on its observer id.

**See also:**

***add_callback***

### Interactive

| | |
|---|---|
| *Axes.can_pan* | Return whether this Axes supports any pan/zoom button functionality. |
| *Axes.can_zoom* | Return whether this Axes supports the zoom box button functionality. |
| *Axes.get_navigate* | Get whether the Axes responds to navigation commands. |
| *Axes.set_navigate* | Set whether the Axes responds to navigation toolbar commands. |
| *Axes.get_navigate_mode* | Get the navigation toolbar button status: 'PAN', 'ZOOM', or None. |
| *Axes.set_navigate_mode* | Set the navigation toolbar button status. |
| *Axes.get_forward_navigation_events* | Get how pan/zoom events are forwarded to Axes below this one. |
| *Axes.set_forward_navigation_events* | Set how pan/zoom events are forwarded to Axes below this one. |
| *Axes.start_pan* | Called when a pan operation has started. |
| *Axes.drag_pan* | Called when the mouse moves during a pan operation. |
| *Axes.end_pan* | Called when a pan operation completes (when the mouse button is up.) |
| *Axes.format_coord* | Return a format string formatting the *x*, *y* coordinates. |
| *Axes.format_cursor_data* | Return a string representation of *data*. |
| *Axes.format_xdata* | Return *x* formatted as an x-value. |
| *Axes.format_ydata* | Return *y* formatted as a y-value. |
| *Axes.mouseover* | Return whether this artist is queried for custom context information when the mouse cursor moves over it. |
| *Axes.in_axes* | Return whether the given event (in display coords) is in the Axes. |
| *Axes.contains* | Test whether the artist contains the mouse event. |
| *Axes.contains_point* | Return whether *point* (pair of pixel coordinates) is inside the Axes patch. |
| *Axes.get_cursor_data* | Return the cursor data for a given event. |

### matplotlib.axes.Axes.can_pan

Axes.**can_pan**()

Return whether this Axes supports any pan/zoom button functionality.

### matplotlib.axes.Axes.can_zoom

Axes.**can_zoom**()

Return whether this Axes supports the zoom box button functionality.

### matplotlib.axes.Axes.get_navigate

Axes.**get_navigate**()

Get whether the Axes responds to navigation commands.

### matplotlib.axes.Axes.set_navigate

Axes.**set_navigate**(*b*)

Set whether the Axes responds to navigation toolbar commands.

**Parameters**

**b**

[bool]

**See also:**

*matplotlib.axes.Axes.set_forward_navigation_events*

### matplotlib.axes.Axes.get_navigate_mode

Axes.**get_navigate_mode**()

Get the navigation toolbar button status: 'PAN', 'ZOOM', or None.

### matplotlib.axes.Axes.set_navigate_mode

Axes.**set_navigate_mode**(*b*)

Set the navigation toolbar button status.

> **Warning:** This is not a user-API function.

### matplotlib.axes.Axes.get_forward_navigation_events

Axes.**get_forward_navigation_events**()

Get how pan/zoom events are forwarded to Axes below this one.

### matplotlib.axes.Axes.set_forward_navigation_events

Axes.**set_forward_navigation_events**(*forward*)

Set how pan/zoom events are forwarded to Axes below this one.

**Parameters**

**forward**

[bool or "auto"] Possible values:

- True: Forward events to other axes with lower or equal zorder.
- False: Events are only executed on this axes.
- "auto": Default behaviour (*True* for axes with an invisible patch and *False* otherwise)

**See also:**

***matplotlib.axes.Axes.set_navigate***

### Examples using `matplotlib.axes.Axes.set_forward_navigation_events`

- *Pan/zoom events of overlapping axes*

### matplotlib.axes.Axes.start_pan

Axes.**start_pan**(*x*, *y*, *button*)

Called when a pan operation has started.

**Parameters**

**x, y**

[float] The mouse coordinates in display coords.

**button**

[*MouseButton*] The pressed mouse button.

#### Notes

This is intended to be overridden by new projection types.

### matplotlib.axes.Axes.drag_pan

Axes.**drag_pan**(*button*, *key*, *x*, *y*)

Called when the mouse moves during a pan operation.

**Parameters**

**button**

[*MouseButton*] The pressed mouse button.

**key**

[str or None] The pressed key, if any.

**x, y**

[float] The mouse coordinates in display coords.

#### Notes

This is intended to be overridden by new projection types.

### matplotlib.axes.Axes.end_pan

Axes.**end_pan**()

Called when a pan operation completes (when the mouse button is up.)

#### Notes

This is intended to be overridden by new projection types.

### matplotlib.axes.Axes.format_coord

Axes.**format_coord**(*x*, *y*)

Return a format string formatting the *x*, *y* coordinates.

### Examples using `matplotlib.axes.Axes.format_coord`

- *Modifying the coordinate formatter*

### matplotlib.axes.Axes.format_cursor_data

Axes.**format_cursor_data**(*data*)

Return a string representation of *data*.

---

**Note:** This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

---

The default implementation converts ints and floats and arrays of ints and floats into a comma-separated string enclosed in square brackets, unless the artist has an associated colorbar, in which case scalar values are formatted using the colorbar's formatter.

**See also:**

***get_cursor_data***

### matplotlib.axes.Axes.format_xdata

Axes.**format_xdata**(*x*)

Return *x* formatted as an x-value.

This function will use the `fmt_xdata` attribute if it is not None, else will fall back on the xaxis major formatter.

### matplotlib.axes.Axes.format_ydata

Axes.**format_ydata**(*y*)

Return *y* formatted as a y-value.

This function will use the `fmt_ydata` attribute if it is not None, else will fall back on the yaxis major formatter.

### matplotlib.axes.Axes.mouseover

**property** Axes.**mouseover**

Return whether this artist is queried for custom context information when the mouse cursor moves over it.

### matplotlib.axes.Axes.in_axes

Axes.**in_axes**(*mouseevent*)

Return whether the given event (in display coords) is in the Axes.

### matplotlib.axes.Axes.contains

Axes.**contains**(*mouseevent*)

Test whether the artist contains the mouse event.

**Parameters**

**mouseevent**

[*MouseEvent*]

**Returns**

**contains**

[bool] Whether any values are within the radius.

**details**

[dict] An artist-specific dictionary of details of the event context, such as which points are contained in the pick radius. See the individual Artist subclasses for details.

### matplotlib.axes.Axes.contains_point

Axes.**contains_point**(*point*)

Return whether *point* (pair of pixel coordinates) is inside the Axes patch.

### matplotlib.axes.Axes.get_cursor_data

Axes.**get_cursor_data**(*event*)

Return the cursor data for a given event.

---

**Note:** This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

---

Cursor data can be used by Artists to provide additional context information for a given event. The default implementation just returns *None*.

Subclasses can override the method and return arbitrary data. However, when doing so, they must ensure that *format_cursor_data* can convert the data to a string representation.

The only current use case is displaying the z-value of an *AxesImage* in the status bar of a plot window, while moving the mouse.

**Parameters**

**event**

[*MouseEvent*]

**See also:**

***format_cursor_data***

### Children

| | |
|---|---|
| *Axes.get_children* | Return a list of the child *Artist*s of this *Artist*. |
| *Axes.get_images* | Return a list of *AxesImage*s contained by the Axes. |
| *Axes.get_lines* | Return a list of lines contained by the Axes. |
| *Axes.findobj* | Find artist objects. |

### matplotlib.axes.Axes.get_children

Axes.**get_children**()

Return a list of the child *Artist*s of this *Artist*.

### matplotlib.axes.Axes.get_images

Axes.**get_images**()

Return a list of *AxesImage*s contained by the Axes.

### matplotlib.axes.Axes.get_lines

Axes.**get_lines**()

Return a list of lines contained by the Axes.

### matplotlib.axes.Axes.findobj

Axes.**findobj**(*match=None*, *include_self=True*)

Find artist objects.

Recursively find all *Artist* instances contained in the artist.

**Parameters**

**match**

A filter criterion for the matches. This can be

- *None*: Return all objects contained in artist.
- A function with signature `def match(artist: Artist) -> bool`. The result will only contain artists for which the function returns *True*.
- A class instance: e.g., *Line2D*. The result will only contain artists of this class or its subclasses (`isinstance` check).

**include_self**

[bool] Include *self* in the list to be checked for a match.

**Returns**

**list of *Artist***

## Drawing

| | |
|---|---|
| *Axes.draw* | Draw the Artist (and its children) using the given renderer. |
| *Axes.draw_artist* | Efficiently redraw a single artist. |
| *Axes.redraw_in_frame* | Efficiently redraw Axes data, but not axis ticks, labels, etc. |
| *Axes.get_rasterization_zorder* | Return the zorder value below which artists will be rasterized. |
| *Axes.set_rasterization_zorder* | Set the zorder threshold for rasterization for vector graphics output. |
| *Axes.get_window_extent* | Return the Axes bounding box in display space. |
| *Axes.get_tightbbox* | Return the tight bounding box of the Axes, including axis and their decorators (xlabel, title, etc). |

## matplotlib.axes.Axes.draw

Axes.**draw**(*renderer*)

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

**Parameters**

**renderer**

[*RendererBase* subclass.]

### Notes

This method is overridden in the Artist subclasses.

### matplotlib.axes.Axes.draw_artist

Axes.**draw_artist**(*a*)

Efficiently redraw a single artist.

### Examples using `matplotlib.axes.Axes.draw_artist`

- *Faster rendering by using blitting*

### matplotlib.axes.Axes.redraw_in_frame

Axes.**redraw_in_frame**()

Efficiently redraw Axes data, but not axis ticks, labels, etc.

### matplotlib.axes.Axes.get_rasterization_zorder

Axes.**get_rasterization_zorder**()

Return the zorder value below which artists will be rasterized.

### matplotlib.axes.Axes.set_rasterization_zorder

Axes.**set_rasterization_zorder**(*z*)

Set the zorder threshold for rasterization for vector graphics output.

All artists with a zorder below the given value will be rasterized if they support rasterization.

This setting is ignored for pixel-based output.

See also *Rasterization for vector graphics*.

**Parameters**

**z**

[float or None] The zorder below which artists are rasterized. If `None` rasterization based on zorder is deactivated.

### Examples using `matplotlib.axes.Axes.set_rasterization_zorder`

- *Rasterization for vector graphics*

## matplotlib.axes.Axes.get_window_extent

`Axes.`**`get_window_extent`**(*renderer=None*)

Return the Axes bounding box in display space.

This bounding box does not include the spines, ticks, ticklabels, or other labels. For a bounding box including these elements use *`get_tightbbox`*.

**See also:**

***matplotlib.axes.Axes.get_tightbbox***
***matplotlib.axis.Axis.get_tightbbox***
***matplotlib.spines.Spine.get_window_extent***

### Examples using `matplotlib.axes.Axes.get_window_extent`

- *Annotations*

## matplotlib.axes.Axes.get_tightbbox

`Axes.`**`get_tightbbox`**(*renderer=None*, *, *call_axes_locator=True*, *bbox_extra_artists=None*, *for_layout_only=False*)

Return the tight bounding box of the Axes, including axis and their decorators (xlabel, title, etc).

Artists that have `artist.set_in_layout(False)` are not included in the bbox.

**Parameters**

**renderer**

[*`RendererBase`* subclass] renderer that will be used to draw the figures (i.e. `fig.canvas.get_renderer()`)

**bbox_extra_artists**

[list of *`Artist`* or `None`] List of artists to include in the tight bounding box. If `None` (default), then all artist children of the Axes are included in the tight bounding box.

**call_axes_locator**

[bool, default: True] If *call_axes_locator* is `False`, it does not call the `_axes_locator` attribute, which is necessary to get the correct bounding box.

`call_axes_locator=False` can be used if the caller is only interested in the relative size of the tightbbox compared to the Axes bbox.

**for_layout_only**

[default: False] The bounding box will *not* include the x-extent of the title and the xlabel, or the y-extent of the ylabel.

**Returns**

***BboxBase***

Bounding box in figure pixel coordinates.

**See also:**

***matplotlib.axes.Axes.get_window_extent***
***matplotlib.axis.Axis.get_tightbbox***
***matplotlib.spines.Spine.get_window_extent***

## Projection

Methods used by *Axis* that must be overridden for non-rectilinear Axes.

| | |
|---|---|
| *Axes.name* | |
| *Axes.get_xaxis_transform* | Get the transformation used for drawing x-axis labels, ticks and gridlines. |
| *Axes.get_yaxis_transform* | Get the transformation used for drawing y-axis labels, ticks and gridlines. |
| *Axes.get_data_ratio* | Return the aspect ratio of the scaled data. |
| *Axes.get_xaxis_text1_transform* | **Returns** |
| *Axes.get_xaxis_text2_transform* | **Returns** |
| *Axes.get_yaxis_text1_transform* | **Returns** |
| *Axes.get_yaxis_text2_transform* | **Returns** |

### matplotlib.axes.Axes.name

Axes.**name = 'rectilinear'**

### matplotlib.axes.Axes.get_xaxis_transform

Axes.**get_xaxis_transform**(*which='grid'*)

Get the transformation used for drawing x-axis labels, ticks and gridlines. The x-direction is in data coordinates and the y-direction is in axis coordinates.

---

**Note:** This transformation is primarily used by the *Axis* class, and is meant to be overridden by new kinds of projections that may need to place axis elements in different locations.

---

**Parameters**

**which**

[{'grid', 'tick1', 'tick2'}]

### Examples using matplotlib.axes.Axes.get_xaxis_transform

- *Filling the area between lines*
- *hlines and vlines*
- *Boxplots*
- *Scale invariant angle label*
- *Centered spines with arrows*

### matplotlib.axes.Axes.get_yaxis_transform

Axes.**get_yaxis_transform**(*which='grid'*)

Get the transformation used for drawing y-axis labels, ticks and gridlines. The x-direction is in axis coordinates and the y-direction is in data coordinates.

---

**Note:** This transformation is primarily used by the *Axis* class, and is meant to be overridden by new kinds of projections that may need to place axis elements in different locations.

---

**Parameters**

**which**

[{'grid', 'tick1', 'tick2'}]

### Examples using `matplotlib.axes.Axes.get_yaxis_transform`

- *Centered spines with arrows*

## matplotlib.axes.Axes.get_data_ratio

`Axes.get_data_ratio()`

Return the aspect ratio of the scaled data.

### Notes

This method is intended to be overridden by new projection types.

## matplotlib.axes.Axes.get_xaxis_text1_transform

`Axes.get_xaxis_text1_transform(pad_points)`

**Returns**

**transform**

[Transform] The transform used for drawing x-axis labels, which will add *pad_points* of padding (in points) between the axis and the label. The x-direction is in data coordinates and the y-direction is in axis coordinates

**valign**

[{'center', 'top', 'bottom', 'baseline', 'center_baseline'}] The text vertical alignment.

**halign**

[{'center', 'left', 'right'}] The text horizontal alignment.

### Notes

This transformation is primarily used by the *Axis* class, and is meant to be overridden by new kinds of projections that may need to place axis elements in different locations.

### matplotlib.axes.Axes.get_xaxis_text2_transform

Axes.**get_xaxis_text2_transform**(*pad_points*)

**Returns**

**transform**

[Transform] The transform used for drawing secondary x-axis labels, which will add *pad_points* of padding (in points) between the axis and the label. The x-direction is in data coordinates and the y-direction is in axis coordinates

**valign**

[{'center', 'top', 'bottom', 'baseline', 'center_baseline'}] The text vertical alignment.

**halign**

[{'center', 'left', 'right'}] The text horizontal alignment.

#### Notes

This transformation is primarily used by the *Axis* class, and is meant to be overridden by new kinds of projections that may need to place axis elements in different locations.

### matplotlib.axes.Axes.get_yaxis_text1_transform

Axes.**get_yaxis_text1_transform**(*pad_points*)

**Returns**

**transform**

[Transform] The transform used for drawing y-axis labels, which will add *pad_points* of padding (in points) between the axis and the label. The x-direction is in axis coordinates and the y-direction is in data coordinates

**valign**

[{'center', 'top', 'bottom', 'baseline', 'center_baseline'}] The text vertical alignment.

**halign**

[{'center', 'left', 'right'}] The text horizontal alignment.

### Notes

This transformation is primarily used by the *Axis* class, and is meant to be overridden by new kinds of projections that may need to place axis elements in different locations.

## matplotlib.axes.Axes.get_yaxis_text2_transform

Axes.**get_yaxis_text2_transform**(*pad_points*)

**Returns**

**transform**

[Transform] The transform used for drawing secondart y-axis labels, which will add *pad_points* of padding (in points) between the axis and the label. The x-direction is in axis coordinates and the y-direction is in data coordinates

**valign**

[{'center', 'top', 'bottom', 'baseline', 'center_baseline'}] The text vertical alignment.

**halign**

[{'center', 'left', 'right'}] The text horizontal alignment.

### Notes

This transformation is primarily used by the *Axis* class, and is meant to be overridden by new kinds of projections that may need to place axis elements in different locations.

### Other

| | |
|---|---|
| *Axes.zorder* | |
| *Axes.get_default_bbox_extra_artists* | Return a default list of artists that are used for the bounding box calculation. |
| *Axes.get_transformed_clip_path_and_* | Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation. |
| *Axes.has_data* | Return whether any artists have been added to the Axes. |
| *Axes.set* | Set multiple properties at once. |

### matplotlib.axes.Axes.zorder

Axes.**zorder = 0**

### matplotlib.axes.Axes.get_default_bbox_extra_artists

Axes.**get_default_bbox_extra_artists**()

Return a default list of artists that are used for the bounding box calculation.

Artists are excluded either by not being visible or `artist.set_in_layout(False)`.

### matplotlib.axes.Axes.get_transformed_clip_path_and_affine

Axes.**get_transformed_clip_path_and_affine**()

Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

### matplotlib.axes.Axes.has_data

Axes.**has_data**()

Return whether any artists have been added to the Axes.

This should not be used to determine whether the *dataLim* need to be updated, and may not actually be useful for anything.

### matplotlib.axes.Axes.set

Axes.**set**(*, *adjustable=<UNSET>*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *anchor=<UNSET>*, *animated=<UNSET>*, *aspect=<UNSET>*, *autoscale_on=<UNSET>*, *autoscalex_on=<UNSET>*, *autoscaley_on=<UNSET>*, *axes_locator=<UNSET>*, *axisbelow=<UNSET>*, *box_aspect=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *facecolor=<UNSET>*, *forward_navigation_events=<UNSET>*, *frame_on=<UNSET>*, *gid=<UNSET>*, *in_layout=<UNSET>*, *label=<UNSET>*, *mouseover=<UNSET>*, *navigate=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *position=<UNSET>*, *prop_cycle=<UNSET>*, *rasterization_zorder=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *subplotspec=<UNSET>*, *title=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *xbound=<UNSET>*, *xlabel=<UNSET>*, *xlim=<UNSET>*, *xmargin=<UNSET>*, *xscale=<UNSET>*, *xticklabels=<UNSET>*, *xticks=<UNSET>*, *ybound=<UNSET>*, *ylabel=<UNSET>*, *ylim=<UNSET>*, *ymargin=<UNSET>*, *yscale=<UNSET>*, *yticklabels=<UNSET>*, *yticks=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *adjustable* | {'box', 'datalim'} |
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and |
| *alpha* | scalar or None |
| *anchor* | (float, float) or {'C', 'SW', 'S', 'SE', 'E', 'NE', ...} |
| *animated* | bool |
| *aspect* | {'auto', 'equal'} or float |
| *autoscale_on* | bool |
| *autoscalex_on* | unknown |
| *autoscaley_on* | unknown |
| *axes_locator* | Callable[[Axes, Renderer], Bbox] |
| *axisbelow* | bool or 'line' |
| *box_aspect* | float or None |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *facecolor* or fc | *color* |
| *figure* | *Figure* |
| *forward_navigation_events* | bool or "auto" |
| *frame_on* | bool |
| *gid* | str |
| *in_layout* | bool |
| *label* | object |
| *mouseover* | bool |
| *navigate* | bool |
| *navigate_mode* | unknown |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *position* | [left, bottom, width, height] or *Bbox* |
| *prop_cycle* | *Cycler* |
| *rasterization_zorder* | float or None |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *subplotspec* | unknown |
| *title* | str |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *xbound* | (lower: float, upper: float) |
| *xlabel* | str |
| *xlim* | (left: float, right: float) |
| *xmargin* | float greater than -0.5 |
| *xscale* | unknown |
| *xticklabels* | unknown |

Table 32 – continued from previou

| Property | Description |
|---|---|
| *xticks* | unknown |
| *ybound* | (lower: float, upper: float) |
| *ylabel* | str |
| *ylim* | (bottom: float, top: float) |
| *ymargin* | float greater than -0.5 |
| *yscale* | unknown |
| *yticklabels* | unknown |
| *yticks* | unknown |
| *zorder* | float |

### Examples using `matplotlib.axes.Axes.set`

- *Bar Label Demo*
- *Curve with error band*
- *Bar chart with gradients*
- *Scatter plot with histograms*
- *Simple Plot*
- *Creating a timeline with lines, dates, and text*
- *Contour plot of irregularly spaced data*
- *Axes Demo*
- *Creating multiple subplots using plt.subplots*
- *Boxplots*
- *Hexagonal binned plot*
- *Nested pie charts*
- *Angle annotations on bracket arrows*
- *Annotating Plots*
- *Arrow Demo*
- *Text Commands*
- *Usetex Baseline Test*
- *Reference for Matplotlib artists*
- *Ellipse Demo*
- *Drawing fancy boxes*

- *Demo Axes Grid*
- *Inset locator demo*
- *axis_direction demo*
- *Parasite Axes demo*
- *Parasite axis demo*
- *Animated 3D random walk*
- *Pick event demo*
- *Viewlims*
- *Zoom Window*
- *Manual Contour*
- *Plotting with keywords*
- *3D box surface plot*
- *Project contour profiles onto a graph*
- *Project filled contour onto a graph*
- *Generate polygons to fill under 3D line graph*
- *3D stem*
- *3D voxel / volumetric plot with RGB colors*
- *Log Demo*
- *Multiple y-axis with Spines*
- *Mouse Cursor*
- *Annotate Explain*
- *Connection styles for annotations*
- *Nested GridSpecs*
- *Simple Annotate01*
- *The Lifecycle of a Plot*
- *plot(x, y)*
- *scatter(x, y)*
- *bar(x, height)*
- *stem(x, y)*
- *fill_between(x, y1, y2)*
- *stackplot(x, y)*
- *stairs(values)*

- *hist(x)*
- *boxplot(X)*
- *errorbar(x, y, yerr, xerr)*
- *violinplot(D)*
- *eventplot(D)*
- *hist2d(x, y)*
- *hexbin(x, y, C)*
- *pie(x)*
- *barbs(X, Y, U, V)*
- *quiver(X, Y, U, V)*
- *tricontour(x, y, z)*
- *tricontourf(x, y, z)*
- *tripcolor(x, y, z)*
- *triplot(x, y)*
- *scatter(xs, ys, zs)*
- *plot_surface(X, Y, Z)*
- *plot_trisurf(x, y, z)*
- *voxels([x, y, z], filled)*
- *plot_wireframe(X, Y, Z)*
- *Animations using Matplotlib*
- *Arranging multiple Axes in a Figure*
- *Annotations*

**class** matplotlib.axes.Axes.**ArtistList**(*axes*, *prop_name*, *valid_types=None*, *invalid_types=None*)

A sublist of Axes children based on their type.

The type-specific children sublists were made immutable in Matplotlib 3.7. In the future these artist lists may be replaced by tuples. Use as if this is a tuple already.

**Parameters**

**axes**

[*Axes*] The Axes from which this sublist will pull the children Artists.

**prop_name**

[str] The property name used to access this sublist from the Axes; used to generate deprecation warnings.

**valid_types**

[list of type, optional] A list of types that determine which children will be returned by this sublist. If specified, then the Artists in the sublist must be instances of any of these types. If unspecified, then any type of Artist is valid (unless limited by *invalid_types*.)

**invalid_types**

[tuple, optional] A list of types that determine which children will *not* be returned by this sublist. If specified, then Artists in the sublist will never be an instance of these types. Otherwise, no types will be excluded.

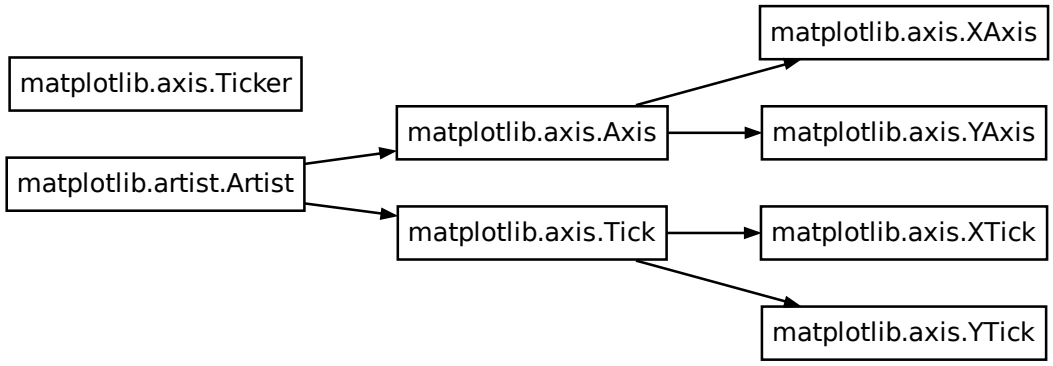
### 7.2.6 `matplotlib.axis`

**Table of Contents**


- *Inheritance*
- *`Axis` objects*
  - *Formatters and Locators*
  - *Axis Label*
  - *Ticks, tick labels and Offset text*
  - *Data and view intervals*
  - *Rendering helpers*
  - *Interactive*
  - *Units*
  - *XAxis Specific*
  - *YAxis Specific*
  - *Other*
  - *Discouraged*
- *`Tick` objects*


Classes for the ticks and x- and y-axis.

### Inheritance




### `Axis` objects

**class** matplotlib.axis.**Axis**(*axes*, *, *pickradius=15*, *clear=True*)

Base class for *XAxis* and *YAxis*.

**Attributes**

**isDefault_label**

[bool]

***axes***

[*Axes*] The *Axes* instance the artist resides in, or *None*.

**major**

[*Ticker*] Determines the major tick positions and their label format.

**minor**

[*Ticker*] Determines the minor tick positions and their label format.

**callbacks**

[*CallbackRegistry*]

**label**

[*Text*] The axis label.

**labelpad**

[float] The distance between the axis label and the tick labels. Defaults to *rcParams["axes.labelpad"]* (default: `4.0`) = 4.

**offsetText**

[*Text*] A *Text* object containing the data offset of the ticks (if any).

***pickradius***

[float] The acceptance radius for containment tests.

**majorTicks**

[list of *Tick*] The major ticks.

> **Warning:** Ticks are not guaranteed to be persistent. Various operations can create, delete and modify the Tick instances. There is an imminent risk that changes to individual ticks will not survive if you work on the figure further (including also panning/zooming on a displayed figure).
>
> Working on the individual ticks is a method of last resort. Use *set_tick_params* instead if possible.

**minorTicks**

[list of *Tick*] The minor ticks.

**Parameters**

**axes**

[*Axes*] The *Axes* to which the created Axis belongs.

**pickradius**

[float] The acceptance radius for containment tests. See also *Axis.contains*.

**clear**

[bool, default: True] Whether to clear the Axis on creation. This is not required, e.g., when creating an Axis as part of an Axes, as `Axes.clear` will call `Axis.clear`. .. versionadded:: 3.8

**class** matplotlib.axis.**XAxis**(**args*, ***kwargs*)

**Parameters**

**axes**

[*Axes*] The *Axes* to which the created Axis belongs.

**pickradius**

[float] The acceptance radius for containment tests. See also *Axis.contains*.

**clear**

[bool, default: True] Whether to clear the Axis on creation. This is not required, e.g., when creating an Axis as part of an Axes, as `Axes.clear` will call `Axis.clear`. .. versionadded:: 3.8

**class** matplotlib.axis.**YAxis**(**args*, ***kwargs*)

**Parameters**

**axes**

[*Axes*] The *Axes* to which the created Axis belongs.

**pickradius**

[float] The acceptance radius for containment tests. See also *Axis.contains*.

**clear**

[bool, default: True] Whether to clear the Axis on creation. This is not required, e.g., when creating an Axis as part of an Axes, as `Axes.clear` will call `Axis.clear`. .. versionadded:: 3.8

**class** matplotlib.axis.**Ticker**

A container for the objects defining tick position and format.

**Attributes**

**locator**

[*Locator* subclass] Determines the positions of the ticks.

**formatter**

[*Formatter* subclass] Determines the format of the tick labels.

| | |
|---|---|
| *Axis.clear* | Clear the axis. |
| *Axis.get_scale* | Return this Axis' scale (as a str). |

### matplotlib.axis.Axis.clear

Axis.**clear**()

Clear the axis.

This resets axis properties to their default values:

- the label
- the scale
- locators, formatters and ticks
- major and minor grid

- units
- registered callbacks

### matplotlib.axis.Axis.get_scale

Axis.**get_scale**()

Return this Axis' scale (as a str).

### Examples using `matplotlib.axis.Axis.get_scale`

- *Axis scales*

### Formatters and Locators

| | |
|---|---|
| *Axis.get_major_formatter* | Get the formatter of the major ticker. |
| *Axis.get_major_locator* | Get the locator of the major ticker. |
| *Axis.get_minor_formatter* | Get the formatter of the minor ticker. |
| *Axis.get_minor_locator* | Get the locator of the minor ticker. |
| *Axis.set_major_formatter* | Set the formatter of the major ticker. |
| *Axis.set_major_locator* | Set the locator of the major ticker. |
| *Axis.set_minor_formatter* | Set the formatter of the minor ticker. |
| *Axis.set_minor_locator* | Set the locator of the minor ticker. |
| *Axis.remove_overlapping_locs* | If minor ticker locations that overlap with major ticker locations should be trimmed. |
| *Axis.get_remove_overlapping_locs* | |
| *Axis.set_remove_overlapping_locs* | |

### matplotlib.axis.Axis.get_major_formatter

Axis.**get_major_formatter**()

Get the formatter of the major ticker.

### Examples using `matplotlib.axis.Axis.get_major_formatter`

- *Axis scales*
- *Plotting dates and strings*

## matplotlib.axis.Axis.get_major_locator

Axis.**get_major_locator**()

Get the locator of the major ticker.

### Examples using `matplotlib.axis.Axis.get_major_locator`

- *Date tick labels*
- *Inset locator demo 2*
- *Quick start guide*
- *Axis scales*
- *Plotting dates and strings*

## matplotlib.axis.Axis.get_minor_formatter

Axis.**get_minor_formatter**()

Get the formatter of the minor ticker.

## matplotlib.axis.Axis.get_minor_locator

Axis.**get_minor_locator**()

Get the locator of the minor ticker.

## matplotlib.axis.Axis.set_major_formatter

Axis.**set_major_formatter**(*formatter*)

Set the formatter of the major ticker.

In addition to a *Formatter* instance, this also accepts a `str` or function.

For a `str` a *StrMethodFormatter* is used. The field used for the value must be labeled `'x'` and the field used for the position must be labeled `'pos'`. See the *StrMethodFormatter* documentation for more information.

For a function, a *FuncFormatter* is used. The function must take two inputs (a tick value `x` and a position `pos`), and return a string containing the corresponding tick label. See the *FuncFormatter* documentation for more information.

**Parameters**

**formatter**

[*Formatter*, `str`, or function]

## Examples using `matplotlib.axis.Axis.set_major_formatter`

- *Creating a timeline with lines, dates, and text*
- *Date tick labels*
- *Labeling ticks using engineering notation*
- *3D surface (colormap)*
- *SkewT-logP diagram: using transforms and custom projections*
- *Centering labels between ticks*
- *Custom Ticker*
- *Formatting date ticks using ConciseDateFormatter*
- *Date Demo Convert*
- *Placing date ticks using recurrence rules*
- *Date tick locators and formatters*
- *Custom tick formatter for time series*
- *Dollar ticks*
- *Major and minor ticks*
- *Multilevel (nested) ticks*
- *Tick formatters*
- *Setting tick labels from a list of values*
- *The Lifecycle of a Plot*
- *Quick start guide*
- *Axis ticks*
- *Plotting dates and strings*
- *Choosing Colormaps in Matplotlib*
- *Text in Matplotlib*

### matplotlib.axis.Axis.set_major_locator

Axis.**set_major_locator**(*locator*)

Set the locator of the major ticker.

**Parameters**

**locator**

[*Locator*]

### Examples using `matplotlib.axis.Axis.set_major_locator`

- *Hatch-filled histograms*
- *Creating a timeline with lines, dates, and text*
- *Date tick labels*
- *Anatomy of a figure*
- *3D surface (colormap)*
- *3D surface (checkerboard)*
- *Scales*
- *SkewT-logP diagram: using transforms and custom projections*
- *Centering labels between ticks*
- *Formatting date ticks using ConciseDateFormatter*
- *Date Demo Convert*
- *Placing date ticks using recurrence rules*
- *Date tick locators and formatters*
- *Custom tick formatter for time series*
- *Major and minor ticks*
- *Multilevel (nested) ticks*
- *Tick locators*
- *Setting tick labels from a list of values*
- *Axis scales*
- *Axis ticks*
- *Plotting dates and strings*
- *Choosing Colormaps in Matplotlib*
- *Text in Matplotlib*

### matplotlib.axis.Axis.set_minor_formatter

Axis.**set_minor_formatter**(*formatter*)

Set the formatter of the minor ticker.

In addition to a *Formatter* instance, this also accepts a `str` or function. See *Axis.set_major_formatter* for more information.

**Parameters**

**formatter**

[*Formatter*, `str`, or function]

### Examples using `matplotlib.axis.Axis.set_minor_formatter`

- *Anatomy of a figure*
- *Scales*
- *SkewT-logP diagram: using transforms and custom projections*
- *Centering labels between ticks*
- *Axis scales*

### matplotlib.axis.Axis.set_minor_locator

Axis.**set_minor_locator**(*locator*)

Set the locator of the minor ticker.

**Parameters**

**locator**

[*Locator*]

### Examples using `matplotlib.axis.Axis.set_minor_locator`

- *Stackplots and streamgraphs*
- *Secondary Axis*
- *Date tick labels*
- *Anatomy of a figure*
- *Centering labels between ticks*
- *Date Demo Convert*

- *Major and minor ticks*
- *Tick locators*
- *Axis ticks*

### matplotlib.axis.Axis.remove_overlapping_locs

**property** Axis.**remove_overlapping_locs**

If minor ticker locations that overlap with major ticker locations should be trimmed.

### matplotlib.axis.Axis.get_remove_overlapping_locs

Axis.**get_remove_overlapping_locs**()

### matplotlib.axis.Axis.set_remove_overlapping_locs

Axis.**set_remove_overlapping_locs**(*val*)

### Axis Label

| | |
|---|---|
| *Axis.set_label_coords* | Set the coordinates of the label. |
| *Axis.set_label_position* | Set the label position (top or bottom) |
| *Axis.set_label_text* | Set the text value of the axis label. |
| *Axis.get_label* | Return the axis label as a Text instance. |
| *Axis.get_label_position* | Return the label position (top or bottom) |
| *Axis.get_label_text* | Get the text of the label. |

### matplotlib.axis.Axis.set_label_coords

Axis.**set_label_coords**(*x*, *y*, *transform=None*)

Set the coordinates of the label.

By default, the x coordinate of the y label and the y coordinate of the x label are determined by the tick label bounding boxes, but this can lead to poor alignment of multiple labels if there are multiple Axes.

You can also specify the coordinate system of the label with the transform. If None, the default coordinate system will be the axes coordinate system: (0, 0) is bottom left, (0.5, 0.5) is center, etc.

### Examples using `matplotlib.axis.Axis.set_label_coords`

- *Align y-labels*

## matplotlib.axis.Axis.set_label_position

`Axis.`**`set_label_position`**(*position*)

Set the label position (top or bottom)

**Parameters**

**position**

[{'top', 'bottom'}]

### Examples using `matplotlib.axis.Axis.set_label_position`

- *Title positioning*

## matplotlib.axis.Axis.set_label_text

`Axis.`**`set_label_text`**(*label*, *fontdict=None*, ***kwargs*)

Set the text value of the axis label.

**Parameters**

**label**

[str] Text string.

**fontdict**

[dict] Text properties.

---

**Discouraged**

The use of *fontdict* is discouraged. Parameters should be passed as individual keyword arguments or using dictionary-unpacking `set_label_text(..., **fontdict)`.

---

****kwargs**

Merged into fontdict.

### matplotlib.axis.Axis.get_label

Axis.**get_label**()
Return the axis label as a Text instance.

### Examples using `matplotlib.axis.Axis.get_label`

- *Parasite Simple*

### matplotlib.axis.Axis.get_label_position

Axis.**get_label_position**()
Return the label position (top or bottom)

### matplotlib.axis.Axis.get_label_text

Axis.**get_label_text**()
Get the text of the label.

### Ticks, tick labels and Offset text

| | |
|---|---|
| *Axis.get_major_ticks* | Return the list of major *Tick*s. |
| *Axis.get_majorticklabels* | Return this Axis' major tick labels, as a list of *Text*. |
| *Axis.get_majorticklines* | Return this Axis' major tick lines as a list of *Line2D*s. |
| *Axis.get_majorticklocs* | Return this Axis' major tick locations in data coordinates. |
| *Axis.get_minor_ticks* | Return the list of minor *Tick*s. |
| *Axis.get_minorticklabels* | Return this Axis' minor tick labels, as a list of *Text*. |
| *Axis.get_minorticklines* | Return this Axis' minor tick lines as a list of *Line2D*s. |
| *Axis.get_minorticklocs* | Return this Axis' minor tick locations in data coordinates. |
| *Axis.get_offset_text* | Return the axis offsetText as a Text instance. |
| *Axis.get_tick_padding* | |
| *Axis.get_tick_params* | Get appearance parameters for ticks, ticklabels, and gridlines. |
| *Axis.get_ticklabels* | Get this Axis' tick labels. |
| *Axis.get_ticklines* | Return this Axis' tick lines as a list of *Line2D*s. |
| *Axis.get_ticklocs* | Return this Axis' tick locations in data coordinates. |
| *Axis.get_gridlines* | Return this Axis' grid lines as a list of *Line2D*s. |
| *Axis.grid* | Configure the grid lines. |
| *Axis.set_tick_params* | Set appearance parameters for ticks, ticklabels, and gridlines. |
| *Axis.axis_date* | Set up axis ticks and labels to treat data along this Axis as dates. |
| *Axis.minorticks_off* | Remove minor ticks from the Axis. |
| *Axis.minorticks_on* | Display default minor ticks on the Axis, depending on the scale (*get_scale*). |

### matplotlib.axis.Axis.get_major_ticks

Axis.**get_major_ticks**(*numticks=None*)

Return the list of major *Tick*s.

> **Warning:** Ticks are not guaranteed to be persistent. Various operations can create, delete and modify the Tick instances. There is an imminent risk that changes to individual ticks will not survive if you work on the figure further (including also panning/zooming on a displayed figure).
>
> Working on the individual ticks is a method of last resort. Use *set_tick_params* instead if possible.

### matplotlib.axis.Axis.get_majorticklabels

Axis.**get_majorticklabels**()

Return this Axis' major tick labels, as a list of *Text*.

### matplotlib.axis.Axis.get_majorticklines

Axis.**get_majorticklines**()

Return this Axis' major tick lines as a list of *Line2D*s.

### matplotlib.axis.Axis.get_majorticklocs

Axis.**get_majorticklocs**()

Return this Axis' major tick locations in data coordinates.

### matplotlib.axis.Axis.get_minor_ticks

Axis.**get_minor_ticks**(*numticks=None*)

Return the list of minor *Tick*s.

> **Warning:** Ticks are not guaranteed to be persistent. Various operations can create, delete and modify the Tick instances. There is an imminent risk that changes to individual ticks will not survive if you work on the figure further (including also panning/zooming on a displayed figure).
>
> Working on the individual ticks is a method of last resort. Use *set_tick_params* instead if possible.

### matplotlib.axis.Axis.get_minorticklabels

Axis.**get_minorticklabels**()

Return this Axis' minor tick labels, as a list of *Text*.

### matplotlib.axis.Axis.get_minorticklines

Axis.**get_minorticklines**()

Return this Axis' minor tick lines as a list of *Line2D*s.

### matplotlib.axis.Axis.get_minorticklocs

Axis.**get_minorticklocs**()

Return this Axis' minor tick locations in data coordinates.

### matplotlib.axis.Axis.get_offset_text

Axis.**get_offset_text**()

Return the axis offsetText as a Text instance.

### matplotlib.axis.Axis.get_tick_padding

Axis.**get_tick_padding**()

### matplotlib.axis.Axis.get_tick_params

Axis.**get_tick_params**(*which='major'*)

Get appearance parameters for ticks, ticklabels, and gridlines.

New in version 3.7.

**Parameters**

**which**

[{'major', 'minor'}, default: 'major'] The group of ticks for which the parameters are retrieved.

**Returns**

**dict**

Properties for styling tick elements added to the axis.

#### Notes

This method returns the appearance parameters for styling *new* elements added to this axis and may be different from the values on current elements if they were modified directly by the user (e.g., via `set_*` methods on individual tick objects).

### Examples

```
>>> ax.yaxis.set_tick_params(labelsize=30, labelcolor='red',
...                          direction='out', which='major')
>>> ax.yaxis.get_tick_params(which='major')
{'direction': 'out',
'left': True,
'right': False,
'labelleft': True,
'labelright': False,
'gridOn': False,
'labelsize': 30,
'labelcolor': 'red'}
>>> ax.yaxis.get_tick_params(which='minor')
{'left': True,
'right': False,
'labelleft': True,
'labelright': False,
'gridOn': False}
```

## matplotlib.axis.Axis.get_ticklabels

Axis.**get_ticklabels**(*minor=False*, *which=None*)

Get this Axis' tick labels.

**Parameters**

**minor**

[bool] Whether to return the minor or the major ticklabels.

**which**

[None, ('minor', 'major', 'both')] Overrides *minor*.

Selects which ticklabels to return

**Returns**

**list of *Text***

### Examples using `matplotlib.axis.Axis.get_ticklabels`

- *Fig Axes Customize Simple*
- *Artist tutorial*

## matplotlib.axis.Axis.get_ticklines

`Axis.get_ticklines`(*minor=False*)

Return this Axis' tick lines as a list of `Line2D`s.

### Examples using `matplotlib.axis.Axis.get_ticklines`

- *Fig Axes Customize Simple*
- *Artist tutorial*

## matplotlib.axis.Axis.get_ticklocs

`Axis.get_ticklocs`(*, *minor=False*)

Return this Axis' tick locations in data coordinates.

The locations are not clipped to the current axis limits and hence may contain locations that are not visible in the output.

**Parameters**

**minor**

[bool, default: False] True to return the minor tick directions, False to return the major tick directions.

**Returns**

**array of tick locations**

### Examples using `matplotlib.axis.Axis.get_ticklocs`

- *Artist tutorial*

### matplotlib.axis.Axis.get_gridlines

Axis.**get_gridlines**()

Return this Axis' grid lines as a list of *Line2D*s.

### matplotlib.axis.Axis.grid

Axis.**grid**(*visible=None*, *which='major'*, ***kwargs*)

Configure the grid lines.

**Parameters**

**visible**

[bool or None] Whether to show the grid lines. If any *kwargs* are supplied, it is assumed you want the grid on and *visible* will be set to True.

If *visible* is *None* and there are no *kwargs*, this toggles the visibility of the lines.

**which**

[{'major', 'minor', 'both'}] The grid lines to apply the changes on.

****kwargs**

[*Line2D* properties] Define the line properties of the grid, e.g.:

```
grid(color='r', linestyle='-', linewidth=2)
```

### Examples using `matplotlib.axis.Axis.grid`

- *Boxplots*
- *Box plot vs. violin plot comparison*
- *Symlog Demo*

### matplotlib.axis.Axis.set_tick_params

Axis.**set_tick_params**(*which='major'*, *reset=False*, ***kwargs*)

Set appearance parameters for ticks, ticklabels, and gridlines.

For documentation of keyword arguments, see *matplotlib.axes.Axes.tick_params()*.

**See also:**

***Axis.get_tick_params***

View the current style settings for ticks, ticklabels, and gridlines.

### Examples using `matplotlib.axis.Axis.set_tick_params`

- *Secondary Axis*
- *Scatter Histogram (Locatable Axes)*
- *Placing date ticks using recurrence rules*
- *Date Precision and Epochs*
- *Dollar ticks*
- *Choosing Colormaps in Matplotlib*

## matplotlib.axis.Axis.axis_date

Axis.**axis_date**(*tz=None*)

Set up axis ticks and labels to treat data along this Axis as dates.

**Parameters**

**tz**

[str or `datetime.tzinfo`, default: *`rcParams["timezone"]`* (default: `'UTC'`)] The timezone used to create date labels.

## matplotlib.axis.Axis.minorticks_off

Axis.**minorticks_off**()

Remove minor ticks from the Axis.

## matplotlib.axis.Axis.minorticks_on

Axis.**minorticks_on**()

Display default minor ticks on the Axis, depending on the scale (*`get_scale`*).

Scales use specific minor locators:

- log: *`LogLocator`*
- symlog: *`SymmetricalLogLocator`*
- asinh: *`AsinhLocator`*
- logit: *`LogitLocator`*
- default: *`AutoMinorLocator`*

Displaying minor ticks may reduce performance; you may turn them off using *`minorticks_off()`* if drawing speed is a problem.

### Data and view intervals

| | |
|---|---|
| *Axis.get_data_interval* | Return the `(min, max)` data limits of this axis. |
| *Axis.get_view_interval* | Return the `(min, max)` view limits of this axis. |
| *Axis.get_inverted* | Return whether this Axis is oriented in the "inverse" direction. |
| *Axis.set_data_interval* | Set the axis data limits. |
| *Axis.set_view_interval* | Set the axis view limits. |
| *Axis.set_inverted* | Set whether this Axis is oriented in the "inverse" direction. |

#### matplotlib.axis.Axis.get_data_interval

`Axis.`**`get_data_interval`**`()`

Return the `(min, max)` data limits of this axis.

#### matplotlib.axis.Axis.get_view_interval

`Axis.`**`get_view_interval`**`()`

Return the `(min, max)` view limits of this axis.

#### matplotlib.axis.Axis.get_inverted

`Axis.`**`get_inverted`**`()`

Return whether this Axis is oriented in the "inverse" direction.

The "normal" direction is increasing to the right for the x-axis and to the top for the y-axis; the "inverse" direction is increasing to the left for the x-axis and to the bottom for the y-axis.

#### matplotlib.axis.Axis.set_data_interval

`Axis.`**`set_data_interval`**`(`*vmin*, *vmax*, *ignore=False*`)`

Set the axis data limits. This method is for internal use.

If *ignore* is False (the default), this method will never reduce the preexisting data limits, only expand them if *vmin* or *vmax* are not within them. Moreover, the order of *vmin* and *vmax* does not matter; the orientation of the axis will not change.

If *ignore* is True, the data limits will be set exactly to `(vmin, vmax)` in that order.

### matplotlib.axis.Axis.set_view_interval

Axis.**set_view_interval**(*vmin*, *vmax*, *ignore=False*)

Set the axis view limits. This method is for internal use; Matplotlib users should typically use e.g. *set_xlim* or *set_ylim*.

If *ignore* is False (the default), this method will never reduce the preexisting view limits, only expand them if *vmin* or *vmax* are not within them. Moreover, the order of *vmin* and *vmax* does not matter; the orientation of the axis will not change.

If *ignore* is True, the view limits will be set exactly to `(vmin, vmax)` in that order.

### matplotlib.axis.Axis.set_inverted

Axis.**set_inverted**(*inverted*)

Set whether this Axis is oriented in the "inverse" direction.

The "normal" direction is increasing to the right for the x-axis and to the top for the y-axis; the "inverse" direction is increasing to the left for the x-axis and to the bottom for the y-axis.

### Examples using `matplotlib.axis.Axis.set_inverted`

- *Inverted axis*

### Rendering helpers

| | |
|---|---|
| *Axis.get_minpos* | |
| *Axis.get_tick_space* | Return the estimated number of ticks that can fit on the axis. |
| *Axis.get_tightbbox* | Return a bounding box that encloses the axis. |

### matplotlib.axis.Axis.get_minpos

Axis.**get_minpos**()

### matplotlib.axis.Axis.get_tick_space

Axis.**get_tick_space**()

Return the estimated number of ticks that can fit on the axis.

### matplotlib.axis.Axis.get_tightbbox

Axis.**get_tightbbox**(*renderer=None*, *, *for_layout_only=False*)

Return a bounding box that encloses the axis. It only accounts tick labels, axis label, and offsetText.

If *for_layout_only* is True, then the width of the label (if this is an x-axis) or the height of the label (if this is a y-axis) is collapsed to near zero. This allows tight/constrained_layout to ignore too-long labels when doing their layout.

## Interactive

| | |
|---|---|
| *Axis.contains* | Test whether the artist contains the mouse event. |
| *Axis.pickradius* | The acceptance radius for containment tests. |
| *Axis.get_pickradius* | Return the depth of the axis used by the picker. |
| *Axis.set_pickradius* | Set the depth of the axis used by the picker. |

### matplotlib.axis.Axis.contains

Axis.**contains**(*mouseevent*)

Test whether the artist contains the mouse event.

**Parameters**

**mouseevent**

[*MouseEvent*]

**Returns**

**contains**

[bool] Whether any values are within the radius.

**details**

[dict] An artist-specific dictionary of details of the event context, such as which points are contained in the pick radius. See the individual Artist subclasses for details.

### matplotlib.axis.Axis.pickradius

**property** Axis.**pickradius**

The acceptance radius for containment tests. See also *Axis.contains*.

### matplotlib.axis.Axis.get_pickradius

Axis.**get_pickradius**()

Return the depth of the axis used by the picker.

### matplotlib.axis.Axis.set_pickradius

Axis.**set_pickradius**(*pickradius*)

Set the depth of the axis used by the picker.

**Parameters**

**pickradius**

[float] The acceptance radius for containment tests. See also *Axis.contains*.

## Units

| | |
|---|---|
| *Axis.convert_units* | |
| *Axis.set_units* | Set the units for axis. |
| *Axis.get_units* | Return the units for axis. |
| *Axis.update_units* | Introspect *data* for units converter and update the `axis.converter` instance if necessary. |

### matplotlib.axis.Axis.convert_units

Axis.**convert_units**(*x*)

### matplotlib.axis.Axis.set_units

Axis.**set_units**(*u*)

Set the units for axis.

**Parameters**

**u**

[units tag]

#### Notes

The units of any shared axis will also be updated.

### Examples using `matplotlib.axis.Axis.set_units`

- *Artist tests*
- *Group barchart with units*
- *Unit handling*

### matplotlib.axis.Axis.get_units

Axis.**get_units**()

Return the units for axis.

### matplotlib.axis.Axis.update_units

Axis.**update_units**(*data*)

Introspect *data* for units converter and update the `axis.converter` instance if necessary. Return *True* if *data* is registered for unit conversion.

### XAxis Specific

| | |
|---|---|
| *XAxis.axis_name* | Read-only name identifying the axis. |
| *XAxis.get_ticks_position* | Return the ticks position ("top", "bottom", "default", or "unknown"). |
| *XAxis.set_ticks_position* | Set the ticks position. |
| *XAxis.set_label_position* | Set the label position (top or bottom) |
| *XAxis.tick_bottom* | Move ticks and ticklabels (if present) to the bottom of the Axes. |
| *XAxis.tick_top* | Move ticks and ticklabels (if present) to the top of the Axes. |

### matplotlib.axis.XAxis.axis_name

XAxis.**axis_name = 'x'**

Read-only name identifying the axis.

### matplotlib.axis.XAxis.get_ticks_position

XAxis.**get_ticks_position**()

Return the ticks position ("top", "bottom", "default", or "unknown").

### matplotlib.axis.XAxis.set_ticks_position

XAxis.**set_ticks_position**(*position*)

Set the ticks position.

**Parameters**

**position**

[{'top', 'bottom', 'both', 'default', 'none'}] 'both' sets the ticks to appear on both positions, but does not change the tick labels. 'default' resets the tick positions to the default: ticks on both positions, labels at bottom. 'none' can be used if you don't want any ticks. 'none' and 'both' affect only the ticks, not the labels.

### Examples using `matplotlib.axis.XAxis.set_ticks_position`

- *Colorbar with AxesDivider*
- *Controlling the position and size of colorbars with Inset Axes*
- *XKCD*
- *Axis ticks*
- *Choosing Colormaps in Matplotlib*

## matplotlib.axis.XAxis.set_label_position

XAxis.**set_label_position**(*position*)

Set the label position (top or bottom)

**Parameters**

**position**

[{'top', 'bottom'}]

### Examples using `matplotlib.axis.XAxis.set_label_position`

- *Title positioning*

## matplotlib.axis.XAxis.tick_bottom

XAxis.**tick_bottom**()

Move ticks and ticklabels (if present) to the bottom of the Axes.

### Examples using `matplotlib.axis.XAxis.tick_bottom`

- *Broken Axis*
- *Stock prices over 32 years*

### matplotlib.axis.XAxis.tick_top

XAxis.**tick_top**()

Move ticks and ticklabels (if present) to the top of the Axes.

### Examples using `matplotlib.axis.XAxis.tick_top`

- *Aligning Labels and Titles*
- *Broken Axis*
- *Title positioning*

### YAxis Specific

| | |
|---|---|
| *YAxis.axis_name* | Read-only name identifying the axis. |
| *YAxis.get_ticks_position* | Return the ticks position ("left", "right", "default", or "unknown"). |
| *YAxis.set_offset_position* | **Parameters** |
| *YAxis.set_ticks_position* | Set the ticks position. |
| *YAxis.set_label_position* | Set the label position (left or right) |
| *YAxis.tick_left* | Move ticks and ticklabels (if present) to the left of the Axes. |
| *YAxis.tick_right* | Move ticks and ticklabels (if present) to the right of the Axes. |

### matplotlib.axis.YAxis.axis_name

YAxis.**axis_name = 'y'**

Read-only name identifying the axis.

### matplotlib.axis.YAxis.get_ticks_position

YAxis.**get_ticks_position**()

Return the ticks position ("left", "right", "default", or "unknown").

### matplotlib.axis.YAxis.set_offset_position

YAxis.**set_offset_position**(*position*)

**Parameters**

**position**

[{'left', 'right'}]

### matplotlib.axis.YAxis.set_ticks_position

YAxis.**set_ticks_position**(*position*)

Set the ticks position.

**Parameters**

**position**

[{'left', 'right', 'both', 'default', 'none'}] 'both' sets the ticks to appear on both positions, but does not change the tick labels. 'default' resets the tick positions to the default: ticks on both positions, labels at left. 'none' can be used if you don't want any ticks. 'none' and 'both' affect only the ticks, not the labels.

### matplotlib.axis.YAxis.set_label_position

YAxis.**set_label_position**(*position*)

Set the label position (left or right)

**Parameters**

**position**

[{'left', 'right'}]

### matplotlib.axis.YAxis.tick_left

YAxis.**tick_left**()

Move ticks and ticklabels (if present) to the left of the Axes.

### Examples using `matplotlib.axis.YAxis.tick_left`

- *Stock prices over 32 years*
- *Set default y-axis tick labels on the right*

### matplotlib.axis.YAxis.tick_right

YAxis.**tick_right**()

Move ticks and ticklabels (if present) to the right of the Axes.

### Other

| | |
|---|---|
| *Axis.OFFSETTEXTPAD* | |
| *Axis.axes* | The *Axes* instance the artist resides in, or *None*. |
| *Axis.limit_range_for_scale* | |
| *Axis.reset_ticks* | Re-initialize the major and minor Tick lists. |
| *Axis.set_default_intervals* | Set the default limits for the axis data and view interval if they have not been not mutated yet. |

### matplotlib.axis.Axis.OFFSETTEXTPAD

Axis.**OFFSETTEXTPAD = 3**

### matplotlib.axis.Axis.axes

**property** Axis.**axes**

The *Axes* instance the artist resides in, or *None*.

### matplotlib.axis.Axis.limit_range_for_scale

Axis.**limit_range_for_scale**(*vmin*, *vmax*)

### matplotlib.axis.Axis.reset_ticks

Axis.**reset_ticks**()

Re-initialize the major and minor Tick lists.

Each list starts with a single fresh Tick.

### matplotlib.axis.Axis.set_default_intervals

Axis.**set_default_intervals**()

Set the default limits for the axis data and view interval if they have not been not mutated yet.

### Discouraged

These methods should be used together with care, calling `set_ticks` to specify the desired tick locations **before** calling `set_ticklabels` to specify a matching series of labels. Calling `set_ticks` makes a *FixedLocator*; it's list of locations is then used by `set_ticklabels` to make an appropriate *FuncFormatter*.

| | |
|---|---|
| *Axis.set_ticks* | Set this Axis' tick locations and optionally tick labels. |
| *Axis.set_ticklabels* | [*Discouraged*] Set this Axis' tick labels with list of string labels. |

### matplotlib.axis.Axis.set_ticks

Axis.**set_ticks**(*ticks*, *labels=None*, *, *minor=False*, ***kwargs*)

Set this Axis' tick locations and optionally tick labels.

If necessary, the view limits of the Axis are expanded so that all given ticks are visible.

**Parameters**

**ticks**

[1D array-like] Array of tick locations (either floats or in axis units). The axis *Locator* is replaced by a *FixedLocator*.

Pass an empty list (`set_ticks([])`) to remove all ticks.

Some tick formatters will not label arbitrary tick positions; e.g. log formatters only label decade ticks by default. In such a case you can set a formatter explicitly on the axis using *Axis.set_major_formatter* or provide formatted *labels* yourself.

**labels**

[list of str, optional] Tick labels for each location in *ticks*; must have the same length as *ticks*. If set, the labels are used as is, via a *FixedFormatter*. If not set, the labels are generated using the axis tick *Formatter*.

**minor**

[bool, default: False] If `False`, set only the major ticks; if `True`, only the minor ticks.

****kwargs**

*Text* properties for the labels. Using these is only allowed if you pass *labels*. In other cases, please use *tick_params*.

### Notes

The mandatory expansion of the view limits is an intentional design choice to prevent the surprise of a non-visible tick. If you need other limits, you should set the limits explicitly after setting the ticks.

## Examples using `matplotlib.axis.Axis.set_ticks`

- *Multiple Axes animation*

## matplotlib.axis.Axis.set_ticklabels

`Axis.`**`set_ticklabels`**(*labels*, *, *minor=False*, *fontdict=None*, ***kwargs*)

[*Discouraged*] Set this Axis' tick labels with list of string labels.

---

**Discouraged**

The use of this method is discouraged, because of the dependency on tick positions. In most cases, you'll want to use `Axes.set_[x/y/z]ticks(positions, labels)` or `Axis.set_ticks` instead.

If you are using this method, you should always fix the tick positions before, e.g. by using *Axis.set_ticks* or by explicitly setting a *FixedLocator*. Otherwise, ticks are free to move and the labels may end up in unexpected positions.

---

**Parameters**

**labels**

[sequence of str or of *Text*s] Texts for labeling each tick location in the sequence set by *Axis.set_ticks*; the number of labels must match the number of lo-

cations. The labels are used as is, via a *FixedFormatter* (without further formatting).

**minor**

[bool] If True, set minor ticks instead of major ticks.

**fontdict**

[dict, optional]

---

**Discouraged**

The use of *fontdict* is discouraged. Parameters should be passed as individual keyword arguments or using dictionary-unpacking `set_ticklabels(..., **fontdict)`.

---

A dictionary controlling the appearance of the ticklabels. The default *fontdict* is:

```
{'fontsize': rcParams['axes.titlesize'],
 'fontweight': rcParams['axes.titleweight'],
 'verticalalignment': 'baseline',
 'horizontalalignment': loc}
```

****kwargs**

Text properties.

**Warning:** This only sets the properties of the current ticks. Ticks are not guaranteed to be persistent. Various operations can create, delete and modify the Tick instances. There is an imminent risk that these settings can get lost if you work on the figure further (including also panning/zooming on a displayed figure).

Use *set_tick_params* instead if possible.

**Returns**

**list of *Text*s**

For each tick, includes `tick.label1` if it is visible, then `tick.label2` if it is visible, in that order.

### `Tick` objects

**class** matplotlib.axis.**Tick**(*axes*, *loc*, ***, *size=None*, *width=None*, *color=None*, *tickdir=None*, *pad=None*, *labelsize=None*, *labelcolor=None*, *labelfontfamily=None*, *zorder=None*, *gridOn=None*, *tick1On=True*, *tick2On=True*, *label1On=True*, *label2On=False*, *major=True*, *labelrotation=0*, *grid_color=None*, *grid_linestyle=None*, *grid_linewidth=None*, *grid_alpha=None*, ***kwargs*)

Abstract base class for the axis ticks, grid lines and labels.

Ticks mark a position on an Axis. They contain two lines as markers and two labels; one each for the bottom and top positions (in case of an *XAxis*) or for the left and right positions (in case of a *YAxis*).

**Attributes**

**tick1line**

[*Line2D*] The left/bottom tick marker.

**tick2line**

[*Line2D*] The right/top tick marker.

**gridline**

[*Line2D*] The grid line associated with the label position.

**label1**

[*Text*] The left/bottom tick label.

**label2**

[*Text*] The right/top tick label.

bbox is the Bound2D bounding box in display coords of the Axes loc is the tick location in data coords size is the tick size in points

**class** matplotlib.axis.**XTick**(**args*, ***kwargs*)

Contains all the Artists needed to make an x tick - the tick line, the label text and the grid line

bbox is the Bound2D bounding box in display coords of the Axes loc is the tick location in data coords size is the tick size in points

**class** matplotlib.axis.**YTick**(**args*, ***kwargs*)

Contains all the Artists needed to make a Y tick - the tick line, the label text and the grid line

bbox is the Bound2D bounding box in display coords of the Axes loc is the tick location in data coords size is the tick size in points

| | |
|---|---|
| *Tick.get_loc* | Return the tick location (data coords) as a scalar. |
| *Tick.get_pad* | Get the value of the tick label pad in points. |
| *Tick.get_tick_padding* | Get the length of the tick outside of the Axes. |
| *Tick.get_tickdir* | |
| *Tick.get_view_interval* | Return the view limits `(min, max)` of the axis the tick belongs to. |
| *Tick.set_label1* | [*Deprecated*] Set the label1 text. |
| *Tick.set_label2* | [*Deprecated*] Set the label2 text. |
| *Tick.set_pad* | Set the tick label pad in points |
| *Tick.set_url* | Set the url of label1 and label2. |
| *Tick.update_position* | Set the location of tick in data coords with scalar *loc*. |

### matplotlib.axis.Tick.get_loc

`Tick.`**`get_loc`**`()`

Return the tick location (data coords) as a scalar.

### matplotlib.axis.Tick.get_pad

`Tick.`**`get_pad`**`()`

Get the value of the tick label pad in points.

### matplotlib.axis.Tick.get_tick_padding

`Tick.`**`get_tick_padding`**`()`

Get the length of the tick outside of the Axes.

### matplotlib.axis.Tick.get_tickdir

`Tick.`**`get_tickdir`**`()`

### matplotlib.axis.Tick.get_view_interval

Tick.**get_view_interval**()

Return the view limits `(min, max)` of the axis the tick belongs to.

### matplotlib.axis.Tick.set_label1

Tick.**set_label1**(*s*)

[*Deprecated*] Set the label1 text.

**Parameters**

**s**

[str]

#### Notes

Deprecated since version 3.8.

### matplotlib.axis.Tick.set_label2

Tick.**set_label2**(*s*)

[*Deprecated*] Set the label2 text.

**Parameters**

**s**

[str]

#### Notes

Deprecated since version 3.8.

### matplotlib.axis.Tick.set_pad

Tick.**set_pad**(*val*)

Set the tick label pad in points

**Parameters**

**val**

[float]

### matplotlib.axis.Tick.set_url

Tick.**set_url**(*url*)

Set the url of label1 and label2.

**Parameters**

**url**

[str]

### matplotlib.axis.Tick.update_position

Tick.**update_position**(*loc*)

Set the location of tick in data coords with scalar *loc*.

## 7.2.7 matplotlib.backend_bases

Abstract base classes define the primitives that renderers and graphics contexts must implement to serve as a Matplotlib backend.

***RendererBase***

An abstract base class to handle drawing/rendering operations.

***FigureCanvasBase***

The abstraction layer that separates the *Figure* from the backend specific details like a user interface drawing area.

***GraphicsContextBase***

An abstract base class that provides color, line styles, etc.

***Event***

The base class for all of the Matplotlib event handling. Derived classes such as *KeyEvent* and *MouseEvent* store the meta data like keys and buttons pressed, x and y locations in pixel and *Axes* coordinates.

***ShowBase***

The base class for the Show class of each interactive backend; the 'show' callable is then set to Show.__call__.

***ToolContainerBase***

The base class for the Toolbar class of each interactive backend.

**class** matplotlib.backend_bases.**CloseEvent**(*name*, *canvas*, *guiEvent=None*)

Bases: *Event*

An event triggered by a figure being closed.

**class** matplotlib.backend_bases.**DrawEvent**(*name*, *canvas*, *renderer*)

Bases: *Event*

An event triggered by a draw operation on the canvas.

In most backends, callbacks subscribed to this event will be fired after the rendering is complete but before the screen is updated. Any extra artists drawn to the canvas's renderer will be reflected without an explicit call to `blit`.

**Warning:** Calling `canvas.draw` and `canvas.blit` in these callbacks may not be safe with all backends and may cause infinite recursion.

A DrawEvent has a number of special attributes in addition to those defined by the parent *Event* class.

**Attributes**

**renderer**

[*RendererBase*] The renderer for the draw event.

**class** matplotlib.backend_bases.**Event**(*name*, *canvas*, *guiEvent=None*)

Bases: `object`

A Matplotlib event.

The following attributes are defined and shown with their default values. Subclasses may define additional attributes.

**Attributes**

**name**

[str] The event name.

**canvas**

[*FigureCanvasBase*] The backend-specific canvas instance generating the event.

**guiEvent**

The GUI event that triggered the Matplotlib event.

**property guiEvent**

**class** matplotlib.backend_bases.**FigureCanvasBase**(*figure=None*)

Bases: `object`

The canvas the figure renders into.

**Attributes**

**figure**

[*Figure*] A high-level figure instance.

**blit**(*bbox=None*)

Blit the canvas in bbox (default entire canvas).

**property button_pick_id**

**property callbacks**

**property device_pixel_ratio**

The ratio of physical to logical pixels used for the canvas on screen.

By default, this is 1, meaning physical and logical pixels are the same size. Subclasses that support High DPI screens may set this property to indicate that said ratio is different. All Matplotlib interaction, unless working directly with the canvas, remains in logical pixels.

**draw**(**args*, ***kwargs*)

Render the *Figure*.

This method must walk the artist tree, even if no output is produced, because it triggers deferred work that users may want to access before saving output to disk. For example computing limits, auto-limits, and tick values.

**draw_idle**(**args*, ***kwargs*)

Request a widget redraw once control returns to the GUI event loop.

Even if multiple calls to *draw_idle* occur before control returns to the GUI event loop, the figure will only be rendered once.

### Notes

Backends may choose to override the method and implement their own strategy to prevent multiple renderings.

```
events = ['resize_event', 'draw_event', 'key_press_event',
'key_release_event', 'button_press_event', 'button_release_event',
'scroll_event', 'motion_notify_event', 'pick_event',
'figure_enter_event', 'figure_leave_event', 'axes_enter_event',
'axes_leave_event', 'close_event']
```

```
filetypes = {'eps': 'Encapsulated Postscript', 'jpeg': 'Joint
Photographic Experts Group', 'jpg': 'Joint Photographic Experts
Group', 'pdf': 'Portable Document Format', 'pgf': 'PGF code for
LaTeX', 'png': 'Portable Network Graphics', 'ps': 'Postscript',
'raw': 'Raw RGBA bitmap', 'rgba': 'Raw RGBA bitmap', 'svg':
'Scalable Vector Graphics', 'svgz': 'Scalable Vector Graphics',
'tif': 'Tagged Image File Format', 'tiff': 'Tagged Image File
Format', 'webp': 'WebP Image Format'}
```

**fixed_dpi = None**

**flush_events**()

Flush the GUI events for the figure.

Interactive backends need to reimplement this method.

**get_default_filename**()

Return a string, which includes extension, suitable for use as a default filename.

**classmethod get_default_filetype**()

Return the default savefig file format as specified in *rcParams["savefig.format"]* (default: 'png').

The returned string does not include a period. This method is overridden in backends that only support a single file type.

**classmethod get_supported_filetypes**()

Return dict of savefig file formats supported by this backend.

**classmethod get_supported_filetypes_grouped**()

Return a dict of savefig file formats supported by this backend, where the keys are a file type name, such as 'Joint Photographic Experts Group', and the values are a list of filename extensions used for that filetype, such as ['jpg', 'jpeg'].

**get_width_height**(*, *physical=False*)

Return the figure width and height in integral points or pixels.

When the figure is used on High DPI screens (and the backend supports it), the truncation to integers occurs after scaling by the device pixel ratio.

**Parameters**

**physical**

[bool, default: False] Whether to return true physical pixels or logical pixels. Physical pixels may be used by backends that support HiDPI, but still configure the canvas using its actual size.

**Returns**

**width, height**

[int] The size of the figure, in points or pixels, depending on the backend.

**grab_mouse**(*ax*)

Set the child *Axes* which is grabbing the mouse events.

Usually called by the widgets themselves. It is an error to call this if the mouse is already grabbed by another Axes.

**inaxes**(*xy*)

Return the topmost visible *Axes* containing the point *xy*.

**Parameters**

**xy**

[(float, float)] (x, y) pixel positions from left/bottom of the canvas.

**Returns**

***Axes* or None**

The topmost visible Axes containing the point, or None if there is no Axes at the point.

**is_saving**()

Return whether the renderer is in the process of saving to a file, rather than rendering for an on-screen buffer.

**manager_class**

alias of *FigureManagerBase*

**mpl_connect**(*s*, *func*)

Bind function *func* to event *s*.

**Parameters**

**s**

[str] One of the following events ids:

- 'button_press_event'
- 'button_release_event'
- 'draw_event'
- 'key_press_event'
- 'key_release_event'
- 'motion_notify_event'
- 'pick_event'
- 'resize_event'
- 'scroll_event'
- 'figure_enter_event',
- 'figure_leave_event',
- 'axes_enter_event',
- 'axes_leave_event'

- 'close_event'.

**func**

[callable] The callback function to be executed, which must have the signature:

```
def func(event: Event) -> Any
```

For the location events (button and key press/release), if the mouse is over the Axes, the `inaxes` attribute of the event will be set to the *Axes* the event occurs is over, and additionally, the variables `xdata` and `ydata` attributes will be set to the mouse location in data coordinates. See *KeyEvent* and *MouseEvent* for more info.

---

**Note:** If func is a method, this only stores a weak reference to the method. Thus, the figure does not influence the lifetime of the associated object. Usually, you want to make sure that the object is kept alive throughout the lifetime of the figure by holding a reference to it.

---

**Returns**

**cid**

A connection id that can be used with *FigureCanvasBase.mpl_disconnect*.

### Examples

```
def on_press(event):
    print('you pressed', event.button, event.xdata, event.ydata)

cid = canvas.mpl_connect('button_press_event', on_press)
```

**mpl_disconnect**(*cid*)

Disconnect the callback with id *cid*.

### Examples

```
cid = canvas.mpl_connect('button_press_event', on_press)
# ... later
canvas.mpl_disconnect(cid)
```

**classmethod new_manager**(*figure*, *num*)

Create a new figure manager for *figure*, using this canvas class.

### Notes

This method should not be reimplemented in subclasses. If custom manager creation logic is needed, please reimplement `FigureManager.create_with_canvas`.

**new_timer**(*interval=None*, *callbacks=None*)

Create a new backend-specific subclass of `Timer`.

This is useful for getting periodic events through the backend's native event loop. Implemented only for backends with GUIs.

**Parameters**

**interval**

[int] Timer interval in milliseconds.

**callbacks**

[list[tuple[callable, tuple, dict]]] Sequence of (func, args, kwargs) where `func(*args, **kwargs)` will be executed by the timer every *interval*.

Callbacks which return `False` or `0` will be removed from the timer.

### Examples

```
>>> timer = fig.canvas.new_timer(callbacks=[(f1, (1,), {'a': 3})])
```

**print_figure**(*filename*, *dpi=None*, *facecolor=None*, *edgecolor=None*, *orientation='portrait'*, *format=None*, ***, *bbox_inches=None*, *pad_inches=None*, *bbox_extra_artists=None*, *backend=None*, ***kwargs*)

Render the figure to hardcopy. Set the figure patch face and edge colors. This is useful because some of the GUIs have a gray figure face color background and you'll probably want to override this on hardcopy.

**Parameters**

**filename**

[str or path-like or file-like] The file where the figure is saved.

**dpi**

[float, default: *rcParams["savefig.dpi"]* (default: `'figure'`)] The dots per inch to save the figure in.

**facecolor**

[*color* or 'auto', default: *rcParams["savefig.facecolor"]* (default: `'auto'`)] The facecolor of the figure. If 'auto', use the current figure facecolor.

**edgecolor**

[*color* or 'auto', default: *rcParams["savefig.edgecolor"]* (default: `'auto'`)] The edgecolor of the figure. If 'auto', use the current figure edgecolor.

**orientation**

[{'landscape', 'portrait'}, default: 'portrait'] Only currently applies to PostScript printing.

**format**

[str, optional] Force a specific file format. If not given, the format is inferred from the *filename* extension, and if that fails from *rcParams["savefig.format"]* (default: `'png'`).

**bbox_inches**

['tight' or *Bbox*, default: *rcParams["savefig.bbox"]* (default: `None`)] Bounding box in inches: only the given portion of the figure is saved. If 'tight', try to figure out the tight bbox of the figure.

**pad_inches**

[float or 'layout', default: *rcParams["savefig.pad_inches"]* (default: `0.1`)] Amount of padding in inches around the figure when bbox_inches is 'tight'. If 'layout' use the padding from the constrained or compressed layout engine; ignored if one of those engines is not in use.

**bbox_extra_artists**

[list of *Artist*, optional] A list of extra artists that will be considered when the tight bbox is calculated.

**backend**

[str, optional] Use a non-default backend to render the file, e.g. to render a png file with the "cairo" backend rather than the default "agg", or a pdf file with the "pgf" backend rather than the default "pdf". Note that the default backend is normally sufficient. See *The builtin backends* for a list of valid backends for each file format. Custom backends can be referenced as "module://...".

**release_mouse**(*ax*)

Release the mouse grab held by the *Axes* *ax*.

Usually called by the widgets. It is ok to call this even if *ax* doesn't have the mouse grab currently.

**required_interactive_framework = None**

**property scroll_pick_id**

**set_cursor**(*cursor*)

Set the current cursor.

This may have no effect if the backend does not display anything.

If required by the backend, this method should trigger an update in the backend event loop after the cursor is set, as this method may be called e.g. before a long-running task during which the GUI is not updated.

**Parameters**

**cursor**

[*Cursors*] The cursor to display over the canvas. Note: some backends may change the cursor for the entire window.

**start_event_loop**(*timeout=0*)

Start a blocking event loop.

Such an event loop is used by interactive functions, such as *ginput* and *waitforbuttonpress*, to wait for events.

The event loop blocks until a callback function triggers *stop_event_loop*, or *timeout* is reached.

If *timeout* is 0 or negative, never timeout.

Only interactive backends need to reimplement this method and it relies on *flush_events* being properly implemented.

Interactive backends should implement this in a more native way.

**stop_event_loop**()

Stop the current blocking event loop.

Interactive backends need to reimplement this to match *start_event_loop*

**supports_blit = False**

**switch_backends**(*FigureCanvasClass*)

[*Deprecated*] Instantiate an instance of FigureCanvasClass

This is used for backend switching, e.g., to instantiate a FigureCanvasPS from a FigureCanvasGTK. Note, deep copying is not done, so any changes to one of the instances (e.g., setting figure size or line props), will be reflected in the other

### Notes

Deprecated since version 3.8.

**class** matplotlib.backend_bases.**FigureManagerBase**(*canvas*, *num*)

Bases: object

A backend-independent abstraction of a figure container and controller.

The figure manager is used by pyplot to interact with the window in a backend-independent way. It's an adapter for the real (GUI) framework that represents the visual figure on screen.

The figure manager is connected to a specific canvas instance, which in turn is connected to a specific figure instance. To access a figure manager for a given figure in user code, you typically use `fig.canvas.manager`.

GUI backends derive from this class to translate common operations such as *show* or *resize* to the GUI-specific code. Non-GUI backends do not support these operations and can just use the base class.

This following basic operations are accessible:

**Window operations**

- *show*
- *destroy*
- *full_screen_toggle*
- *resize*
- *get_window_title*
- *set_window_title*

**Key and mouse button press handling**

The figure manager sets up default key and mouse button press handling by hooking up the *key_press_handler* to the matplotlib event system. This ensures the same shortcuts and mouse actions across backends.

**Other operations**

Subclasses will have additional attributes and functions to access additional functionality. This is of course backend-specific. For example, most GUI backends have `window` and `toolbar` attributes that give access to the native GUI widgets of the respective framework.

**Attributes**

**canvas**

[*FigureCanvasBase*] The backend-specific canvas instance.

**num**

[int or str] The figure number.

**key_press_handler_id**

[int] The default key handler cid, when using the toolmanager. To disable the default key press handling use:

```
figure.canvas.mpl_disconnect(
    figure.canvas.manager.key_press_handler_id)
```

**button_press_handler_id**

[int] The default mouse button handler cid, when using the toolmanager. To disable the default button press handling use:

```
figure.canvas.mpl_disconnect(
    figure.canvas.manager.button_press_handler_id)
```

**classmethod create_with_canvas**(*canvas_class*, *figure*, *num*)

Create a manager for a given *figure* using a specific *canvas_class*.

Backends should override this method if they have specific needs for setting up the canvas or the manager.

**destroy**()

**full_screen_toggle**()

**get_window_title**()

Return the title text of the window containing the figure, or None if there is no window (e.g., a PS backend).

**classmethod pyplot_show**(*, *block=None*)

Show all figures. This method is the implementation of *pyplot.show*.

To customize the behavior of *pyplot.show*, interactive backends should usually override *start_main_loop*; if more customized logic is necessary, *pyplot_show* can also be overridden.

**Parameters**

**block**

[bool, optional] Whether to block by calling `start_main_loop`. The default, None, means to block if we are neither in IPython's `%pylab` mode nor in `interactive` mode.

**resize**(*w*, *h*)

For GUI backends, resize the window (in physical pixels).

**set_window_title**(*title*)

Set the title text of the window containing the figure.

This has no effect for non-GUI (e.g., PS) backends.

### Examples

```
>>> fig = plt.figure()
>>> fig.canvas.manager.set_window_title('My figure')
```

**show**()

For GUI backends, show the figure window and redraw. For non-GUI backends, raise an exception, unless running headless (i.e. on Linux with an unset DISPLAY); this exception is converted to a warning in *Figure.show*.

**classmethod start_main_loop**()

Start the main event loop.

This method is called by *FigureManagerBase.pyplot_show*, which is the implementation of *pyplot.show*. To customize the behavior of *pyplot.show*, interactive backends should usually override *start_main_loop*; if more customized logic is necessary, *pyplot_show* can also be overridden.

**class** matplotlib.backend_bases.**GraphicsContextBase**

Bases: object

An abstract base class that provides color, line styles, etc.

**copy_properties**(*gc*)

Copy properties from *gc* to self.

**get_alpha**()

Return the alpha value used for blending - not supported on all backends.

**get_antialiased**()

Return whether the object should try to do antialiased rendering.

**get_capstyle**()

Return the *CapStyle*.

**get_clip_path**()

Return the clip path in the form (path, transform), where path is a *Path* instance, and transform is an affine transform to apply to the path before clipping.

**get_clip_rectangle**()

Return the clip rectangle as a *Bbox* instance.

**get_dashes**()

Return the dash style as an (offset, dash-list) pair.

See *set_dashes* for details.

Default value is (None, None).

**get_forced_alpha**()

Return whether the value given by get_alpha() should be used to override any other alpha-channel values.

**get_gid**()

Return the object identifier if one is set, None otherwise.

**get_hatch**()

Get the current hatch style.

**get_hatch_color**()

Get the hatch color.

**get_hatch_linewidth**()

Get the hatch linewidth.

**get_hatch_path**(*density=6.0*)

Return a *Path* for the current hatch.

**get_joinstyle**()

Return the *JoinStyle*.

**get_linewidth**()

Return the line width in points.

**get_rgb**()

Return a tuple of three or four floats from 0-1.

**get_sketch_params**()

Return the sketch parameters for the artist.

**Returns**

**tuple or None**

A 3-tuple with the following elements:

- `scale`: The amplitude of the wiggle perpendicular to the source line.
- `length`: The length of the wiggle along the line.
- `randomness`: The scale factor by which the length is shrunken or expanded.

May return `None` if no sketch parameters were set.

**get_snap**()

Return the snap setting, which can be:

- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

**get_url**()

Return a url if one is set, None otherwise.

**restore**()

Restore the graphics context from the stack - needed only for backends that save graphics contexts on a stack.

**set_alpha**(*alpha*)

Set the alpha value used for blending - not supported on all backends.

If `alpha=None` (the default), the alpha components of the foreground and fill colors will be used to set their respective transparencies (where applicable); otherwise, `alpha` will override them.

**set_antialiased**(*b*)

Set whether object should be drawn with antialiased rendering.

**set_capstyle**(*cs*)

Set how to draw endpoints of lines.

**Parameters**

**cs**

[*CapStyle* or {'butt', 'projecting', 'round'}]

**set_clip_path**(*path*)

Set the clip path to a *TransformedPath* or None.

**set_clip_rectangle**(*rectangle*)

Set the clip rectangle to a *Bbox* or None.

**set_dashes**(*dash_offset*, *dash_list*)

Set the dash style for the gc.

**Parameters**

**dash_offset**

[float] Distance, in points, into the dash pattern at which to start the pattern. It is usually set to 0.

**dash_list**

[array-like or None] The on-off sequence as points. None specifies a solid line. All values must otherwise be non-negative (≥ 0).

## Notes

See p. 666 of the PostScript Language Reference for more info.

**set_foreground**(*fg*, *isRGBA=False*)

Set the foreground color.

**Parameters**

**fg**

[*color*]

**isRGBA**

[bool] If *fg* is known to be an `(r, g, b, a)` tuple, *isRGBA* can be set to True to improve performance.

**set_gid**(*id*)

Set the id.

**set_hatch**(*hatch*)

Set the hatch style (for fills).

**set_hatch_color**(*hatch_color*)

Set the hatch color.

**set_joinstyle**(*js*)

Set how to draw connections between line segments.

**Parameters**

**js**

[*JoinStyle* or {'miter', 'round', 'bevel'}]

**set_linewidth**(*w*)

Set the linewidth in points.

**set_sketch_params**(*scale=None*, *length=None*, *randomness=None*)

Set the sketch parameters.

**Parameters**

**scale**

[float, optional] The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is `None`, or not provided, no sketch filter will be provided.

**length**

[float, default: 128] The length of the wiggle along the line, in pixels.

**randomness**

[float, default: 16] The scale factor by which the length is shrunken or expanded.

**set_snap**(*snap*)

Set the snap setting which may be:

- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

**set_url**(*url*)

Set the url for links in compatible backends.

**class** matplotlib.backend_bases.**KeyEvent**(*name*, *canvas*, *key*, *x=0*, *y=0*, *guiEvent=None*)

Bases: *LocationEvent*

A key event (key press, key release).

A KeyEvent has a number of special attributes in addition to those defined by the parent *Event* and *LocationEvent* classes.

### Notes

Modifier keys will be prefixed to the pressed key and will be in the order "ctrl", "alt", "super". The exception to this rule is when the pressed key is itself a modifier key, therefore "ctrl+alt" and "alt+control" can both be valid key values.

### Examples

```
def on_key(event):
    print('you pressed', event.key, event.xdata, event.ydata)

cid = fig.canvas.mpl_connect('key_press_event', on_key)
```

**Attributes**

**key**

[None or str] The key(s) pressed. Could be *None*, a single case sensitive Unicode character ("g", "G", "#", etc.), a special key ("control", "shift", "f1", "up", etc.) or a combination of the above (e.g., "ctrl+alt+g", "ctrl+alt+G").

**class** matplotlib.backend_bases.**LocationEvent**(*name*, *canvas*, *x*, *y*, *guiEvent=None*, ***, *modifiers=None*)

Bases: *Event*

An event that has a screen location.

A LocationEvent has a number of special attributes in addition to those defined by the parent *Event* class.

**Attributes**

**x, y**

[int or None] Event location in pixels from bottom left of canvas.

**inaxes**

[*Axes* or None] The *Axes* instance over which the mouse is, if any.

**xdata, ydata**

[float or None] Data coordinates of the mouse within *inaxes*, or *None* if the mouse is not over an Axes.

**modifiers**

[frozenset] The keyboard modifiers currently being pressed (except for KeyEvent).

**lastevent = <matplotlib.backend_bases.MouseEvent object>**

**class** matplotlib.backend_bases.**MouseButton**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntEnum

**BACK = 8**

**FORWARD = 9**

**LEFT = 1**

**MIDDLE = 2**

**RIGHT = 3**

**class** matplotlib.backend_bases.**MouseEvent**(*name*, *canvas*, *x*, *y*, *button=None*, *key=None*, *step=0*, *dblclick=False*, *guiEvent=None*, ***, *modifiers=None*)

Bases: *LocationEvent*

A mouse event ('button_press_event', 'button_release_event', 'scroll_event', 'motion_notify_event').

A MouseEvent has a number of special attributes in addition to those defined by the parent *Event* and *LocationEvent* classes.

## Examples

```
def on_press(event):
    print('you pressed', event.button, event.xdata, event.ydata)

cid = fig.canvas.mpl_connect('button_press_event', on_press)
```

**Attributes**

**button**

[None or *MouseButton* or {'up', 'down'}] The button pressed. 'up' and 'down' are used for scroll events.

Note that LEFT and RIGHT actually refer to the "primary" and "secondary" buttons, i.e. if the user inverts their left and right buttons ("left-handed setting") then the LEFT button will be the one physically on the right.

If this is unset, *name* is "scroll_event", and *step* is nonzero, then this will be set to "up" or "down" depending on the sign of *step*.

**key**

[None or str] The key pressed when the mouse event triggered, e.g. 'shift'. See *KeyEvent*.

> **Warning:** This key is currently obtained from the last 'key_press_event' or 'key_release_event' that occurred within the canvas. Thus, if the last change of keyboard state occurred while the canvas did not have focus, this attribute will be wrong. On the other hand, the `modifiers` attribute should always be correct, but it can only report on modifier keys.

**step**

[float] The number of scroll steps (positive for 'up', negative for 'down'). This applies only to 'scroll_event' and defaults to 0 otherwise.

**dblclick**

[bool] Whether the event is a double-click. This applies only to 'button_press_event' and is False otherwise. In particular, it's not used in 'button_release_event'.

**class** matplotlib.backend_bases.**NavigationToolbar2**(*canvas*)

Bases: object

Base class for the navigation cursor, version 2.

Backends must implement a canvas that handles connections for 'button_press_event' and 'button_release_event'. See *FigureCanvasBase.mpl_connect()* for more information.

They must also define

***save_figure()***

Save the current figure.

***draw_rubberband()*** **(optional)**

Draw the zoom to rect "rubberband" rectangle.

***set_message()*** **(optional)**

Display message.

***set_history_buttons()*** **(optional)**

You can change the history back / forward buttons to indicate disabled / enabled state.

and override `__init__` to set up the toolbar -- without forgetting to call the base-class init. Typically, `__init__` needs to set up toolbar buttons connected to the *home*, *back*, *forward*, *pan*, *zoom*, and *save_figure* methods and using standard icons in the "images" subdirectory of the data path.

That's it, we'll do the rest!

**back**(**args*)

Move back up the view lim stack.

For convenience of being directly connected as a GUI callback, which often get passed additional parameters, this method accepts arbitrary parameters, but does not use them.

**configure_subplots**(**args*)

**drag_pan**(*event*)

Callback for dragging in pan/zoom mode.

**drag_zoom**(*event*)

Callback for dragging in zoom mode.

**draw_rubberband**(*event*, *x0*, *y0*, *x1*, *y1*)

Draw a rectangle rubberband to indicate zoom limits.

Note that it is not guaranteed that `x0 <= x1` and `y0 <= y1`.

**forward**(**args*)

Move forward in the view lim stack.

For convenience of being directly connected as a GUI callback, which often get passed additional parameters, this method accepts arbitrary parameters, but does not use them.

**home**(**args*)

Restore the original view.

For convenience of being directly connected as a GUI callback, which often get passed additional parameters, this method accepts arbitrary parameters, but does not use them.

**mouse_move**(*event*)

**pan**(**args*)

Toggle the pan/zoom tool.

Pan with left button, zoom with right.

**press_pan**(*event*)

Callback for mouse button press in pan/zoom mode.

**press_zoom**(*event*)

Callback for mouse button press in zoom to rect mode.

**push_current**()

Push the current view limits and position onto the stack.

**release_pan**(*event*)

Callback for mouse button release in pan/zoom mode.

**release_zoom**(*event*)

Callback for mouse button release in zoom to rect mode.

**remove_rubberband**()

Remove the rubberband.

**save_figure**(**args*)

Save the current figure.

**set_history_buttons**()

Enable or disable the back/forward button.

**set_message**(*s*)

Display a message on toolbar or in status bar.

**toolitems = (('Home', 'Reset original view', 'home', 'home'), ('Back', 'Back to previous view', 'back', 'back'), ('Forward', 'Forward to next view', 'forward', 'forward'), (None, None, None, None), ('Pan', 'Left button pans, Right button zooms\nx/y fixes axis, CTRL fixes aspect', 'move', 'pan'), ('Zoom', 'Zoom to rectangle\nx/y fixes axis', 'zoom_to_rect', 'zoom'), ('Subplots', 'Configure subplots', 'subplots', 'configure_subplots'), (None, None, None, None), ('Save', 'Save the figure', 'filesave', 'save_figure'))**

**update**()

Reset the Axes stack.

**zoom**(**args*)

**exception** matplotlib.backend_bases.**NonGuiException**

Bases: Exception

Raised when trying show a figure in a non-GUI backend.

**class** matplotlib.backend_bases.**PickEvent**(*name*, *canvas*, *mouseevent*, *artist*, *guiEvent=None*, ***kwargs*)

Bases: *Event*

A pick event.

This event is fired when the user picks a location on the canvas sufficiently close to an artist that has been made pickable with *Artist.set_picker*.

A PickEvent has a number of special attributes in addition to those defined by the parent *Event* class.

### Examples

Bind a function `on_pick()` to pick events, that prints the coordinates of the picked data point:

```
ax.plot(np.rand(100), 'o', picker=5)  # 5 points tolerance

def on_pick(event):
    line = event.artist
    xdata, ydata = line.get_data()
    ind = event.ind
    print(f'on pick line: {xdata[ind]:.3f}, {ydata[ind]:.3f}')

cid = fig.canvas.mpl_connect('pick_event', on_pick)
```

**Attributes**

**mouseevent**

[*MouseEvent*] The mouse event that generated the pick.

**artist**

[*Artist*] The picked artist. Note that artists are not pickable by default (see *Artist.set_picker*).

**other**

Additional attributes may be present depending on the type of the picked object; e.g., a *Line2D* pick may define different extra attributes than a *PatchCollection* pick.

**class** matplotlib.backend_bases.**RendererBase**

Bases: object

An abstract base class to handle drawing/rendering operations.

The following methods must be implemented in the backend for full functionality (though just implementing *draw_path* alone would give a highly capable backend):

- *draw_path*
- *draw_image*
- *draw_gouraud_triangles*

The following methods *should* be implemented in the backend for optimization reasons:

- *draw_text*
- *draw_markers*
- *draw_path_collection*
- *draw_quad_mesh*

**close_group**(*s*)

Close a grouping element with label *s*.

Only used by the SVG renderer.

**draw_gouraud_triangles**(*gc*, *triangles_array*, *colors_array*, *transform*)

Draw a series of Gouraud triangles.

**Parameters**

**gc**

[*GraphicsContextBase*] The graphics context.

**triangles_array**

[(N, 3, 2) array-like] Array of *N* (x, y) points for the triangles.

**colors_array**

[(N, 3, 4) array-like] Array of *N* RGBA colors for each point of the triangles.

**transform**

[*Transform*] An affine transform to apply to the points.

**draw_image**(*gc*, *x*, *y*, *im*, *transform=None*)

Draw an RGBA image.

**Parameters**

**gc**

[*GraphicsContextBase*] A graphics context with clipping information.

**x**

[scalar] The distance in physical units (i.e., dots or pixels) from the left hand side of the canvas.

**y**

[scalar] The distance in physical units (i.e., dots or pixels) from the bottom side of the canvas.

**im**

[(N, M, 4) array of `numpy.uint8`] An array of RGBA pixels.

**transform**

[*Affine2DBase*] If and only if the concrete backend is written such that *option_scale_image* returns `True`, an affine transformation (i.e., an *Affine2DBase*) *may* be passed to *draw_image*. The translation vector of the transformation is given in physical units (i.e., dots or pixels). Note that the transformation does not override *x* and *y*, and has to be applied *before* translating

the result by *x* and *y* (this can be accomplished by adding *x* and *y* to the translation vector defined by *transform*).

**draw_markers**(*gc*, *marker_path*, *marker_trans*, *path*, *trans*, *rgbFace=None*)

Draw a marker at each of *path*'s vertices (excluding control points).

The base (fallback) implementation makes multiple calls to `draw_path`. Backends may want to override this method in order to draw the marker only once and reuse it multiple times.

**Parameters**

**gc**

[`GraphicsContextBase`] The graphics context.

**marker_path**

[`Path`] The path for the marker.

**marker_trans**

[`Transform`] An affine transform applied to the marker.

**path**

[`Path`] The locations to draw the markers.

**trans**

[`Transform`] An affine transform applied to the path.

**rgbFace**

[*color*, optional]

**draw_path**(*gc*, *path*, *transform*, *rgbFace=None*)

Draw a `Path` instance using the given affine transform.

**draw_path_collection**(*gc*, *master_transform*, *paths*, *all_transforms*, *offsets*, *offset_trans*, *facecolors*, *edgecolors*, *linewidths*, *linestyles*, *antialiaseds*, *urls*, *offset_position*)

Draw a collection of *paths*.

Each path is first transformed by the corresponding entry in *all_transforms* (a list of (3, 3) matrices) and then by *master_transform*. They are then translated by the corresponding entry in *offsets*, which has been first transformed by *offset_trans*.

*facecolors*, *edgecolors*, *linewidths*, *linestyles*, and *antialiased* are lists that set the corresponding properties.

*offset_position* is unused now, but the argument is kept for backwards compatibility.

The base (fallback) implementation makes multiple calls to `draw_path`. Backends may want to override this in order to render each set of path data only once, and then reference that path multiple times with the different offsets, colors, styles etc. The generator methods `_iter_collection_raw_paths` and `_iter_collection` are provided to help with

(and standardize) the implementation across backends. It is highly recommended to use those generators, so that changes to the behavior of *draw_path_collection* can be made globally.

**draw_quad_mesh**(*gc*, *master_transform*, *meshWidth*, *meshHeight*, *coordinates*, *offsets*, *offsetTrans*, *facecolors*, *antialiased*, *edgecolors*)

Draw a quadmesh.

The base (fallback) implementation converts the quadmesh to paths and then calls *draw_path_collection*.

**draw_tex**(*gc*, *x*, *y*, *s*, *prop*, *angle*, *, *mtext=None*)

Draw a TeX instance.

**Parameters**

**gc**

[*GraphicsContextBase*] The graphics context.

**x**

[float] The x location of the text in display coords.

**y**

[float] The y location of the text baseline in display coords.

**s**

[str] The TeX text string.

**prop**

[*FontProperties*] The font properties.

**angle**

[float] The rotation angle in degrees anti-clockwise.

**mtext**

[*Text*] The original text object to be rendered.

**draw_text**(*gc*, *x*, *y*, *s*, *prop*, *angle*, *ismath=False*, *mtext=None*)

Draw a text instance.

**Parameters**

**gc**

[*GraphicsContextBase*] The graphics context.

**x**

[float] The x location of the text in display coords.

**y**

[float] The y location of the text baseline in display coords.

**s**

[str] The text string.

**prop**

[*FontProperties*] The font properties.

**angle**

[float] The rotation angle in degrees anti-clockwise.

**ismath**

[bool or "TeX"] If True, use mathtext parser. If "TeX", use tex for rendering.

**mtext**

[*Text*] The original text object to be rendered.

### Notes

**Note for backend implementers:**

When you are trying to determine if you have gotten your bounding box right (which is what enables the text layout/alignment to work properly), it helps to change the line in text.py:

```
if 0: bbox_artist(self, renderer)
```

to if 1, and then the actual bounding box will be plotted along with your text.

**flipy**()

Return whether y values increase from top to bottom.

Note that this only affects drawing of texts.

**get_canvas_width_height**()

Return the canvas width and height in display coords.

**get_image_magnification**()

Get the factor by which to magnify images passed to *draw_image*. Allows a backend to have images at a different resolution to other artists.

**get_texmanager**()

Return the *TexManager* instance.

**get_text_width_height_descent**(*s*, *prop*, *ismath*)

Get the width, height, and descent (offset from the bottom to the baseline), in display coords, of the string *s* with *FontProperties* *prop*.

Whitespace at the start and the end of *s* is included in the reported width.

**new_gc**()

Return an instance of a *GraphicsContextBase*.

**open_group**(*s*, *gid=None*)

Open a grouping element with label *s* and *gid* (if set) as id.

Only used by the SVG renderer.

**option_image_nocomposite**()

Return whether image composition by Matplotlib should be skipped.

Raster backends should usually return False (letting the C-level rasterizer take care of image composition); vector backends should usually return `not rcParams["image.composite_image"]`.

**option_scale_image**()

Return whether arbitrary affine transformations in *draw_image* are supported (True for most vector backends).

**points_to_pixels**(*points*)

Convert points to display units.

You need to override this function (unless your backend doesn't have a dpi, e.g., postscript or svg). Some imaging systems assume some value for pixels per inch:

```
points to pixels = points * pixels_per_inch/72 * dpi/72
```

**Parameters**

**points**

[float or array-like]

**Returns**

**Points converted to pixels**

**start_filter**()

Switch to a temporary renderer for image filtering effects.

Currently only supported by the agg renderer.

**start_rasterizing**()

Switch to the raster renderer.

Used by *MixedModeRenderer*.

**stop_filter**(*filter_func*)

Switch back to the original renderer. The contents of the temporary renderer is processed with the *filter_func* and is drawn on the original renderer as an image.

Currently only supported by the agg renderer.

**stop_rasterizing**()

Switch back to the vector renderer and draw the contents of the raster renderer as an image on the vector renderer.

Used by *MixedModeRenderer*.

**class** matplotlib.backend_bases.**ResizeEvent**(*name*, *canvas*)

Bases: *Event*

An event triggered by a canvas resize.

A ResizeEvent has a number of special attributes in addition to those defined by the parent *Event* class.

**Attributes**

**width**

[int] Width of the canvas in pixels.

**height**

[int] Height of the canvas in pixels.

**class** matplotlib.backend_bases.**ShowBase**

Bases: _Backend

Simple base class to generate a show() function in backends.

Subclass must override mainloop() method.

**class** matplotlib.backend_bases.**TimerBase**(*interval=None*, *callbacks=None*)

Bases: object

A base class for providing timer events, useful for things animations. Backends need to implement a few specific methods in order to use their own timing mechanisms so that the timer events are integrated into their event loops.

Subclasses must override the following methods:

- _timer_start: Backend-specific code for starting the timer.
- _timer_stop: Backend-specific code for stopping the timer.

Subclasses may additionally override the following methods:

- _timer_set_single_shot: Code for setting the timer to single shot operating mode, if supported by the timer object. If not, the Timer class itself will store the flag and the _on_timer method should be overridden to support such behavior.
- _timer_set_interval: Code for setting the interval on the timer, if there is a method for doing so on the timer object.
- _on_timer: The internal function that any timer object should call, which will handle the task of running all callbacks that have been set.

**Parameters**

**interval**

[int, default: 1000ms] The time between timer events in milliseconds. Will be stored as `timer.interval`.

**callbacks**

[list[tuple[callable, tuple, dict]]] List of (func, args, kwargs) tuples that will be called upon timer events. This list is accessible as `timer.callbacks` and can be manipulated directly, or the functions *add_callback* and *remove_callback* can be used.

**add_callback**(*func*, **args*, ***kwargs*)

Register *func* to be called by timer when the event fires. Any additional arguments provided will be passed to *func*.

This function returns *func*, which makes it possible to use it as a decorator.

**property interval**

The time between timer events, in milliseconds.

**remove_callback**(*func*, **args*, ***kwargs*)

Remove *func* from list of callbacks.

*args* and *kwargs* are optional and used to distinguish between copies of the same function registered to be called with different arguments. This behavior is deprecated. In the future, `*args, **kwargs` won't be considered anymore; to keep a specific callback removable by itself, pass it to *add_callback* as a `functools.partial` object.

**property single_shot**

Whether this timer should stop after a single run.

**start**(*interval=<deprecated parameter>*)

Start the timer object.

**Parameters**

**interval**

[int, optional] Timer interval in milliseconds; overrides a previously set interval if provided.

**stop**()

Stop the timer.

**class** matplotlib.backend_bases.**ToolContainerBase**(*toolmanager*)

Bases: `object`

Base class for all tool containers, e.g. toolbars.

**Attributes**

**toolmanager**

[*ToolManager*] The tools with which this `ToolContainer` wants to communicate.

**add_tool**(*tool*, *group*, *position=-1*)

Add a tool to this container.

**Parameters**

**tool**

[tool_like] The tool to add, see *ToolManager.get_tool*.

**group**

[str] The name of the group to add this tool to.

**position**

[int, default: -1] The position within the group to place this tool.

**add_toolitem**(*name*, *group*, *position*, *image*, *description*, *toggle*)

A hook to add a toolitem to the container.

This hook must be implemented in each backend and contains the backend-specific code to add an element to the toolbar.

**Warning:** This is part of the backend implementation and should not be called by end-users. They should instead call *ToolContainerBase.add_tool*.

The callback associated with the button click event must be *exactly* `self.trigger_tool(name)`.

**Parameters**

**name**

[str] Name of the tool to add, this gets used as the tool's ID and as the default label of the buttons.

**group**

[str] Name of the group that this tool belongs to.

**position**

[int] Position of the tool within its group, if -1 it goes at the end.

**image**

[str] Filename of the image for the button or `None`.

**description**

[str] Description of the tool, used for the tooltips.

**toggle**

[bool]

- `True` : The button is a toggle (change the pressed/unpressed state between consecutive clicks).
- `False` : The button is a normal button (returns to unpressed state after release).

**remove_toolitem**(*name*)

A hook to remove a toolitem from the container.

This hook must be implemented in each backend and contains the backend-specific code to remove an element from the toolbar; it is called when *ToolManager* emits a `tool_removed_event`.

Because some tools are present only on the *ToolManager* but not on the `ToolContainer`, this method must be a no-op when called on a tool absent from the container.

> **Warning:** This is part of the backend implementation and should not be called by end-users. They should instead call *ToolManager.remove_tool*.

**Parameters**

**name**

[str] Name of the tool to remove.

**set_message**(*s*)

Display a message on the toolbar.

**Parameters**

**s**

[str] Message text.

**toggle_toolitem**(*name*, *toggled*)

A hook to toggle a toolitem without firing an event.

This hook must be implemented in each backend and contains the backend-specific code to silently toggle a toolbar element.

**Warning:** This is part of the backend implementation and should not be called by end-users. They should instead call *ToolManager.trigger_tool* or *ToolContainerBase.trigger_tool* (which are equivalent).

**Parameters**

**name**

[str] Id of the tool to toggle.

**toggled**

[bool] Whether to set this tool as toggled or not.

**trigger_tool**(*name*)

Trigger the tool.

**Parameters**

**name**

[str] Name (id) of the tool triggered from within the container.

matplotlib.backend_bases.**button_press_handler**(*event*, *canvas=None*, *toolbar=None*)

The default Matplotlib button actions for extra mouse buttons.

Parameters are as for *key_press_handler*, except that *event* is a *MouseEvent*.

matplotlib.backend_bases.**get_registered_canvas_class**(*format*)

Return the registered default canvas for given file format. Handles deferred import of required backend.

matplotlib.backend_bases.**key_press_handler**(*event*, *canvas=None*, *toolbar=None*)

Implement the default Matplotlib key bindings for the canvas and toolbar described at *Navigation keyboard shortcuts*.

**Parameters**

**event**

[*KeyEvent*] A key press/release event.

**canvas**

[*FigureCanvasBase*, default: `event.canvas`] The backend-specific canvas instance. This parameter is kept for back-compatibility, but, if set, should always be equal to `event.canvas`.

**toolbar**

[*NavigationToolbar2*, default: `event.canvas.toolbar`] The navigation cursor toolbar. This parameter is kept for back-compatibility, but, if set, should always be equal to `event.canvas.toolbar`.

matplotlib.backend_bases.**register_backend**(*format*, *backend*, *description=None*)

Register a backend for saving to a given file format.

**Parameters**

**format**

[str] File extension

**backend**

[module string or canvas class] Backend for handling file output

**description**

[str, default: ""] Description of the file type.

### 7.2.8 matplotlib.backend_managers

**class** matplotlib.backend_managers.**ToolEvent**(*name*, *sender*, *tool*, *data=None*)

Bases: object

Event for tool manipulation (add/remove).

**class** matplotlib.backend_managers.**ToolManager**(*figure=None*)

Bases: object

Manager for actions triggered by user interactions (key press, toolbar clicks, ...) on a Figure.

**Attributes**

***figure***

[*Figure*] Figure that holds the canvas.

**keypresslock**

[*LockDraw*] *LockDraw* object to know if the *canvas* key_press_event is locked.

**messagelock**

[*LockDraw*] *LockDraw* object to know if the message is available to write.

**property active_toggle**

Currently toggled tools.

**add_tool**(*name*, *tool*, **args*, ***kwargs*)

Add *tool* to *ToolManager*.

If successful, adds a new event `tool_trigger_{name}` where `{name}` is the *name* of the tool; the event is fired every time the tool is triggered.

**Parameters**

**name**

[str] Name of the tool, treated as the ID, has to be unique.

**tool**

[type] Class of the tool to be added. A subclass will be used instead if one was registered for the current canvas class.

***args, **kwargs**

Passed to the *tool*'s constructor.

**See also:**

***matplotlib.backend_tools.ToolBase***

The base class for tools.

**property canvas**

Canvas managed by FigureManager.

**property figure**

Figure that holds the canvas.

**get_tool**(*name*, *warn=True*)

Return the tool object with the given name.

For convenience, this passes tool objects through.

**Parameters**

**name**

[str or *ToolBase*] Name of the tool, or the tool itself.

**warn**

[bool, default: True] Whether a warning should be emitted it no tool with the given name exists.

**Returns**

***ToolBase* or None**

The tool or None if no tool with the given name exists.

**get_tool_keymap**(*name*)

Return the keymap associated with the specified tool.

**Parameters**

**name**

[str] Name of the Tool.

**Returns**

**list of str**

List of keys associated with the tool.

**message_event**(*message*, *sender=None*)

Emit a *ToolManagerMessageEvent*.

**remove_tool**(*name*)

Remove tool named *name*.

**Parameters**

**name**

[str] Name of the tool.

**set_figure**(*figure*, *update_tools=True*)

Bind the given figure to the tools.

**Parameters**

**figure**

[*Figure*]

**update_tools**

[bool, default: True] Force tools to update figure.

**toolmanager_connect**(*s*, *func*)

Connect event with string *s* to *func*.

**Parameters**

**s**

[str] The name of the event. The following events are recognized:

- 'tool_message_event'
- 'tool_removed_event'
- 'tool_added_event'

For every tool added a new event is created

- 'tool_trigger_TOOLNAME', where TOOLNAME is the id of the tool.

**func**

[callable] Callback function for the toolmanager event with signature:

```
def func(event: ToolEvent) -> Any
```

**Returns**

**cid**

The callback id for the connection. This can be used in *toolmanager_disconnect*.

**toolmanager_disconnect**(*cid*)

Disconnect callback id *cid*.

Example usage:

```
cid = toolmanager.toolmanager_connect('tool_trigger_zoom', onpress)
#...later
toolmanager.toolmanager_disconnect(cid)
```

**property tools**

A dict mapping tool name -> controlled tool.

**trigger_tool**(*name*, *sender=None*, *canvasevent=None*, *data=None*)

Trigger a tool and emit the `tool_trigger_{name}` event.

**Parameters**

**name**

[str] Name of the tool.

**sender**

[object] Object that wishes to trigger the tool.

**canvasevent**

[Event] Original Canvas event or None.

**data**

[object] Extra data to pass to the tool when triggering.

**update_keymap**(*name*, *key*)

Set the keymap to associate with the specified tool.

**Parameters**

**name**

[str] Name of the Tool.

**key**

[str or list of str] Keys to associate with the tool.

**class** matplotlib.backend_managers.**ToolManagerMessageEvent**(*name*, *sender*, *message*)

Bases: `object`

Event carrying messages from toolmanager.

Messages usually get displayed to the user by the toolbar.

**class** matplotlib.backend_managers.**ToolTriggerEvent**(*name*, *sender*, *tool*, *canvasevent=None*, *data=None*)

Bases: *ToolEvent*

Event to inform that a tool has been triggered.

### 7.2.9 matplotlib.backend_tools

Abstract base classes define the primitives for Tools. These tools are used by *matplotlib.backend_managers.ToolManager*

***ToolBase***

Simple stateless tool

***ToolToggleBase***

Tool that has two states, only one Toggle tool can be active at any given time for the same *matplotlib.backend_managers.ToolManager*

**class** matplotlib.backend_tools.**AxisScaleBase**(**args*, ***kwargs*)

Bases: *ToolToggleBase*

Base Tool to toggle between linear and logarithmic.

**disable**(*event=None*)

Disable the toggle tool.

*trigger* call this method when `toggled` is True.

This can happen in different circumstances.

- Click on the toolbar tool button.
- Call to *matplotlib.backend_managers.ToolManager.trigger_tool*.
- Another *ToolToggleBase* derived tool is triggered (from the same *ToolManager*).

**enable**(*event=None*)

Enable the toggle tool.

*trigger* calls this method when `toggled` is False.

**trigger**(*sender*, *event*, *data=None*)

Calls *enable* or *disable* based on `toggled` value.

**class** matplotlib.backend_tools.**ConfigureSubplotsBase**(*toolmanager*, *name*)

Bases: *ToolBase*

Base tool for the configuration of subplots.

**description = 'Configure subplots'**

Description of the Tool.

str: Tooltip used if the Tool is included in a Toolbar.

**image = 'mpl-data/images/subplots'**

Icon filename.

str | None: Filename of the Toolbar icon; either absolute, or relative to the directory containing the Python source file where the Tool.image class attribute is defined (in the latter case, this cannot be defined as an instance attribute). In either case, the extension is optional; leaving it off lets individual backends select the icon format they prefer. If None, the *name* is used as a label in the toolbar button.

**class** matplotlib.backend_tools.**Cursors**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntEnum

Backend-independent cursor types.

**HAND = 2**

**MOVE = 4**

**POINTER = 1**

**RESIZE_HORIZONTAL = 6**

**RESIZE_VERTICAL = 7**

**SELECT_REGION = 3**

**WAIT = 5**

**class** matplotlib.backend_tools.**RubberbandBase**(*toolmanager*, *name*)

Bases: *ToolBase*

Draw and remove a rubberband.

**draw_rubberband**(**data*)

Draw rubberband.

This method must get implemented per backend.

**remove_rubberband**()

Remove rubberband.

This method should get implemented per backend.

**trigger**(*sender*, *event*, *data=None*)

Call *draw_rubberband* or *remove_rubberband* based on data.

**class** matplotlib.backend_tools.**SaveFigureBase**(*toolmanager*, *name*)

Bases: *ToolBase*

Base tool for figure saving.

**property default_keymap**

**description = 'Save the figure'**

Description of the Tool.

str: Tooltip used if the Tool is included in a Toolbar.

**image = 'mpl-data/images/filesave'**

Icon filename.

str | None: Filename of the Toolbar icon; either absolute, or relative to the directory containing the Python source file where the Tool.image class attribute is defined (in the latter case, this cannot be defined as an instance attribute). In either case, the extension is optional; leaving it off lets individual backends select the icon format they prefer. If None, the *name* is used as a label in the toolbar button.

**class** matplotlib.backend_tools.**ToolBack**(*toolmanager*, *name*)

Bases: *ViewsPositionsBase*

Move back up the view limits stack.

**property default_keymap**

**description = 'Back to previous view'**

Description of the Tool.

str: Tooltip used if the Tool is included in a Toolbar.

**image = 'mpl-data/images/back'**

Icon filename.

str | None: Filename of the Toolbar icon; either absolute, or relative to the directory containing the Python source file where the Tool.image class attribute is defined (in the latter case, this cannot be defined as an instance attribute). In either case, the extension is optional; leaving it off lets individual backends select the icon format they prefer. If None, the *name* is used as a label in the toolbar button.

**class** matplotlib.backend_tools.**ToolBase**(*toolmanager*, *name*)

Bases: object

Base tool class.

A base tool, only implements *trigger* method or no method at all. The tool is instantiated by *matplotlib.backend_managers.ToolManager*.

**property canvas**

The canvas of the figure affected by this tool, or None.

**default_keymap = None**

Keymap to associate with this tool.

`list[str]`: List of keys that will trigger this tool when a keypress event is emitted on `self.figure.canvas`. Note that this attribute is looked up on the instance, and can therefore be a property (this is used e.g. by the built-in tools to load the rcParams at instantiation time).

**description = None**

Description of the Tool.

`str`: Tooltip used if the Tool is included in a Toolbar.

**property figure**

The Figure affected by this tool, or None.

**image = None**

Icon filename.

`str | None`: Filename of the Toolbar icon; either absolute, or relative to the directory containing the Python source file where the `Tool.image` class attribute is defined (in the latter case, this cannot be defined as an instance attribute). In either case, the extension is optional; leaving it off lets individual backends select the icon format they prefer. If None, the *name* is used as a label in the toolbar button.

**property name**

The tool id (str, must be unique among tools of a tool manager).

**set_figure**(*figure*)

**property toolmanager**

The *ToolManager* that controls this tool.

**trigger**(*sender*, *event*, *data=None*)

Called when this tool gets used.

This method is called by *ToolManager.trigger_tool*.

**Parameters**

**event**

[*Event*] The canvas event that caused this tool to be called.

**sender**

[object] Object that requested the tool to be triggered.

**data**

[object] Extra data.

**class** matplotlib.backend_tools.**ToolCopyToClipboardBase**(*toolmanager*, *name*)

Bases: *ToolBase*

Tool to copy the figure to the clipboard.

**property default_keymap**

**description = 'Copy the canvas figure to clipboard'**

Description of the Tool.

str: Tooltip used if the Tool is included in a Toolbar.

**trigger**(**args*, ***kwargs*)

Called when this tool gets used.

This method is called by *ToolManager.trigger_tool*.

**Parameters**

**event**

[*Event*] The canvas event that caused this tool to be called.

**sender**

[object] Object that requested the tool to be triggered.

**data**

[object] Extra data.

**class** matplotlib.backend_tools.**ToolCursorPosition**(**args*, ***kwargs*)

Bases: *ToolBase*

Send message with the current pointer position.

This tool runs in the background reporting the position of the cursor.

**send_message**(*event*)

Call *matplotlib.backend_managers.ToolManager.message_event*.

**set_figure**(*figure*)

**class** matplotlib.backend_tools.**ToolForward**(*toolmanager*, *name*)

Bases: *ViewsPositionsBase*

Move forward in the view lim stack.

**property default_keymap**

**description = 'Forward to next view'**

Description of the Tool.

str: Tooltip used if the Tool is included in a Toolbar.

**image = 'mpl-data/images/forward'**

Icon filename.

`str | None`: Filename of the Toolbar icon; either absolute, or relative to the directory containing the Python source file where the `Tool.image` class attribute is defined (in the latter case, this cannot be defined as an instance attribute). In either case, the extension is optional; leaving it off lets individual backends select the icon format they prefer. If None, the *name* is used as a label in the toolbar button.

**class** matplotlib.backend_tools.**ToolFullScreen**(*toolmanager*, *name*)

Bases: *ToolBase*

Tool to toggle full screen.

**property default_keymap**

**description = 'Toggle fullscreen mode'**

Description of the Tool.

`str`: Tooltip used if the Tool is included in a Toolbar.

**trigger**(*sender*, *event*, *data=None*)

Called when this tool gets used.

This method is called by *ToolManager.trigger_tool*.

**Parameters**

**event**

[*Event*] The canvas event that caused this tool to be called.

**sender**

[object] Object that requested the tool to be triggered.

**data**

[object] Extra data.

**class** matplotlib.backend_tools.**ToolGrid**(*toolmanager*, *name*)

Bases: *ToolBase*

Tool to toggle the major grids of the figure.

**property default_keymap**

**description = 'Toggle major grids'**

Description of the Tool.

`str`: Tooltip used if the Tool is included in a Toolbar.

**trigger**(*sender*, *event*, *data=None*)

Called when this tool gets used.

This method is called by *ToolManager.trigger_tool*.

**Parameters**

**event**

[*Event*] The canvas event that caused this tool to be called.

**sender**

[object] Object that requested the tool to be triggered.

**data**

[object] Extra data.

**class** matplotlib.backend_tools.**ToolHelpBase**(*toolmanager*, *name*)

Bases: *ToolBase*

**property default_keymap**

**description = 'Print tool list, shortcuts and description'**

Description of the Tool.

`str`: Tooltip used if the Tool is included in a Toolbar.

**static format_shortcut**(*key_sequence*)

Convert a shortcut string from the notation used in rc config to the standard notation for displaying shortcuts, e.g. 'ctrl+a' -> 'Ctrl+A'.

**image = 'mpl-data/images/help'**

Icon filename.

`str | None`: Filename of the Toolbar icon; either absolute, or relative to the directory containing the Python source file where the `Tool.image` class attribute is defined (in the latter case, this cannot be defined as an instance attribute). In either case, the extension is optional; leaving it off lets individual backends select the icon format they prefer. If None, the *name* is used as a label in the toolbar button.

**class** matplotlib.backend_tools.**ToolHome**(*toolmanager*, *name*)

Bases: *ViewsPositionsBase*

Restore the original view limits.

**property default_keymap**

**description = 'Reset original view'**

Description of the Tool.

`str`: Tooltip used if the Tool is included in a Toolbar.

**image = 'mpl-data/images/home'**

Icon filename.

`str | None`: Filename of the Toolbar icon; either absolute, or relative to the directory containing the Python source file where the `Tool.image` class attribute is defined (in the latter case, this cannot be defined as an instance attribute). In either case, the extension is optional;

leaving it off lets individual backends select the icon format they prefer. If None, the *name* is used as a label in the toolbar button.

**class** matplotlib.backend_tools.**ToolMinorGrid**(*toolmanager*, *name*)

Bases: *ToolBase*

Tool to toggle the major and minor grids of the figure.

**property default_keymap**

**description = 'Toggle major and minor grids'**

Description of the Tool.

str: Tooltip used if the Tool is included in a Toolbar.

**trigger**(*sender*, *event*, *data=None*)

Called when this tool gets used.

This method is called by *ToolManager.trigger_tool*.

**Parameters**

**event**

[*Event*] The canvas event that caused this tool to be called.

**sender**

[object] Object that requested the tool to be triggered.

**data**

[object] Extra data.

**class** matplotlib.backend_tools.**ToolPan**(**args*)

Bases: *ZoomPanBase*

Pan Axes with left mouse, zoom with right.

**cursor = 4**

Cursor to use when the tool is active.

**property default_keymap**

**description = 'Pan axes with left mouse, zoom with right'**

Description of the Tool.

str: Tooltip used if the Tool is included in a Toolbar.

**image = 'mpl-data/images/move'**

Icon filename.

str | None: Filename of the Toolbar icon; either absolute, or relative to the directory containing the Python source file where the Tool.image class attribute is defined (in the latter case, this cannot be defined as an instance attribute). In either case, the extension is optional;

leaving it off lets individual backends select the icon format they prefer. If None, the *name* is used as a label in the toolbar button.

**radio_group = 'default'**

Attribute to group 'radio' like tools (mutually exclusive).

`str` that identifies the group or **None** if not belonging to a group.

**class** matplotlib.backend_tools.**ToolQuit**(*toolmanager*, *name*)

Bases: *ToolBase*

Tool to call the figure manager destroy method.

**property default_keymap**

**description = 'Quit the figure'**

Description of the Tool.

`str`: Tooltip used if the Tool is included in a Toolbar.

**trigger**(*sender*, *event*, *data=None*)

Called when this tool gets used.

This method is called by *ToolManager.trigger_tool*.

**Parameters**

**event**

[*Event*] The canvas event that caused this tool to be called.

**sender**

[object] Object that requested the tool to be triggered.

**data**

[object] Extra data.

**class** matplotlib.backend_tools.**ToolQuitAll**(*toolmanager*, *name*)

Bases: *ToolBase*

Tool to call the figure manager destroy method.

**property default_keymap**

**description = 'Quit all figures'**

Description of the Tool.

`str`: Tooltip used if the Tool is included in a Toolbar.

**trigger**(*sender*, *event*, *data=None*)

Called when this tool gets used.

This method is called by *ToolManager.trigger_tool*.

**Parameters**

**event**

[*Event*] The canvas event that caused this tool to be called.

**sender**

[object] Object that requested the tool to be triggered.

**data**

[object] Extra data.

**class** matplotlib.backend_tools.**ToolSetCursor**(**args*, ***kwargs*)

Bases: *ToolBase*

Change to the current cursor while inaxes.

This tool, keeps track of all *ToolToggleBase* derived tools, and updates the cursor when a tool gets triggered.

**set_figure**(*figure*)

**class** matplotlib.backend_tools.**ToolToggleBase**(**args*, ***kwargs*)

Bases: *ToolBase*

Toggleable tool.

Every time it is triggered, it switches between enable and disable.

**Parameters**

**``*args``**

Variable length argument to be used by the Tool.

**``**kwargs``**

*toggled* if present and True, sets the initial state of the Tool Arbitrary keyword arguments to be consumed by the Tool

**cursor = None**

Cursor to use when the tool is active.

**default_toggled = False**

Default of toggled state.

**disable**(*event=None*)

Disable the toggle tool.

*trigger* call this method when *toggled* is True.

This can happen in different circumstances.

- Click on the toolbar tool button.
- Call to *matplotlib.backend_managers.ToolManager.trigger_tool*.

- Another *ToolToggleBase* derived tool is triggered (from the same *ToolManager*).

**enable**(*event=None*)

Enable the toggle tool.

*trigger* calls this method when *toggled* is False.

**radio_group = None**

Attribute to group 'radio' like tools (mutually exclusive).

str that identifies the group or **None** if not belonging to a group.

**set_figure**(*figure*)

**property toggled**

State of the toggled tool.

**trigger**(*sender*, *event*, *data=None*)

Calls *enable* or *disable* based on *toggled* value.

**class** matplotlib.backend_tools.**ToolViewsPositions**(**args*, ***kwargs*)

Bases: *ToolBase*

Auxiliary Tool to handle changes in views and positions.

Runs in the background and should get used by all the tools that need to access the figure's history of views and positions, e.g.

- *ToolZoom*
- *ToolPan*
- *ToolHome*
- *ToolBack*
- *ToolForward*

**add_figure**(*figure*)

Add the current figure to the stack of views and positions.

**back**()

Back one step in the stack of views and positions.

**clear**(*figure*)

Reset the Axes stack.

**forward**()

Forward one step in the stack of views and positions.

**home**()

Recall the first view and position from the stack.

**push_current**(*figure=None*)

Push the current view limits and position onto their respective stacks.

**update_home_views**(*figure=None*)

Make sure that `self.home_views` has an entry for all Axes present in the figure.

**update_view**()

Update the view limits and position for each Axes from the current stack position. If any Axes are present in the figure that aren't in the current stack position, use the home view limits for those Axes and don't update *any* positions.

**class** matplotlib.backend_tools.**ToolXScale**(**args*, ***kwargs*)

Bases: *AxisScaleBase*

Tool to toggle between linear and logarithmic scales on the X axis.

**property default_keymap**

**description = 'Toggle scale X axis'**

Description of the Tool.

`str`: Tooltip used if the Tool is included in a Toolbar.

**set_scale**(*ax*, *scale*)

**class** matplotlib.backend_tools.**ToolYScale**(**args*, ***kwargs*)

Bases: *AxisScaleBase*

Tool to toggle between linear and logarithmic scales on the Y axis.

**property default_keymap**

**description = 'Toggle scale Y axis'**

Description of the Tool.

`str`: Tooltip used if the Tool is included in a Toolbar.

**set_scale**(*ax*, *scale*)

**class** matplotlib.backend_tools.**ToolZoom**(**args*)

Bases: *ZoomPanBase*

A Tool for zooming using a rectangle selector.

**cursor = 3**

Cursor to use when the tool is active.

**property default_keymap**

**description = 'Zoom to rectangle'**

Description of the Tool.

`str`: Tooltip used if the Tool is included in a Toolbar.

**image = 'mpl-data/images/zoom_to_rect'**

Icon filename.

`str | None`: Filename of the Toolbar icon; either absolute, or relative to the directory containing the Python source file where the `Tool.image` class attribute is defined (in the latter case, this cannot be defined as an instance attribute). In either case, the extension is optional; leaving it off lets individual backends select the icon format they prefer. If None, the *name* is used as a label in the toolbar button.

**radio_group = 'default'**

Attribute to group 'radio' like tools (mutually exclusive).

`str` that identifies the group or **None** if not belonging to a group.

**class** matplotlib.backend_tools.**ViewsPositionsBase**(*toolmanager*, *name*)

Bases: *ToolBase*

Base class for *ToolHome*, *ToolBack* and *ToolForward*.

**trigger**(*sender*, *event*, *data=None*)

Called when this tool gets used.

This method is called by *ToolManager.trigger_tool*.

**Parameters**

**event**

[*Event*] The canvas event that caused this tool to be called.

**sender**

[object] Object that requested the tool to be triggered.

**data**

[object] Extra data.

**class** matplotlib.backend_tools.**ZoomPanBase**(**args*)

Bases: *ToolToggleBase*

Base class for *ToolZoom* and *ToolPan*.

**disable**(*event=None*)

Release the canvas and disconnect press/release events.

**enable**(*event=None*)

Connect press/release events and lock the canvas.

**scroll_zoom**(*event*)

**trigger**(*sender*, *event*, *data=None*)

Calls *enable* or *disable* based on `toggled` value.

matplotlib.backend_tools.**add_tools_to_container**(*container*, *tools=[['navigation', ['home', 'back', 'forward']], ['zoompan', ['pan', 'zoom', 'subplots']], ['io', ['save', 'help']]]*)

Add multiple tools to the container.

**Parameters**

**container**

[Container] *backend_bases.ToolContainerBase* object that will get the tools added.

**tools**

[list, optional] List in the form `[[group1, [tool1, tool2 ...]], [group2, [...]]]` where the tools `[tool1, tool2, ...]` will display in group1. See *backend_bases.ToolContainerBase.add_tool* for details.

matplotlib.backend_tools.**add_tools_to_manager**(*toolmanager*, *tools={'back': <class 'matplotlib.backend_tools.ToolBack'>, 'copy': <class 'matplotlib.backend_tools.ToolCopyToClipboardBase'>, 'cursor': <class 'matplotlib.backend_tools.ToolSetCursor'>, 'forward': <class 'matplotlib.backend_tools.ToolForward'>, 'fullscreen': <class 'matplotlib.backend_tools.ToolFullScreen'>, 'grid': <class 'matplotlib.backend_tools.ToolGrid'>, 'grid_minor': <class 'matplotlib.backend_tools.ToolMinorGrid'>, 'help': <class 'matplotlib.backend_tools.ToolHelpBase'>, 'home': <class 'matplotlib.backend_tools.ToolHome'>, 'pan': <class 'matplotlib.backend_tools.ToolPan'>, 'position': <class 'matplotlib.backend_tools.ToolCursorPosition'>, 'quit': <class 'matplotlib.backend_tools.ToolQuit'>, 'quit_all': <class 'matplotlib.backend_tools.ToolQuitAll'>, 'rubberband': <class 'matplotlib.backend_tools.RubberbandBase'>, 'save': <class 'matplotlib.backend_tools.SaveFigureBase'>, 'subplots': <class 'matplotlib.backend_tools.ConfigureSubplotsBase'>, 'viewpos': <class 'matplotlib.backend_tools.ToolViewsPositions'>, 'xscale': <class 'matplotlib.backend_tools.ToolXScale'>, 'yscale': <class 'matplotlib.backend_tools.ToolYScale'>, 'zoom': <class 'matplotlib.backend_tools.ToolZoom'>}*)

Add multiple tools to a *ToolManager*.

**Parameters**

**toolmanager**

[*backend_managers.ToolManager*] Manager to which the tools are added.

**tools**

[{str: class_like}, optional] The tools to add in a {name: tool} dict, see *backend_managers.ToolManager.add_tool* for more info.

matplotlib.backend_tools.**cursors**

alias of *Cursors*

### 7.2.10 matplotlib.backends

#### matplotlib.backends.backend_mixed

**class** matplotlib.backends.backend_mixed.**MixedModeRenderer**(*figure*, *width*, *height*, *dpi*, *vector_renderer*, *raster_renderer_class=None*, *bbox_inches_restore=None*)

Bases: object

A helper class to implement a renderer that switches between vector and raster drawing. An example may be a PDF writer, where most things are drawn with PDF vector commands, but some very complex objects, such as quad meshes, are rasterised and then output as images.

**Parameters**

**figure**

[*Figure*] The figure instance.

**width**

[scalar] The width of the canvas in logical units

**height**

[scalar] The height of the canvas in logical units

**dpi**

[float] The dpi of the canvas

**vector_renderer**

[*RendererBase*] An instance of a subclass of *RendererBase* that will be used for the vector drawing.

**raster_renderer_class**

[*RendererBase*] The renderer class to use for the raster drawing. If not provided, this will use the Agg backend (which is currently the only viable option anyway.)

**start_rasterizing**()

Enter "raster" mode. All subsequent drawing commands (until *stop_rasterizing* is called) will be drawn with the raster backend.

**stop_rasterizing**()

Exit "raster" mode. All of the drawing that was done since the last *start_rasterizing* call will be copied to the vector backend by calling draw_image.

### matplotlib.backends.backend_template

A fully functional, do-nothing backend intended as a template for backend writers. It is fully functional in that you can select it as a backend e.g. with

```
import matplotlib
matplotlib.use("template")
```

and your program will (should!) run without error, though no output is produced. This provides a starting point for backend writers; you can selectively implement drawing methods (*draw_path*, *draw_image*, etc.) and slowly see your figure come to life instead having to have a full-blown implementation before getting any results.

Copy this file to a directory outside the Matplotlib source tree, somewhere where Python can import it (by adding the directory to your `sys.path` or by packaging it as a normal Python package); if the backend is importable as `import my.backend` you can then select it using

```
import matplotlib
matplotlib.use("module://my.backend")
```

If your backend implements support for saving figures (i.e. has a `print_xyz` method), you can register it as the default handler for a given file type:

```
from matplotlib.backend_bases import register_backend
register_backend('xyz', 'my_backend', 'XYZ File Format')
...
plt.savefig("figure.xyz")
```

matplotlib.backends.backend_template.**FigureCanvas**

alias of *FigureCanvasTemplate*

**class** matplotlib.backends.backend_template.**FigureCanvasTemplate**(*figure=None*)

Bases: *FigureCanvasBase*

The canvas the figure renders into. Calls the draw and print fig methods, creates the renderers, etc.

Note: GUI templates will want to connect events for button presses, mouse movements and key presses to functions that call the base class methods button_press_event, button_release_event, motion_notify_event, key_press_event, and key_release_event. See the implementations of the interactive backends for examples.

**Attributes**

**figure**

[*Figure*] A high-level Figure instance

**draw**()

Draw the figure using the renderer.

It is important that this method actually walk the artist tree even if not output is produced because this will trigger deferred work (like computing limits auto-limits and tick values) that users may want access to before saving to disk.

**filetypes = {'eps': 'Encapsulated Postscript', 'foo': 'My magic Foo format', 'jpeg': 'Joint Photographic Experts Group', 'jpg': 'Joint Photographic Experts Group', 'pdf': 'Portable Document Format', 'pgf': 'PGF code for LaTeX', 'png': 'Portable Network Graphics', 'ps': 'Postscript', 'raw': 'Raw RGBA bitmap', 'rgba': 'Raw RGBA bitmap', 'svg': 'Scalable Vector Graphics', 'svgz': 'Scalable Vector Graphics', 'tif': 'Tagged Image File Format', 'tiff': 'Tagged Image File Format', 'webp': 'WebP Image Format'}**

**get_default_filetype**()

Return the default savefig file format as specified in *rcParams["savefig.format"]* (default: `'png'`).

The returned string does not include a period. This method is overridden in backends that only support a single file type.

**manager_class**

alias of *FigureManagerTemplate*

**print_foo**(*filename*, ***kwargs*)

Write out format foo.

This method is normally called via *Figure.savefig* and *FigureCanvasBase.print_figure*, which take care of setting the figure facecolor, edgecolor, and dpi to the desired output values, and will restore them to the original values. Therefore, *print_foo* does not need to handle these settings.

matplotlib.backends.backend_template.**FigureManager**

alias of *FigureManagerTemplate*

**class** matplotlib.backends.backend_template.**FigureManagerTemplate**(*canvas*, *num*)

Bases: *FigureManagerBase*

Helper class for pyplot mode, wraps everything up into a neat bundle.

For non-interactive backends, the base class is sufficient. For interactive backends, see the documentation of the *FigureManagerBase* class for the list of methods that can/should be overridden.

**class** matplotlib.backends.backend_template.**GraphicsContextTemplate**

Bases: *GraphicsContextBase*

The graphics context provides the color, line styles, etc. See the cairo and postscript backends for examples of mapping the graphics context attributes (cap styles, join styles, line widths, colors) to a particular backend. In cairo this is done by wrapping a cairo.Context object and forwarding the appropriate calls to it using a dictionary mapping styles to gdk constants. In Postscript, all the work is done by the renderer, mapping line styles to postscript calls.

If it's more appropriate to do the mapping at the renderer level (as in the postscript backend), you don't need to override any of the GC methods. If it's more appropriate to wrap an instance (as in the cairo backend) and do the mapping here, you'll need to override several of the setter methods.

The base GraphicsContext stores colors as an RGB tuple on the unit interval, e.g., (0.5, 0.0, 1.0). You may need to map this to colors appropriate for your backend.

**class** matplotlib.backends.backend_template.**RendererTemplate**(*dpi*)

Bases: *RendererBase*

The renderer handles drawing/rendering operations.

This is a minimal do-nothing class that can be used to get started when writing a new backend. Refer to *backend_bases.RendererBase* for documentation of the methods.

**draw_image**(*gc*, *x*, *y*, *im*)

Draw an RGBA image.

**Parameters**

**gc**

[*GraphicsContextBase*] A graphics context with clipping information.

**x**

[scalar] The distance in physical units (i.e., dots or pixels) from the left hand side of the canvas.

**y**

[scalar] The distance in physical units (i.e., dots or pixels) from the bottom side of the canvas.

**im**

[(N, M, 4) array of numpy.uint8] An array of RGBA pixels.

**transform**

[*Affine2DBase*] If and only if the concrete backend is written such that option_scale_image returns True, an affine transformation (i.e., an *Affine2DBase*) *may* be passed to *draw_image*. The translation vector of the transformation is given in physical units (i.e., dots or pixels). Note that the transformation does not override *x* and *y*, and has to be applied *before* translating the result by *x* and *y* (this can be accomplished by adding *x* and *y* to the translation vector defined by *transform*).

**draw_path**(*gc*, *path*, *transform*, *rgbFace=None*)

Draw a *Path* instance using the given affine transform.

**draw_text**(*gc*, *x*, *y*, *s*, *prop*, *angle*, *ismath=False*, *mtext=None*)

Draw a text instance.

**Parameters**

**gc**

[*GraphicsContextBase*] The graphics context.

**x**

[float] The x location of the text in display coords.

**y**

[float] The y location of the text baseline in display coords.

**s**

[str] The text string.

**prop**

[*FontProperties*] The font properties.

**angle**

[float] The rotation angle in degrees anti-clockwise.

**ismath**

[bool or "TeX"] If True, use mathtext parser. If "TeX", use tex for rendering.

**mtext**

[*Text*] The original text object to be rendered.

### Notes

**Note for backend implementers:**

When you are trying to determine if you have gotten your bounding box right (which is what enables the text layout/alignment to work properly), it helps to change the line in text.py:

```
if 0: bbox_artist(self, renderer)
```

to if 1, and then the actual bounding box will be plotted along with your text.

**flipy**()

Return whether y values increase from top to bottom.

Note that this only affects drawing of texts.

**get_canvas_width_height**()

Return the canvas width and height in display coords.

**get_text_width_height_descent**(*s*, *prop*, *ismath*)

Get the width, height, and descent (offset from the bottom to the baseline), in display coords, of the string *s* with *FontProperties* *prop*.

Whitespace at the start and the end of *s* is included in the reported width.

**new_gc**()

Return an instance of a *GraphicsContextBase*.

**points_to_pixels**(*points*)

Convert points to display units.

You need to override this function (unless your backend doesn't have a dpi, e.g., postscript or svg). Some imaging systems assume some value for pixels per inch:

```
points to pixels = points * pixels_per_inch/72 * dpi/72
```

**Parameters**

**points**

[float or array-like]

**Returns**

**Points converted to pixels**

## matplotlib.backends.backend_agg

An Anti-Grain Geometry (AGG) backend.

Features that are implemented:

- capstyles and join styles
- dashes
- linewidth
- lines, rectangles, ellipses
- clipping to a rectangle
- output to RGBA and Pillow-supported image formats
- alpha blending
- DPI scaling properly - everything scales properly (dashes, linewidths, etc)
- draw polygon

- freetype2 w/ ft2font

Still TODO:

- integrate screen dpi w/ ppi and text

matplotlib.backends.backend_agg.**FigureCanvas**
alias of *FigureCanvasAgg*

**class** matplotlib.backends.backend_agg.**FigureCanvasAgg**(*figure=None*)
Bases: *FigureCanvasBase*

**buffer_rgba**()
Get the image as a memoryview to the renderer's buffer.

*draw* must be called at least once before this function will work and to update the renderer for any subsequent changes to the Figure.

**copy_from_bbox**(*bbox*)

**draw**()
Render the *Figure*.

This method must walk the artist tree, even if no output is produced, because it triggers deferred work that users may want to access before saving output to disk. For example computing limits, auto-limits, and tick values.

**get_renderer**()

**print_jpeg**(*filename_or_obj*, *, *metadata=None*, *pil_kwargs=None*)
Write the figure to a JPEG file.

**Parameters**

**filename_or_obj**

[str or path-like or file-like] The file to write to.

**pil_kwargs**

[dict, optional] Additional keyword arguments that are passed to PIL.Image.Image.save when saving the figure.

**print_jpg**(*filename_or_obj*, *, *metadata=None*, *pil_kwargs=None*)
Write the figure to a JPEG file.

**Parameters**

**filename_or_obj**

[str or path-like or file-like] The file to write to.

**pil_kwargs**

[dict, optional] Additional keyword arguments that are passed to PIL.Image.Image.save when saving the figure.

**print_png**(*filename_or_obj*, *, *metadata=None*, *pil_kwargs=None*)

Write the figure to a PNG file.

**Parameters**

**filename_or_obj**

[str or path-like or file-like] The file to write to.

**metadata**

[dict, optional] Metadata in the PNG file as key-value pairs of bytes or latin-1 encodable strings. According to the PNG specification, keys must be shorter than 79 chars.

The PNG specification defines some common keywords that may be used as appropriate:

- Title: Short (one line) title or caption for image.
- Author: Name of image's creator.
- Description: Description of image (possibly long).
- Copyright: Copyright notice.
- Creation Time: Time of original image creation (usually RFC 1123 format).
- Software: Software used to create the image.
- Disclaimer: Legal disclaimer.
- Warning: Warning of nature of content.
- Source: Device used to create the image.
- Comment: Miscellaneous comment; conversion from other image format.

Other keywords may be invented for other purposes.

If 'Software' is not given, an autogenerated value for Matplotlib will be used. This can be removed by setting it to *None*.

For more details see the PNG specification.

**pil_kwargs**

[dict, optional] Keyword arguments passed to `PIL.Image.Image.save`.

If the 'pnginfo' key is present, it completely overrides *metadata*, including the default 'Software' key.

**print_raw**(*filename_or_obj*, *, *metadata=None*)

**print_rgba**(*filename_or_obj*, *, *metadata=None*)

**print_tif**(*filename_or_obj*, *, *metadata=None*, *pil_kwargs=None*)

Write the figure to a TIFF file.

**Parameters**

**filename_or_obj**

[str or path-like or file-like] The file to write to.

**pil_kwargs**

[dict, optional] Additional keyword arguments that are passed to `PIL.Image.Image.save` when saving the figure.

**print_tiff**(*filename_or_obj*, *, *metadata=None*, *pil_kwargs=None*)

Write the figure to a TIFF file.

**Parameters**

**filename_or_obj**

[str or path-like or file-like] The file to write to.

**pil_kwargs**

[dict, optional] Additional keyword arguments that are passed to `PIL.Image.Image.save` when saving the figure.

**print_to_buffer**()

**print_webp**(*filename_or_obj*, *, *metadata=None*, *pil_kwargs=None*)

Write the figure to a WebP file.

**Parameters**

**filename_or_obj**

[str or path-like or file-like] The file to write to.

**pil_kwargs**

[dict, optional] Additional keyword arguments that are passed to `PIL.Image.Image.save` when saving the figure.

**restore_region**(*region*, *bbox=None*, *xy=None*)

**tostring_argb**()

Get the image as ARGB `bytes`.

*draw* must be called at least once before this function will work and to update the renderer for any subsequent changes to the Figure.

**tostring_rgb**()

[*Deprecated*] Get the image as RGB `bytes`.

*draw* must be called at least once before this function will work and to update the renderer for any subsequent changes to the Figure.

### Notes

Deprecated since version 3.8: Use buffer_rgba instead.

**class** matplotlib.backends.backend_agg.**RendererAgg**(*width*, *height*, *dpi*)

Bases: *RendererBase*

The renderer handles all the drawing primitives using a graphics context instance that controls the colors/styles

**buffer_rgba**()

**clear**()

**draw_mathtext**(*gc*, *x*, *y*, *s*, *prop*, *angle*)

Draw mathtext using *matplotlib.mathtext*.

**draw_path**(*gc*, *path*, *transform*, *rgbFace=None*)

Draw a *Path* instance using the given affine transform.

**draw_tex**(*gc*, *x*, *y*, *s*, *prop*, *angle*, ***, *mtext=None*)

Draw a TeX instance.

**Parameters**

**gc**

[*GraphicsContextBase*] The graphics context.

**x**

[float] The x location of the text in display coords.

**y**

[float] The y location of the text baseline in display coords.

**s**

[str] The TeX text string.

**prop**

[*FontProperties*] The font properties.

**angle**

[float] The rotation angle in degrees anti-clockwise.

**mtext**

[*Text*] The original text object to be rendered.

**draw_text**(*gc*, *x*, *y*, *s*, *prop*, *angle*, *ismath=False*, *mtext=None*)

Draw a text instance.

**Parameters**

**gc**

[*GraphicsContextBase*] The graphics context.

**x**

[float] The x location of the text in display coords.

**y**

[float] The y location of the text baseline in display coords.

**s**

[str] The text string.

**prop**

[*FontProperties*] The font properties.

**angle**

[float] The rotation angle in degrees anti-clockwise.

**ismath**

[bool or "TeX"] If True, use mathtext parser. If "TeX", use tex for rendering.

**mtext**

[*Text*] The original text object to be rendered.

### Notes

**Note for backend implementers:**

When you are trying to determine if you have gotten your bounding box right (which is what enables the text layout/alignment to work properly), it helps to change the line in text.py:

```
if 0: bbox_artist(self, renderer)
```

to if 1, and then the actual bounding box will be plotted along with your text.

**get_canvas_width_height**()

Return the canvas width and height in display coords.

**get_text_width_height_descent**(*s*, *prop*, *ismath*)

Get the width, height, and descent (offset from the bottom to the baseline), in display coords, of the string *s* with *FontProperties* *prop*.

Whitespace at the start and the end of *s* is included in the reported width.

**option_image_nocomposite**()

Return whether image composition by Matplotlib should be skipped.

Raster backends should usually return False (letting the C-level rasterizer take care of image composition); vector backends should usually return `not rcParams["image.composite_image"]`.

**option_scale_image**()

Return whether arbitrary affine transformations in `draw_image` are supported (True for most vector backends).

**points_to_pixels**(*points*)

Convert points to display units.

You need to override this function (unless your backend doesn't have a dpi, e.g., postscript or svg). Some imaging systems assume some value for pixels per inch:

```
points to pixels = points * pixels_per_inch/72 * dpi/72
```

**Parameters**

**points**

[float or array-like]

**Returns**

**Points converted to pixels**

**restore_region**(*region*, *bbox=None*, *xy=None*)

Restore the saved region. If bbox (instance of BboxBase, or its extents) is given, only the region specified by the bbox will be restored. *xy* (a pair of floats) optionally specifies the new position (the LLC of the original region, not the LLC of the bbox) where the region will be restored.

```
>>> region = renderer.copy_from_bbox()
>>> x1, y1, x2, y2 = region.get_extents()
>>> renderer.restore_region(region, bbox=(x1+dx, y1, x2, y2),
...                         xy=(x1-dx, y1))
```

**start_filter**()

Start filtering. It simply creates a new canvas (the old one is saved).

**stop_filter**(*post_processing*)

Save the current canvas as an image and apply post processing.

The *post_processing* function:

```
def post_processing(image, dpi):
  # ny, nx, depth = image.shape
  # image (numpy array) has RGBA channels and has a depth of 4.
  ...
  # create a new_image (numpy array of 4 channels, size can be
  # different). The resulting image may have offsets from
  # lower-left corner of the original image
  return new_image, offset_x, offset_y
```

The saved renderer is restored and the returned image from post_processing is plotted (using draw_image) on it.

**tostring_argb**()

**tostring_rgb**()

[*Deprecated*]

### Notes

Deprecated since version 3.8: Use buffer_rgba instead.

matplotlib.backends.backend_agg.**get_hinting_flag**()

## matplotlib.backends.backend_cairo

### A Cairo backend for Matplotlib

**Author**

Steve Chaplin and others

This backend depends on cairocffi or pycairo.

matplotlib.backends.backend_cairo.**FigureCanvas**

alias of *FigureCanvasCairo*

**class** matplotlib.backends.backend_cairo.**FigureCanvasCairo**(*figure=None*)

Bases: *FigureCanvasBase*

**copy_from_bbox**(*bbox*)

**get_renderer**()

**print_pdf**(*fobj*, *, *orientation='portrait'*)

**print_png**(*fobj*)

**print_ps**(*fobj*, *, *orientation='portrait'*)

**print_raw**(*fobj*)

**print_rgba**(*fobj*)

**print_svg**(*fobj*, *, *orientation='portrait'*)

**print_svgz**(*fobj*, *, *orientation='portrait'*)

**restore_region**(*region*)

**class** matplotlib.backends.backend_cairo.**GraphicsContextCairo**(*renderer*)

Bases: *GraphicsContextBase*

**get_antialiased**()

Return whether the object should try to do antialiased rendering.

**get_rgb**()

Return a tuple of three or four floats from 0-1.

**restore**()

Restore the graphics context from the stack - needed only for backends that save graphics contexts on a stack.

**set_alpha**(*alpha*)

Set the alpha value used for blending - not supported on all backends.

If `alpha=None` (the default), the alpha components of the foreground and fill colors will be used to set their respective transparencies (where applicable); otherwise, `alpha` will override them.

**set_antialiased**(*b*)

Set whether object should be drawn with antialiased rendering.

**set_capstyle**(*cs*)

Set how to draw endpoints of lines.

**Parameters**

**cs**

[*CapStyle* or {'butt', 'projecting', 'round'}]

**set_clip_path**(*path*)

Set the clip path to a *TransformedPath* or None.

**set_clip_rectangle**(*rectangle*)

Set the clip rectangle to a *Bbox* or None.

**set_dashes**(*offset*, *dashes*)

Set the dash style for the gc.

**Parameters**

**dash_offset**

[float] Distance, in points, into the dash pattern at which to start the pattern. It is usually set to 0.

**dash_list**

[array-like or None] The on-off sequence as points. None specifies a solid line. All values must otherwise be non-negative (≥ 0).

### Notes

See p. 666 of the PostScript Language Reference for more info.

**set_foreground**(*fg*, *isRGBA=None*)

Set the foreground color.

**Parameters**

**fg**

[*color*]

**isRGBA**

[bool] If *fg* is known to be an `(r, g, b, a)` tuple, *isRGBA* can be set to True to improve performance.

**set_joinstyle**(*js*)

Set how to draw connections between line segments.

**Parameters**

**js**

[*JoinStyle* or {'miter', 'round', 'bevel'}]

**set_linewidth**(*w*)

Set the linewidth in points.

**class** matplotlib.backends.backend_cairo.**RendererCairo**(*dpi*)

Bases: *RendererBase*

**draw_image**(*gc*, *x*, *y*, *im*)

Draw an RGBA image.

**Parameters**

**gc**

[*GraphicsContextBase*] A graphics context with clipping information.

**x**

[scalar] The distance in physical units (i.e., dots or pixels) from the left hand side of the canvas.

**y**

[scalar] The distance in physical units (i.e., dots or pixels) from the bottom side of the canvas.

**im**

[(N, M, 4) array of `numpy.uint8`] An array of RGBA pixels.

**transform**

[*Affine2DBase*] If and only if the concrete backend is written such that `option_scale_image` returns `True`, an affine transformation (i.e., an *Affine2DBase*) *may* be passed to *draw_image*. The translation vector of the transformation is given in physical units (i.e., dots or pixels). Note that the transformation does not override *x* and *y*, and has to be applied *before* translating the result by *x* and *y* (this can be accomplished by adding *x* and *y* to the translation vector defined by *transform*).

**draw_markers**(*gc*, *marker_path*, *marker_trans*, *path*, *transform*, *rgbFace=None*)

Draw a marker at each of *path*'s vertices (excluding control points).

The base (fallback) implementation makes multiple calls to *draw_path*. Backends may want to override this method in order to draw the marker only once and reuse it multiple times.

**Parameters**

**gc**

[*GraphicsContextBase*] The graphics context.

**marker_path**

[*Path*] The path for the marker.

**marker_trans**

[*Transform*] An affine transform applied to the marker.

**path**

[*Path*] The locations to draw the markers.

**trans**

[*Transform*] An affine transform applied to the path.

**rgbFace**

[*color*, optional]

**draw_path**(*gc*, *path*, *transform*, *rgbFace=None*)

Draw a *Path* instance using the given affine transform.

**draw_text**(*gc*, *x*, *y*, *s*, *prop*, *angle*, *ismath=False*, *mtext=None*)

Draw a text instance.

**Parameters**

**gc**

[*GraphicsContextBase*] The graphics context.

**x**

[float] The x location of the text in display coords.

**y**

[float] The y location of the text baseline in display coords.

**s**

[str] The text string.

**prop**

[*FontProperties*] The font properties.

**angle**

[float] The rotation angle in degrees anti-clockwise.

**ismath**

[bool or "TeX"] If True, use mathtext parser. If "TeX", use tex for rendering.

**mtext**

[*Text*] The original text object to be rendered.

## Notes

**Note for backend implementers:**

When you are trying to determine if you have gotten your bounding box right (which is what enables the text layout/alignment to work properly), it helps to change the line in text.py:

```
if 0: bbox_artist(self, renderer)
```

to if 1, and then the actual bounding box will be plotted along with your text.

**get_canvas_width_height**()

Return the canvas width and height in display coords.

**get_text_width_height_descent**(*s*, *prop*, *ismath*)

Get the width, height, and descent (offset from the bottom to the baseline), in display coords, of the string *s* with *FontProperties* *prop*.

Whitespace at the start and the end of *s* is included in the reported width.

**new_gc**()

Return an instance of a *GraphicsContextBase*.

**points_to_pixels**(*points*)

Convert points to display units.

You need to override this function (unless your backend doesn't have a dpi, e.g., postscript or svg). Some imaging systems assume some value for pixels per inch:

```
points to pixels = points * pixels_per_inch/72 * dpi/72
```

**Parameters**

**points**

[float or array-like]

**Returns**

**Points converted to pixels**

**set_context**(*ctx*)

**matplotlib.backends.backend_gtk3agg, matplotlib.backends.backend_gtk3cairo**

**NOTE** These *Backends* are not documented here, to avoid adding a dependency to building the docs.

**matplotlib.backends.backend_gtk4agg, matplotlib.backends.backend_gtk4cairo**

**NOTE** These *Backends* are not documented here, to avoid adding a dependency to building the docs.

### matplotlib.backends.backend_nbagg

Interactive figures in the IPython notebook.

**class** matplotlib.backends.backend_nbagg.**CommSocket**(*manager*)

Bases: object

Manages the Comm connection between IPython and the browser (client).

Comms are 2 way, with the CommSocket being able to publish a message via the send_json method, and handle a message with on_message. On the JS side figure.send_message and figure.ws.onmessage do the sending and receiving respectively.

**is_open**()

**on_close**()

**on_message**(*message*)

**send_binary**(*blob*)

**send_json**(*content*)

matplotlib.backends.backend_nbagg.**FigureCanvas**

alias of *FigureCanvasNbAgg*

**class** matplotlib.backends.backend_nbagg.**FigureCanvasNbAgg**(**args*, ***kwargs*)

Bases: *FigureCanvasWebAggCore*

**manager_class**

alias of *FigureManagerNbAgg*

matplotlib.backends.backend_nbagg.**FigureManager**

alias of *FigureManagerNbAgg*

**class** matplotlib.backends.backend_nbagg.**FigureManagerNbAgg**(*canvas*, *num*)

Bases: *FigureManagerWebAgg*

**ToolbarCls**

alias of *NavigationIPy*

**clearup_closed**()

Clear up any closed Comms.

**property connected**

**classmethod create_with_canvas**(*canvas_class*, *figure*, *num*)

Create a manager for a given *figure* using a specific *canvas_class*.

Backends should override this method if they have specific needs for setting up the canvas or the manager.

**destroy**()

**display_js**()

**classmethod get_javascript**(*stream=None*)

**remove_comm**(*comm_id*)

**reshow**()

A special method to re-show the figure in the notebook.

**show**()

For GUI backends, show the figure window and redraw. For non-GUI backends, raise an exception, unless running headless (i.e. on Linux with an unset DISPLAY); this exception is converted to a warning in *Figure.show*.

**class** matplotlib.backends.backend_nbagg.**NavigationIPy**(*canvas*)

Bases: *NavigationToolbar2WebAgg*

**toolitems = [('Home', 'Reset original view', 'fa fa-home', 'home'), ('Back', 'Back to previous view', 'fa fa-arrow-left', 'back'), ('Forward', 'Forward to next view', 'fa fa-arrow-right', 'forward'), (None, None, None, None), ('Pan', 'Left button pans, Right button zooms\nx/y fixes axis, CTRL fixes aspect', 'fa fa-arrows', 'pan'), ('Zoom', 'Zoom to rectangle\nx/y fixes axis', 'fa fa-square-o', 'zoom'), (None, None, None, None), ('Download', 'Download plot', 'fa fa-floppy-o', 'download')]**

matplotlib.backends.backend_nbagg.**connection_info**()

Return a string showing the figure and connection status for the backend.

This is intended as a diagnostic tool, and not for general use.

### matplotlib.backends.backend_pdf

A PDF Matplotlib backend.

Author: Jouni K Seppänen <jks@iki.fi> and others.

matplotlib.backends.backend_pdf.**FigureCanvas**

alias of *FigureCanvasPdf*

**class** matplotlib.backends.backend_pdf.**FigureCanvasPdf**(*figure=None*)

Bases: *FigureCanvasBase*

**draw**()

Render the *Figure*.

This method must walk the artist tree, even if no output is produced, because it triggers deferred work that users may want to access before saving output to disk. For example computing limits, auto-limits, and tick values.

**filetypes = {'pdf': 'Portable Document Format'}**

**fixed_dpi = 72**

**get_default_filetype**()

Return the default savefig file format as specified in *rcParams["savefig.format"]* (default: 'png').

The returned string does not include a period. This method is overridden in backends that only support a single file type.

**print_pdf**(*filename*, *, *bbox_inches_restore=None*, *metadata=None*)

**class** matplotlib.backends.backend_pdf.**GraphicsContextPdf**(*file*)

Bases: *GraphicsContextBase*

**alpha_cmd**(*alpha*, *forced*, *effective_alphas*)

**capstyle_cmd**(*style*)

**capstyles = {'butt': 0, 'projecting': 2, 'round': 1}**

**clip_cmd**(*cliprect*, *clippath*)

Set clip rectangle. Calls *pop()* and *push()*.

**commands = ((('_cliprect', '_clippath'), <function GraphicsContextPdf.clip_cmd>), (('_alpha', '_forced_alpha', '_effective_alphas'), <function GraphicsContextPdf.alpha_cmd>), (('_capstyle',), <function GraphicsContextPdf.capstyle_cmd>), (('_fillcolor',), <function GraphicsContextPdf.fillcolor_cmd>), (('_joinstyle',), <function GraphicsContextPdf.joinstyle_cmd>), (('_linewidth',), <function GraphicsContextPdf.linewidth_cmd>), (('_dashes',), <function GraphicsContextPdf.dash_cmd>), (('_rgb',), <function GraphicsContextPdf.rgb_cmd>), (('_hatch', '_hatch_color'), <function GraphicsContextPdf.hatch_cmd>))**

**copy_properties**(*other*)

Copy properties of other into self.

**dash_cmd**(*dashes*)

**delta**(*other*)

Copy properties of other into self and return PDF commands needed to transform *self* into *other*.

**fill**(**args*)

Predicate: does the path need to be filled?

An optional argument can be used to specify an alternative _fillcolor, as needed by RendererPdf.draw_markers.

**fillcolor_cmd**(*rgb*)

**finalize**()

Make sure every pushed graphics state is popped.

**hatch_cmd**(*hatch*, *hatch_color*)

**joinstyle_cmd**(*style*)

**joinstyles = {'bevel': 2, 'miter': 0, 'round': 1}**

**linewidth_cmd**(*width*)

**paint**()

Return the appropriate pdf operator to cause the path to be stroked, filled, or both.

**pop**()

**push**()

**rgb_cmd**(*rgb*)

**stroke**()

Predicate: does the path need to be stroked (its outline drawn)? This tests for the various conditions that disable stroking the path, in which case it would presumably be filled.

**class** matplotlib.backends.backend_pdf.**Name**(*name*)

Bases: object

PDF name object.

**name**

**pdfRepr**()

**class** matplotlib.backends.backend_pdf.**Op**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

PDF operators (not an exhaustive list).

**begin_text = b'BT'**

**clip = b'W'**

**close_fill_stroke = b'b'**

**close_stroke = b's'**

**closepath = b'h'**

**concat_matrix = b'cm'**

**curveto = b'c'**

**end_text = b'ET'**

**endpath = b'n'**

**fill = b'f'**

**fill_stroke = b'B'**

**grestore = b'Q'**

**gsave = b'q'**

**lineto = b'l'**

**moveto = b'm'**

**classmethod paint_path**(*fill*, *stroke*)

Return the PDF operator to paint a path.

**Parameters**

**fill**

[bool] Fill the path with the fill color.

**stroke**

[bool] Stroke the outline of the path with the line color.

**pdfRepr**()

**rectangle = b're'**

**selectfont = b'Tf'**

**setcolor_nonstroke = b'scn'**

**setcolor_stroke = b'SCN'**

**setcolorspace_nonstroke = b'cs'**

**setcolorspace_stroke = b'CS'**

**setdash = b'd'**

**setgray_nonstroke = b'g'**

**setgray_stroke = b'G'**

**setgstate = b'gs'**

**setlinecap = b'J'**

**setlinejoin = b'j'**

**setlinewidth = b'w'**

**setrgb_nonstroke = b'rg'**

**setrgb_stroke = b'RG'**

**shading = b'sh'**

**show = b'Tj'**

**showkern = b'TJ'**

**stroke = b'S'**

**textmatrix = b'Tm'**

**textpos = b'Td'**

**use_xobject = b'Do'**

**class** matplotlib.backends.backend_pdf.**PdfFile**(*filename*, *metadata=None*)

Bases: object

PDF file object.

**Parameters**

**filename**

[str or path-like or file-like] Output target; if a string, a file will be opened for writing.

**metadata**

[dict from strings to strings and dates] Information dictionary object (see PDF reference section 10.2.1 'Document Information Dictionary'), e.g.: `{'Creator': 'My software', 'Author': 'Me', 'Title': 'Awesome'}`.

The standard keys are 'Title', 'Author', 'Subject', 'Keywords', 'Creator', 'Producer', 'CreationDate', 'ModDate', and 'Trapped'. Values have been predefined for 'Creator', 'Producer' and 'CreationDate'. They can be removed by setting them to `None`.

**addGouraudTriangles**(*points*, *colors*)

Add a Gouraud triangle shading.

**Parameters**

**points**

[np.ndarray] Triangle vertices, shape (n, 3, 2) where n = number of triangles, 3 = vertices, 2 = x, y.

**colors**

[np.ndarray] Vertex colors, shape (n, 3, 1) or (n, 3, 4) as with points, but last dimension is either (gray,) or (r, g, b, alpha).

**Returns**

**Name, Reference**

**alphaState**(*alpha*)

Return name of an ExtGState that sets alpha to the given value.

**beginStream**(*id*, *len*, *extra=None*, *png=None*)

**close**()

Flush all buffers and free all resources.

**createType1Descriptor**(*t1font*, *fontfile*)

**dviFontName**(*dvifont*)

Given a dvi font object, return a name suitable for Op.selectfont. This registers the font information in `self.dviFontInfo` if not yet registered.

**embedTTF**(*filename*, *characters*)

Embed the TTF font from the named file into the document.

**endStream**()

**finalize**()

Write out the various deferred objects and the pdf end matter.

**fontName**(*fontprop*)

Select a font based on fontprop and return a name suitable for Op.selectfont. If fontprop is a string, it will be interpreted as the filename of the font.

**hatchPattern**(*hatch_style*)

**imageObject**(*image*)

Return name of an image XObject representing the given image.

**markerObject**(*path*, *trans*, *fill*, *stroke*, *lw*, *joinstyle*, *capstyle*)

Return name of a marker XObject representing the given path.

**newPage**(*width*, *height*)

**newTextnote**(*text*, *positionRect=[-100, -100, 0, 0]*)

**output**(**data*)

**outputStream**(*ref*, *data*, ***, *extra=None*)

**pathCollectionObject**(*gc*, *path*, *trans*, *padding*, *filled*, *stroked*)

**static pathOperations**(*path*, *transform*, *clip=None*, *simplify=None*, *sketch=None*)

**recordXref**(*id*)

**reserveObject**(*name=''*)

Reserve an ID for an indirect object.

The name is used for debugging in case we forget to print out the object with writeObject.

**write**(*data*)

**writeExtGSTates**()

**writeFonts**()

**writeGouraudTriangles**()

**writeHatches**()

**writeImages**()

**writeInfoDict**()

Write out the info dictionary, checking it for good form

**writeMarkers**()

**writeObject**(*object*, *contents*)

**writePath**(*path*, *transform*, *clip=False*, *sketch=None*)

**writePathCollectionTemplates**()

**writeTrailer**()

Write out the PDF trailer.

**writeXref**()

Write out the xref table.

**class** matplotlib.backends.backend_pdf.**PdfPages**(*filename*, *keep_empty=<object object>*, *metadata=None*)

Bases: object

A multi-page PDF file.

### Notes

In reality *PdfPages* is a thin wrapper around *PdfFile*, in order to avoid confusion when using *savefig* and forgetting the format argument.

### Examples

```
>>> import matplotlib.pyplot as plt
>>> # Initialize:
>>> with PdfPages('foo.pdf') as pdf:
...     # As many times as you like, create a figure fig and save it:
...     fig = plt.figure()
...     pdf.savefig(fig)
...     # When no figure is specified the current figure is saved
...     pdf.savefig()
```

Create a new PdfPages object.

**Parameters**

**filename**

[str or path-like or file-like] Plots using *PdfPages.savefig* will be written to a file at this location. The file is opened when a figure is saved for the first time (overwriting any older file with the same name).

**keep_empty**

[bool, optional] If set to False, then empty pdf files will be deleted automatically when closed.

**metadata**

[dict, optional] Information dictionary object (see PDF reference section 10.2.1 'Document Information Dictionary'), e.g.: `{'Creator': 'My software', 'Author': 'Me', 'Title': 'Awesome'}`.

The standard keys are 'Title', 'Author', 'Subject', 'Keywords', 'Creator', 'Producer', 'CreationDate', 'ModDate', and 'Trapped'. Values have been predefined for 'Creator', 'Producer' and 'CreationDate'. They can be removed by setting them to `None`.

**attach_note**(*text*, *positionRect=[-100, -100, 0, 0]*)

Add a new text note to the page to be saved next. The optional positionRect specifies the position of the new note on the page. It is outside the page per default to make sure it is invisible on printouts.

**close**()

Finalize this object, making the underlying file a complete PDF file.

**get_pagecount**()

Return the current number of pages in the multipage pdf file.

**infodict**()

Return a modifiable information dictionary object (see PDF reference section 10.2.1 'Document Information Dictionary').

**property keep_empty**

[*Deprecated*]

### Notes

Deprecated since version 3.8:

**savefig**(*figure=None*, ***kwargs*)

Save a *Figure* to this file as a new page.

Any other keyword arguments are passed to *savefig*.

**Parameters**

**figure**

[*Figure* or int, default: the active figure] The figure, or index of the figure, that is saved to the file.

**class** matplotlib.backends.backend_pdf.**Reference**(*id*)

Bases: object

PDF reference object.

Use PdfFile.reserveObject() to create References.

**pdfRepr**()

**write**(*contents*, *file*)

**class** matplotlib.backends.backend_pdf.**RendererPdf**(*file*, *image_dpi*, *height*, *width*)

Bases: RendererPDFPSBase

**check_gc**(*gc*, *fillcolor=None*)

**draw_gouraud_triangles**(*gc*, *points*, *colors*, *trans*)

Draw a series of Gouraud triangles.

**Parameters**

**gc**

[*GraphicsContextBase*] The graphics context.

**triangles_array**

[(N, 3, 2) array-like] Array of *N* (x, y) points for the triangles.

**colors_array**

[(N, 3, 4) array-like] Array of *N* RGBA colors for each point of the triangles.

**transform**

[*Transform*] An affine transform to apply to the points.

**draw_image**(*gc*, *x*, *y*, *im*, *transform=None*)

Draw an RGBA image.

**Parameters**

**gc**

[*GraphicsContextBase*] A graphics context with clipping information.

**x**

[scalar] The distance in physical units (i.e., dots or pixels) from the left hand side of the canvas.

**y**

[scalar] The distance in physical units (i.e., dots or pixels) from the bottom side of the canvas.

**im**

[(N, M, 4) array of `numpy.uint8`] An array of RGBA pixels.

**transform**

[*Affine2DBase*] If and only if the concrete backend is written such that `option_scale_image` returns `True`, an affine transformation (i.e., an *Affine2DBase*) *may* be passed to *draw_image*. The translation vector of the transformation is given in physical units (i.e., dots or pixels). Note that the transformation does not override *x* and *y*, and has to be applied *before* translating the result by *x* and *y* (this can be accomplished by adding *x* and *y* to the translation vector defined by *transform*).

**draw_markers**(*gc*, *marker_path*, *marker_trans*, *path*, *trans*, *rgbFace=None*)

Draw a marker at each of *path*'s vertices (excluding control points).

The base (fallback) implementation makes multiple calls to *draw_path*. Backends may want to override this method in order to draw the marker only once and reuse it multiple times.

**Parameters**

**gc**

[*GraphicsContextBase*] The graphics context.

**marker_path**

[*Path*] The path for the marker.

**marker_trans**

[*Transform*] An affine transform applied to the marker.

**path**

[*Path*] The locations to draw the markers.

**trans**

[*Transform*] An affine transform applied to the path.

**rgbFace**

[*color*, optional]

**draw_mathtext**(*gc*, *x*, *y*, *s*, *prop*, *angle*)

**draw_path**(*gc*, *path*, *transform*, *rgbFace=None*)

Draw a *Path* instance using the given affine transform.

**draw_path_collection**(*gc*, *master_transform*, *paths*, *all_transforms*, *offsets*, *offset_trans*, *facecolors*, *edgecolors*, *linewidths*, *linestyles*, *antialiaseds*, *urls*, *offset_position*)

Draw a collection of *paths*.

Each path is first transformed by the corresponding entry in *all_transforms* (a list of (3, 3) matrices) and then by *master_transform*. They are then translated by the corresponding entry in *offsets*, which has been first transformed by *offset_trans*.

*facecolors*, *edgecolors*, *linewidths*, *linestyles*, and *antialiased* are lists that set the corresponding properties.

*offset_position* is unused now, but the argument is kept for backwards compatibility.

The base (fallback) implementation makes multiple calls to *draw_path*. Backends may want to override this in order to render each set of path data only once, and then reference that path multiple times with the different offsets, colors, styles etc. The generator methods `_iter_collection_raw_paths` and `_iter_collection` are provided to help with (and standardize) the implementation across backends. It is highly recommended to use those generators, so that changes to the behavior of *draw_path_collection* can be made globally.

**draw_tex**(*gc*, *x*, *y*, *s*, *prop*, *angle*, *, *mtext=None*)

Draw a TeX instance.

**Parameters**

**gc**

[*GraphicsContextBase*] The graphics context.

**x**

[float] The x location of the text in display coords.

**y**

[float] The y location of the text baseline in display coords.

**s**

[str] The TeX text string.

**prop**

[*FontProperties*] The font properties.

**angle**

[float] The rotation angle in degrees anti-clockwise.

**mtext**

[*Text*] The original text object to be rendered.

**draw_text**(*gc*, *x*, *y*, *s*, *prop*, *angle*, *ismath=False*, *mtext=None*)

Draw a text instance.

**Parameters**

**gc**

[*GraphicsContextBase*] The graphics context.

**x**

[float] The x location of the text in display coords.

**y**

[float] The y location of the text baseline in display coords.

**s**

[str] The text string.

**prop**

[*FontProperties*] The font properties.

**angle**

[float] The rotation angle in degrees anti-clockwise.

**ismath**

[bool or "TeX"] If True, use mathtext parser. If "TeX", use tex for rendering.

**mtext**

[*Text*] The original text object to be rendered.

### Notes

**Note for backend implementers:**

When you are trying to determine if you have gotten your bounding box right (which is what enables the text layout/alignment to work properly), it helps to change the line in text.py:

```
if 0: bbox_artist(self, renderer)
```

to if 1, and then the actual bounding box will be plotted along with your text.

**encode_string**(*s*, *fonttype*)

**finalize**()

**get_image_magnification**()

Get the factor by which to magnify images passed to *draw_image*. Allows a backend to have images at a different resolution to other artists.

**new_gc**()

Return an instance of a *GraphicsContextBase*.

**class** matplotlib.backends.backend_pdf.**Stream**(*id*, *len*, *file*, *extra=None*, *png=None*)

Bases: object

PDF stream object.

This has no pdfRepr method. Instead, call begin(), then output the contents of the stream by calling write(), and finally call end().

**Parameters**

**id**

[int] Object id of the stream.

**len**

[Reference or None] An unused Reference object for the length of the stream; None means to use a memory buffer so the length can be inlined.

**file**

[PdfFile] The underlying object to write the stream to.

**extra**

[dict from Name to anything, or None] Extra key-value pairs to include in the stream header.

**png**

[dict or None] If the data is already png encoded, the decode parameters.

**compressobj**

**end**()
Finalize stream.

**extra**

**file**

**id**

**len**

**pdfFile**

**pos**

**write**(*data*)
Write some data on the stream.

**class** matplotlib.backends.backend_pdf.**Verbatim**(*x*)
Bases: object

Store verbatim PDF command content for later inclusion in the stream.

**pdfRepr**()

matplotlib.backends.backend_pdf.**pdfRepr**(*obj*)
Map Python objects to PDF syntax.

## matplotlib.backends.backend_pgf

matplotlib.backends.backend_pgf.**FigureCanvas**
alias of *FigureCanvasPgf*

**class** matplotlib.backends.backend_pgf.**FigureCanvasPgf**(*figure=None*)
Bases: *FigureCanvasBase*

**draw**()
Render the *Figure*.

This method must walk the artist tree, even if no output is produced, because it triggers deferred work that users may want to access before saving output to disk. For example computing limits, auto-limits, and tick values.

**filetypes = {'pdf': 'LaTeX compiled PGF picture', 'pgf': 'LaTeX PGF picture', 'png': 'Portable Network Graphics'}**

**get_default_filetype**()
Return the default savefig file format as specified in *rcParams["savefig.format"]* (default: 'png').

The returned string does not include a period. This method is overridden in backends that only support a single file type.

**get_renderer**()

**print_pdf**(*fname_or_fh*, *, *metadata=None*, ***kwargs*)

Use LaTeX to compile a pgf generated figure to pdf.

**print_pgf**(*fname_or_fh*, ***kwargs*)

Output pgf macros for drawing the figure so it can be included and rendered in latex documents.

**print_png**(*fname_or_fh*, ***kwargs*)

Use LaTeX to compile a pgf figure to pdf and convert it to png.

**exception** matplotlib.backends.backend_pgf.**LatexError**(*message*, *latex_output=''*)

Bases: Exception

**class** matplotlib.backends.backend_pgf.**LatexManager**

Bases: object

The LatexManager opens an instance of the LaTeX application for determining the metrics of text elements. The LaTeX environment can be modified by setting fonts and/or a custom preamble in *rcParams*.

**get_width_height_descent**(*text*, *prop*)

Get the width, total height, and descent (in TeX points) for a text typeset by the current LaTeX environment.

**class** matplotlib.backends.backend_pgf.**PdfPages**(*filename*, *, *keep_empty=<object object>*, *metadata=None*)

Bases: object

A multi-page PDF file using the pgf backend

### Examples

```
>>> import matplotlib.pyplot as plt
>>> # Initialize:
>>> with PdfPages('foo.pdf') as pdf:
...     # As many times as you like, create a figure fig and save it:
...     fig = plt.figure()
...     pdf.savefig(fig)
...     # When no figure is specified the current figure is saved
...     pdf.savefig()
```

Create a new PdfPages object.

**Parameters**

**filename**

[str or path-like] Plots using *PdfPages.savefig* will be written to a file at this location. Any older file with the same name is overwritten.

**keep_empty**

[bool, default: True] If set to False, then empty pdf files will be deleted automatically when closed.

**metadata**

[dict, optional] Information dictionary object (see PDF reference section 10.2.1 'Document Information Dictionary'), e.g.: `{'Creator': 'My software', 'Author': 'Me', 'Title': 'Awesome'}`.

The standard keys are 'Title', 'Author', 'Subject', 'Keywords', 'Creator', 'Producer', 'CreationDate', 'ModDate', and 'Trapped'. Values have been predefined for 'Creator', 'Producer' and 'CreationDate'. They can be removed by setting them to `None`.

Note that some versions of LaTeX engines may ignore the 'Producer' key and set it to themselves.

**close**()

Finalize this object, running LaTeX in a temporary directory and moving the final pdf file to *filename*.

**get_pagecount**()

Return the current number of pages in the multipage pdf file.

**property keep_empty**

[*Deprecated*]

### Notes

Deprecated since version 3.8:

**savefig**(*figure=None*, ***kwargs*)

Save a *Figure* to this file as a new page.

Any other keyword arguments are passed to *savefig*.

**Parameters**

**figure**

[*Figure* or int, default: the active figure] The figure, or index of the figure, that is saved to the file.

**class** matplotlib.backends.backend_pgf.**RendererPgf**(*figure*, *fh*)

Bases: *RendererBase*

Create a new PGF renderer that translates any drawing instruction into text commands to be interpreted in a latex pgfpicture environment.

**Attributes**

**figure**

[*Figure*] Matplotlib figure to initialize height, width and dpi from.

**fh**

[file-like] File handle for the output of the drawing commands.

**draw_image**(*gc*, *x*, *y*, *im*, *transform=None*)

Draw an RGBA image.

**Parameters**

**gc**

[*GraphicsContextBase*] A graphics context with clipping information.

**x**

[scalar] The distance in physical units (i.e., dots or pixels) from the left hand side of the canvas.

**y**

[scalar] The distance in physical units (i.e., dots or pixels) from the bottom side of the canvas.

**im**

[(N, M, 4) array of `numpy.uint8`] An array of RGBA pixels.

**transform**

[*Affine2DBase*] If and only if the concrete backend is written such that *option_scale_image* returns `True`, an affine transformation (i.e., an *Affine2DBase*) *may* be passed to *draw_image*. The translation vector of the transformation is given in physical units (i.e., dots or pixels). Note that the transformation does not override *x* and *y*, and has to be applied *before* translating the result by *x* and *y* (this can be accomplished by adding *x* and *y* to the translation vector defined by *transform*).

**draw_markers**(*gc*, *marker_path*, *marker_trans*, *path*, *trans*, *rgbFace=None*)

Draw a marker at each of *path*'s vertices (excluding control points).

The base (fallback) implementation makes multiple calls to *draw_path*. Backends may want to override this method in order to draw the marker only once and reuse it multiple times.

**Parameters**

**gc**

[*GraphicsContextBase*] The graphics context.

**marker_path**

[*Path*] The path for the marker.

**marker_trans**

[*Transform*] An affine transform applied to the marker.

**path**

[*Path*] The locations to draw the markers.

**trans**

[*Transform*] An affine transform applied to the path.

**rgbFace**

[*color*, optional]

**draw_path**(*gc*, *path*, *transform*, *rgbFace=None*)

Draw a *Path* instance using the given affine transform.

**draw_tex**(*gc*, *x*, *y*, *s*, *prop*, *angle*, *, *mtext=None*)

Draw a TeX instance.

**Parameters**

**gc**

[*GraphicsContextBase*] The graphics context.

**x**

[float] The x location of the text in display coords.

**y**

[float] The y location of the text baseline in display coords.

**s**

[str] The TeX text string.

**prop**

[*FontProperties*] The font properties.

**angle**

[float] The rotation angle in degrees anti-clockwise.

**mtext**

[*Text*] The original text object to be rendered.

**draw_text**(*gc*, *x*, *y*, *s*, *prop*, *angle*, *ismath=False*, *mtext=None*)

Draw a text instance.

**Parameters**

**gc**

[*GraphicsContextBase*] The graphics context.

**x**

[float] The x location of the text in display coords.

**y**

[float] The y location of the text baseline in display coords.

**s**

[str] The text string.

**prop**

[*FontProperties*] The font properties.

**angle**

[float] The rotation angle in degrees anti-clockwise.

**ismath**

[bool or "TeX"] If True, use mathtext parser. If "TeX", use tex for rendering.

**mtext**

[*Text*] The original text object to be rendered.

### Notes

**Note for backend implementers:**

When you are trying to determine if you have gotten your bounding box right (which is what enables the text layout/alignment to work properly), it helps to change the line in text.py:

```
if 0: bbox_artist(self, renderer)
```

to if 1, and then the actual bounding box will be plotted along with your text.

**flipy**()

Return whether y values increase from top to bottom.

Note that this only affects drawing of texts.

**get_canvas_width_height**()

Return the canvas width and height in display coords.

**get_text_width_height_descent**(*s*, *prop*, *ismath*)

Get the width, height, and descent (offset from the bottom to the baseline), in display coords, of the string *s* with *FontProperties* *prop*.

Whitespace at the start and the end of *s* is included in the reported width.

**option_image_nocomposite**()

Return whether image composition by Matplotlib should be skipped.

Raster backends should usually return False (letting the C-level rasterizer take care of image composition); vector backends should usually return `not rcParams["image.composite_image"]`.

**option_scale_image**()

Return whether arbitrary affine transformations in *draw_image* are supported (True for most vector backends).

**points_to_pixels**(*points*)

Convert points to display units.

You need to override this function (unless your backend doesn't have a dpi, e.g., postscript or svg). Some imaging systems assume some value for pixels per inch:

```
points to pixels = points * pixels_per_inch/72 * dpi/72
```

**Parameters**

**points**

[float or array-like]

**Returns**

**Points converted to pixels**

matplotlib.backends.backend_pgf.**make_pdf_to_png_converter**()

Return a function that converts a pdf file to a png file.

## matplotlib.backends.backend_ps

A PostScript backend, which can produce both PostScript .ps and .eps.

matplotlib.backends.backend_ps.**FigureCanvas**

alias of *FigureCanvasPS*

**class** matplotlib.backends.backend_ps.**FigureCanvasPS**(*figure=None*)

Bases: *FigureCanvasBase*

**draw**()

Render the *Figure*.

This method must walk the artist tree, even if no output is produced, because it triggers deferred work that users may want to access before saving output to disk. For example computing limits, auto-limits, and tick values.

**filetypes = {'eps': 'Encapsulated Postscript', 'ps': 'Postscript'}**

**fixed_dpi = 72**

**get_default_filetype**()

Return the default savefig file format as specified in *rcParams["savefig.format"]* (default: 'png').

The returned string does not include a period. This method is overridden in backends that only support a single file type.

**print_eps**(*outfile*, *, *metadata=None*, *papertype=None*, *orientation='portrait'*, *bbox_inches_restore=None*, **kwargs*)

**print_ps**(*outfile*, *, *metadata=None*, *papertype=None*, *orientation='portrait'*, *bbox_inches_restore=None*, **kwargs*)

**class** matplotlib.backends.backend_ps.**RendererPS**(*width*, *height*, *pswriter*, *imagedpi=72*)

Bases: RendererPDFPSBase

The renderer handles all the drawing primitives using a graphics context instance that controls the colors/styles.

**create_hatch**(*hatch*)

**draw_gouraud_triangles**(*gc*, *points*, *colors*, *trans*)

Draw a series of Gouraud triangles.

**Parameters**

**gc**

[*GraphicsContextBase*] The graphics context.

**triangles_array**

[(N, 3, 2) array-like] Array of *N* (x, y) points for the triangles.

**colors_array**

[(N, 3, 4) array-like] Array of *N* RGBA colors for each point of the triangles.

**transform**

[*Transform*] An affine transform to apply to the points.

**draw_image**(*gc*, *x*, *y*, *im*, *transform=None*)

Draw an RGBA image.

**Parameters**

**gc**

[*GraphicsContextBase*] A graphics context with clipping information.

**x**

[scalar] The distance in physical units (i.e., dots or pixels) from the left hand side of the canvas.

**y**

[scalar] The distance in physical units (i.e., dots or pixels) from the bottom side of the canvas.

**im**

[(N, M, 4) array of `numpy.uint8`] An array of RGBA pixels.

**transform**

[*Affine2DBase*] If and only if the concrete backend is written such that `option_scale_image` returns `True`, an affine transformation (i.e., an *Affine2DBase*) *may* be passed to *draw_image*. The translation vector of the transformation is given in physical units (i.e., dots or pixels). Note that the transformation does not override *x* and *y*, and has to be applied *before* translating the result by *x* and *y* (this can be accomplished by adding *x* and *y* to the translation vector defined by *transform*).

**draw_markers**(*gc*, *marker_path*, *marker_trans*, *path*, *trans*, *rgbFace=None*)

Draw a marker at each of *path*'s vertices (excluding control points).

The base (fallback) implementation makes multiple calls to *draw_path*. Backends may want to override this method in order to draw the marker only once and reuse it multiple times.

**Parameters**

**gc**

[*GraphicsContextBase*] The graphics context.

**marker_path**

[*Path*] The path for the marker.

**marker_trans**

[*Transform*] An affine transform applied to the marker.

**path**

[*Path*] The locations to draw the markers.

**trans**

[*Transform*] An affine transform applied to the path.

**rgbFace**

[*color*, optional]

**draw_mathtext**(*gc*, *x*, *y*, *s*, *prop*, *angle*)

Draw the math text using matplotlib.mathtext.

**draw_path**(*gc*, *path*, *transform*, *rgbFace=None*)

Draw a *Path* instance using the given affine transform.

**draw_path_collection**(*gc*, *master_transform*, *paths*, *all_transforms*, *offsets*, *offset_trans*, *facecolors*, *edgecolors*, *linewidths*, *linestyles*, *antialiaseds*, *urls*, *offset_position*)

Draw a collection of *paths*.

Each path is first transformed by the corresponding entry in *all_transforms* (a list of (3, 3) matrices) and then by *master_transform*. They are then translated by the corresponding entry in *offsets*, which has been first transformed by *offset_trans*.

*facecolors*, *edgecolors*, *linewidths*, *linestyles*, and *antialiased* are lists that set the corresponding properties.

*offset_position* is unused now, but the argument is kept for backwards compatibility.

The base (fallback) implementation makes multiple calls to *draw_path*. Backends may want to override this in order to render each set of path data only once, and then reference that path multiple times with the different offsets, colors, styles etc. The generator methods `_iter_collection_raw_paths` and `_iter_collection` are provided to help with (and standardize) the implementation across backends. It is highly recommended to use those generators, so that changes to the behavior of *draw_path_collection* can be made globally.

**draw_tex**(*gc*, *x*, *y*, *s*, *prop*, *angle*, *, *mtext=None*)

Draw a TeX instance.

**Parameters**

**gc**

[*GraphicsContextBase*] The graphics context.

**x**

[float] The x location of the text in display coords.

**y**

[float] The y location of the text baseline in display coords.

**s**

[str] The TeX text string.

**prop**

[*FontProperties*] The font properties.

**angle**

[float] The rotation angle in degrees anti-clockwise.

**mtext**

[*Text*] The original text object to be rendered.

**draw_text**(*gc*, *x*, *y*, *s*, *prop*, *angle*, *ismath=False*, *mtext=None*)

Draw a text instance.

**Parameters**

**gc**

[*GraphicsContextBase*] The graphics context.

**x**

[float] The x location of the text in display coords.

**y**

[float] The y location of the text baseline in display coords.

**s**

[str] The text string.

**prop**

[*FontProperties*] The font properties.

**angle**

[float] The rotation angle in degrees anti-clockwise.

**ismath**

[bool or "TeX"] If True, use mathtext parser. If "TeX", use tex for rendering.

**mtext**

[*Text*] The original text object to be rendered.

## Notes

**Note for backend implementers:**

When you are trying to determine if you have gotten your bounding box right (which is what enables the text layout/alignment to work properly), it helps to change the line in text.py:

```
if 0: bbox_artist(self, renderer)
```

to if 1, and then the actual bounding box will be plotted along with your text.

**get_image_magnification**()

Get the factor by which to magnify images passed to draw_image. Allows a backend to have images at a different resolution to other artists.

**set_color**(*r*, *g*, *b*, *store=True*)

**set_font**(*fontname*, *fontsize*, *store=True*)

**set_linecap**(*linecap*, *store=True*)

**set_linedash**(*offset*, *seq*, *store=True*)

**set_linejoin**(*linejoin*, *store=True*)

**set_linewidth**(*linewidth*, *store=True*)

matplotlib.backends.backend_ps.**get_bbox_header**(*lbrt*, *rotated=False*)

[*Deprecated*] Return a postscript header string for the given bbox lbrt=(l, b, r, t). Optionally, return rotate command.

### Notes

Deprecated since version 3.9.

matplotlib.backends.backend_ps.**gs_distill**(*tmpfile*, *eps=False*, *ptype='letter'*, *bbox=None*, *rotated=False*)

Use ghostscript's pswrite or epswrite device to distill a file. This yields smaller files without illegal encapsulated postscript operators. The output is low-level, converting text to outlines.

matplotlib.backends.backend_ps.**pstoeps**(*tmpfile*, *bbox=None*, *rotated=False*)

Convert the postscript to encapsulated postscript. The bbox of the eps file will be replaced with the given *bbox* argument. If None, original bbox will be used.

matplotlib.backends.backend_ps.**xpdf_distill**(*tmpfile*, *eps=False*, *ptype='letter'*, *bbox=None*, *rotated=False*)

Use ghostscript's ps2pdf and xpdf's/poppler's pdftops to distill a file. This yields smaller files without illegal encapsulated postscript operators. This distiller is preferred, generating high-level postscript output that treats text as text.

## matplotlib.backends.registry

**class** matplotlib.backends.registry.**BackendFilter**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

Filter used with *list_builtin()*

New in version 3.9.

**INTERACTIVE = 0**

**NON_INTERACTIVE = 1**

**class** matplotlib.backends.registry.**BackendRegistry**

Bases: object

Registry of backends available within Matplotlib.

This is the single source of truth for available backends.

All use of BackendRegistry should be via the singleton instance backend_registry which can be imported from matplotlib.backends.

Each backend has a name, a module name containing the backend code, and an optional GUI framework that must be running if the backend is interactive. There are three sources of backends: built-in (source code is within the Matplotlib repository), explicit module://some.backend syntax (backend is obtained by loading the module), or via an entry point (self-registering backend in an external package).

New in version 3.9.

**backend_for_gui_framework**(*framework*)

Return the name of the backend corresponding to the specified GUI framework.

**Parameters**

**framework**

[str] GUI framework such as "qt".

**Returns**

**str or None**

Backend name or None if GUI framework not recognised.

**is_valid_backend**(*backend*)

Return True if the backend name is valid, False otherwise.

A backend name is valid if it is one of the built-in backends or has been dynamically added via an entry point. Those beginning with module:// are always considered valid and are added to the current list of all backends within this function.

Even if a name is valid, it may not be importable or usable. This can only be determined by loading and using the backend module.

**Parameters**

**backend**

[str] Name of backend.

**Returns**

**bool**

True if backend is valid, False otherwise.

**list_all**()

Return list of all known backends.

These include built-in backends and those obtained at runtime either from entry points or explicit `module://some.backend` syntax.

Entry points will be loaded if they haven't been already.

**Returns**

**list of str**

Backend names.

**list_builtin**(*filter_=None*)

Return list of backends that are built into Matplotlib.

**Parameters**

**filter_**

[*BackendFilter*, optional] Filter to apply to returned backends. For example, to return only non-interactive backends use *BackendFilter.NON_INTERACTIVE*.

**Returns**

**list of str**

Backend names.

**list_gui_frameworks**()

Return list of GUI frameworks used by Matplotlib backends.

**Returns**

**list of str**

GUI framework names.

**load_backend_module**(*backend*)

Load and return the module containing the specified backend.

**Parameters**

**backend**

[str] Name of backend to load.

**Returns**

**Module**

Module containing backend.

**resolve_backend**(*backend*)

Return the backend and GUI framework for the specified backend name.

If the GUI framework is not yet known then it will be determined by loading the backend module and checking the `FigureCanvas.required_interactive_framework` attribute.

This function only loads entry points if they have not already been loaded and the backend is not built-in and not of `module://some.backend` format.

**Parameters**

**backend**

[str or None] Name of backend, or None to use the default backend.

**Returns**

**backend**

[str] The backend name.

**framework**

[str or None] The GUI framework, which will be None for a backend that is non-interactive.

**resolve_gui_or_backend**(*gui_or_backend*)

Return the backend and GUI framework for the specified string that may be either a GUI framework or a backend name, tested in that order.

This is for use with the IPython %matplotlib magic command which may be a GUI framework such as `%matplotlib qt` or a backend name such as `%matplotlib qtagg`.

This function only loads entry points if they have not already been loaded and the backend is not built-in and not of `module://some.backend` format.

**Parameters**

**gui_or_backend**

[str or None] Name of GUI framework or backend, or None to use the default backend.

**Returns**

**backend**

[str] The backend name.

**framework**

[str or None] The GUI framework, which will be None for a backend that is non-interactive.

### matplotlib.backends.backend_qtagg, matplotlib.backends.backend_qtcairo

**NOTE** These *Backends* are not (auto) documented here, to avoid adding a dependency to building the docs.

## Qt Bindings

There are currently 2 actively supported Qt versions, Qt5 and Qt6, and two supported Python bindings per version -- PyQt5 and PySide2 for Qt5 and PyQt6 and PySide6 for Qt6[1]. Matplotlib's qtagg and qtcairo backends (`matplotlib.backends.backend_qtagg` and `matplotlib.backend.backend_qtcairo`) support all these bindings, with common parts factored out in the `matplotlib.backends.backend_qt` module.

At runtime, these backends select the actual binding used as follows:

1. If a binding's `QtCore` subpackage is already imported, that binding is selected (the order for the check is `PyQt6`, `PySide6`, `PyQt5`, `PySide2`).
2. If the *`QT_API`* environment variable is set to one of "PyQt6", "PySide6", "PyQt5", "PySide2" (case-insensitive), that binding is selected. (See also the documentation on *Environment variables*.)
3. Otherwise, the first available backend in the order `PyQt6`, `PySide6`, `PyQt5`, `PySide2` is selected.

In the past, Matplotlib used to have separate backends for each version of Qt (e.g. qt4agg/`matplotlib.backends.backend_qt4agg` and qt5agg/`matplotlib.backends.backend_qt5agg`). This scheme was dropped when support for Qt6 was added. For back-compatibility, qt5agg/`backend_qt5agg` and qt5cairo/`backend_qt5cairo` remain available; selecting one of these backends forces the use of a Qt5 binding. Their use is discouraged and `backend_qtagg` or `backend_qtcairo` should be preferred instead. However, these modules will not be deprecated until we drop support for Qt5.

While both PyQt and Qt for Python (aka PySide) closely mirror the underlying C++ API they are wrapping, they are not drop-in replacements for each other[2]. To account for this, Matplotlib has an internal API compatibility layer in *`matplotlib.backends.qt_compat`* which covers our needs. Despite being a public module, we do not consider this to be a stable user-facing API and it may change without warning[3].

[1] There is also PyQt4 and PySide for Qt4 but these are no longer supported by Matplotlib and upstream support for Qt4 ended in 2015.

[2] Despite the slight API differences, the more important distinction between the PyQt and Qt for Python series of bindings is licensing.

[3] If you are looking for a general purpose compatibility library please see qtpy.

### matplotlib.backends.backend_svg

matplotlib.backends.backend_svg.**FigureCanvas**

alias of *FigureCanvasSVG*

**class** matplotlib.backends.backend_svg.**FigureCanvasSVG**(*figure=None*)

Bases: *FigureCanvasBase*

**draw**()

Render the *Figure*.

This method must walk the artist tree, even if no output is produced, because it triggers deferred work that users may want to access before saving output to disk. For example computing limits, auto-limits, and tick values.

**filetypes = {'svg': 'Scalable Vector Graphics', 'svgz': 'Scalable Vector Graphics'}**

**fixed_dpi = 72**

**get_default_filetype**()

Return the default savefig file format as specified in *rcParams["savefig.format"]* (default: 'png').

The returned string does not include a period. This method is overridden in backends that only support a single file type.

**print_svg**(*filename*, *, *bbox_inches_restore=None*, *metadata=None*)

**Parameters**

**filename**

[str or path-like or file-like] Output target; if a string, a file will be opened for writing.

**metadata**

[dict[str, Any], optional] Metadata in the SVG file defined as key-value pairs of strings, datetimes, or lists of strings, e.g., {'Creator': 'My software', 'Contributor': ['Me', 'My Friend'], 'Title': 'Awesome'}.

The standard keys and their value types are:

- *str*: 'Coverage', 'Description', 'Format', 'Identifier', 'Language', 'Relation', 'Source', 'Title', and 'Type'.
- *str* or *list of str*: 'Contributor', 'Creator', 'Keywords', 'Publisher', and 'Rights'.
- *str*, *date*, *datetime*, or *tuple* of same: 'Date'. If a non-*str*, then it will be formatted as ISO 8601.

Values have been predefined for `'Creator'`, `'Date'`, `'Format'`, and `'Type'`. They can be removed by setting them to `None`.

Information is encoded as Dublin Core Metadata.

**print_svgz**(*filename*, ***kwargs*)

**class** matplotlib.backends.backend_svg.**RendererSVG**(*width*, *height*, *svgwriter*, *basename=None*, *image_dpi=72*, ***, *metadata=None*)

Bases: *RendererBase*

**close_group**(*s*)

Close a grouping element with label *s*.

Only used by the SVG renderer.

**draw_gouraud_triangles**(*gc*, *triangles_array*, *colors_array*, *transform*)

Draw a series of Gouraud triangles.

**Parameters**

**gc**

[*GraphicsContextBase*] The graphics context.

**triangles_array**

[(N, 3, 2) array-like] Array of *N* (x, y) points for the triangles.

**colors_array**

[(N, 3, 4) array-like] Array of *N* RGBA colors for each point of the triangles.

**transform**

[*Transform*] An affine transform to apply to the points.

**draw_image**(*gc*, *x*, *y*, *im*, *transform=None*)

Draw an RGBA image.

**Parameters**

**gc**

[*GraphicsContextBase*] A graphics context with clipping information.

**x**

[scalar] The distance in physical units (i.e., dots or pixels) from the left hand side of the canvas.

**y**

[scalar] The distance in physical units (i.e., dots or pixels) from the bottom side of the canvas.

**im**

[(N, M, 4) array of `numpy.uint8`] An array of RGBA pixels.

**transform**

[*`Affine2DBase`*] If and only if the concrete backend is written such that *`option_scale_image`* returns `True`, an affine transformation (i.e., an *`Affine2DBase`*) *may* be passed to *`draw_image`*. The translation vector of the transformation is given in physical units (i.e., dots or pixels). Note that the transformation does not override *x* and *y*, and has to be applied *before* translating the result by *x* and *y* (this can be accomplished by adding *x* and *y* to the translation vector defined by *transform*).

**`draw_markers`**(*gc*, *marker_path*, *marker_trans*, *path*, *trans*, *rgbFace=None*)

Draw a marker at each of *path*'s vertices (excluding control points).

The base (fallback) implementation makes multiple calls to *`draw_path`*. Backends may want to override this method in order to draw the marker only once and reuse it multiple times.

**Parameters**

**gc**

[*`GraphicsContextBase`*] The graphics context.

**marker_path**

[*`Path`*] The path for the marker.

**marker_trans**

[*`Transform`*] An affine transform applied to the marker.

**path**

[*`Path`*] The locations to draw the markers.

**trans**

[*`Transform`*] An affine transform applied to the path.

**rgbFace**

[*color*, optional]

**`draw_path`**(*gc*, *path*, *transform*, *rgbFace=None*)

Draw a *`Path`* instance using the given affine transform.

**draw_path_collection**(*gc*, *master_transform*, *paths*, *all_transforms*, *offsets*, *offset_trans*, *facecolors*, *edgecolors*, *linewidths*, *linestyles*, *antialiaseds*, *urls*, *offset_position*)

Draw a collection of *paths*.

Each path is first transformed by the corresponding entry in *all_transforms* (a list of (3, 3) matrices) and then by *master_transform*. They are then translated by the corresponding entry in *offsets*, which has been first transformed by *offset_trans*.

*facecolors*, *edgecolors*, *linewidths*, *linestyles*, and *antialiased* are lists that set the corresponding properties.

*offset_position* is unused now, but the argument is kept for backwards compatibility.

The base (fallback) implementation makes multiple calls to *draw_path*. Backends may want to override this in order to render each set of path data only once, and then reference that path multiple times with the different offsets, colors, styles etc. The generator methods `_iter_collection_raw_paths` and `_iter_collection` are provided to help with (and standardize) the implementation across backends. It is highly recommended to use those generators, so that changes to the behavior of *draw_path_collection* can be made globally.

**draw_text**(*gc*, *x*, *y*, *s*, *prop*, *angle*, *ismath=False*, *mtext=None*)

Draw a text instance.

**Parameters**

**gc**

[*GraphicsContextBase*] The graphics context.

**x**

[float] The x location of the text in display coords.

**y**

[float] The y location of the text baseline in display coords.

**s**

[str] The text string.

**prop**

[*FontProperties*] The font properties.

**angle**

[float] The rotation angle in degrees anti-clockwise.

**ismath**

[bool or "TeX"] If True, use mathtext parser. If "TeX", use tex for rendering.

**mtext**

[*Text*] The original text object to be rendered.

### Notes

**Note for backend implementers:**

When you are trying to determine if you have gotten your bounding box right (which is what enables the text layout/alignment to work properly), it helps to change the line in text.py:

```
if 0: bbox_artist(self, renderer)
```

to if 1, and then the actual bounding box will be plotted along with your text.

**finalize**()

**flipy**()

Return whether y values increase from top to bottom.

Note that this only affects drawing of texts.

**get_canvas_width_height**()

Return the canvas width and height in display coords.

**get_image_magnification**()

Get the factor by which to magnify images passed to *draw_image*. Allows a backend to have images at a different resolution to other artists.

**get_text_width_height_descent**(*s*, *prop*, *ismath*)

Get the width, height, and descent (offset from the bottom to the baseline), in display coords, of the string *s* with *FontProperties* *prop*.

Whitespace at the start and the end of *s* is included in the reported width.

**open_group**(*s*, *gid=None*)

Open a grouping element with label *s* and *gid* (if set) as id.

Only used by the SVG renderer.

**option_image_nocomposite**()

Return whether image composition by Matplotlib should be skipped.

Raster backends should usually return False (letting the C-level rasterizer take care of image composition); vector backends should usually return `not rcParams["image.composite_image"]`.

**option_scale_image**()

Return whether arbitrary affine transformations in *draw_image* are supported (True for most vector backends).

**class** matplotlib.backends.backend_svg.**XMLWriter**(*file*)

Bases: object

**Parameters**

**file**

[writable text file-like object]

**close**(*id*)

Close open elements, up to (and including) the element identified by the given identifier.

**Parameters**

**id**

Element identifier, as returned by the *start()* method.

**comment**(*comment*)

Add a comment to the output stream.

**Parameters**

**comment**

[str] Comment text.

**data**(*text*)

Add character data to the output stream.

**Parameters**

**text**

[str] Character data.

**element**(*tag*, *text=None*, *attrib={}*, ***extra*)

Add an entire element. This is the same as calling *start()*, *data()*, and *end()* in sequence. The *text* argument can be omitted.

**end**(*tag=None*, *indent=True*)

Close the current element (opened by the most recent call to *start()*).

**Parameters**

**tag**

Element tag. If given, the tag must match the start tag. If omitted, the current element is closed.

**indent**

[bool, default: True]

**flush**()

Flush the output stream.

**start**(*tag*, *attrib={}*, ***extra*)

Open a new element. Attributes can be given as keyword arguments, or as a string/string dictionary. The method returns an opaque identifier that can be passed to the *close()* method, to close all open elements up to and including this one.

**Parameters**

**tag**

Element tag.

**attrib**

Attribute dictionary. Alternatively, attributes can be given as keyword arguments.

**Returns**

**An element identifier.**

### matplotlib.backends.backend_tkagg, matplotlib.backends.backend_tkcairo

matplotlib.backends.backend_tkagg.**FigureCanvas**

alias of *FigureCanvasTkAgg*

**class** matplotlib.backends.backend_tkagg.**FigureCanvasTkAgg**(*figure=None*, *master=None*)

Bases: *FigureCanvasAgg*, FigureCanvasTk

**blit**(*bbox=None*)

Blit the canvas in bbox (default entire canvas).

**draw**()

Render the *Figure*.

This method must walk the artist tree, even if no output is produced, because it triggers deferred work that users may want to access before saving output to disk. For example computing limits, auto-limits, and tick values.

matplotlib.backends.backend_tkcairo.**FigureCanvas**

alias of *FigureCanvasTkCairo*

**class** matplotlib.backends.backend_tkcairo.**FigureCanvasTkCairo**(*figure=None*, *master=None*)

Bases: *FigureCanvasCairo*, FigureCanvasTk

**draw**()

Render the *Figure*.

This method must walk the artist tree, even if no output is produced, because it triggers deferred work that users may want to access before saving output to disk. For example computing limits, auto-limits, and tick values.

### matplotlib.backends.backend_webagg_core

Displays Agg images in the browser, with interactivity.

matplotlib.backends.backend_webagg_core.**FigureCanvas**

alias of *FigureCanvasWebAggCore*

**class** matplotlib.backends.backend_webagg_core.**FigureCanvasWebAggCore**(**args*, ***kwargs*)

Bases: *FigureCanvasAgg*

**blit**(*bbox=None*)

Blit the canvas in bbox (default entire canvas).

**draw**()

Render the *Figure*.

This method must walk the artist tree, even if no output is produced, because it triggers deferred work that users may want to access before saving output to disk. For example computing limits, auto-limits, and tick values.

**draw_idle**()

Request a widget redraw once control returns to the GUI event loop.

Even if multiple calls to *draw_idle* occur before control returns to the GUI event loop, the figure will only be rendered once.

#### Notes

Backends may choose to override the method and implement their own strategy to prevent multiple renderings.

**get_diff_image**()

**handle_ack**(*event*)

**handle_button_press**(*event*)

**handle_button_release**(*event*)

**handle_dblclick**(*event*)

**handle_draw**(*event*)

**handle_event**(*event*)

**handle_figure_enter**(*event*)

**handle_figure_leave**(*event*)

**handle_key_press**(*event*)

**handle_key_release**(*event*)

**handle_motion_notify**(*event*)

**handle_refresh**(*event*)

**handle_resize**(*event*)

**handle_scroll**(*event*)

**handle_send_image_mode**(*event*)

**handle_set_device_pixel_ratio**(*event*)

**handle_set_dpi_ratio**(*event*)

**handle_toolbar_button**(*event*)

**handle_unknown_event**(*event*)

**manager_class**

alias of *FigureManagerWebAgg*

**send_event**(*event_type*, ***kwargs*)

**set_cursor**(*cursor*)

Set the current cursor.

This may have no effect if the backend does not display anything.

If required by the backend, this method should trigger an update in the backend event loop after the cursor is set, as this method may be called e.g. before a long-running task during which the GUI is not updated.

**Parameters**

**cursor**

[*Cursors*] The cursor to display over the canvas. Note: some backends may change the cursor for the entire window.

**set_image_mode**(*mode*)

Set the image mode for any subsequent images which will be sent to the clients. The modes may currently be either 'full' or 'diff'.

Note: diff images may not contain transparency, therefore upon draw this mode may be changed if the resulting image has any transparent component.

**show**()

**supports_blit = False**

matplotlib.backends.backend_webagg_core.**FigureManager**

alias of *FigureManagerWebAgg*

**class** matplotlib.backends.backend_webagg_core.**FigureManagerWebAgg**(*canvas*, *num*)

Bases: *FigureManagerBase*

**ToolbarCls**

alias of *NavigationToolbar2WebAgg*

**add_web_socket**(*web_socket*)

**classmethod get_javascript**(*stream=None*)

**classmethod get_static_file_path**()

**get_window_title**()

Return the title text of the window containing the figure, or None if there is no window (e.g., a PS backend).

**handle_json**(*content*)

**refresh_all**()

**remove_web_socket**(*web_socket*)

**resize**(*w*, *h*, *forward=True*)

For GUI backends, resize the window (in physical pixels).

**set_window_title**(*title*)

Set the title text of the window containing the figure.

This has no effect for non-GUI (e.g., PS) backends.

### Examples

```
>>> fig = plt.figure()
>>> fig.canvas.manager.set_window_title('My figure')
```

**show**()

For GUI backends, show the figure window and redraw. For non-GUI backends, raise an exception, unless running headless (i.e. on Linux with an unset DISPLAY); this exception is converted to a warning in *Figure.show*.

**class** matplotlib.backends.backend_webagg_core.**NavigationToolbar2WebAgg**(*canvas*)

Bases: *NavigationToolbar2*

**draw_rubberband**(*event*, *x0*, *y0*, *x1*, *y1*)

Draw a rectangle rubberband to indicate zoom limits.

Note that it is not guaranteed that `x0 <= x1` and `y0 <= y1`.

**pan**()

Toggle the pan/zoom tool.

Pan with left button, zoom with right.

**remove_rubberband**()

Remove the rubberband.

**save_figure**(**args*)

Save the current figure

**set_history_buttons**()

Enable or disable the back/forward button.

**set_message**(*message*)

Display a message on toolbar or in status bar.

**toolitems = [('Home', 'Reset original view', 'home', 'home'), ('Back', 'Back to previous view', 'back', 'back'), ('Forward', 'Forward to next view', 'forward', 'forward'), (None, None, None, None), ('Pan', 'Left button pans, Right button zooms\nx/y fixes axis, CTRL fixes aspect', 'move', 'pan'), ('Zoom', 'Zoom to rectangle\nx/y fixes axis', 'zoom_to_rect', 'zoom'), (None, None, None, None), ('Download', 'Download plot', 'filesave', 'download')]**

**zoom**()

**class** matplotlib.backends.backend_webagg_core.**TimerAsyncio**(**args*, ***kwargs*)

Bases: *TimerBase*

**Parameters**

**interval**

[int, default: 1000ms] The time between timer events in milliseconds. Will be stored as `timer.interval`.

**callbacks**

[list[tuple[callable, tuple, dict]]] List of (func, args, kwargs) tuples that will be called upon timer events. This list is accessible as `timer.callbacks` and can be manipulated directly, or the functions *add_callback* and *remove_callback* can be used.

**class** matplotlib.backends.backend_webagg_core.**TimerTornado**(**args*, ***kwargs*)

Bases: *TimerBase*

**Parameters**

**interval**

[int, default: 1000ms] The time between timer events in milliseconds. Will be stored as `timer.interval`.

**callbacks**

[list[tuple[callable, tuple, dict]]] List of (func, args, kwargs) tuples that will be called upon timer events. This list is accessible as `timer.callbacks` and can be manipulated directly, or the functions *add_callback* and *remove_callback* can be used.

## matplotlib.backends.backend_webagg

Displays Agg images in the browser, with interactivity.

matplotlib.backends.backend_webagg.**FigureCanvas**

alias of *FigureCanvasWebAgg*

**class** matplotlib.backends.backend_webagg.**FigureCanvasWebAgg**(**args*, ***kwargs*)

Bases: *FigureCanvasWebAggCore*

**manager_class**

alias of *FigureManagerWebAgg*

matplotlib.backends.backend_webagg.**FigureManager**

alias of *FigureManagerWebAgg*

**class** matplotlib.backends.backend_webagg.**FigureManagerWebAgg**(*canvas*, *num*)

Bases: *FigureManagerWebAgg*

**classmethod pyplot_show**(*, *block=None*)

Show all figures. This method is the implementation of *pyplot.show*.

To customize the behavior of *pyplot.show*, interactive backends should usually override *start_main_loop*; if more customized logic is necessary, *pyplot_show* can also be overridden.

**Parameters**

**block**

[bool, optional] Whether to block by calling `start_main_loop`. The default, None, means to block if we are neither in IPython's `%pylab` mode nor in `interactive` mode.

**class** matplotlib.backends.backend_webagg.**WebAggApplication**(*url_prefix=''*)

Bases: Application

**class AllFiguresPage**(*application*, *request*, *, *url_prefix=''*, ***kwargs*)

Bases: RequestHandler

**get**()

**class Download**(*application*, *request*, ***kwargs*)

Bases: RequestHandler

**get**(*fignum*, *fmt*)

**class FavIcon**(*application*, *request*, ***kwargs*)

Bases: RequestHandler

**get**()

**class MplJs**(*application*, *request*, ***kwargs*)

Bases: RequestHandler

**get**()

**class SingleFigurePage**(*application*, *request*, *, *url_prefix=''*, ***kwargs*)

Bases: RequestHandler

**get**(*fignum*)

**class WebSocket**(*application*, *request*, ***kwargs*)

Bases: WebSocketHandler

**on_close**()

Invoked when the WebSocket is closed.

If the connection was closed cleanly and a status code or reason phrase was supplied, these values will be available as the attributes `self.close_code` and `self.close_reason`.

Changed in version 4.0: Added `close_code` and `close_reason` attributes.

**on_message**(*message*)

Handle incoming messages on the WebSocket

This method must be overridden.

Changed in version 4.5: `on_message` can be a coroutine.

**open**(*fignum*)

Invoked when a new WebSocket is opened.

The arguments to *open* are extracted from the `tornado.web.URLSpec` regular expression, just like the arguments to `tornado.web.RequestHandler.get`.

*open* may be a coroutine. *on_message* will not be called until *open* has returned.

Changed in version 5.1: `open` may be a coroutine.

**send_binary**(*blob*)

**send_json**(*content*)

**supports_binary = True**

**classmethod initialize**(*url_prefix=''*, *port=None*, *address=None*)

**initialized = False**

**classmethod start**()

**started = False**

matplotlib.backends.backend_webagg.**ipython_inline_display**(*figure*)

#### matplotlib.backends.backend_wxagg, matplotlib.backends.backend_wxcairo

**NOTE** These *Backends* are not documented here, to avoid adding a dependency to building the docs.

### 7.2.11 matplotlib.bezier

A module providing some utility functions regarding Bézier path manipulation.

**class** matplotlib.bezier.**BezierSegment**(*control_points*)

Bases: object

A d-dimensional Bézier segment.

**Parameters**

**control_points**

[(N, d) array] Location of the *N* control points.

**axis_aligned_extrema**()

Return the dimension and location of the curve's interior extrema.

The extrema are the points along the curve where one of its partial derivatives is zero.

**Returns**

**dims**

[array of int] Index $i$ of the partial derivative which is zero at each interior extrema.

**dzeros**

[array of float] Of same size as dims. The $t$ such that $d/dx_i B(t) = 0$

**property control_points**

The control points of the curve.

**property degree**

Degree of the polynomial. One less the number of control points.

**property dimension**

The dimension of the curve.

**point_at_t**(*t*)

Evaluate the curve at a single point, returning a tuple of *d* floats.

**property polynomial_coefficients**

The polynomial coefficients of the Bézier curve.

> **Warning:** Follows opposite convention from `numpy.polyval`.

**Returns**

**(n+1, d) array**

Coefficients after expanding in polynomial basis, where $n$ is the degree of the Bézier curve and $d$ its dimension. These are the numbers ($C_j$) such that the curve can be written $\sum_{j=0}^{n} C_j t^j$.

### Notes

The coefficients are calculated as

$$\binom{n}{j} \sum_{i=0}^{j} (-1)^{i+j} \binom{j}{i} P_i$$

where $P_i$ are the control points of the curve.

**exception** matplotlib.bezier.**NonIntersectingPathException**

Bases: `ValueError`

matplotlib.bezier.**check_if_parallel**(*dx1*, *dy1*, *dx2*, *dy2*, *tolerance=1e-05*)

Check if two lines are parallel.

**Parameters**

**dx1, dy1, dx2, dy2**

[float] The gradients *dy*/*dx* of the two lines.

**tolerance**

[float] The angular tolerance in radians up to which the lines are considered parallel.

**Returns**

**is_parallel**

- 1 if two lines are parallel in same direction.
- -1 if two lines are parallel in opposite direction.
- False otherwise.

matplotlib.bezier.**find_bezier_t_intersecting_with_closedpath**(*bezier_point_at_t*, *inside_closedpath*, *t0=0.0*, *t1=1.0*, *tolerance=0.01*)

Find the intersection of the Bézier curve with a closed path.

The intersection point *t* is approximated by two parameters *t0*, *t1* such that *t0* <= *t* <= *t1*.

Search starts from *t0* and *t1* and uses a simple bisecting algorithm therefore one of the end points must be inside the path while the other doesn't. The search stops when the distance of the points parametrized by *t0* and *t1* gets smaller than the given *tolerance*.

**Parameters**

**bezier_point_at_t**

[callable] A function returning x, y coordinates of the Bézier at parameter *t*. It must have the signature:

```
bezier_point_at_t(t: float) -> tuple[float, float]
```

**inside_closedpath**

[callable] A function returning True if a given point (x, y) is inside the closed path. It must have the signature:

```
inside_closedpath(point: tuple[float, float]) -> bool
```

**t0, t1**

[float] Start parameters for the search.

**tolerance**

[float] Maximal allowed distance between the final points.

**Returns**

**t0, t1**

[float] The Bézier path parameters.

matplotlib.bezier.**find_control_points**(*c1x*, *c1y*, *mmx*, *mmy*, *c2x*, *c2y*)

Find control points of the Bézier curve passing through (*c1x*, *c1y*), (*mmx*, *mmy*), and (*c2x*, *c2y*), at parametric values 0, 0.5, and 1.

matplotlib.bezier.**get_cos_sin**(*x0*, *y0*, *x1*, *y1*)

matplotlib.bezier.**get_intersection**(*cx1*, *cy1*, *cos_t1*, *sin_t1*, *cx2*, *cy2*, *cos_t2*, *sin_t2*)

Return the intersection between the line through (*cx1*, *cy1*) at angle *t1* and the line through (*cx2*, *cy2*) at angle *t2*.

matplotlib.bezier.**get_normal_points**(*cx*, *cy*, *cos_t*, *sin_t*, *length*)

For a line passing through (*cx*, *cy*) and having an angle *t*, return locations of the two points located along its perpendicular line at the distance of *length*.

matplotlib.bezier.**get_parallels**(*bezier2*, *width*)

Given the quadratic Bézier control points *bezier2*, returns control points of quadratic Bézier lines roughly parallel to given one separated by *width*.

matplotlib.bezier.**inside_circle**(*cx*, *cy*, *r*)

Return a function that checks whether a point is in a circle with center (*cx*, *cy*) and radius *r*.

The returned function has the signature:

```
f(xy: tuple[float, float]) -> bool
```

matplotlib.bezier.**make_wedged_bezier2**(*bezier2*, *width*, *w1=1.0*, *wm=0.5*, *w2=0.0*)

Being similar to *get_parallels*, returns control points of two quadratic Bézier lines having a width roughly parallel to given one separated by *width*.

matplotlib.bezier.**split_bezier_intersecting_with_closedpath**(*bezier*, *inside_closedpath*, *tolerance=0.01*)

Split a Bézier curve into two at the intersection with a closed path.

**Parameters**

**bezier**

[(N, 2) array-like] Control points of the Bézier segment. See *BezierSegment*.

**inside_closedpath**

[callable] A function returning True if a given point (x, y) is inside the closed path. See also *find_bezier_t_intersecting_with_closedpath*.

**tolerance**

[float] The tolerance for the intersection. See also *find_bezier_t_intersecting_with_closedpath*.

**Returns**

**left, right**

Lists of control points for the two Bézier segments.

matplotlib.bezier.**split_de_casteljau**(*beta*, *t*)

Split a Bézier segment defined by its control points *beta* into two separate segments divided at *t* and return their control points.

matplotlib.bezier.**split_path_inout**(*path*, *inside*, *tolerance=0.01*, *reorder_inout=False*)

Divide a path into two segments at the point where `inside(x, y)` becomes False.

### 7.2.12 matplotlib.category

Plotting of string "category" data: `plot(['d', 'f', 'a'], [1, 2, 3])` will plot three points with x-axis values of 'd', 'f', 'a'.

See *Plotting categorical variables* for an example.

The module uses Matplotlib's *matplotlib.units* mechanism to convert from strings to integers and provides a tick locator, a tick formatter, and the *UnitData* class that creates and stores the string-to-integer mapping.

**class** matplotlib.category.**StrCategoryConverter**

Bases: *ConversionInterface*

**static axisinfo**(*unit*, *axis*)

Set the default axis ticks and labels.

**Parameters**

**unit**

[*UnitData*] object string unit information for value

**axis**

[*Axis*] axis for which information is being set

---

**Note:** *axis* is not used

---

**Returns**

***AxisInfo***

Information to support default tick labeling

**static convert**(*value*, *unit*, *axis*)

Convert strings in *value* to floats using mapping information stored in the *unit* object.

**Parameters**

**value**

[str or iterable] Value or list of values to be converted.

**unit**

[*UnitData*] An object mapping strings to integers.

**axis**

[*Axis*] The axis on which the converted value is plotted.

---

**Note:** *axis* is unused.

---

**Returns**

**float or ndarray of float**

**static default_units**(*data*, *axis*)

Set and update the *Axis* units.

**Parameters**

**data**

[str or iterable of str]

**axis**

[*Axis*] axis on which the data is plotted

**Returns**

***UnitData***

object storing string to integer mapping

**class** matplotlib.category.**StrCategoryFormatter**(*units_mapping*)

Bases: *Formatter*

String representation of the data at every tick.

**Parameters**

**units_mapping**

[dict] Mapping of category names (str) to indices (int).

**format_ticks**(*values*)

Return the tick labels for all the ticks at once.

**class** matplotlib.category.**StrCategoryLocator**(*units_mapping*)

Bases: *Locator*

Tick at every integer mapping of the string data.

**Parameters**

**units_mapping**

[dict] Mapping of category names (str) to indices (int).

**tick_values**(*vmin*, *vmax*)

Return the values of the located ticks given **vmin** and **vmax**.

---

**Note:** To get tick locations with the vmin and vmax values defined automatically for the associated `axis` simply call the Locator instance:

```
>>> print(type(loc))
<type 'Locator'>
>>> print(loc())
[1, 2, 3, 4]
```

---

**class** matplotlib.category.**UnitData**(*data=None*)

Bases: object

Create mapping between unique categorical values and integer ids.

**Parameters**

**data**

[iterable] sequence of string values

**update**(*data*)

Map new values to integer identifiers.

**Parameters**

**data**

[iterable of str or bytes]

**Raises**

**TypeError**

If elements in *data* are neither str nor bytes.

### 7.2.13 matplotlib.cbook

A collection of utility functions and classes. Originally, many (but not all) were from the Python Cookbook -- hence the name cbook.

**class** matplotlib.cbook.**CallbackRegistry**(*exception_handler=<function _exception_printer>*, ***, *signals=None*)

Bases: object

Handle registering, processing, blocking, and disconnecting for a set of signals and callbacks:

```
>>> def oneat(x):
...     print('eat', x)
>>> def ondrink(x):
...     print('drink', x)
```

```
>>> from matplotlib.cbook import CallbackRegistry
>>> callbacks = CallbackRegistry()
```

```
>>> id_eat = callbacks.connect('eat', oneat)
>>> id_drink = callbacks.connect('drink', ondrink)
```

```
>>> callbacks.process('drink', 123)
drink 123
>>> callbacks.process('eat', 456)
eat 456
>>> callbacks.process('be merry', 456)   # nothing will be called
```

```
>>> callbacks.disconnect(id_eat)
>>> callbacks.process('eat', 456)        # nothing will be called
```

```
>>> with callbacks.blocked(signal='drink'):
...     callbacks.process('drink', 123)  # nothing will be called
>>> callbacks.process('drink', 123)
drink 123
```

In practice, one should always disconnect all callbacks when they are no longer needed to avoid dangling references (and thus memory leaks). However, real code in Matplotlib rarely does so, and due to its design, it is rather difficult to place this kind of code. To get around this, and prevent this class of memory leaks, we instead store weak references to bound methods only, so when the destination object needs to die, the CallbackRegistry won't keep it alive.

**Parameters**

**exception_handler**

[callable, optional] If not None, *exception_handler* must be a function that takes an Exception as single parameter. It gets called with any Exception raised by the callbacks during *CallbackRegistry.process*, and may either re-raise the exception or handle it in another manner.

The default handler prints the exception (with `traceback.print_exc`) if an interactive event loop is running; it re-raises the exception if no interactive event loop is running.

**signals**

[list, optional] If not None, *signals* is a list of signals that this registry handles: attempting to *process* or to *connect* to a signal not in the list throws a `ValueError`. The default, None, does not restrict the handled signals.

**blocked**(*, *signal=None*)

Block callback signals from being processed.

A context manager to temporarily block/disable callback signals from being processed by the registered listeners.

**Parameters**

**signal**

[str, optional] The callback signal to block. The default is to block all signals.

**connect**(*signal*, *func*)

Register *func* to be called when signal *signal* is generated.

**disconnect**(*cid*)

Disconnect the callback registered with callback id *cid*.

No error is raised if such a callback does not exist.

**process**(*s*, **args*, ***kwargs*)

Process signal *s*.

All of the functions registered to receive callbacks on *s* will be called with `*args` and `**kwargs`.

**class** matplotlib.cbook.**Grouper**(*init=()*)

Bases: `object`

A disjoint-set data structure.

Objects can be joined using *join()*, tested for connectedness using *joined()*, and all disjoint sets can be retrieved by using the object as an iterator.

The objects being joined must be hashable and weak-referenceable.

### Examples

```
>>> from matplotlib.cbook import Grouper
>>> class Foo:
...     def __init__(self, s):
...         self.s = s
...     def __repr__(self):
...         return self.s
...
>>> a, b, c, d, e, f = [Foo(x) for x in 'abcdef']
>>> grp = Grouper()
>>> grp.join(a, b)
>>> grp.join(b, c)
>>> grp.join(d, e)
>>> list(grp)
[[a, b, c], [d, e]]
>>> grp.joined(a, b)
True
>>> grp.joined(a, c)
True
>>> grp.joined(a, d)
False
```

**clean**()

[*Deprecated*] Clean dead weak references from the dictionary.

#### Notes

Deprecated since version 3.8: Use none, you no longer need to clean a Grouper instead.

**get_siblings**(*a*)

Return all of the items joined with *a*, including itself.

**join**(*a*, **args*)

Join given arguments into the same set. Accepts one or more arguments.

**joined**(*a*, *b*)

Return whether *a* and *b* are members of the same set.

**remove**(*a*)

Remove *a* from the grouper, doing nothing if it is not there.

**class** matplotlib.cbook.**GrouperView**(*grouper*)

Bases: object

Immutable view over a *Grouper*.

**get_siblings**(*a*)

**joined**(*a*, *b*)

**class** matplotlib.cbook.**Stack**(*default=None*)

Bases: object

[*Deprecated*] Stack of elements with a movable cursor.

Mimics home/back/forward in a web browser.

### Notes

Deprecated since version 3.8.

**back**()

Move the position back and return the current element.

**bubble**(*o*)

Raise all references of *o* to the top of the stack, and return it.

**Raises**

**ValueError**

If *o* is not in the stack.

**clear**()

Empty the stack.

**empty**()

Return whether the stack is empty.

**forward**()

Move the position forward and return the current element.

**home**()

Push the first element onto the top of the stack.

The first element is returned.

**push**(*o*)

Push *o* to the stack at current position. Discard all later elements.

*o* is returned.

**remove**(*o*)

Remove *o* from the stack.

**Raises**

**ValueError**

If *o* is not in the stack.

matplotlib.cbook.**boxplot_stats**(*X*, *whis=1.5*, *bootstrap=None*, *labels=None*, *autorange=False*)

Return a list of dictionaries of statistics used to draw a series of box and whisker plots using *bxp*.

**Parameters**

**X**

[array-like] Data that will be represented in the boxplots. Should have 2 or fewer dimensions.

**whis**

[float or (float, float), default: 1.5] The position of the whiskers.

If a float, the lower whisker is at the lowest datum above `Q1 - whis*(Q3-Q1)`, and the upper whisker at the highest datum below `Q3 + whis*(Q3-Q1)`, where Q1 and Q3 are the first and third quartiles. The default value of `whis = 1.5` corresponds to Tukey's original definition of boxplots.

If a pair of floats, they indicate the percentiles at which to draw the whiskers (e.g., (5, 95)). In particular, setting this to (0, 100) results in whiskers covering the whole range of the data.

In the edge case where `Q1 == Q3`, *whis* is automatically set to (0, 100) (cover the whole range of the data) if *autorange* is True.

Beyond the whiskers, data are considered outliers and are plotted as individual points.

**bootstrap**

[int, optional] Number of times the confidence intervals around the median should be bootstrapped (percentile method).

**labels**

[list of str, optional] Labels for each dataset. Length must be compatible with dimensions of *X*.

**autorange**

[bool, optional (False)] When `True` and the data are distributed such that the 25th and 75th percentiles are equal, `whis` is set to (0, 100) such that the whisker ends are at the minimum and maximum of the data.

**Returns**

**list of dict**

A list of dictionaries containing the results for each column of data. Keys of each dictionary are the following:

| Key | Value Description |
|---|---|
| label | tick label for the boxplot |
| mean | arithmetic mean value |
| med | 50th percentile |
| q1 | first quartile (25th percentile) |
| q3 | third quartile (75th percentile) |
| iqr | interquartile range |
| cilo | lower notch around the median |
| cihi | upper notch around the median |
| whislo | end of the lower whisker |
| whishi | end of the upper whisker |
| fliers | outliers |

### Notes

Non-bootstrapping approach to confidence interval uses Gaussian-based asymptotic approximation:

$$\mathrm{med} \pm 1.57 \times \frac{\mathrm{iqr}}{\sqrt{N}}$$

General approach from: McGill, R., Tukey, J.W., and Larsen, W.A. (1978) "Variations of Boxplots", The American Statistician, 32:12-16.

matplotlib.cbook.**contiguous_regions**(*mask*)

Return a list of (ind0, ind1) such that `mask[ind0:ind1].all()` is True and we cover all such regions.

matplotlib.cbook.**delete_masked_points**(**args*)

Find all masked and/or non-finite points in a set of arguments, and return the arguments with only the unmasked points remaining.

Arguments can be in any of 5 categories:

1) 1-D masked arrays
2) 1-D ndarrays
3) ndarrays with more than one dimension
4) other non-string iterables
5) anything else

The first argument must be in one of the first four categories; any argument with a length differing from that of the first argument (and hence anything in category 5) then will be passed through unchanged.

Masks are obtained from all arguments of the correct length in categories 1, 2, and 4; a point is bad if masked in a masked array or if it is a nan or inf. No attempt is made to extract a mask from categories 2, 3, and 4 if `numpy.isfinite` does not yield a Boolean array.

All input arguments that are not passed unchanged are returned as ndarrays after removing the points or rows corresponding to masks in any of the arguments.

A vastly simpler version of this function was originally written as a helper for Axes.scatter().

matplotlib.cbook.**file_requires_unicode**(*x*)

Return whether the given writable file-like object requires Unicode to be written to it.

matplotlib.cbook.**flatten**(*seq*, *scalarp=<function is_scalar_or_string>*)

Return a generator of flattened nested containers.

For example:

```
>>> from matplotlib.cbook import flatten
>>> l = (('John', ['Hunter']), (1, 23), [[([42, (5, 23)], )]])
>>> print(list(flatten(l)))
['John', 'Hunter', 1, 23, 42, 5, 23]
```

By: Composite of Holger Krekel and Luther Blissett From: https://code.activestate.com/recipes/121294/ and Recipe 1.12 in cookbook

matplotlib.cbook.**get_sample_data**(*fname*, *asfileobj=True*, ***, *np_load=<deprecated parameter>*)

Return a sample data file. *fname* is a path relative to the `mpl-data/sample_data` directory. If *asfileobj* is `True` return a file object, otherwise just a file path.

Sample data files are stored in the 'mpl-data/sample_data' directory within the Matplotlib package.

If the filename ends in .gz, the file is implicitly ungzipped. If the filename ends with .npy or .npz, and *asfileobj* is `True`, the file is loaded with `numpy.load`.

matplotlib.cbook.**index_of**(*y*)

A helper function to create reasonable x values for the given *y*.

This is used for plotting (x, y) if x values are not explicitly given.

First try `y.index` (assuming *y* is a `pandas.Series`), if that fails, use `range(len(y))`.

This will be extended in the future to deal with more types of labeled data.

**Parameters**

**y**

[float or array-like]

**Returns**

**x, y**

[ndarray] The x and y values to plot.

matplotlib.cbook.**is_math_text**(*s*)

Return whether the string *s* contains math expressions.

This is done by checking whether *s* contains an even number of non-escaped dollar signs.

matplotlib.cbook.**is_scalar_or_string**(*val*)

Return whether the given object is a scalar or string like.

matplotlib.cbook.**is_writable_file_like**(*obj*)

Return whether *obj* looks like a file object with a *write* method.

matplotlib.cbook.**ls_mapper = {'-': 'solid', '--': 'dashed', '-.': 'dashdot', ':': 'dotted'}**

Maps short codes for line style to their full name used by backends.

matplotlib.cbook.**ls_mapper_r = {'dashdot': '-.', 'dashed': '--', 'dotted': ':', 'solid': '-'}**

Maps full names for line styles used by backends to their short codes.

matplotlib.cbook.**normalize_kwargs**(*kw*, *alias_mapping=None*)

Helper function to normalize kwarg inputs.

**Parameters**

**kw**

[dict or None] A dict of keyword arguments. None is explicitly supported and treated as an empty dict, to support functions with an optional parameter of the form `props=None`.

**alias_mapping**

[dict or Artist subclass or Artist instance, optional] A mapping between a canonical name to a list of aliases, in order of precedence from lowest to highest.

If the canonical value is not in the list it is assumed to have the highest priority.

If an Artist subclass or instance is passed, use its properties alias mapping.

**Raises**

**TypeError**

To match what Python raises if invalid arguments/keyword arguments are passed to a callable.

matplotlib.cbook.**open_file_cm**(*path_or_file*, *mode='r'*, *encoding=None*)

Pass through file objects and context-manage path-likes.

matplotlib.cbook.**print_cycles**(*objects*, *outstream=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*, *show_progress=False*)

Print loops of cyclic references in the given *objects*.

It is often useful to pass in `gc.garbage` to find the cycles that are preventing some objects from being garbage collected.

**Parameters**

**objects**

A list of objects to find cycles in.

**outstream**

The stream for output.

**show_progress**

[bool] If True, print the number of objects reached as they are found.

`matplotlib.cbook.`**`pts_to_midstep`**(*x*, **args*)

Convert continuous line to mid-steps.

Given a set of `N` points convert to `2N` points which when connected linearly give a step function which changes values at the middle of the intervals.

**Parameters**

**x**

[array] The x location of the steps. May be empty.

**y1, ..., yp**

[array] y arrays to be turned into steps; all must be the same length as `x`.

**Returns**

**array**

The x and y values converted to steps in the same order as the input; can be unpacked as `x_out, y1_out, ..., yp_out`. If the input is length `N`, each of these arrays will be length `2N`.

### Examples

```
>>> x_s, y1_s, y2_s = pts_to_midstep(x, y1, y2)
```

`matplotlib.cbook.`**`pts_to_poststep`**(*x*, **args*)

Convert continuous line to post-steps.

Given a set of `N` points convert to `2N + 1` points, which when connected linearly give a step function which changes values at the end of the intervals.

**Parameters**

**x**

[array] The x location of the steps. May be empty.

**y1, ..., yp**

[array] y arrays to be turned into steps; all must be the same length as `x`.

**Returns**

**array**

The x and y values converted to steps in the same order as the input; can be unpacked as `x_out, y1_out, ..., yp_out`. If the input is length `N`, each of these arrays will be length `2N + 1`. For `N=0`, the length will be 0.

### Examples

```
>>> x_s, y1_s, y2_s = pts_to_poststep(x, y1, y2)
```

`matplotlib.cbook.`**`pts_to_prestep`**(*x*, **args*)

Convert continuous line to pre-steps.

Given a set of `N` points, convert to `2N - 1` points, which when connected linearly give a step function which changes values at the beginning of the intervals.

**Parameters**

**x**

[array] The x location of the steps. May be empty.

**y1, ..., yp**

[array] y arrays to be turned into steps; all must be the same length as `x`.

**Returns**

**array**

The x and y values converted to steps in the same order as the input; can be unpacked as `x_out, y1_out, ..., yp_out`. If the input is length `N`, each of these arrays will be length `2N + 1`. For `N=0`, the length will be 0.

### Examples

```
>>> x_s, y1_s, y2_s = pts_to_prestep(x, y1, y2)
```

matplotlib.cbook.**safe_first_element**(*obj*)

Return the first element in *obj*.

This is a type-independent way of obtaining the first element, supporting both index access and the iterator protocol.

matplotlib.cbook.**safe_masked_invalid**(*x*, *copy=False*)

matplotlib.cbook.**sanitize_sequence**(*data*)

Convert dictview objects to list. Other inputs are returned unchanged.

**class** matplotlib.cbook.**silent_list**(*type*, *seq=None*)

Bases: `list`

A list with a short `repr()`.

This is meant to be used for a homogeneous list of artists, so that they don't cause long, meaningless output.

Instead of

```
[<matplotlib.lines.Line2D object at 0x7f5749fed3c8>,
 <matplotlib.lines.Line2D object at 0x7f5749fed4e0>,
 <matplotlib.lines.Line2D object at 0x7f5758016550>]
```

one will get

```
<a list of 3 Line2D objects>
```

If `self.type` is None, the type name is obtained from the first item in the list (if any).

matplotlib.cbook.**simple_linear_interpolation**(*a*, *steps*)

Resample an array with `steps - 1` points between original point pairs.

Along each column of *a*, `(steps - 1)` points are introduced between each original values; the values are linearly interpolated.

**Parameters**

**a**

[array, shape (n, ...)]

**steps**

[int]

**Returns**

**array**

shape `((n - 1) * steps + 1, ...)`

`matplotlib.cbook.`**`strip_math`**(*s*)

Remove latex formatting from mathtext.

Only handles fully math and fully non-math strings.

`matplotlib.cbook.`**`to_filehandle`**(*fname*, *flag='r'*, *return_opened=False*, *encoding=None*)

Convert a path to an open file handle or pass-through a file-like object.

Consider using *open_file_cm* instead, as it allows one to properly close newly created file objects more easily.

**Parameters**

**fname**

[str or path-like or file-like] If `str` or `os.PathLike`, the file is opened using the flags specified by *flag* and *encoding*. If a file-like object, it is passed through.

**flag**

[str, default: 'r'] Passed as the *mode* argument to `open` when *fname* is `str` or `os.PathLike`; ignored if *fname* is file-like.

**return_opened**

[bool, default: False] If True, return both the file object and a boolean indicating whether this was a new file (that the caller needs to close). If False, return only the new file.

**encoding**

[str or None, default: None] Passed as the *mode* argument to `open` when *fname* is `str` or `os.PathLike`; ignored if *fname* is file-like.

**Returns**

**fh**

[file-like]

**opened**

[bool] *opened* is only returned if *return_opened* is True.

`matplotlib.cbook.`**`violin_stats`**(*X*, *method*, *points=100*, *quantiles=None*)

Return a list of dictionaries of data which can be used to draw a series of violin plots.

See the `Returns` section below to view the required keys of the dictionary.

Users can skip this function and pass a user-defined set of dictionaries with the same keys to *violinplot* instead of using Matplotlib to do the calculations. See the *Returns* section below for the keys that must be present in the dictionaries.

**Parameters**

**X**

[array-like] Sample data that will be used to produce the gaussian kernel density estimates. Must have 2 or fewer dimensions.

**method**

[callable] The method used to calculate the kernel density estimate for each column of data. When called via `method(v, coords)`, it should return a vector of the values of the KDE evaluated at the values specified in coords.

**points**

[int, default: 100] Defines the number of points to evaluate each of the gaussian kernel density estimates at.

**quantiles**

[array-like, default: None] Defines (if not None) a list of floats in interval [0, 1] for each column of data, which represents the quantiles that will be rendered for that column of data. Must have 2 or fewer dimensions. 1D array will be treated as a singleton list containing them.

**Returns**

**list of dict**

A list of dictionaries containing the results for each column of data. The dictionaries contain at least the following:

- coords: A list of scalars containing the coordinates this particular kernel density estimate was evaluated at.
- vals: A list of scalars containing the values of the kernel density estimate at each of the coordinates given in *coords*.
- mean: The mean value for this column of data.
- median: The median value for this column of data.
- min: The minimum value for this column of data.
- max: The maximum value for this column of data.
- quantiles: The quantile values for this column of data.

### 7.2.14 matplotlib.cm

Builtin colormaps, colormap handling utilities, and the *ScalarMappable* mixin.

**See also:**

*Colormap reference* for a list of builtin colormaps.

*Creating Colormaps in Matplotlib* for examples of how to make colormaps.

*Choosing Colormaps in Matplotlib* an in-depth discussion of choosing colormaps.

*Colormap normalization* for more details about data normalization.

**class** matplotlib.cm.**ColormapRegistry**(*cmaps*)

Bases: Mapping

Container for colormaps that are known to Matplotlib by name.

The universal registry instance is *matplotlib.colormaps*. There should be no need for users to instantiate *ColormapRegistry* themselves.

Read access uses a dict-like interface mapping names to *Colormap*s:

```
import matplotlib as mpl
cmap = mpl.colormaps['viridis']
```

Returned *Colormap*s are copies, so that their modification does not change the global definition of the colormap.

Additional colormaps can be added via *ColormapRegistry.register*:

```
mpl.colormaps.register(my_colormap)
```

To get a list of all registered colormaps, you can do:

```
from matplotlib import colormaps
list(colormaps)
```

**get_cmap**(*cmap*)

Return a color map specified through *cmap*.

**Parameters**

**cmap**

[str or *Colormap* or None]

- if a *Colormap*, return it
- if a string, look it up in `mpl.colormaps`
- if None, return the Colormap defined in *rcParams["image.cmap"]* (default: `'viridis'`)

**Returns**

**Colormap**

**register**(*cmap*, *, *name=None*, *force=False*)

Register a new colormap.

The colormap name can then be used as a string argument to any `cmap` parameter in Matplotlib. It is also available in `pyplot.get_cmap`.

The colormap registry stores a copy of the given colormap, so that future changes to the original colormap instance do not affect the registered colormap. Think of this as the registry taking a snapshot of the colormap at registration.

**Parameters**

**cmap**

[matplotlib.colors.Colormap] The colormap to register.

**name**

[str, optional] The name for the colormap. If not given, `cmap.name` is used.

**force**

[bool, default: False] If False, a ValueError is raised if trying to overwrite an already registered name. True supports overwriting registered colormaps other than the builtin colormaps.

**unregister**(*name*)

Remove a colormap from the registry.

You cannot remove built-in colormaps.

If the named colormap is not registered, returns with no error, raises if you try to de-register a default colormap.

**Warning:** Colormap names are currently a shared namespace that may be used by multiple packages. Use *unregister* only if you know you have registered that name before. In particular, do not unregister just in case to clean the name before registering a new colormap.

**Parameters**

**name**

[str] The name of the colormap to be removed.

**Raises**

**ValueError**

If you try to remove a default built-in colormap.

**class** matplotlib.cm.**ScalarMappable**(*norm=None*, *cmap=None*)

Bases: object

A mixin class to map scalar data to RGBA.

The ScalarMappable applies data normalization before returning RGBA colors from the given colormap.

**Parameters**

**norm**

[*Normalize* (or subclass thereof) or str or None] The normalizing object which scales data, typically into the interval `[0, 1]`. If a `str`, a *Normalize* subclass is dynamically generated based on the scale with the corresponding name. If *None*, *norm* defaults to a *colors.Normalize* object which initializes its scaling based on the first data processed.

**cmap**

[str or *Colormap*] The colormap used to map normalized data values to RGBA colors.

**autoscale**()

Autoscale the scalar limits on the norm instance using the current array

**autoscale_None**()

Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

**changed**()

Call this whenever the mappable is changed to notify all the callbackSM listeners to the 'changed' signal.

**colorbar**

The last colorbar associated with this ScalarMappable. May be None.

**get_alpha**()

**Returns**

**float**

Always returns 1.

**get_array**()

Return the array of values, that are mapped to colors.

The base class *ScalarMappable* does not make any assumptions on the dimensionality and shape of the array.

**get_clim**()

Return the values (min, max) that are mapped to the colormap limits.

**get_cmap**()

Return the *Colormap* instance.

**property norm**

**set_array**(*A*)

Set the value array from array-like *A*.

**Parameters**

**A**

[array-like or None] The values that are mapped to colors.

The base class *ScalarMappable* does not make any assumptions on the dimensionality and shape of the value array *A*.

**set_clim**(*vmin=None*, *vmax=None*)

Set the norm limits for image scaling.

**Parameters**

**vmin, vmax**

[float] The limits.

The limits may also be passed as a tuple (*vmin*, *vmax*) as a single positional argument.

**set_cmap**(*cmap*)

Set the colormap for luminance data.

**Parameters**

**cmap**

[*Colormap* or str or None]

**set_norm**(*norm*)

Set the normalization instance.

**Parameters**

**norm**

[*Normalize* or str or None]

### Notes

If there are any colorbars using the mappable for this norm, setting the norm of the mappable will reset the norm, locator, and formatters on the colorbar to default.

**to_rgba**(*x*, *alpha=None*, *bytes=False*, *norm=True*)

Return a normalized RGBA array corresponding to *x*.

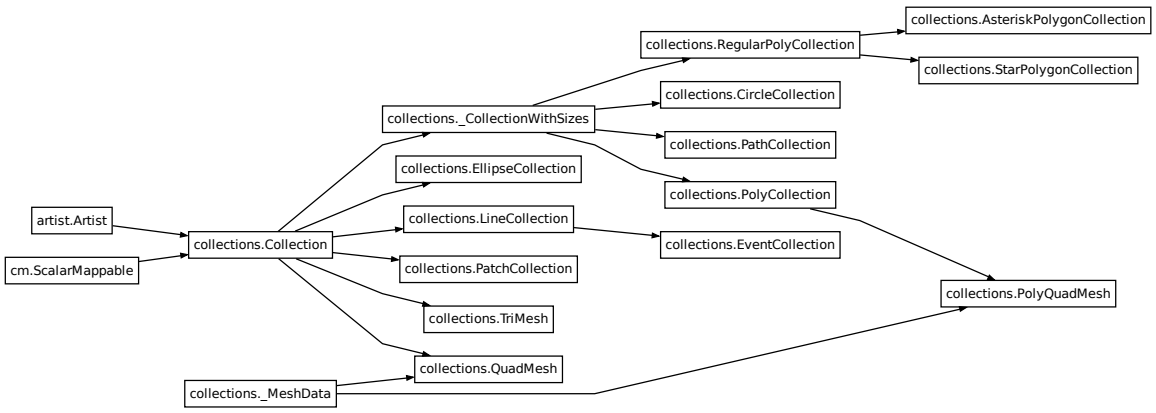
In the normal case, *x* is a 1D or 2D sequence of scalars, and the corresponding `ndarray` of RGBA values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already RGB or RGBA, such as might have been read from an image file. If *x* is an `ndarray` with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an RGB or RGBA array, and no mapping will be done. The array can be `uint8`, or it can be floats with values in the 0-1 range; otherwise a ValueError will be raised. Any NaNs or masked elements will be set to 0 alpha. If the last dimension is 3, the *alpha* kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the *alpha* kwarg is ignored; it does not replace the preexisting alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if *bytes* is *False* (default), the RGBA array will be floats in the 0-1 range; if it is *True*, the returned RGBA array will be `uint8` in the 0 to 255 range.

If norm is False, no normalization of the input data is performed, and it is assumed to be in the range (0-1).

## 7.2.15 matplotlib.collections




Classes for the efficient drawing of large collections of objects that share most properties, e.g., a large number of line segments or polygons.

The classes are not meant to be as flexible as their single element counterparts (e.g., you may not be able to select all line styles) but they are meant to be fast for common use cases (e.g., a large set of solid line segments).

**class** matplotlib.collections.**AsteriskPolygonCollection**(*numsides*, *, *rotation=0*, *sizes=(1,)*, ***kwargs*)

Bases: *RegularPolyCollection*

Draw a collection of regular asterisks with *numsides* points.

**Parameters**

**numsides**

[int] The number of sides of the polygon.

**rotation**

[float] The rotation of the polygon in radians.

**sizes**

[tuple of float] The area of the circle circumscribing the polygon in points^2.

****kwargs**

Forwarded to *Collection*.

## Examples

See *Lasso Demo* for a complete example:

```
offsets = np.random.rand(20, 2)
facecolors = [cm.jet(x) for x in np.random.rand(20)]

collection = RegularPolyCollection(
    numsides=5, # a pentagon
    rotation=0, sizes=(50,),
    facecolors=facecolors,
    edgecolors=("black",),
    linewidths=(1,),
    offsets=offsets,
    offset_transform=ax.transData,
    )
```

**add_callback**(*func*)

Add a callback function that will be called whenever one of the *Artist*'s properties changes.

**Parameters**

**func**

[callable] The callback function. It must have the signature:

```
def func(artist: Artist) -> Any
```

where *artist* is the calling *Artist*. Return values may exist but are ignored.

**Returns**

**int**

The observer id associated with the callback. This id can be used for removing the callback with *remove_callback* later.

**See also:**

***remove_callback***

**autoscale**()

Autoscale the scalar limits on the norm instance using the current array

**autoscale_None**()

Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

**property axes**

The *Axes* instance the artist resides in, or *None*.

**changed**()

Call this whenever the mappable is changed to notify all the callbackSM listeners to the 'changed' signal.

**colorbar**

The last colorbar associated with this ScalarMappable. May be None.

**contains**(*mouseevent*)

Test whether the mouse event occurred in the collection.

Returns `bool, dict(ind=itemlist)`, where every item in itemlist contains the event.

**convert_xunits**(*x*)

Convert *x* using the unit type of the xaxis.

If the artist is not contained in an Axes or if the xaxis does not have units, *x* itself is returned.

**convert_yunits**(*y*)

Convert *y* using the unit type of the yaxis.

If the artist is not contained in an Axes or if the yaxis does not have units, *y* itself is returned.

**draw**(*renderer*)

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

**Parameters**

**renderer**

[*RendererBase* subclass.]

### Notes

This method is overridden in the Artist subclasses.

**findobj**(*match=None*, *include_self=True*)

Find artist objects.

Recursively find all *Artist* instances contained in the artist.

**Parameters**

**match**

A filter criterion for the matches. This can be

- *None*: Return all objects contained in artist.
- A function with signature `def match(artist: Artist) -> bool`. The result will only contain artists for which the function returns *True*.
- A class instance: e.g., *Line2D*. The result will only contain artists of this class or its subclasses (`isinstance` check).

**include_self**

[bool] Include *self* in the list to be checked for a match.

**Returns**

**list of *Artist***

**format_cursor_data**(*data*)

Return a string representation of *data*.

---

**Note:** This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

---

The default implementation converts ints and floats and arrays of ints and floats into a comma-separated string enclosed in square brackets, unless the artist has an associated colorbar, in which case scalar values are formatted using the colorbar's formatter.

**See also:**

***get_cursor_data***

**get_aa**()

Alias for *get_antialiased*.

**get_agg_filter**()

Return filter function to be used for agg filter.

**get_alpha**()

Return the alpha value used for blending - not supported on all backends.

**get_animated**()

Return whether the artist is animated.

**get_antialiased**()

Get the antialiasing state for rendering.

**Returns**

**array of bools**

**get_antialiaseds**()

Alias for *get_antialiased*.

**get_array**()

Return the array of values, that are mapped to colors.

The base class *ScalarMappable* does not make any assumptions on the dimensionality and shape of the array.

**get_capstyle**()

Return the cap style for the collection (for all its elements).

**Returns**

**{'butt', 'projecting', 'round'} or None**

**get_children**()

Return a list of the child *Artist*s of this *Artist*.

**get_clim**()

Return the values (min, max) that are mapped to the colormap limits.

**get_clip_box**()

Return the clipbox.

**get_clip_on**()

Return whether the artist uses clipping.

**get_clip_path**()

Return the clip path.

**get_cmap**()

Return the *Colormap* instance.

**get_cursor_data**(*event*)

Return the cursor data for a given event.

**Note:** This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

Cursor data can be used by Artists to provide additional context information for a given event. The default implementation just returns *None*.

Subclasses can override the method and return arbitrary data. However, when doing so, they must ensure that *format_cursor_data* can convert the data to a string representation.

The only current use case is displaying the z-value of an *AxesImage* in the status bar of a plot window, while moving the mouse.

**Parameters**

**event**

[*MouseEvent*]

**See also:**

***format_cursor_data***

**get_dashes**()
Alias for *get_linestyle*.

**get_datalim**(*transData*)

**get_ec**()
Alias for *get_edgecolor*.

**get_edgecolor**()

**get_edgecolors**()
Alias for *get_edgecolor*.

**get_facecolor**()

**get_facecolors**()
Alias for *get_facecolor*.

**get_fc**()
Alias for *get_facecolor*.

**get_figure**()
Return the *Figure* instance the artist belongs to.

**get_fill**()
Return whether face is colored.

**get_gid**()

Return the group id.

**get_hatch**()

Return the current hatching pattern.

**get_in_layout**()

Return boolean flag, `True` if artist is included in layout calculations.

E.g. *Constrained layout guide*, *`Figure.tight_layout()`*, and `fig.savefig(fname, bbox_inches='tight')`.

**get_joinstyle**()

Return the join style for the collection (for all its elements).

**Returns**

**{'miter', 'round', 'bevel'} or None**

**get_label**()

Return the label used for this artist in the legend.

**get_linestyle**()

**get_linestyles**()

Alias for *`get_linestyle`*.

**get_linewidth**()

**get_linewidths**()

Alias for *`get_linewidth`*.

**get_ls**()

Alias for *`get_linestyle`*.

**get_lw**()

Alias for *`get_linewidth`*.

**get_mouseover**()

Return whether this artist is queried for custom context information when the mouse cursor moves over it.

**get_numsides**()

**get_offset_transform**()

Return the *`Transform`* instance used by this artist offset.

**get_offsets**()

Return the offsets for the collection.

**get_path_effects**()

**get_paths**()

**get_picker**()

Return the picking behavior of the artist.

The possible values are described in *Artist.set_picker*.

**See also:**

***Artist.set_picker***, ***Artist.pickable***, ***Artist.pick***

**get_pickradius**()

**get_rasterized**()

Return whether the artist is to be rasterized.

**get_rotation**()

**get_sizes**()

Return the sizes ('areas') of the elements in the collection.

**Returns**

**array**

The 'area' of each element.

**get_sketch_params**()

Return the sketch parameters for the artist.

**Returns**

**tuple or None**

A 3-tuple with the following elements:

- *scale*: The amplitude of the wiggle perpendicular to the source line.
- *length*: The length of the wiggle along the line.
- *randomness*: The scale factor by which the length is shrunken or expanded.

Returns *None* if no sketch parameters were set.

**get_snap**()

Return the snap setting.

See *set_snap* for details.

**get_tightbbox**(*renderer=None*)

Like *Artist.get_window_extent*, but includes any clipping.

**Parameters**

**renderer**

[*RendererBase* subclass, optional] renderer that will be used to draw the figures (i.e. `fig.canvas.get_renderer()`)

**Returns**

***Bbox* or None**

The enclosing bounding box (in figure pixel coordinates). Returns None if clipping results in no intersection.

**get_transOffset**()

Alias for *get_offset_transform*.

**get_transform**()

Return the *Transform* instance used by this artist.

**get_transformed_clip_path_and_affine**()

Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

**get_transforms**()

**get_url**()

Return the url.

**get_urls**()

Return a list of URLs, one for each element of the collection.

The list contains *None* for elements without a URL. See *Hyperlinks* for an example.

**get_visible**()

Return the visibility.

**get_window_extent**(*renderer=None*)

Get the artist's bounding box in display space.

The bounding box' width and height are nonnegative.

Subclasses should override for inclusion in the bounding box "tight" calculation. Default is to return an empty bounding box at 0, 0.

Be careful when using this function, the results will not update if the artist window extent of the artist changes. The extent can change due to any changes in the transform stack, such as changing the Axes limits, the figure size, or the canvas used (as is done when saving a figure). This can lead to unexpected behavior where interactive figures will look fine on the screen, but will save incorrectly.

**get_zorder**()

Return the artist's zorder.

**have_units**()

Return whether units are set on any axis.

**is_transform_set**()

Return whether the Artist has an explicitly set transform.

This is *True* after *set_transform* has been called.

**property mouseover**

Return whether this artist is queried for custom context information when the mouse cursor moves over it.

**property norm**

**pchanged**()

Call all of the registered callbacks.

This function is triggered internally when a property is changed.

**See also:**

***add_callback***
***remove_callback***

**pick**(*mouseevent*)

Process a pick event.

Each child artist will fire a pick event if *mouseevent* is over the artist and the artist has picker set.

**See also:**

***Artist.set_picker***, ***Artist.get_picker***, ***Artist.pickable***

**pickable**()

Return whether the artist is pickable.

**See also:**

***Artist.set_picker***, ***Artist.get_picker***, ***Artist.pick***

**properties**()

Return a dictionary of all the properties of the artist.

**remove**()

Remove the artist from the figure if possible.

The effect will not be visible until the figure is redrawn, e.g., with *FigureCanvasBase.draw_idle*. Call *relim* to update the Axes limits if desired.

Note: *relim* will not see collections even if the collection was added to the Axes with *autolim* = True.

Note: there is no support for removing the artist's legend entry.

**remove_callback**(*oid*)

Remove a callback based on its observer id.

**See also:**

*add_callback*

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *antialiased=<UNSET>*, *array=<UNSET>*, *capstyle=<UNSET>*, *clim=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *cmap=<UNSET>*, *color=<UNSET>*, *edgecolor=<UNSET>*, *facecolor=<UNSET>*, *gid=<UNSET>*, *hatch=<UNSET>*, *in_layout=<UNSET>*, *joinstyle=<UNSET>*, *label=<UNSET>*, *linestyle=<UNSET>*, *linewidth=<UNSET>*, *mouseover=<UNSET>*, *norm=<UNSET>*, *offset_transform=<UNSET>*, *offsets=<UNSET>*, *path_effects=<UNSET>*, *paths=<UNSET>*, *picker=<UNSET>*, *pickradius=<UNSET>*, *rasterized=<UNSET>*, *sizes=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *urls=<UNSET>*, *visible=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi v |
| *alpha* | array-like or scalar or None |
| *animated* | bool |
| *antialiased* or aa or antialiaseds | bool or list of bools |
| *array* | array-like or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clim* | (vmin: float, vmax: float) |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *cmap* | *Colormap* or str or None |
| *color* | *color* or list of RGBA tuples |
| *edgecolor* or ec or edgecolors | *color* or list of *color* or 'face' |
| *facecolor* or facecolors or fc | *color* or list of *color* |
| *figure* | *Figure* |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or dashes or linestyles or ls | str or tuple or list thereof |
| *linewidth* or linewidths or lw | float or list of floats |
| *mouseover* | bool |
| *norm* | *Normalize* or str or None |

Table 33 – continued from p

| Property | Description |
|---|---|
| *offset_transform* or transOffset | *Transform* |
| *offsets* | (N, 2) or (2,) array-like |
| *path_effects* | list of *AbstractPathEffect* |
| *paths* | unknown |
| *picker* | None or bool or float or callable |
| *pickradius* | float |
| *rasterized* | bool |
| sizes | numpy.ndarray or None |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *urls* | list of str or None |
| *visible* | bool |
| *zorder* | float |

**set_aa**(*aa*)

Alias for *set_antialiased*.

**set_agg_filter**(*filter_func*)

Set the agg filter.

**Parameters**

**filter_func**

[callable] A filter function, which takes a (m, n, depth) float array and a dpi value, and returns a (m, n, depth) array and two offsets from the bottom left corner of the image

**set_alpha**(*alpha*)

Set the alpha value used for blending - not supported on all backends.

**Parameters**

**alpha**

[array-like or scalar or None] All values must be within the 0-1 range, inclusive. Masked values and nans are not supported.

**set_animated**(*b*)

Set whether the artist is intended to be used in an animation.

If True, the artist is excluded from regular drawing of the figure. You have to call *Figure.draw_artist* / *Axes.draw_artist* explicitly on the artist. This approach is used to speed up animations using blitting.

See also *matplotlib.animation* and *Faster rendering by using blitting*.

**Parameters**

**b**

[bool]

**set_antialiased**(*aa*)

Set the antialiasing state for rendering.

**Parameters**

**aa**

[bool or list of bools]

**set_antialiaseds**(*aa*)

Alias for *set_antialiased*.

**set_array**(*A*)

Set the value array from array-like *A*.

**Parameters**

**A**

[array-like or None] The values that are mapped to colors.

The base class *ScalarMappable* does not make any assumptions on the dimensionality and shape of the value array *A*.

**set_capstyle**(*cs*)

Set the *CapStyle* for the collection (for all its elements).

**Parameters**

**cs**

[*CapStyle* or {'butt', 'projecting', 'round'}]

**set_clim**(*vmin=None*, *vmax=None*)

Set the norm limits for image scaling.

**Parameters**

**vmin, vmax**

[float] The limits.

The limits may also be passed as a tuple (*vmin*, *vmax*) as a single positional argument.

**set_clip_box**(*clipbox*)

Set the artist's clip *Bbox*.

**Parameters**

**clipbox**

[*BboxBase* or None] Will typically be created from a *TransformedBbox*. For instance, `TransformedBbox(Bbox([[0, 0], [1, 1]]), ax.transAxes)` is the default clipping for an artist added to an Axes.

**set_clip_on**(*b*)

Set whether the artist uses clipping.

When False, artists will be visible outside the Axes which can lead to unexpected results.

**Parameters**

**b**

[bool]

**set_clip_path**(*path*, *transform=None*)

Set the artist's clip path.

**Parameters**

**path**

[*Patch* or *Path* or *TransformedPath* or None] The clip path. If given a *Path*, *transform* must be provided as well. If *None*, a previously set clip path is removed.

**transform**

[*Transform*, optional] Only used if *path* is a *Path*, in which case the given *Path* is converted to a *TransformedPath* using *transform*.

### Notes

For efficiency, if *path* is a *Rectangle* this method will set the clipping box to the corresponding rectangle and set the clipping path to `None`.

For technical reasons (support of *set*), a tuple (*path*, *transform*) is also accepted as a single positional parameter.

**set_cmap**(*cmap*)

Set the colormap for luminance data.

**Parameters**

**cmap**

[*Colormap* or str or None]

**set_color**(*c*)

Set both the edgecolor and the facecolor.

**Parameters**

**c**

[*color* or list of RGBA tuples]

**See also:**

***Collection.set_facecolor*, *Collection.set_edgecolor***

For setting the edge or face color individually.

**set_dashes**(*ls*)

Alias for *set_linestyle*.

**set_ec**(*c*)

Alias for *set_edgecolor*.

**set_edgecolor**(*c*)

Set the edgecolor(s) of the collection.

**Parameters**

**c**

[*color* or list of *color* or 'face'] The collection edgecolor(s). If a sequence, the patches cycle through it. If 'face', match the facecolor.

**set_edgecolors**(*c*)

Alias for *set_edgecolor*.

**set_facecolor**(*c*)

Set the facecolor(s) of the collection. *c* can be a color (all patches have same color), or a sequence of colors; if it is a sequence the patches will cycle through the sequence.

If *c* is 'none', the patch will not be filled.

**Parameters**

**c**

[*color* or list of *color*]

**set_facecolors**(*c*)

Alias for *set_facecolor*.

**set_fc**(*c*)

Alias for *set_facecolor*.

**set_figure**(*fig*)

Set the *Figure* instance the artist belongs to.

**Parameters**

**fig**

[*Figure*]

**set_gid**(*gid*)

Set the (group) id for the artist.

**Parameters**

**gid**

[str]

**set_hatch**(*hatch*)

Set the hatching pattern

*hatch* can be one of:

```
/   - diagonal hatching
\   - back diagonal
|   - vertical
-   - horizontal
+   - crossed
x   - crossed diagonal
o   - small circle
O   - large circle
.   - dots
*   - stars
```

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

**Parameters**

**hatch**

[{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}]

**set_in_layout**(*in_layout*)

Set if artist is to be included in layout calculations, E.g. *Constrained layout guide*, *Figure.tight_layout()*, and `fig.savefig(fname, bbox_inches='tight')`.

**Parameters**

**in_layout**

[bool]

**set_joinstyle**(*js*)

Set the *JoinStyle* for the collection (for all its elements).

**Parameters**

**js**

[*JoinStyle* or {'miter', 'round', 'bevel'}]

**set_label**(*s*)

Set a label that will be displayed in the legend.

**Parameters**

**s**

[object] *s* will be converted to a string by calling `str`.

**set_linestyle**(*ls*)

Set the linestyle(s) for the collection.

| linestyle | description |
|---|---|
| `'-'` or `'solid'` | solid line |
| `'--'` or `'dashed'` | dashed line |
| `'-.'` or `'dashdot'` | dash-dotted line |
| `':'` or `'dotted'` | dotted line |

Alternatively a dash tuple of the following form can be provided:

```
(offset, onoffseq),
```

where `onoffseq` is an even length tuple of on and off ink in points.

**Parameters**

**ls**

[str or tuple or list thereof] Valid values for individual linestyles include {'-', '--', '-.', ':', '', (offset, on-off-seq)}. See *Line2D.set_linestyle* for a complete description.

**set_linestyles**(*ls*)

Alias for *set_linestyle*.

**set_linewidth**(*lw*)

Set the linewidth(s) for the collection. *lw* can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

**Parameters**

**lw**

[float or list of floats]

**set_linewidths**(*lw*)

Alias for *set_linewidth*.

**set_ls**(*ls*)

Alias for *set_linestyle*.

**set_lw**(*lw*)

Alias for *set_linewidth*.

**set_mouseover**(*mouseover*)

Set whether this artist is queried for custom context information when the mouse cursor moves over it.

**Parameters**

**mouseover**

[bool]

**See also:**

***get_cursor_data***
***ToolCursorPosition***
***NavigationToolbar2***

**set_norm**(*norm*)

Set the normalization instance.

**Parameters**

**norm**

[*Normalize* or str or None]

### Notes

If there are any colorbars using the mappable for this norm, setting the norm of the mappable will reset the norm, locator, and formatters on the colorbar to default.

**set_offset_transform**(*offset_transform*)

Set the artist offset transform.

**Parameters**

**offset_transform**

[*Transform*]

**set_offsets**(*offsets*)

Set the offsets for the collection.

**Parameters**

**offsets**

[(N, 2) or (2,) array-like]

**set_path_effects**(*path_effects*)

Set the path effects.

**Parameters**

**path_effects**

[list of *AbstractPathEffect*]

**set_paths**(*paths*)

**set_picker**(*picker*)

Define the picking behavior of the artist.

**Parameters**

**picker**

[None or bool or float or callable] This can be one of the following:

- *None*: Picking is disabled for this artist (default).
- A boolean: If *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist.
- A float: If picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if its data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event

- A function: If picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

to determine the hit test. if the mouse event is over the artist, return *hit=True* and props is a dictionary of properties you want added to the PickEvent attributes.

**set_pickradius** (*pickradius*)

Set the pick radius used for containment tests.

**Parameters**

**pickradius**

[float] Pick radius, in points.

**set_rasterized** (*rasterized*)

Force rasterized (bitmap) drawing for vector graphics output.

Rasterized drawing is not supported by all artists. If you try to enable this on an artist that does not support it, the command has no effect and a warning will be issued.

This setting is ignored for pixel-based output.

See also *Rasterization for vector graphics*.

**Parameters**

**rasterized**

[bool]

**set_sizes** (*sizes*, *dpi=72.0*)

Set the sizes of each member of the collection.

**Parameters**

**sizes**

[`numpy.ndarray` or None] The size to set for each element of the collection. The value is the 'area' of the element.

**dpi**

[float, default: 72] The dpi of the canvas.

**set_sketch_params** (*scale=None*, *length=None*, *randomness=None*)

Set the sketch parameters.

**Parameters**

**scale**

[float, optional] The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is `None`, or not provided, no sketch filter will be provided.

**length**

[float, optional] The length of the wiggle along the line, in pixels (default 128.0)

**randomness**

[float, optional] The scale factor by which the length is shrunken or expanded (default 16.0)

The PGF backend uses this argument as an RNG seed and not as described above. Using the same seed yields the same random shape.

**set_snap**(*snap*)

Set the snapping behavior.

Snapping aligns positions with the pixel grid, which results in clearer images. For example, if a black line of 1px width was defined at a position in between two pixels, the resulting image would contain the interpolated value of that line in the pixel grid, which would be a grey value on both adjacent pixel positions. In contrast, snapping will move the line to the nearest integer pixel value, so that the resulting image will really contain a 1px wide black line.

Snapping is currently only supported by the Agg and MacOSX backends.

**Parameters**

**snap**

[bool or None] Possible values:

- *True*: Snap vertices to the nearest pixel center.
- *False*: Do not modify vertex positions.
- *None*: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center.

**set_transOffset**(*offset_transform*)

Alias for *set_offset_transform*.

**set_transform**(*t*)

Set the artist transform.

**Parameters**

**t**

[*Transform*]

**set_url**(*url*)

Set the url for the artist.

**Parameters**

**url**

[str]

**set_urls**(*urls*)

**Parameters**

**urls**

[list of str or None]

### Notes

URLs are currently only implemented by the SVG backend. They are ignored by all other backends.

**set_visible**(*b*)

Set the artist's visibility.

**Parameters**

**b**

[bool]

**set_zorder**(*level*)

Set the zorder for the artist. Artists with lower zorder values are drawn first.

**Parameters**

**level**

[float]

**property stale**

Whether the artist is 'stale' and needs to be re-drawn for the output to match the internal state of the artist.

**property sticky_edges**

`x` and `y` sticky edge lists for autoscaling.

When performing autoscaling, if a data limit coincides with a value in the corresponding sticky_edges list, then no margin will be added--the view limit "sticks" to the edge. A typical use case is histograms, where one usually expects no margin on the bottom edge (0) of the histogram.

Moreover, margin expansion "bumps" against sticky edges and cannot cross them. For example, if the upper data limit is 1.0, the upper view limit computed by simple margin application is 1.2, but there is a sticky edge at 1.1, then the actual upper view limit will be 1.1.

This attribute cannot be assigned to; however, the `x` and `y` lists can be modified in place as needed.

### Examples

```
>>> artist.sticky_edges.x[:] = (xmin, xmax)
>>> artist.sticky_edges.y[:] = (ymin, ymax)
```

**to_rgba**(*x*, *alpha=None*, *bytes=False*, *norm=True*)

Return a normalized RGBA array corresponding to *x*.

In the normal case, *x* is a 1D or 2D sequence of scalars, and the corresponding `ndarray` of RGBA values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already RGB or RGBA, such as might have been read from an image file. If *x* is an `ndarray` with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an RGB or RGBA array, and no mapping will be done. The array can be `uint8`, or it can be floats with values in the 0-1 range; otherwise a ValueError will be raised. Any NaNs or masked elements will be set to 0 alpha. If the last dimension is 3, the *alpha* kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the *alpha* kwarg is ignored; it does not replace the preexisting alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if *bytes* is *False* (default), the RGBA array will be floats in the 0-1 range; if it is *True*, the returned RGBA array will be `uint8` in the 0 to 255 range.

If norm is False, no normalization of the input data is performed, and it is assumed to be in the range (0-1).

**update**(*props*)

Update this artist's properties from the dict *props*.

**Parameters**

**props**

[dict]

**update_from**(*other*)

Copy properties from other to self.

**update_scalarmappable**()

Update colors from the scalar mappable array, if any.

Assign colors to edges and faces based on the array and/or colors that were directly set, as appropriate.

**zorder = 0**

**class** matplotlib.collections.**CircleCollection**(*sizes*, ***kwargs*)

Bases: _CollectionWithSizes

A collection of circles, drawn using splines.

**Parameters**

**sizes**

[float or array-like] The area of each circle in points^2.

****kwargs**

Forwarded to *Collection*.

**add_callback**(*func*)

Add a callback function that will be called whenever one of the *Artist*'s properties changes.

**Parameters**

**func**

[callable] The callback function. It must have the signature:

```
def func(artist: Artist) -> Any
```

where *artist* is the calling *Artist*. Return values may exist but are ignored.

**Returns**

**int**

The observer id associated with the callback. This id can be used for removing the callback with *remove_callback* later.

**See also:**

***remove_callback***

**autoscale**()

Autoscale the scalar limits on the norm instance using the current array

**autoscale_None**()

Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

**property axes**

The *Axes* instance the artist resides in, or *None*.

**changed**()

Call this whenever the mappable is changed to notify all the callbackSM listeners to the 'changed' signal.

**colorbar**

The last colorbar associated with this ScalarMappable. May be None.

**contains**(*mouseevent*)

Test whether the mouse event occurred in the collection.

Returns `bool, dict(ind=itemlist)`, where every item in itemlist contains the event.

**convert_xunits**(*x*)

Convert *x* using the unit type of the xaxis.

If the artist is not contained in an Axes or if the xaxis does not have units, *x* itself is returned.

**convert_yunits**(*y*)

Convert *y* using the unit type of the yaxis.

If the artist is not contained in an Axes or if the yaxis does not have units, *y* itself is returned.

**draw**(*renderer*)

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

**Parameters**

**renderer**

[*RendererBase* subclass.]

### Notes

This method is overridden in the Artist subclasses.

**findobj**(*match=None*, *include_self=True*)

Find artist objects.

Recursively find all *Artist* instances contained in the artist.

**Parameters**

**match**

A filter criterion for the matches. This can be

- *None*: Return all objects contained in artist.
- A function with signature `def match(artist: Artist) -> bool`. The result will only contain artists for which the function returns *True*.
- A class instance: e.g., *Line2D*. The result will only contain artists of this class or its subclasses (`isinstance` check).

**include_self**

[bool] Include *self* in the list to be checked for a match.

**Returns**

**list of *Artist***

**format_cursor_data**(*data*)

Return a string representation of *data*.

---

**Note:** This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

---

The default implementation converts ints and floats and arrays of ints and floats into a comma-separated string enclosed in square brackets, unless the artist has an associated colorbar, in which case scalar values are formatted using the colorbar's formatter.

**See also:**

*get_cursor_data*

**get_aa**()

Alias for *get_antialiased*.

**get_agg_filter**()

Return filter function to be used for agg filter.

**get_alpha**()

Return the alpha value used for blending - not supported on all backends.

**get_animated**()

Return whether the artist is animated.

**get_antialiased**()

Get the antialiasing state for rendering.

**Returns**

**array of bools**

**get_antialiaseds**()

Alias for *get_antialiased*.

**get_array**()

Return the array of values, that are mapped to colors.

The base class *ScalarMappable* does not make any assumptions on the dimensionality and shape of the array.

**get_capstyle**()

Return the cap style for the collection (for all its elements).

**Returns**

**{'butt', 'projecting', 'round'} or None**

**get_children**()

Return a list of the child *Artist*s of this *Artist*.

**get_clim**()

Return the values (min, max) that are mapped to the colormap limits.

**get_clip_box**()

Return the clipbox.

**get_clip_on**()

Return whether the artist uses clipping.

**get_clip_path**()

Return the clip path.

**get_cmap**()

Return the *Colormap* instance.

**get_cursor_data**(*event*)

Return the cursor data for a given event.

---

**Note:** This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

---

Cursor data can be used by Artists to provide additional context information for a given event. The default implementation just returns *None*.

Subclasses can override the method and return arbitrary data. However, when doing so, they must ensure that *format_cursor_data* can convert the data to a string representation.

The only current use case is displaying the z-value of an *AxesImage* in the status bar of a plot window, while moving the mouse.

**Parameters**

**event**

[*MouseEvent*]

**See also:**

***format_cursor_data***

**get_dashes**()

Alias for *get_linestyle*.

**get_datalim**(*transData*)

**get_ec**()

Alias for *get_edgecolor*.

**get_edgecolor**()

**get_edgecolors**()

Alias for *get_edgecolor*.

**get_facecolor**()

**get_facecolors**()

Alias for *get_facecolor*.

**get_fc**()

Alias for *get_facecolor*.

**get_figure**()

Return the *Figure* instance the artist belongs to.

**get_fill**()

Return whether face is colored.

**get_gid**()

Return the group id.

**get_hatch**()

Return the current hatching pattern.

**get_in_layout**()

Return boolean flag, `True` if artist is included in layout calculations.

E.g. *Constrained layout guide*, *Figure.tight_layout()*, and `fig.savefig(fname, bbox_inches='tight')`.

**get_joinstyle**()

Return the join style for the collection (for all its elements).

**Returns**

**{'miter', 'round', 'bevel'} or None**

**get_label**()

Return the label used for this artist in the legend.

**get_linestyle**()

**get_linestyles**()

Alias for *get_linestyle*.

**get_linewidth**()

**get_linewidths**()

Alias for *get_linewidth*.

**get_ls**()

Alias for *get_linestyle*.

**get_lw**()

Alias for *get_linewidth*.

**get_mouseover**()

Return whether this artist is queried for custom context information when the mouse cursor moves over it.

**get_offset_transform**()

Return the *Transform* instance used by this artist offset.

**get_offsets**()

Return the offsets for the collection.

**get_path_effects**()

**get_paths**()

**get_picker**()

Return the picking behavior of the artist.

The possible values are described in *Artist.set_picker*.

**See also:**

***Artist.set_picker***, ***Artist.pickable***, ***Artist.pick***

**get_pickradius**()

**get_rasterized**()

Return whether the artist is to be rasterized.

**get_sizes**()

Return the sizes ('areas') of the elements in the collection.

**Returns**

**array**

The 'area' of each element.

**get_sketch_params**()

Return the sketch parameters for the artist.

**Returns**

**tuple or None**

A 3-tuple with the following elements:

- *scale*: The amplitude of the wiggle perpendicular to the source line.
- *length*: The length of the wiggle along the line.
- *randomness*: The scale factor by which the length is shrunken or expanded.

Returns *None* if no sketch parameters were set.

**get_snap**()

Return the snap setting.

See *set_snap* for details.

**get_tightbbox**(*renderer=None*)

Like *Artist.get_window_extent*, but includes any clipping.

**Parameters**

**renderer**

[*RendererBase* subclass, optional] renderer that will be used to draw the figures (i.e. `fig.canvas.get_renderer()`)

**Returns**

***Bbox* or None**

The enclosing bounding box (in figure pixel coordinates). Returns None if clipping results in no intersection.

**get_transOffset**()

Alias for *get_offset_transform*.

**get_transform**()

Return the *Transform* instance used by this artist.

**get_transformed_clip_path_and_affine**()

Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

**get_transforms**()

**get_url**()

Return the url.

**get_urls**()

Return a list of URLs, one for each element of the collection.

The list contains *None* for elements without a URL. See *Hyperlinks* for an example.

**get_visible**()

Return the visibility.

**get_window_extent**(*renderer=None*)

Get the artist's bounding box in display space.

The bounding box' width and height are nonnegative.

Subclasses should override for inclusion in the bounding box "tight" calculation. Default is to return an empty bounding box at 0, 0.

Be careful when using this function, the results will not update if the artist window extent of the artist changes. The extent can change due to any changes in the transform stack, such as changing the Axes limits, the figure size, or the canvas used (as is done when saving a figure). This can lead to unexpected behavior where interactive figures will look fine on the screen, but will save incorrectly.

**get_zorder**()

Return the artist's zorder.

**have_units**()

Return whether units are set on any axis.

**is_transform_set**()

Return whether the Artist has an explicitly set transform.

This is *True* after *set_transform* has been called.

**property mouseover**

Return whether this artist is queried for custom context information when the mouse cursor moves over it.

**property norm**

**pchanged**()

Call all of the registered callbacks.

This function is triggered internally when a property is changed.

**See also:**

***add_callback***
***remove_callback***

**pick**(*mouseevent*)

Process a pick event.

Each child artist will fire a pick event if *mouseevent* is over the artist and the artist has picker set.

**See also:**

***Artist.set_picker***, ***Artist.get_picker***, ***Artist.pickable***

**pickable**()

Return whether the artist is pickable.

**See also:**

*Artist.set_picker*, *Artist.get_picker*, *Artist.pick*

**properties**()

Return a dictionary of all the properties of the artist.

**remove**()

Remove the artist from the figure if possible.

The effect will not be visible until the figure is redrawn, e.g., with *FigureCanvasBase.draw_idle*. Call *relim* to update the Axes limits if desired.

Note: *relim* will not see collections even if the collection was added to the Axes with *autolim* = True.

Note: there is no support for removing the artist's legend entry.

**remove_callback**(*oid*)

Remove a callback based on its observer id.

**See also:**

*add_callback*

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *antialiased=<UNSET>*, *array=<UNSET>*, *capstyle=<UNSET>*, *clim=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *cmap=<UNSET>*, *color=<UNSET>*, *edgecolor=<UNSET>*, *facecolor=<UNSET>*, *gid=<UNSET>*, *hatch=<UNSET>*, *in_layout=<UNSET>*, *joinstyle=<UNSET>*, *label=<UNSET>*, *linestyle=<UNSET>*, *linewidth=<UNSET>*, *mouseover=<UNSET>*, *norm=<UNSET>*, *offset_transform=<UNSET>*, *offsets=<UNSET>*, *path_effects=<UNSET>*, *paths=<UNSET>*, *picker=<UNSET>*, *pickradius=<UNSET>*, *rasterized=<UNSET>*, *sizes=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *urls=<UNSET>*, *visible=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi v |
| *alpha* | array-like or scalar or None |
| *animated* | bool |
| *antialiased* or aa or antialiaseds | bool or list of bools |
| *array* | array-like or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |

Table 34 – continued from p

| Property | Description |
|---|---|
| clim | (vmin: float, vmax: float) |
| clip_box | BboxBase or None |
| clip_on | bool |
| clip_path | Patch or (Path, Transform) or None |
| cmap | Colormap or str or None |
| color | color or list of RGBA tuples |
| edgecolor or ec or edgecolors | color or list of color or 'face' |
| facecolor or facecolors or fc | color or list of color |
| figure | Figure |
| gid | str |
| hatch | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| in_layout | bool |
| joinstyle | JoinStyle or {'miter', 'round', 'bevel'} |
| label | object |
| linestyle or dashes or linestyles or ls | str or tuple or list thereof |
| linewidth or linewidths or lw | float or list of floats |
| mouseover | bool |
| norm | Normalize or str or None |
| offset_transform or transOffset | Transform |
| offsets | (N, 2) or (2,) array-like |
| path_effects | list of AbstractPathEffect |
| paths | unknown |
| picker | None or bool or float or callable |
| pickradius | float |
| rasterized | bool |
| sizes | numpy.ndarray or None |
| sketch_params | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| transform | Transform |
| url | str |
| urls | list of str or None |
| visible | bool |
| zorder | float |

**set_aa**(*aa*)

Alias for set_antialiased.

**set_agg_filter**(*filter_func*)

Set the agg filter.

**Parameters**

**filter_func**

[callable] A filter function, which takes a (m, n, depth) float array and a dpi value, and returns a (m, n, depth) array and two offsets from the bottom left corner of the image

**set_alpha**(*alpha*)

Set the alpha value used for blending - not supported on all backends.

**Parameters**

**alpha**

[array-like or scalar or None] All values must be within the 0-1 range, inclusive. Masked values and nans are not supported.

**set_animated**(*b*)

Set whether the artist is intended to be used in an animation.

If True, the artist is excluded from regular drawing of the figure. You have to call *Figure.draw_artist* / *Axes.draw_artist* explicitly on the artist. This approach is used to speed up animations using blitting.

See also *matplotlib.animation* and *Faster rendering by using blitting*.

**Parameters**

**b**

[bool]

**set_antialiased**(*aa*)

Set the antialiasing state for rendering.

**Parameters**

**aa**

[bool or list of bools]

**set_antialiaseds**(*aa*)

Alias for *set_antialiased*.

**set_array**(*A*)

Set the value array from array-like *A*.

**Parameters**

**A**

[array-like or None] The values that are mapped to colors.

The base class *ScalarMappable* does not make any assumptions on the dimensionality and shape of the value array *A*.

**set_capstyle**(*cs*)

Set the *CapStyle* for the collection (for all its elements).

**Parameters**

**cs**

[*CapStyle* or {'butt', 'projecting', 'round'}]

**set_clim**(*vmin=None*, *vmax=None*)

Set the norm limits for image scaling.

**Parameters**

**vmin, vmax**

[float] The limits.

The limits may also be passed as a tuple (*vmin*, *vmax*) as a single positional argument.

**set_clip_box**(*clipbox*)

Set the artist's clip *Bbox*.

**Parameters**

**clipbox**

[*BboxBase* or None] Will typically be created from a *TransformedBbox*. For instance, `TransformedBbox(Bbox([[0, 0], [1, 1]]), ax.transAxes)` is the default clipping for an artist added to an Axes.

**set_clip_on**(*b*)

Set whether the artist uses clipping.

When False, artists will be visible outside the Axes which can lead to unexpected results.

**Parameters**

**b**

[bool]

**set_clip_path**(*path*, *transform=None*)

Set the artist's clip path.

**Parameters**

**path**

[*Patch* or *Path* or *TransformedPath* or None] The clip path. If given a *Path*, *transform* must be provided as well. If *None*, a previously set clip path is removed.

**transform**

[*Transform*, optional] Only used if *path* is a *Path*, in which case the given *Path* is converted to a *TransformedPath* using *transform*.

### Notes

For efficiency, if *path* is a *Rectangle* this method will set the clipping box to the corresponding rectangle and set the clipping path to `None`.

For technical reasons (support of *set*), a tuple (*path*, *transform*) is also accepted as a single positional parameter.

**set_cmap**(*cmap*)

Set the colormap for luminance data.

**Parameters**

**cmap**

[*Colormap* or str or None]

**set_color**(*c*)

Set both the edgecolor and the facecolor.

**Parameters**

**c**

[*color* or list of RGBA tuples]

**See also:**

***Collection.set_facecolor*, *Collection.set_edgecolor***

For setting the edge or face color individually.

**set_dashes**(*ls*)

Alias for *set_linestyle*.

**set_ec**(*c*)

Alias for *set_edgecolor*.

**set_edgecolor**(*c*)

Set the edgecolor(s) of the collection.

**Parameters**

**c**

[*color* or list of *color* or 'face'] The collection edgecolor(s). If a sequence, the patches cycle through it. If 'face', match the facecolor.

**set_edgecolors**(*c*)

Alias for *set_edgecolor*.

**set_facecolor**(*c*)

Set the facecolor(s) of the collection. *c* can be a color (all patches have same color), or a sequence of colors; if it is a sequence the patches will cycle through the sequence.

If *c* is 'none', the patch will not be filled.

**Parameters**

**c**

[*color* or list of *color*]

**set_facecolors**(*c*)

Alias for *set_facecolor*.

**set_fc**(*c*)

Alias for *set_facecolor*.

**set_figure**(*fig*)

Set the *Figure* instance the artist belongs to.

**Parameters**

**fig**

[*Figure*]

**set_gid**(*gid*)

Set the (group) id for the artist.

**Parameters**

**gid**

[str]

**set_hatch**(*hatch*)

Set the hatching pattern

*hatch* can be one of:

```
/   - diagonal hatching
\   - back diagonal
|   - vertical
-   - horizontal
+   - crossed
x   - crossed diagonal
o   - small circle
O   - large circle
```

(continues on next page)

(continued from previous page)

```
.   - dots
*   - stars
```

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

**Parameters**

**hatch**

[{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}]

**set_in_layout**(*in_layout*)

Set if artist is to be included in layout calculations, E.g. *Constrained layout guide*, *Figure.tight_layout()*, and `fig.savefig(fname, bbox_inches='tight')`.

**Parameters**

**in_layout**

[bool]

**set_joinstyle**(*js*)

Set the *JoinStyle* for the collection (for all its elements).

**Parameters**

**js**

[*JoinStyle* or {'miter', 'round', 'bevel'}]

**set_label**(*s*)

Set a label that will be displayed in the legend.

**Parameters**

**s**

[object] *s* will be converted to a string by calling `str`.

**set_linestyle**(*ls*)

Set the linestyle(s) for the collection.

| linestyle | description |
|---|---|
| `'-'` or `'solid'` | solid line |
| `'--'` or `'dashed'` | dashed line |
| `'-.'` or `'dashdot'` | dash-dotted line |
| `':'` or `'dotted'` | dotted line |

Alternatively a dash tuple of the following form can be provided:

```
(offset, onoffseq),
```

where `onoffseq` is an even length tuple of on and off ink in points.

**Parameters**

**ls**

[str or tuple or list thereof] Valid values for individual linestyles include {'-', '--', '-.', ':', '', (offset, on-off-seq)}. See *Line2D.set_linestyle* for a complete description.

**set_linestyles**(*ls*)

Alias for *set_linestyle*.

**set_linewidth**(*lw*)

Set the linewidth(s) for the collection. *lw* can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

**Parameters**

**lw**

[float or list of floats]

**set_linewidths**(*lw*)

Alias for *set_linewidth*.

**set_ls**(*ls*)

Alias for *set_linestyle*.

**set_lw**(*lw*)

Alias for *set_linewidth*.

**set_mouseover**(*mouseover*)

Set whether this artist is queried for custom context information when the mouse cursor moves over it.

**Parameters**

**mouseover**

[bool]

**See also:**

*get_cursor_data*
*ToolCursorPosition*
*NavigationToolbar2*

**set_norm**(*norm*)

Set the normalization instance.

**Parameters**

**norm**

[*Normalize* or str or None]

### Notes

If there are any colorbars using the mappable for this norm, setting the norm of the mappable will reset the norm, locator, and formatters on the colorbar to default.

**set_offset_transform**(*offset_transform*)

Set the artist offset transform.

**Parameters**

**offset_transform**

[*Transform*]

**set_offsets**(*offsets*)

Set the offsets for the collection.

**Parameters**

**offsets**

[(N, 2) or (2,) array-like]

**set_path_effects**(*path_effects*)

Set the path effects.

**Parameters**

**path_effects**

[list of *AbstractPathEffect*]

**set_paths**(*paths*)

**set_picker**(*picker*)

Define the picking behavior of the artist.

**Parameters**

**picker**

[None or bool or float or callable] This can be one of the following:

- *None*: Picking is disabled for this artist (default).
- A boolean: If *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist.
- A float: If picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if its data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event
- A function: If picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

to determine the hit test. if the mouse event is over the artist, return *hit=True* and props is a dictionary of properties you want added to the PickEvent attributes.

**set_pickradius**(*pickradius*)

Set the pick radius used for containment tests.

**Parameters**

**pickradius**

[float] Pick radius, in points.

**set_rasterized**(*rasterized*)

Force rasterized (bitmap) drawing for vector graphics output.

Rasterized drawing is not supported by all artists. If you try to enable this on an artist that does not support it, the command has no effect and a warning will be issued.

This setting is ignored for pixel-based output.

See also *Rasterization for vector graphics*.

**Parameters**

**rasterized**

[bool]

**set_sizes**(*sizes*, *dpi=72.0*)

Set the sizes of each member of the collection.

**Parameters**

**sizes**

[numpy.ndarray or None] The size to set for each element of the collection. The value is the 'area' of the element.

**dpi**

[float, default: 72] The dpi of the canvas.

**set_sketch_params**(*scale=None*, *length=None*, *randomness=None*)

Set the sketch parameters.

**Parameters**

**scale**

[float, optional] The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is None, or not provided, no sketch filter will be provided.

**length**

[float, optional] The length of the wiggle along the line, in pixels (default 128.0)

**randomness**

[float, optional] The scale factor by which the length is shrunken or expanded (default 16.0)

The PGF backend uses this argument as an RNG seed and not as described above. Using the same seed yields the same random shape.

**set_snap**(*snap*)

Set the snapping behavior.

Snapping aligns positions with the pixel grid, which results in clearer images. For example, if a black line of 1px width was defined at a position in between two pixels, the resulting image would contain the interpolated value of that line in the pixel grid, which would be a grey value on both adjacent pixel positions. In contrast, snapping will move the line to the nearest integer pixel value, so that the resulting image will really contain a 1px wide black line.

Snapping is currently only supported by the Agg and MacOSX backends.

**Parameters**

**snap**

[bool or None] Possible values:

- *True*: Snap vertices to the nearest pixel center.
- *False*: Do not modify vertex positions.
- *None*: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center.

**set_transOffset**(*offset_transform*)

Alias for *set_offset_transform*.

**set_transform**(*t*)

Set the artist transform.

**Parameters**

**t**

[*Transform*]

**set_url**(*url*)

Set the url for the artist.

**Parameters**

**url**

[str]

**set_urls**(*urls*)

**Parameters**

**urls**

[list of str or None]

### Notes

URLs are currently only implemented by the SVG backend. They are ignored by all other backends.

**set_visible**(*b*)

Set the artist's visibility.

**Parameters**

**b**

[bool]

**set_zorder**(*level*)

Set the zorder for the artist. Artists with lower zorder values are drawn first.

**Parameters**

**level**

[float]

**property stale**

Whether the artist is 'stale' and needs to be re-drawn for the output to match the internal state of the artist.

**property sticky_edges**

`x` and `y` sticky edge lists for autoscaling.

When performing autoscaling, if a data limit coincides with a value in the corresponding sticky_edges list, then no margin will be added--the view limit "sticks" to the edge. A typical use case is histograms, where one usually expects no margin on the bottom edge (0) of the histogram.

Moreover, margin expansion "bumps" against sticky edges and cannot cross them. For example, if the upper data limit is 1.0, the upper view limit computed by simple margin application is 1.2, but there is a sticky edge at 1.1, then the actual upper view limit will be 1.1.

This attribute cannot be assigned to; however, the `x` and `y` lists can be modified in place as needed.

### Examples

```
>>> artist.sticky_edges.x[:] = (xmin, xmax)
>>> artist.sticky_edges.y[:] = (ymin, ymax)
```

**to_rgba**(*x*, *alpha=None*, *bytes=False*, *norm=True*)

Return a normalized RGBA array corresponding to *x*.

In the normal case, *x* is a 1D or 2D sequence of scalars, and the corresponding `ndarray` of RGBA values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already RGB or RGBA, such as might have been read from an image file. If *x* is an `ndarray` with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an RGB or RGBA array, and no mapping will be done. The array can be `uint8`, or it can be floats with values in the 0-1 range; otherwise a ValueError will be raised. Any NaNs or masked elements will be set to 0 alpha. If the last dimension is 3, the *alpha* kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the *alpha* kwarg is ignored; it does not replace the preexisting alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if *bytes* is *False* (default), the RGBA array will be floats in the 0-1 range; if it is *True*, the returned RGBA array will be `uint8` in the 0 to 255 range.

If norm is False, no normalization of the input data is performed, and it is assumed to be in the range (0-1).

**update**(*props*)

Update this artist's properties from the dict *props*.

**Parameters**

**props**

[dict]

**update_from**(*other*)

Copy properties from other to self.

**update_scalarmappable**()

Update colors from the scalar mappable array, if any.

Assign colors to edges and faces based on the array and/or colors that were directly set, as appropriate.

**zorder = 0**

**class** matplotlib.collections.**Collection**(*, *edgecolors=None*, *facecolors=None*, *linewidths=None*, *linestyles='solid'*, *capstyle=None*, *joinstyle=None*, *antialiaseds=None*, *offsets=None*, *offset_transform=None*, *norm=None*, *cmap=None*, *pickradius=5.0*, *hatch=None*, *urls=None*, *zorder=1*, ***kwargs*)

Bases: *Artist*, *ScalarMappable*

Base class for Collections. Must be subclassed to be usable.

A Collection represents a sequence of *Patch*es that can be drawn more efficiently together than individually. For example, when a single path is being drawn repeatedly at different offsets, the renderer can typically execute a `draw_marker()` call much more efficiently than a series of repeated calls to `draw_path()` with the offsets put in one-by-one.

Most properties of a collection can be configured per-element. Therefore, Collections have "plural" versions of many of the properties of a *Patch* (e.g. *Collection.get_paths* instead of *Patch.get_path*). Exceptions are the *zorder*, *hatch*, *pickradius*, *capstyle* and *joinstyle* properties, which can only be set globally for the whole collection.

Besides these exceptions, all properties can be specified as single values (applying to all elements) or sequences of values. The property of the `i`th element of the collection is:

```
prop[i % len(prop)]
```

Each Collection can optionally be used as its own *ScalarMappable* by passing the *norm* and *cmap* parameters to its constructor. If the Collection's *ScalarMappable* matrix `_A` has been set (via a call to *Collection.set_array*), then at draw time this internal scalar mappable will be used to set the `facecolors` and `edgecolors`, ignoring those that were manually passed in.

**Parameters**

**edgecolors**

[*color* or list of colors, default: *rcParams["patch.edgecolor"]* (default: `'black'`)] Edge color for each patch making up the collection. The special value 'face' can be passed to make the edgecolor match the facecolor.

**facecolors**

[*color* or list of colors, default: *rcParams["patch.facecolor"]* (default: `'C0'`)] Face color for each patch making up the collection.

**linewidths**

[float or list of floats, default: *rcParams["patch.linewidth"]* (default: `1.0`)] Line width for each patch making up the collection.

**linestyles**

[str or tuple or list thereof, default: 'solid'] Valid strings are ['solid', 'dashed', 'dashdot', 'dotted', '-', '--', '-.', ':']. Dash tuples should be of the form:

```
(offset, onoffseq),
```

where *onoffseq* is an even length tuple of on and off ink lengths in points. For examples, see *Linestyles*.

**capstyle**

[*CapStyle*-like, default: *rcParams["patch.capstyle"]*] Style to use for capping lines for all paths in the collection. Allowed values are {'butt', 'projecting', 'round'}.

**joinstyle**

[*JoinStyle*-like, default: *rcParams["patch.joinstyle"]*] Style to use for joining lines for all paths in the collection. Allowed values are {'miter', 'round', 'bevel'}.

**antialiaseds**

[bool or list of bool, default: *rcParams["patch.antialiased"]* (default: `True`)] Whether each patch in the collection should be drawn with antialiasing.

**offsets**

[(float, float) or list thereof, default: (0, 0)] A vector by which to translate each patch after rendering (default is no translation). The translation is performed in screen (pixel) coordinates (i.e. after the Artist's transform is applied).

**offset_transform**

[*Transform*, default: *IdentityTransform*] A single transform which will be applied to each *offsets* vector before it is used.

**cmap, norm**

Data normalization and colormapping parameters. See *ScalarMappable* for a detailed description.

**hatch**

[str, optional] Hatching pattern to use in filled paths, if any. Valid strings are ['/', '', '|', '-', '+', 'x', 'o', 'O', '.', '*']. See *Hatch style reference* for the meaning of each hatch type.

**pickradius**

[float, default: 5.0] If `pickradius <= 0`, then *Collection.contains* will return `True` whenever the test point is inside of one of the polygons formed by the control points of a Path in the Collection. On the other hand, if it is greater than 0, then we instead check if the test point is contained in a stroke of width `2*pickradius` following any of the Paths in the Collection.

**urls**

[list of str, default: None] A URL for each patch to link to once drawn. Currently only works for the SVG backend. See *Hyperlinks* for examples.

**zorder**

[float, default: 1] The drawing order, shared by all Patches in the Collection. See *Zorder Demo* for all defaults and examples.

****kwargs**

Remaining keyword arguments will be used to set properties as `Collection.set_{key}(val)` for each key-value pair in *kwargs*.

**add_callback**(*func*)

Add a callback function that will be called whenever one of the *Artist*'s properties changes.

**Parameters**

**func**

[callable] The callback function. It must have the signature:

```
def func(artist: Artist) -> Any
```

where *artist* is the calling *Artist*. Return values may exist but are ignored.

**Returns**

**int**

The observer id associated with the callback. This id can be used for removing the callback with *remove_callback* later.

**See also:**

*remove_callback*

**autoscale**()

Autoscale the scalar limits on the norm instance using the current array

**autoscale_None**()

Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

**property axes**

The *Axes* instance the artist resides in, or *None*.

**changed**()

Call this whenever the mappable is changed to notify all the callbackSM listeners to the 'changed' signal.

**colorbar**

The last colorbar associated with this ScalarMappable. May be None.

**contains**(*mouseevent*)

Test whether the mouse event occurred in the collection.

Returns `bool, dict(ind=itemlist)`, where every item in itemlist contains the event.

**convert_xunits**(*x*)

Convert *x* using the unit type of the xaxis.

If the artist is not contained in an Axes or if the xaxis does not have units, *x* itself is returned.

**convert_yunits**(*y*)

Convert *y* using the unit type of the yaxis.

If the artist is not contained in an Axes or if the yaxis does not have units, *y* itself is returned.

**draw**(*renderer*)

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

**Parameters**

**renderer**

[*RendererBase* subclass.]

### Notes

This method is overridden in the Artist subclasses.

**findobj**(*match=None*, *include_self=True*)

Find artist objects.

Recursively find all *Artist* instances contained in the artist.

**Parameters**

**match**

A filter criterion for the matches. This can be

- *None*: Return all objects contained in artist.
- A function with signature `def match(artist: Artist) -> bool`. The result will only contain artists for which the function returns *True*.
- A class instance: e.g., *Line2D*. The result will only contain artists of this class or its subclasses (`isinstance` check).

**include_self**

[bool] Include *self* in the list to be checked for a match.

**Returns**

**list of *Artist***

**format_cursor_data**(*data*)

Return a string representation of *data*.

---

**Note:** This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

---

The default implementation converts ints and floats and arrays of ints and floats into a comma-separated string enclosed in square brackets, unless the artist has an associated colorbar, in which case scalar values are formatted using the colorbar's formatter.

**See also:**

***get_cursor_data***

**get_aa**()

Alias for *get_antialiased*.

**get_agg_filter**()

Return filter function to be used for agg filter.

**get_alpha**()

Return the alpha value used for blending - not supported on all backends.

**get_animated**()

Return whether the artist is animated.

**get_antialiased**()

Get the antialiasing state for rendering.

**Returns**

**array of bools**

**get_antialiaseds**()

Alias for *get_antialiased*.

**get_array**()

Return the array of values, that are mapped to colors.

The base class *ScalarMappable* does not make any assumptions on the dimensionality and shape of the array.

**get_capstyle**()

Return the cap style for the collection (for all its elements).

**Returns**

**{'butt', 'projecting', 'round'} or None**

**get_children**()

Return a list of the child *Artist*s of this *Artist*.

**get_clim**()

Return the values (min, max) that are mapped to the colormap limits.

**get_clip_box**()

Return the clipbox.

**get_clip_on**()

Return whether the artist uses clipping.

**get_clip_path**()

Return the clip path.

**get_cmap**()

Return the *Colormap* instance.

**get_cursor_data**(*event*)

Return the cursor data for a given event.

**Note:** This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

Cursor data can be used by Artists to provide additional context information for a given event. The default implementation just returns *None*.

Subclasses can override the method and return arbitrary data. However, when doing so, they must ensure that *format_cursor_data* can convert the data to a string representation.

The only current use case is displaying the z-value of an *AxesImage* in the status bar of a plot window, while moving the mouse.

**Parameters**

**event**

[*MouseEvent*]

**See also:**

***format_cursor_data***

**get_dashes**()
Alias for *get_linestyle*.

**get_datalim**(*transData*)

**get_ec**()
Alias for *get_edgecolor*.

**get_edgecolor**()

**get_edgecolors**()
Alias for *get_edgecolor*.

**get_facecolor**()

**get_facecolors**()
Alias for *get_facecolor*.

**get_fc**()
Alias for *get_facecolor*.

**get_figure**()
Return the *Figure* instance the artist belongs to.

**get_fill**()
Return whether face is colored.

**get_gid**()

Return the group id.

**get_hatch**()

Return the current hatching pattern.

**get_in_layout**()

Return boolean flag, `True` if artist is included in layout calculations.

E.g. *Constrained layout guide*, *`Figure.tight_layout()`*, and `fig.savefig(fname, bbox_inches='tight')`.

**get_joinstyle**()

Return the join style for the collection (for all its elements).

**Returns**

**{'miter', 'round', 'bevel'} or None**

**get_label**()

Return the label used for this artist in the legend.

**get_linestyle**()

**get_linestyles**()

Alias for *`get_linestyle`*.

**get_linewidth**()

**get_linewidths**()

Alias for *`get_linewidth`*.

**get_ls**()

Alias for *`get_linestyle`*.

**get_lw**()

Alias for *`get_linewidth`*.

**get_mouseover**()

Return whether this artist is queried for custom context information when the mouse cursor moves over it.

**get_offset_transform**()

Return the *`Transform`* instance used by this artist offset.

**get_offsets**()

Return the offsets for the collection.

**get_path_effects**()

**get_paths**()

**get_picker**()

Return the picking behavior of the artist.

The possible values are described in *Artist.set_picker*.

**See also:**

***Artist.set_picker***, ***Artist.pickable***, ***Artist.pick***

**get_pickradius**()

**get_rasterized**()

Return whether the artist is to be rasterized.

**get_sketch_params**()

Return the sketch parameters for the artist.

**Returns**

**tuple or None**

A 3-tuple with the following elements:

- *scale*: The amplitude of the wiggle perpendicular to the source line.
- *length*: The length of the wiggle along the line.
- *randomness*: The scale factor by which the length is shrunken or expanded.

Returns *None* if no sketch parameters were set.

**get_snap**()

Return the snap setting.

See *set_snap* for details.

**get_tightbbox**(*renderer=None*)

Like *Artist.get_window_extent*, but includes any clipping.

**Parameters**

**renderer**

[*RendererBase* subclass, optional] renderer that will be used to draw the figures (i.e. `fig.canvas.get_renderer()`)

**Returns**

***Bbox*** **or None**

The enclosing bounding box (in figure pixel coordinates). Returns None if clipping results in no intersection.

**get_transOffset**()

Alias for *get_offset_transform*.

**get_transform**()

Return the *Transform* instance used by this artist.

**get_transformed_clip_path_and_affine**()

Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

**get_transforms**()

**get_url**()

Return the url.

**get_urls**()

Return a list of URLs, one for each element of the collection.

The list contains *None* for elements without a URL. See *Hyperlinks* for an example.

**get_visible**()

Return the visibility.

**get_window_extent**(*renderer=None*)

Get the artist's bounding box in display space.

The bounding box' width and height are nonnegative.

Subclasses should override for inclusion in the bounding box "tight" calculation. Default is to return an empty bounding box at 0, 0.

Be careful when using this function, the results will not update if the artist window extent of the artist changes. The extent can change due to any changes in the transform stack, such as changing the Axes limits, the figure size, or the canvas used (as is done when saving a figure). This can lead to unexpected behavior where interactive figures will look fine on the screen, but will save incorrectly.

**get_zorder**()

Return the artist's zorder.

**have_units**()

Return whether units are set on any axis.

**is_transform_set**()

Return whether the Artist has an explicitly set transform.

This is *True* after *set_transform* has been called.

**property mouseover**

Return whether this artist is queried for custom context information when the mouse cursor moves over it.

**property norm**

**pchanged**()

Call all of the registered callbacks.

This function is triggered internally when a property is changed.

**See also:**

*add_callback*
*remove_callback*

**pick**(*mouseevent*)

Process a pick event.

Each child artist will fire a pick event if *mouseevent* is over the artist and the artist has picker set.

**See also:**

*Artist.set_picker*, *Artist.get_picker*, *Artist.pickable*

**pickable**()

Return whether the artist is pickable.

**See also:**

*Artist.set_picker*, *Artist.get_picker*, *Artist.pick*

**properties**()

Return a dictionary of all the properties of the artist.

**remove**()

Remove the artist from the figure if possible.

The effect will not be visible until the figure is redrawn, e.g., with *FigureCanvasBase.draw_idle*. Call *relim* to update the Axes limits if desired.

Note: *relim* will not see collections even if the collection was added to the Axes with *autolim* = True.

Note: there is no support for removing the artist's legend entry.

**remove_callback**(*oid*)

Remove a callback based on its observer id.

**See also:**

*add_callback*

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *antialiased=<UNSET>*, *array=<UNSET>*, *capstyle=<UNSET>*, *clim=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *cmap=<UNSET>*, *color=<UNSET>*, *edgecolor=<UNSET>*, *facecolor=<UNSET>*, *gid=<UNSET>*, *hatch=<UNSET>*, *in_layout=<UNSET>*, *joinstyle=<UNSET>*, *label=<UNSET>*, *linestyle=<UNSET>*, *linewidth=<UNSET>*, *mouseover=<UNSET>*, *norm=<UNSET>*, *offset_transform=<UNSET>*, *offsets=<UNSET>*, *path_effects=<UNSET>*, *paths=<UNSET>*, *picker=<UNSET>*, *pickradius=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *urls=<UNSET>*, *visible=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns |
| *alpha* | array-like or scalar or None |
| *animated* | bool |
| *antialiased* | bool or list of bools |
| *array* | array-like or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clim* | (vmin: float, vmax: float) |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *cmap* | *Colormap* or str or None |
| *color* | *color* or list of RGBA tuples |
| *edgecolor* | *color* or list of *color* or 'face' |
| *facecolor* | *color* or list of *color* |
| *figure* | *Figure* |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* | str or tuple or list thereof |
| *linewidth* | float or list of floats |
| *mouseover* | bool |
| *norm* | *Normalize* or str or None |
| *offset_transform* | *Transform* |
| *offsets* | (N, 2) or (2,) array-like |
| *path_effects* | list of *AbstractPathEffect* |
| *paths* | unknown |
| *picker* | None or bool or float or callable |
| *pickradius* | float |
| *rasterized* | bool |

Table 35 – continued from previous

| Property | Description |
|---|---|
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *urls* | list of str or None |
| *visible* | bool |
| *zorder* | float |

**set_aa**(*aa*)

Alias for *set_antialiased*.

**set_agg_filter**(*filter_func*)

Set the agg filter.

**Parameters**

**filter_func**

[callable] A filter function, which takes a (m, n, depth) float array and a dpi value, and returns a (m, n, depth) array and two offsets from the bottom left corner of the image

**set_alpha**(*alpha*)

Set the alpha value used for blending - not supported on all backends.

**Parameters**

**alpha**

[array-like or scalar or None] All values must be within the 0-1 range, inclusive. Masked values and nans are not supported.

**set_animated**(*b*)

Set whether the artist is intended to be used in an animation.

If True, the artist is excluded from regular drawing of the figure. You have to call *Figure.draw_artist* / *Axes.draw_artist* explicitly on the artist. This approach is used to speed up animations using blitting.

See also *matplotlib.animation* and *Faster rendering by using blitting*.

**Parameters**

**b**

[bool]

**set_antialiased**(*aa*)

Set the antialiasing state for rendering.

**Parameters**

**aa**

[bool or list of bools]

**set_antialiaseds**(*aa*)

Alias for *set_antialiased*.

**set_array**(*A*)

Set the value array from array-like *A*.

**Parameters**

**A**

[array-like or None] The values that are mapped to colors.

The base class *ScalarMappable* does not make any assumptions on the dimensionality and shape of the value array *A*.

**set_capstyle**(*cs*)

Set the *CapStyle* for the collection (for all its elements).

**Parameters**

**cs**

[*CapStyle* or {'butt', 'projecting', 'round'}]

**set_clim**(*vmin=None*, *vmax=None*)

Set the norm limits for image scaling.

**Parameters**

**vmin, vmax**

[float] The limits.

The limits may also be passed as a tuple (*vmin*, *vmax*) as a single positional argument.

**set_clip_box**(*clipbox*)

Set the artist's clip *Bbox*.

**Parameters**

**clipbox**

[*BboxBase* or None] Will typically be created from a *TransformedBbox*. For instance, `TransformedBbox(Bbox([[0, 0], [1, 1]]), ax.transAxes)` is the default clipping for an artist added to an Axes.

**set_clip_on**(*b*)

Set whether the artist uses clipping.

When False, artists will be visible outside the Axes which can lead to unexpected results.

**Parameters**

**b**

[bool]

**set_clip_path**(*path*, *transform=None*)

Set the artist's clip path.

**Parameters**

**path**

[*Patch* or *Path* or *TransformedPath* or None] The clip path. If given a *Path*, *transform* must be provided as well. If *None*, a previously set clip path is removed.

**transform**

[*Transform*, optional] Only used if *path* is a *Path*, in which case the given *Path* is converted to a *TransformedPath* using *transform*.

### Notes

For efficiency, if *path* is a *Rectangle* this method will set the clipping box to the corresponding rectangle and set the clipping path to `None`.

For technical reasons (support of *set*), a tuple (*path*, *transform*) is also accepted as a single positional parameter.

**set_cmap**(*cmap*)

Set the colormap for luminance data.

**Parameters**

**cmap**

[*Colormap* or str or None]

**set_color**(*c*)

Set both the edgecolor and the facecolor.

**Parameters**

**c**

[*color* or list of RGBA tuples]

**See also:**

***Collection.set_facecolor*, *Collection.set_edgecolor***

For setting the edge or face color individually.

**set_dashes**(*ls*)

Alias for *set_linestyle*.

**set_ec**(*c*)

Alias for *set_edgecolor*.

**set_edgecolor**(*c*)

Set the edgecolor(s) of the collection.

**Parameters**

**c**

[*color* or list of *color* or 'face'] The collection edgecolor(s). If a sequence, the patches cycle through it. If 'face', match the facecolor.

**set_edgecolors**(*c*)

Alias for *set_edgecolor*.

**set_facecolor**(*c*)

Set the facecolor(s) of the collection. *c* can be a color (all patches have same color), or a sequence of colors; if it is a sequence the patches will cycle through the sequence.

If *c* is 'none', the patch will not be filled.

**Parameters**

**c**

[*color* or list of *color*]

**set_facecolors**(*c*)

Alias for *set_facecolor*.

**set_fc**(*c*)

Alias for *set_facecolor*.

**set_figure**(*fig*)

Set the *Figure* instance the artist belongs to.

**Parameters**

**fig**

[*Figure*]

**set_gid**(*gid*)

Set the (group) id for the artist.

**Parameters**

**gid**

[str]

**set_hatch**(*hatch*)

Set the hatching pattern

*hatch* can be one of:

```
/   - diagonal hatching
\   - back diagonal
|   - vertical
-   - horizontal
+   - crossed
x   - crossed diagonal
o   - small circle
O   - large circle
.   - dots
*   - stars
```

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

**Parameters**

**hatch**

[{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}]

**set_in_layout**(*in_layout*)

Set if artist is to be included in layout calculations, E.g. *Constrained layout guide*, *Figure.tight_layout()*, and `fig.savefig(fname, bbox_inches='tight')`.

**Parameters**

**in_layout**

[bool]

**set_joinstyle**(*js*)

Set the *JoinStyle* for the collection (for all its elements).

**Parameters**

**js**

[*JoinStyle* or {'miter', 'round', 'bevel'}]

**set_label**(*s*)

Set a label that will be displayed in the legend.

**Parameters**

**s**

[object] *s* will be converted to a string by calling `str`.

**set_linestyle**(*ls*)

Set the linestyle(s) for the collection.

| linestyle | description |
|---|---|
| `'-'` or `'solid'` | solid line |
| `'--'` or `'dashed'` | dashed line |
| `'-.'` or `'dashdot'` | dash-dotted line |
| `':'` or `'dotted'` | dotted line |

Alternatively a dash tuple of the following form can be provided:

```
(offset, onoffseq),
```

where `onoffseq` is an even length tuple of on and off ink in points.

**Parameters**

**ls**

[str or tuple or list thereof] Valid values for individual linestyles include {'-', '--', '-.', ':', '', (offset, on-off-seq)}. See *Line2D.set_linestyle* for a complete description.

**set_linestyles**(*ls*)

Alias for *set_linestyle*.

**set_linewidth**(*lw*)

Set the linewidth(s) for the collection. *lw* can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

**Parameters**

**lw**

[float or list of floats]

**set_linewidths**(*lw*)

Alias for *set_linewidth*.

**set_ls**(*ls*)

Alias for *set_linestyle*.

**set_lw**(*lw*)

Alias for *set_linewidth*.

**set_mouseover**(*mouseover*)

Set whether this artist is queried for custom context information when the mouse cursor moves over it.

**Parameters**

**mouseover**

[bool]

**See also:**

***get_cursor_data***
***ToolCursorPosition***
***NavigationToolbar2***

**set_norm**(*norm*)

Set the normalization instance.

**Parameters**

**norm**

[*Normalize* or str or None]

### Notes

If there are any colorbars using the mappable for this norm, setting the norm of the mappable will reset the norm, locator, and formatters on the colorbar to default.

**set_offset_transform**(*offset_transform*)

Set the artist offset transform.

**Parameters**

**offset_transform**

[*Transform*]

**set_offsets**(*offsets*)

Set the offsets for the collection.

**Parameters**

**offsets**

[(N, 2) or (2,) array-like]

**set_path_effects**(*path_effects*)

Set the path effects.

**Parameters**

**path_effects**

[list of *AbstractPathEffect*]

**set_paths**(*paths*)

**set_picker**(*picker*)

Define the picking behavior of the artist.

**Parameters**

**picker**

[None or bool or float or callable] This can be one of the following:

- *None*: Picking is disabled for this artist (default).
- A boolean: If *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist.
- A float: If picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if its data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event

- A function: If picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

to determine the hit test. if the mouse event is over the artist, return *hit=True* and props is a dictionary of properties you want added to the PickEvent attributes.

**set_pickradius**(*pickradius*)

Set the pick radius used for containment tests.

**Parameters**

**pickradius**

[float] Pick radius, in points.

**set_rasterized**(*rasterized*)

Force rasterized (bitmap) drawing for vector graphics output.

Rasterized drawing is not supported by all artists. If you try to enable this on an artist that does not support it, the command has no effect and a warning will be issued.

This setting is ignored for pixel-based output.

See also *Rasterization for vector graphics*.

**Parameters**

**rasterized**

[bool]

**set_sketch_params**(*scale=None*, *length=None*, *randomness=None*)

Set the sketch parameters.

**Parameters**

**scale**

[float, optional] The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is `None`, or not provided, no sketch filter will be provided.

**length**

[float, optional] The length of the wiggle along the line, in pixels (default 128.0)

**randomness**

[float, optional] The scale factor by which the length is shrunken or expanded (default 16.0)

The PGF backend uses this argument as an RNG seed and not as described above. Using the same seed yields the same random shape.

**set_snap**(*snap*)

Set the snapping behavior.

Snapping aligns positions with the pixel grid, which results in clearer images. For example, if a black line of 1px width was defined at a position in between two pixels, the resulting image would contain the interpolated value of that line in the pixel grid, which would be a grey value on both adjacent pixel positions. In contrast, snapping will move the line to the nearest integer pixel value, so that the resulting image will really contain a 1px wide black line.

Snapping is currently only supported by the Agg and MacOSX backends.

**Parameters**

**snap**

[bool or None] Possible values:

- *True*: Snap vertices to the nearest pixel center.
- *False*: Do not modify vertex positions.
- *None*: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center.

**set_transOffset**(*offset_transform*)

Alias for *set_offset_transform*.

**set_transform**(*t*)

Set the artist transform.

**Parameters**

**t**

[*Transform*]

**set_url**(*url*)

Set the url for the artist.

**Parameters**

**url**

[str]

**set_urls**(*urls*)

**Parameters**

**urls**

[list of str or None]

### Notes

URLs are currently only implemented by the SVG backend. They are ignored by all other backends.

**set_visible**(*b*)

Set the artist's visibility.

**Parameters**

**b**

[bool]

**set_zorder**(*level*)

Set the zorder for the artist. Artists with lower zorder values are drawn first.

**Parameters**

**level**

[float]

**property stale**

Whether the artist is 'stale' and needs to be re-drawn for the output to match the internal state of the artist.

**property sticky_edges**

`x` and `y` sticky edge lists for autoscaling.

When performing autoscaling, if a data limit coincides with a value in the corresponding sticky_edges list, then no margin will be added--the view limit "sticks" to the edge. A typical use case is histograms, where one usually expects no margin on the bottom edge (0) of the histogram.

Moreover, margin expansion "bumps" against sticky edges and cannot cross them. For example, if the upper data limit is 1.0, the upper view limit computed by simple margin application is 1.2, but there is a sticky edge at 1.1, then the actual upper view limit will be 1.1.

This attribute cannot be assigned to; however, the `x` and `y` lists can be modified in place as needed.

### Examples

```
>>> artist.sticky_edges.x[:] = (xmin, xmax)
>>> artist.sticky_edges.y[:] = (ymin, ymax)
```

**to_rgba**(*x*, *alpha=None*, *bytes=False*, *norm=True*)

Return a normalized RGBA array corresponding to *x*.

In the normal case, *x* is a 1D or 2D sequence of scalars, and the corresponding `ndarray` of RGBA values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already RGB or RGBA, such as might have been read from an image file. If *x* is an `ndarray` with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an RGB or RGBA array, and no mapping will be done. The array can be `uint8`, or it can be floats with values in the 0-1 range; otherwise a ValueError will be raised. Any NaNs or masked elements will be set to 0 alpha. If the last dimension is 3, the *alpha* kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the *alpha* kwarg is ignored; it does not replace the preexisting alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if *bytes* is *False* (default), the RGBA array will be floats in the 0-1 range; if it is *True*, the returned RGBA array will be `uint8` in the 0 to 255 range.

If norm is False, no normalization of the input data is performed, and it is assumed to be in the range (0-1).

**update**(*props*)

Update this artist's properties from the dict *props*.

**Parameters**

**props**

[dict]

**update_from**(*other*)

Copy properties from other to self.

**update_scalarmappable**()

Update colors from the scalar mappable array, if any.

Assign colors to edges and faces based on the array and/or colors that were directly set, as appropriate.

**zorder = 0**

**class** matplotlib.collections.**EllipseCollection**(*widths*, *heights*, *angles*, ***, *units='points'*, ***kwargs*)

Bases: *Collection*

A collection of ellipses, drawn using splines.

**Parameters**

**widths**

[array-like] The lengths of the first axes (e.g., major axis lengths).

**heights**

[array-like] The lengths of second axes.

**angles**

[array-like] The angles of the first axes, degrees CCW from the x-axis.

**units**

[{'points', 'inches', 'dots', 'width', 'height', 'x', 'y', 'xy'}] The units in which majors and minors are given; 'width' and 'height' refer to the dimensions of the axes, while 'x' and 'y' refer to the *offsets* data units. 'xy' differs from all others in that the angle as plotted varies with the aspect ratio, and equals the specified angle only when the aspect ratio is unity. Hence it behaves the same as the *Ellipse* with `axes.transData` as its transform.

****kwargs**

Forwarded to *Collection*.

**add_callback**(*func*)

Add a callback function that will be called whenever one of the *Artist*'s properties changes.

**Parameters**

**func**

[callable] The callback function. It must have the signature:

```
def func(artist: Artist) -> Any
```

where *artist* is the calling *Artist*. Return values may exist but are ignored.

**Returns**

**int**

The observer id associated with the callback. This id can be used for removing the callback with *remove_callback* later.

**See also:**

***remove_callback***

**autoscale**()

Autoscale the scalar limits on the norm instance using the current array

**autoscale_None**()

Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

**property axes**

The *Axes* instance the artist resides in, or *None*.

**changed**()

Call this whenever the mappable is changed to notify all the callbackSM listeners to the 'changed' signal.

**colorbar**

The last colorbar associated with this ScalarMappable. May be None.

**contains**(*mouseevent*)

Test whether the mouse event occurred in the collection.

Returns `bool, dict(ind=itemlist)`, where every item in itemlist contains the event.

**convert_xunits**(*x*)

Convert *x* using the unit type of the xaxis.

If the artist is not contained in an Axes or if the xaxis does not have units, *x* itself is returned.

**convert_yunits**(*y*)

Convert *y* using the unit type of the yaxis.

If the artist is not contained in an Axes or if the yaxis does not have units, *y* itself is returned.

**draw**(*renderer*)

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

**Parameters**

**renderer**

[*RendererBase* subclass.]

## Notes

This method is overridden in the Artist subclasses.

**findobj**(*match=None*, *include_self=True*)

Find artist objects.

Recursively find all *Artist* instances contained in the artist.

**Parameters**

**match**

A filter criterion for the matches. This can be

- *None*: Return all objects contained in artist.
- A function with signature `def match(artist: Artist) -> bool`. The result will only contain artists for which the function returns *True*.
- A class instance: e.g., *Line2D*. The result will only contain artists of this class or its subclasses (`isinstance` check).

**include_self**

[bool] Include *self* in the list to be checked for a match.

**Returns**

**list of *Artist***

**format_cursor_data**(*data*)

Return a string representation of *data*.

---

**Note:** This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

---

The default implementation converts ints and floats and arrays of ints and floats into a comma-separated string enclosed in square brackets, unless the artist has an associated colorbar, in which case scalar values are formatted using the colorbar's formatter.

**See also:**

***get_cursor_data***

**get_aa**()

Alias for *get_antialiased*.

**get_agg_filter**()

Return filter function to be used for agg filter.

**get_alpha**()

Return the alpha value used for blending - not supported on all backends.

**get_angles**()

Get the angles of the first axes, degrees CCW from the x-axis.

**get_animated**()

Return whether the artist is animated.

**get_antialiased**()

Get the antialiasing state for rendering.

**Returns**

**array of bools**

**get_antialiaseds**()

Alias for *get_antialiased*.

**get_array**()

Return the array of values, that are mapped to colors.

The base class *ScalarMappable* does not make any assumptions on the dimensionality and shape of the array.

**get_capstyle**()

Return the cap style for the collection (for all its elements).

**Returns**

**{'butt', 'projecting', 'round'} or None**

**get_children**()

Return a list of the child *Artist*s of this *Artist*.

**get_clim**()

Return the values (min, max) that are mapped to the colormap limits.

**get_clip_box**()

Return the clipbox.

**get_clip_on**()

Return whether the artist uses clipping.

**get_clip_path**()

Return the clip path.

**get_cmap**()

Return the *Colormap* instance.

**get_cursor_data**(*event*)

Return the cursor data for a given event.

---

**Note:** This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

---

Cursor data can be used by Artists to provide additional context information for a given event. The default implementation just returns *None*.

Subclasses can override the method and return arbitrary data. However, when doing so, they must ensure that *format_cursor_data* can convert the data to a string representation.

The only current use case is displaying the z-value of an *AxesImage* in the status bar of a plot window, while moving the mouse.

**Parameters**

**event**

[*MouseEvent*]

**See also:**

***format_cursor_data***

**get_dashes**()
Alias for *get_linestyle*.

**get_datalim**(*transData*)

**get_ec**()
Alias for *get_edgecolor*.

**get_edgecolor**()

**get_edgecolors**()
Alias for *get_edgecolor*.

**get_facecolor**()

**get_facecolors**()
Alias for *get_facecolor*.

**get_fc**()
Alias for *get_facecolor*.

**get_figure**()
Return the *Figure* instance the artist belongs to.

**get_fill**()
Return whether face is colored.

**get_gid**()
Return the group id.

**get_hatch**()
Return the current hatching pattern.

**get_heights**()
Set the lengths of second axes (e.g., minor axes).

**get_in_layout**()
Return boolean flag, `True` if artist is included in layout calculations.

E.g. *Constrained layout guide*, *`Figure.tight_layout()`*, and `fig.savefig(fname, bbox_inches='tight')`.

**get_joinstyle**()

Return the join style for the collection (for all its elements).

**Returns**

**{'miter', 'round', 'bevel'} or None**

**get_label**()

Return the label used for this artist in the legend.

**get_linestyle**()

**get_linestyles**()

Alias for *get_linestyle*.

**get_linewidth**()

**get_linewidths**()

Alias for *get_linewidth*.

**get_ls**()

Alias for *get_linestyle*.

**get_lw**()

Alias for *get_linewidth*.

**get_mouseover**()

Return whether this artist is queried for custom context information when the mouse cursor moves over it.

**get_offset_transform**()

Return the *Transform* instance used by this artist offset.

**get_offsets**()

Return the offsets for the collection.

**get_path_effects**()

**get_paths**()

**get_picker**()

Return the picking behavior of the artist.

The possible values are described in *Artist.set_picker*.

**See also:**

***Artist.set_picker***, ***Artist.pickable***, ***Artist.pick***

**get_pickradius**()

**get_rasterized**()

Return whether the artist is to be rasterized.

**get_sketch_params**()

Return the sketch parameters for the artist.

**Returns**

**tuple or None**

A 3-tuple with the following elements:

- *scale*: The amplitude of the wiggle perpendicular to the source line.
- *length*: The length of the wiggle along the line.
- *randomness*: The scale factor by which the length is shrunken or expanded.

Returns *None* if no sketch parameters were set.

**get_snap**()

Return the snap setting.

See *set_snap* for details.

**get_tightbbox**(*renderer=None*)

Like *Artist.get_window_extent*, but includes any clipping.

**Parameters**

**renderer**

[*RendererBase* subclass, optional] renderer that will be used to draw the figures (i.e. `fig.canvas.get_renderer()`)

**Returns**

***Bbox* or None**

The enclosing bounding box (in figure pixel coordinates). Returns None if clipping results in no intersection.

**get_transOffset**()

Alias for *get_offset_transform*.

**get_transform**()

Return the *Transform* instance used by this artist.

**get_transformed_clip_path_and_affine**()

Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

**get_transforms**()

**get_url**()

Return the url.

**get_urls**()

Return a list of URLs, one for each element of the collection.

The list contains *None* for elements without a URL. See *Hyperlinks* for an example.

**get_visible**()

Return the visibility.

**get_widths**()

Get the lengths of the first axes (e.g., major axis).

**get_window_extent**(*renderer=None*)

Get the artist's bounding box in display space.

The bounding box' width and height are nonnegative.

Subclasses should override for inclusion in the bounding box "tight" calculation. Default is to return an empty bounding box at 0, 0.

Be careful when using this function, the results will not update if the artist window extent of the artist changes. The extent can change due to any changes in the transform stack, such as changing the Axes limits, the figure size, or the canvas used (as is done when saving a figure). This can lead to unexpected behavior where interactive figures will look fine on the screen, but will save incorrectly.

**get_zorder**()

Return the artist's zorder.

**have_units**()

Return whether units are set on any axis.

**is_transform_set**()

Return whether the Artist has an explicitly set transform.

This is *True* after *set_transform* has been called.

**property mouseover**

Return whether this artist is queried for custom context information when the mouse cursor moves over it.

**property norm**

**pchanged**()

Call all of the registered callbacks.

This function is triggered internally when a property is changed.

**See also:**

*add_callback*
*remove_callback*

**pick**(*mouseevent*)

Process a pick event.

Each child artist will fire a pick event if *mouseevent* is over the artist and the artist has picker set.

**See also:**

***Artist.set_picker*, *Artist.get_picker*, *Artist.pickable***

**pickable**()

Return whether the artist is pickable.

**See also:**

***Artist.set_picker*, *Artist.get_picker*, *Artist.pick***

**properties**()

Return a dictionary of all the properties of the artist.

**remove**()

Remove the artist from the figure if possible.

The effect will not be visible until the figure is redrawn, e.g., with *FigureCanvasBase.draw_idle*. Call *relim* to update the Axes limits if desired.

Note: *relim* will not see collections even if the collection was added to the Axes with *autolim* = True.

Note: there is no support for removing the artist's legend entry.

**remove_callback**(*oid*)

Remove a callback based on its observer id.

**See also:**

***add_callback***

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *angles=<UNSET>*, *animated=<UNSET>*, *antialiased=<UNSET>*, *array=<UNSET>*, *capstyle=<UNSET>*, *clim=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *cmap=<UNSET>*, *color=<UNSET>*, *edgecolor=<UNSET>*, *facecolor=<UNSET>*, *gid=<UNSET>*, *hatch=<UNSET>*, *heights=<UNSET>*, *in_layout=<UNSET>*, *joinstyle=<UNSET>*, *label=<UNSET>*, *linestyle=<UNSET>*, *linewidth=<UNSET>*, *mouseover=<UNSET>*, *norm=<UNSET>*, *offset_transform=<UNSET>*, *offsets=<UNSET>*, *path_effects=<UNSET>*, *paths=<UNSET>*, *picker=<UNSET>*, *pickradius=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *urls=<UNSET>*, *visible=<UNSET>*, *widths=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi v |
| *alpha* | array-like or scalar or None |
| *angles* | unknown |
| *animated* | bool |
| *antialiased* or aa or antialiaseds | bool or list of bools |
| *array* | array-like or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clim* | (vmin: float, vmax: float) |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *cmap* | *Colormap* or str or None |
| *color* | *color* or list of RGBA tuples |
| *edgecolor* or ec or edgecolors | *color* or list of *color* or 'face' |
| *facecolor* or facecolors or fc | *color* or list of *color* |
| *figure* | *Figure* |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *heights* | unknown |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or dashes or linestyles or ls | str or tuple or list thereof |
| *linewidth* or linewidths or lw | float or list of floats |
| *mouseover* | bool |
| *norm* | *Normalize* or str or None |
| *offset_transform* or transOffset | *Transform* |
| *offsets* | (N, 2) or (2,) array-like |
| *path_effects* | list of *AbstractPathEffect* |
| *paths* | unknown |
| *picker* | None or bool or float or callable |
| *pickradius* | float |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *urls* | list of str or None |
| *visible* | bool |
| *widths* | unknown |
| *zorder* | float |

**set_aa**(*aa*)

Alias for *set_antialiased*.

**set_agg_filter**(*filter_func*)

Set the agg filter.

**Parameters**

**filter_func**

[callable] A filter function, which takes a (m, n, depth) float array and a dpi value, and returns a (m, n, depth) array and two offsets from the bottom left corner of the image

**set_alpha**(*alpha*)

Set the alpha value used for blending - not supported on all backends.

**Parameters**

**alpha**

[array-like or scalar or None] All values must be within the 0-1 range, inclusive. Masked values and nans are not supported.

**set_angles**(*angles*)

Set the angles of the first axes, degrees CCW from the x-axis.

**set_animated**(*b*)

Set whether the artist is intended to be used in an animation.

If True, the artist is excluded from regular drawing of the figure. You have to call *Figure.draw_artist* / *Axes.draw_artist* explicitly on the artist. This approach is used to speed up animations using blitting.

See also *matplotlib.animation* and *Faster rendering by using blitting*.

**Parameters**

**b**

[bool]

**set_antialiased**(*aa*)

Set the antialiasing state for rendering.

**Parameters**

**aa**

[bool or list of bools]

**set_antialiaseds**(*aa*)

Alias for *set_antialiased*.

**set_array**(*A*)

Set the value array from array-like *A*.

**Parameters**

**A**

[array-like or None] The values that are mapped to colors.

The base class *ScalarMappable* does not make any assumptions on the dimensionality and shape of the value array *A*.

**set_capstyle**(*cs*)

Set the *CapStyle* for the collection (for all its elements).

**Parameters**

**cs**

[*CapStyle* or {'butt', 'projecting', 'round'}]

**set_clim**(*vmin=None*, *vmax=None*)

Set the norm limits for image scaling.

**Parameters**

**vmin, vmax**

[float] The limits.

The limits may also be passed as a tuple (*vmin*, *vmax*) as a single positional argument.

**set_clip_box**(*clipbox*)

Set the artist's clip *Bbox*.

**Parameters**

**clipbox**

[*BboxBase* or None] Will typically be created from a *TransformedBbox*. For instance, `TransformedBbox(Bbox([[0, 0], [1, 1]]), ax.transAxes)` is the default clipping for an artist added to an Axes.

**set_clip_on**(*b*)

Set whether the artist uses clipping.

When False, artists will be visible outside the Axes which can lead to unexpected results.

**Parameters**

**b**

[bool]

**set_clip_path**(*path*, *transform=None*)

Set the artist's clip path.

**Parameters**

**path**

[*Patch* or *Path* or *TransformedPath* or None] The clip path. If given a *Path*, *transform* must be provided as well. If *None*, a previously set clip path is removed.

**transform**

[*Transform*, optional] Only used if *path* is a *Path*, in which case the given *Path* is converted to a *TransformedPath* using *transform*.

### Notes

For efficiency, if *path* is a *Rectangle* this method will set the clipping box to the corresponding rectangle and set the clipping path to `None`.

For technical reasons (support of *set*), a tuple (*path*, *transform*) is also accepted as a single positional parameter.

**set_cmap**(*cmap*)

Set the colormap for luminance data.

**Parameters**

**cmap**

[*Colormap* or str or None]

**set_color**(*c*)

Set both the edgecolor and the facecolor.

**Parameters**

**c**

[*color* or list of RGBA tuples]

**See also:**

***Collection.set_facecolor*, *Collection.set_edgecolor***

For setting the edge or face color individually.

**set_dashes**(*ls*)

Alias for *set_linestyle*.

**set_ec**(*c*)

Alias for *set_edgecolor*.

**set_edgecolor**(*c*)

Set the edgecolor(s) of the collection.

**Parameters**

**c**

[*color* or list of *color* or 'face'] The collection edgecolor(s). If a sequence, the patches cycle through it. If 'face', match the facecolor.

**set_edgecolors**(*c*)

Alias for *set_edgecolor*.

**set_facecolor**(*c*)

Set the facecolor(s) of the collection. *c* can be a color (all patches have same color), or a sequence of colors; if it is a sequence the patches will cycle through the sequence.

If *c* is 'none', the patch will not be filled.

**Parameters**

**c**

[*color* or list of *color*]

**set_facecolors**(*c*)

Alias for *set_facecolor*.

**set_fc**(*c*)

Alias for *set_facecolor*.

**set_figure**(*fig*)

Set the *Figure* instance the artist belongs to.

**Parameters**

**fig**

[*Figure*]

**set_gid**(*gid*)

Set the (group) id for the artist.

**Parameters**

**gid**

[str]

**set_hatch**(*hatch*)

Set the hatching pattern

*hatch* can be one of:

```
/   - diagonal hatching
\   - back diagonal
|   - vertical
-   - horizontal
+   - crossed
x   - crossed diagonal
o   - small circle
O   - large circle
.   - dots
*   - stars
```

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

**Parameters**

**hatch**

[{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}]

**set_heights**(*heights*)

Set the lengths of second axes (e.g., minor axes).

**set_in_layout**(*in_layout*)

Set if artist is to be included in layout calculations, E.g. *Constrained layout guide*, *Figure.tight_layout()*, and `fig.savefig(fname, bbox_inches='tight')`.

**Parameters**

**in_layout**

[bool]

**set_joinstyle**(*js*)

Set the *JoinStyle* for the collection (for all its elements).

**Parameters**

**js**

[*JoinStyle* or {'miter', 'round', 'bevel'}]

**set_label**(*s*)

Set a label that will be displayed in the legend.

**Parameters**

**s**

[object] *s* will be converted to a string by calling `str`.

**set_linestyle**(*ls*)

Set the linestyle(s) for the collection.

| linestyle | description |
|---|---|
| `'-'` or `'solid'` | solid line |
| `'--'` or `'dashed'` | dashed line |
| `'-.'` or `'dashdot'` | dash-dotted line |
| `':'` or `'dotted'` | dotted line |

Alternatively a dash tuple of the following form can be provided:

```
(offset, onoffseq),
```

where `onoffseq` is an even length tuple of on and off ink in points.

**Parameters**

**ls**

[str or tuple or list thereof] Valid values for individual linestyles include {'-', '--', '-.', ':', '', (offset, on-off-seq)}. See *Line2D.set_linestyle* for a complete description.

**set_linestyles**(*ls*)

Alias for *set_linestyle*.

**set_linewidth**(*lw*)

Set the linewidth(s) for the collection. *lw* can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

**Parameters**

**lw**

[float or list of floats]

**set_linewidths**(*lw*)

Alias for *set_linewidth*.

**set_ls**(*ls*)

Alias for *set_linestyle*.

**set_lw**(*lw*)

Alias for *set_linewidth*.

**set_mouseover**(*mouseover*)

Set whether this artist is queried for custom context information when the mouse cursor moves over it.

**Parameters**

**mouseover**

[bool]

**See also:**

*get_cursor_data*
*ToolCursorPosition*
*NavigationToolbar2*

**set_norm**(*norm*)

Set the normalization instance.

**Parameters**

**norm**

[*Normalize* or str or None]

### Notes

If there are any colorbars using the mappable for this norm, setting the norm of the mappable will reset the norm, locator, and formatters on the colorbar to default.

**set_offset_transform**(*offset_transform*)

Set the artist offset transform.

**Parameters**

**offset_transform**

[*Transform*]

**set_offsets**(*offsets*)

Set the offsets for the collection.

**Parameters**

**offsets**

[(N, 2) or (2,) array-like]

**set_path_effects**(*path_effects*)

Set the path effects.

**Parameters**

**path_effects**

[list of *AbstractPathEffect*]

**set_paths**(*paths*)

**set_picker**(*picker*)

Define the picking behavior of the artist.

**Parameters**

**picker**

[None or bool or float or callable] This can be one of the following:

- *None*: Picking is disabled for this artist (default).
- A boolean: If *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist.
- A float: If picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if its data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event
- A function: If picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

to determine the hit test. if the mouse event is over the artist, return *hit=True* and props is a dictionary of properties you want added to the PickEvent attributes.

**set_pickradius**(*pickradius*)

Set the pick radius used for containment tests.

**Parameters**

**pickradius**

[float] Pick radius, in points.

**set_rasterized**(*rasterized*)

Force rasterized (bitmap) drawing for vector graphics output.

Rasterized drawing is not supported by all artists. If you try to enable this on an artist that does not support it, the command has no effect and a warning will be issued.

This setting is ignored for pixel-based output.

See also *Rasterization for vector graphics*.

**Parameters**

**rasterized**

[bool]

**set_sketch_params**(*scale=None*, *length=None*, *randomness=None*)

Set the sketch parameters.

**Parameters**

**scale**

[float, optional] The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is `None`, or not provided, no sketch filter will be provided.

**length**

[float, optional] The length of the wiggle along the line, in pixels (default 128.0)

**randomness**

[float, optional] The scale factor by which the length is shrunken or expanded (default 16.0)

The PGF backend uses this argument as an RNG seed and not as described above. Using the same seed yields the same random shape.

**set_snap**(*snap*)

Set the snapping behavior.

Snapping aligns positions with the pixel grid, which results in clearer images. For example, if a black line of 1px width was defined at a position in between two pixels, the resulting image would contain the interpolated value of that line in the pixel grid, which would be a grey value on both adjacent pixel positions. In contrast, snapping will move the line to the nearest integer pixel value, so that the resulting image will really contain a 1px wide black line.

Snapping is currently only supported by the Agg and MacOSX backends.

**Parameters**

**snap**

[bool or None] Possible values:

- *True*: Snap vertices to the nearest pixel center.
- *False*: Do not modify vertex positions.
- *None*: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center.

**set_transOffset**(*offset_transform*)

Alias for *set_offset_transform*.

**set_transform**(*t*)

Set the artist transform.

**Parameters**

**t**

[*Transform*]

**set_url**(*url*)

Set the url for the artist.

**Parameters**

**url**

[str]

**set_urls**(*urls*)

**Parameters**

**urls**

[list of str or None]

### Notes

URLs are currently only implemented by the SVG backend. They are ignored by all other backends.

**set_visible**(*b*)

Set the artist's visibility.

**Parameters**

**b**

[bool]

**set_widths**(*widths*)

Set the lengths of the first axes (e.g., major axis).

**set_zorder**(*level*)

Set the zorder for the artist. Artists with lower zorder values are drawn first.

**Parameters**

**level**

[float]

**property stale**

Whether the artist is 'stale' and needs to be re-drawn for the output to match the internal state of the artist.

**property sticky_edges**

`x` and `y` sticky edge lists for autoscaling.

When performing autoscaling, if a data limit coincides with a value in the corresponding sticky_edges list, then no margin will be added--the view limit "sticks" to the edge. A typical use case is histograms, where one usually expects no margin on the bottom edge (0) of the histogram.

Moreover, margin expansion "bumps" against sticky edges and cannot cross them. For example, if the upper data limit is 1.0, the upper view limit computed by simple margin application is 1.2, but there is a sticky edge at 1.1, then the actual upper view limit will be 1.1.

This attribute cannot be assigned to; however, the `x` and `y` lists can be modified in place as needed.

### Examples

```
>>> artist.sticky_edges.x[:] = (xmin, xmax)
>>> artist.sticky_edges.y[:] = (ymin, ymax)
```

**to_rgba**(*x*, *alpha=None*, *bytes=False*, *norm=True*)

Return a normalized RGBA array corresponding to *x*.

In the normal case, *x* is a 1D or 2D sequence of scalars, and the corresponding `ndarray` of RGBA values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already RGB or RGBA, such as might have been read from an image file. If *x* is an `ndarray` with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an RGB or RGBA array, and no mapping will be done. The array can be `uint8`, or it can be floats with values in the 0-1 range; otherwise a ValueError will be raised. Any NaNs or masked elements will be set to 0 alpha. If the last dimension is 3, the *alpha* kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the *alpha* kwarg is ignored; it does not replace the preexisting alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if *bytes* is *False* (default), the RGBA array will be floats in the 0-1 range; if it is *True*, the returned RGBA array will be `uint8` in the 0 to 255 range.

If norm is False, no normalization of the input data is performed, and it is assumed to be in the range (0-1).

**update**(*props*)

Update this artist's properties from the dict *props*.

**Parameters**

**props**

[dict]

**update_from**(*other*)

Copy properties from other to self.

**update_scalarmappable**()

Update colors from the scalar mappable array, if any.

Assign colors to edges and faces based on the array and/or colors that were directly set, as appropriate.

**zorder = 0**

**class** matplotlib.collections.**EventCollection**(*positions*, *orientation='horizontal'*, ***, *lineoffset=0*, *linelength=1*, *linewidth=None*, *color=None*, *linestyle='solid'*, *antialiased=None*, ***kwargs*)

Bases: *LineCollection*

A collection of locations along a single axis at which an "event" occurred.

The events are given by a 1-dimensional array. They do not have an amplitude and are displayed as parallel lines.

**Parameters**

**positions**

[1D array-like] Each value is an event.

**orientation**

[{'horizontal', 'vertical'}, default: 'horizontal'] The sequence of events is plotted along this direction. The marker lines of the single events are along the orthogonal direction.

**lineoffset**

[float, default: 0] The offset of the center of the markers from the origin, in the direction orthogonal to *orientation*.

**linelength**

[float, default: 1] The total height of the marker (i.e. the marker stretches from `lineoffset - linelength/2` to `lineoffset + linelength/2`).

**linewidth**

[float or list thereof, default: *rcParams["lines.linewidth"]* (default: `1.5`)] The line width of the event lines, in points.

**color**

[*color* or list of *color*, default: *rcParams["lines.color"]* (default: `'C0'`)] The color of the event lines.

**linestyle**

[str or tuple or list thereof, default: 'solid'] Valid strings are ['solid', 'dashed', 'dashdot', 'dotted', '-', '--', '-.', ':']. Dash tuples should be of the form:

```
(offset, onoffseq),
```

where *onoffseq* is an even length tuple of on and off ink in points.

**antialiased**

[bool or list thereof, default: *rcParams["lines.antialiased"]* (default: `True`)] Whether to use antialiasing for drawing the lines.

****kwargs**

Forwarded to *LineCollection*.

## Examples

**add_callback**(*func*)

Add a callback function that will be called whenever one of the *Artist*'s properties changes.

**Parameters**

**func**

[callable] The callback function. It must have the signature:

```
def func(artist: Artist) -> Any
```

where *artist* is the calling *Artist*. Return values may exist but are ignored.

**Returns**

**int**

The observer id associated with the callback. This id can be used for removing the callback with *remove_callback* later.

**See also:**

***remove_callback***

**add_positions**(*position*)

Add one or more events at the specified positions.

**append_positions**(*position*)

Add one or more events at the specified positions.

**autoscale**()

Autoscale the scalar limits on the norm instance using the current array

**autoscale_None**()

Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

**property axes**

The *Axes* instance the artist resides in, or *None*.

**changed**()

Call this whenever the mappable is changed to notify all the callbackSM listeners to the 'changed' signal.

**colorbar**

The last colorbar associated with this ScalarMappable. May be None.

**contains**(*mouseevent*)

Test whether the mouse event occurred in the collection.

Returns `bool, dict(ind=itemlist)`, where every item in itemlist contains the event.

**convert_xunits**(*x*)

Convert *x* using the unit type of the xaxis.

If the artist is not contained in an Axes or if the xaxis does not have units, *x* itself is returned.

**convert_yunits**(*y*)

Convert *y* using the unit type of the yaxis.

If the artist is not contained in an Axes or if the yaxis does not have units, *y* itself is returned.

**draw**(*renderer*)

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

**Parameters**

**renderer**

[*RendererBase* subclass.]

### Notes

This method is overridden in the Artist subclasses.

**extend_positions**(*position*)

Add one or more events at the specified positions.

**findobj**(*match=None*, *include_self=True*)

Find artist objects.

Recursively find all *Artist* instances contained in the artist.

**Parameters**

**match**

A filter criterion for the matches. This can be

- *None*: Return all objects contained in artist.
- A function with signature `def match(artist: Artist) -> bool`. The result will only contain artists for which the function returns *True*.
- A class instance: e.g., *Line2D*. The result will only contain artists of this class or its subclasses (`isinstance` check).

**include_self**

[bool] Include *self* in the list to be checked for a match.

**Returns**

**list of *Artist***

**format_cursor_data**(*data*)

Return a string representation of *data*.

---

**Note:** This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

---

The default implementation converts ints and floats and arrays of ints and floats into a comma-separated string enclosed in square brackets, unless the artist has an associated colorbar, in which case scalar values are formatted using the colorbar's formatter.

**See also:**

*get_cursor_data*

**get_aa**()

Alias for *get_antialiased*.

**get_agg_filter**()

Return filter function to be used for agg filter.

**get_alpha**()

Return the alpha value used for blending - not supported on all backends.

**get_animated**()

Return whether the artist is animated.

**get_antialiased**()

Get the antialiasing state for rendering.

**Returns**

**array of bools**

**get_antialiaseds**()

Alias for *get_antialiased*.

**get_array**()

Return the array of values, that are mapped to colors.

The base class *ScalarMappable* does not make any assumptions on the dimensionality and shape of the array.

**get_capstyle**()

Return the cap style for the collection (for all its elements).

**Returns**

**{'butt', 'projecting', 'round'} or None**

**get_children**()

Return a list of the child *Artist*s of this *Artist*.

**get_clim**()

Return the values (min, max) that are mapped to the colormap limits.

**get_clip_box**()

Return the clipbox.

**get_clip_on**()

Return whether the artist uses clipping.

**get_clip_path**()

Return the clip path.

**get_cmap**()

Return the *Colormap* instance.

**get_color**()

Return the color of the lines used to mark each event.

**get_colors**()

**get_cursor_data**(*event*)

Return the cursor data for a given event.

---

**Note:** This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

---

Cursor data can be used by Artists to provide additional context information for a given event. The default implementation just returns *None*.

Subclasses can override the method and return arbitrary data. However, when doing so, they must ensure that *format_cursor_data* can convert the data to a string representation.

The only current use case is displaying the z-value of an *AxesImage* in the status bar of a plot window, while moving the mouse.

**Parameters**

**event**

[*MouseEvent*]

**See also:**

***format_cursor_data***

**get_dashes**()

Alias for *get_linestyle*.

**get_datalim**(*transData*)

**get_ec**()

Alias for *get_edgecolor*.

**get_edgecolor**()

**get_edgecolors**()

Alias for *get_edgecolor*.

**get_facecolor**()

**get_facecolors**()

Alias for *get_facecolor*.

**get_fc**()

Alias for *get_facecolor*.

**get_figure**()

Return the *Figure* instance the artist belongs to.

**get_fill**()

Return whether face is colored.

**get_gapcolor**()

**get_gid**()

Return the group id.

**get_hatch**()

Return the current hatching pattern.

**get_in_layout**()

Return boolean flag, `True` if artist is included in layout calculations.

E.g. *Constrained layout guide*, *Figure.tight_layout()*, and `fig.savefig(fname, bbox_inches='tight')`.

**get_joinstyle**()

Return the join style for the collection (for all its elements).

**Returns**

**{'miter', 'round', 'bevel'} or None**

**get_label**()

Return the label used for this artist in the legend.

**get_linelength**()

Return the length of the lines used to mark each event.

**get_lineoffset**()

Return the offset of the lines used to mark each event.

**get_linestyle**()

**get_linestyles**()

Alias for *get_linestyle*.

**get_linewidth**()

Get the width of the lines used to mark each event.

**get_linewidths**()

Alias for *get_linewidth*.

**get_ls**()

Alias for *get_linestyle*.

**get_lw**()

Alias for *get_linewidth*.

**get_mouseover**()

Return whether this artist is queried for custom context information when the mouse cursor moves over it.

**get_offset_transform**()

Return the *Transform* instance used by this artist offset.

**get_offsets**()

Return the offsets for the collection.

**get_orientation**()

Return the orientation of the event line ('horizontal' or 'vertical').

**get_path_effects**()

**get_paths**()

**get_picker**()

Return the picking behavior of the artist.

The possible values are described in *Artist.set_picker*.

**See also:**

***Artist.set_picker***, ***Artist.pickable***, ***Artist.pick***

**get_pickradius**()

**get_positions**()

Return an array containing the floating-point values of the positions.

**get_rasterized**()

Return whether the artist is to be rasterized.

**get_segments**()

**Returns**

**list**

List of segments in the LineCollection. Each list item contains an array of vertices.

**get_sketch_params**()

Return the sketch parameters for the artist.

**Returns**

**tuple or None**

A 3-tuple with the following elements:

- *scale*: The amplitude of the wiggle perpendicular to the source line.
- *length*: The length of the wiggle along the line.
- *randomness*: The scale factor by which the length is shrunken or expanded.

Returns *None* if no sketch parameters were set.

**get_snap**()

Return the snap setting.

See *set_snap* for details.

**get_tightbbox**(*renderer=None*)

Like *Artist.get_window_extent*, but includes any clipping.

**Parameters**

**renderer**

[*RendererBase* subclass, optional] renderer that will be used to draw the figures (i.e. `fig.canvas.get_renderer()`)

**Returns**

***Bbox* or None**

The enclosing bounding box (in figure pixel coordinates). Returns None if clipping results in no intersection.

**get_transOffset**()

Alias for *get_offset_transform*.

**get_transform**()

Return the *Transform* instance used by this artist.

**get_transformed_clip_path_and_affine**()

Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

**get_transforms**()

**get_url**()

Return the url.

**get_urls**()

Return a list of URLs, one for each element of the collection.

The list contains *None* for elements without a URL. See *Hyperlinks* for an example.

**get_visible**()

Return the visibility.

**get_window_extent**(*renderer=None*)

Get the artist's bounding box in display space.

The bounding box' width and height are nonnegative.

Subclasses should override for inclusion in the bounding box "tight" calculation. Default is to return an empty bounding box at 0, 0.

Be careful when using this function, the results will not update if the artist window extent of the artist changes. The extent can change due to any changes in the transform stack, such as changing the Axes limits, the figure size, or the canvas used (as is done when saving a figure). This can lead to unexpected behavior where interactive figures will look fine on the screen, but will save incorrectly.

**get_zorder**()

Return the artist's zorder.

**have_units**()

Return whether units are set on any axis.

**is_horizontal**()

True if the eventcollection is horizontal, False if vertical.

**is_transform_set**()

Return whether the Artist has an explicitly set transform.

This is *True* after *set_transform* has been called.

**property mouseover**

Return whether this artist is queried for custom context information when the mouse cursor moves over it.

**property norm**

**pchanged**()

Call all of the registered callbacks.

This function is triggered internally when a property is changed.

**See also:**

***add_callback***
***remove_callback***

**pick**(*mouseevent*)

Process a pick event.

Each child artist will fire a pick event if *mouseevent* is over the artist and the artist has picker set.

**See also:**

*Artist.set_picker*, *Artist.get_picker*, *Artist.pickable*

**pickable**()

Return whether the artist is pickable.

**See also:**

*Artist.set_picker*, *Artist.get_picker*, *Artist.pick*

**properties**()

Return a dictionary of all the properties of the artist.

**remove**()

Remove the artist from the figure if possible.

The effect will not be visible until the figure is redrawn, e.g., with *FigureCanvasBase.draw_idle*. Call *relim* to update the Axes limits if desired.

Note: *relim* will not see collections even if the collection was added to the Axes with *autolim* = True.

Note: there is no support for removing the artist's legend entry.

**remove_callback**(*oid*)

Remove a callback based on its observer id.

**See also:**

*add_callback*

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *antialiased=<UNSET>*, *array=<UNSET>*, *capstyle=<UNSET>*, *clim=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *cmap=<UNSET>*, *color=<UNSET>*, *colors=<UNSET>*, *edgecolor=<UNSET>*, *facecolor=<UNSET>*, *gapcolor=<UNSET>*, *gid=<UNSET>*, *hatch=<UNSET>*, *in_layout=<UNSET>*, *joinstyle=<UNSET>*, *label=<UNSET>*, *linelength=<UNSET>*, *lineoffset=<UNSET>*, *linestyle=<UNSET>*, *linewidth=<UNSET>*, *mouseover=<UNSET>*, *norm=<UNSET>*, *offset_transform=<UNSET>*, *offsets=<UNSET>*, *orientation=<UNSET>*, *path_effects=<UNSET>*, *paths=<UNSET>*, *picker=<UNSET>*, *pickradius=<UNSET>*, *positions=<UNSET>*, *rasterized=<UNSET>*, *segments=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *urls=<UNSET>*, *verts=<UNSET>*, *visible=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi v |
| *alpha* | array-like or scalar or None |

Table 37 – continued from p

| Property | Description |
|---|---|
| animated | bool |
| antialiased or aa or antialiaseds | bool or list of bools |
| array | array-like or None |
| capstyle | CapStyle or {'butt', 'projecting', 'round'} |
| clim | (vmin: float, vmax: float) |
| clip_box | BboxBase or None |
| clip_on | bool |
| clip_path | Patch or (Path, Transform) or None |
| cmap | Colormap or str or None |
| color | color or list of color |
| colors | color or list of color |
| edgecolor or ec or edgecolors | color or list of color or 'face' |
| facecolor or facecolors or fc | color or list of color |
| figure | Figure |
| gapcolor | color or list of color or None |
| gid | str |
| hatch | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| in_layout | bool |
| joinstyle | JoinStyle or {'miter', 'round', 'bevel'} |
| label | object |
| linelength | unknown |
| lineoffset | unknown |
| linestyle or dashes or linestyles or ls | str or tuple or list thereof |
| linewidth or linewidths or lw | float or list of floats |
| mouseover | bool |
| norm | Normalize or str or None |
| offset_transform or transOffset | Transform |
| offsets | (N, 2) or (2,) array-like |
| orientation | {'horizontal', 'vertical'} |
| path_effects | list of AbstractPathEffect |
| paths | unknown |
| picker | None or bool or float or callable |
| pickradius | float |
| positions | unknown |
| rasterized | bool |
| segments | unknown |
| sketch_params | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| transform | Transform |
| url | str |
| urls | list of str or None |
| verts | unknown |
| visible | bool |
| zorder | float |

Table 37 – continued from p

| Property | Description |
|---|---|

**set_aa**(*aa*)

Alias for *set_antialiased*.

**set_agg_filter**(*filter_func*)

Set the agg filter.

**Parameters**

**filter_func**

[callable] A filter function, which takes a (m, n, depth) float array and a dpi value, and returns a (m, n, depth) array and two offsets from the bottom left corner of the image

**set_alpha**(*alpha*)

Set the alpha value used for blending - not supported on all backends.

**Parameters**

**alpha**

[array-like or scalar or None] All values must be within the 0-1 range, inclusive. Masked values and nans are not supported.

**set_animated**(*b*)

Set whether the artist is intended to be used in an animation.

If True, the artist is excluded from regular drawing of the figure. You have to call *Figure.draw_artist* / *Axes.draw_artist* explicitly on the artist. This approach is used to speed up animations using blitting.

See also *matplotlib.animation* and *Faster rendering by using blitting*.

**Parameters**

**b**

[bool]

**set_antialiased**(*aa*)

Set the antialiasing state for rendering.

**Parameters**

**aa**

[bool or list of bools]

**set_antialiaseds**(*aa*)

Alias for *set_antialiased*.

**set_array**(*A*)

Set the value array from array-like *A*.

**Parameters**

**A**

[array-like or None] The values that are mapped to colors.

The base class *ScalarMappable* does not make any assumptions on the dimensionality and shape of the value array *A*.

**set_capstyle**(*cs*)

Set the *CapStyle* for the collection (for all its elements).

**Parameters**

**cs**

[*CapStyle* or {'butt', 'projecting', 'round'}]

**set_clim**(*vmin=None*, *vmax=None*)

Set the norm limits for image scaling.

**Parameters**

**vmin, vmax**

[float] The limits.

The limits may also be passed as a tuple (*vmin*, *vmax*) as a single positional argument.

**set_clip_box**(*clipbox*)

Set the artist's clip *Bbox*.

**Parameters**

**clipbox**

[*BboxBase* or None] Will typically be created from a *TransformedBbox*. For instance, `TransformedBbox(Bbox([[0, 0], [1, 1]]), ax.transAxes)` is the default clipping for an artist added to an Axes.

**set_clip_on**(*b*)

Set whether the artist uses clipping.

When False, artists will be visible outside the Axes which can lead to unexpected results.

**Parameters**

**b**

[bool]

**set_clip_path**(*path*, *transform=None*)

Set the artist's clip path.

**Parameters**

**path**

[*Patch* or *Path* or *TransformedPath* or None] The clip path. If given a *Path*, *transform* must be provided as well. If *None*, a previously set clip path is removed.

**transform**

[*Transform*, optional] Only used if *path* is a *Path*, in which case the given *Path* is converted to a *TransformedPath* using *transform*.

### Notes

For efficiency, if *path* is a *Rectangle* this method will set the clipping box to the corresponding rectangle and set the clipping path to `None`.

For technical reasons (support of *set*), a tuple (*path*, *transform*) is also accepted as a single positional parameter.

**set_cmap**(*cmap*)

Set the colormap for luminance data.

**Parameters**

**cmap**

[*Colormap* or str or None]

**set_color**(*c*)

Set the edgecolor(s) of the LineCollection.

**Parameters**

**c**

[*color* or list of *color*] Single color (all lines have same color), or a sequence of RGBA tuples; if it is a sequence the lines will cycle through the sequence.

**set_colors**(*c*)

Set the edgecolor(s) of the LineCollection.

**Parameters**

**c**

[*color* or list of *color*] Single color (all lines have same color), or a sequence of RGBA tuples; if it is a sequence the lines will cycle through the sequence.

**set_dashes**(*ls*)

Alias for *set_linestyle*.

**set_ec**(*c*)

Alias for *set_edgecolor*.

**set_edgecolor**(*c*)

Set the edgecolor(s) of the collection.

**Parameters**

**c**

[*color* or list of *color* or 'face'] The collection edgecolor(s). If a sequence, the patches cycle through it. If 'face', match the facecolor.

**set_edgecolors**(*c*)

Alias for *set_edgecolor*.

**set_facecolor**(*c*)

Set the facecolor(s) of the collection. *c* can be a color (all patches have same color), or a sequence of colors; if it is a sequence the patches will cycle through the sequence.

If *c* is 'none', the patch will not be filled.

**Parameters**

**c**

[*color* or list of *color*]

**set_facecolors**(*c*)

Alias for *set_facecolor*.

**set_fc**(*c*)

Alias for *set_facecolor*.

**set_figure**(*fig*)

Set the *Figure* instance the artist belongs to.

**Parameters**

**fig**

[*Figure*]

**set_gapcolor**(*gapcolor*)

Set a color to fill the gaps in the dashed line style.

---

**Note:** Striped lines are created by drawing two interleaved dashed lines. There can be overlaps between those two, which may result in artifacts when using transparency.

This functionality is experimental and may change.

---

**Parameters**

**gapcolor**

[*color* or list of *color* or None] The color with which to fill the gaps. If None, the gaps are unfilled.

**set_gid**(*gid*)

Set the (group) id for the artist.

**Parameters**

**gid**

[str]

**set_hatch**(*hatch*)

Set the hatching pattern

*hatch* can be one of:

```
/   - diagonal hatching
\   - back diagonal
|   - vertical
-   - horizontal
+   - crossed
x   - crossed diagonal
o   - small circle
O   - large circle
.   - dots
*   - stars
```

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

**Parameters**

**hatch**

[{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}]

**set_in_layout**(*in_layout*)

Set if artist is to be included in layout calculations, E.g. *Constrained layout guide*, *Figure.tight_layout()*, and `fig.savefig(fname, bbox_inches='tight')`.

**Parameters**

**in_layout**

[bool]

**set_joinstyle**(*js*)

Set the *JoinStyle* for the collection (for all its elements).

**Parameters**

**js**

[*JoinStyle* or {'miter', 'round', 'bevel'}]

**set_label**(*s*)

Set a label that will be displayed in the legend.

**Parameters**

**s**

[object] *s* will be converted to a string by calling `str`.

**set_linelength**(*linelength*)

Set the length of the lines used to mark each event.

**set_lineoffset**(*lineoffset*)

Set the offset of the lines used to mark each event.

**set_linestyle**(*ls*)

Set the linestyle(s) for the collection.

| linestyle | description |
|---|---|
| `'-'` or `'solid'` | solid line |
| `'--'` or `'dashed'` | dashed line |
| `'-.'` or `'dashdot'` | dash-dotted line |
| `':'` or `'dotted'` | dotted line |

Alternatively a dash tuple of the following form can be provided:

```
(offset, onoffseq),
```

where `onoffseq` is an even length tuple of on and off ink in points.

**Parameters**

**ls**

[str or tuple or list thereof] Valid values for individual linestyles include {'-', '--', '-.', ':', '', (offset, on-off-seq)}. See *Line2D.set_linestyle* for a complete description.

**set_linestyles**(*ls*)

Alias for *set_linestyle*.

**set_linewidth**(*lw*)

Set the linewidth(s) for the collection. *lw* can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

**Parameters**

**lw**

[float or list of floats]

**set_linewidths**(*lw*)

Alias for *set_linewidth*.

**set_ls**(*ls*)

Alias for *set_linestyle*.

**set_lw**(*lw*)

Alias for *set_linewidth*.

**set_mouseover**(*mouseover*)

Set whether this artist is queried for custom context information when the mouse cursor moves over it.

**Parameters**

**mouseover**

[bool]

**See also:**

***get_cursor_data***
***ToolCursorPosition***
***NavigationToolbar2***

**set_norm**(*norm*)

Set the normalization instance.

**Parameters**

**norm**

[*Normalize* or str or None]

### Notes

If there are any colorbars using the mappable for this norm, setting the norm of the mappable will reset the norm, locator, and formatters on the colorbar to default.

**set_offset_transform**(*offset_transform*)

Set the artist offset transform.

**Parameters**

**offset_transform**

[*Transform*]

**set_offsets**(*offsets*)

Set the offsets for the collection.

**Parameters**

**offsets**

[(N, 2) or (2,) array-like]

**set_orientation**(*orientation*)

Set the orientation of the event line.

**Parameters**

**orientation**

[{'horizontal', 'vertical'}]

**set_path_effects**(*path_effects*)

Set the path effects.

**Parameters**

**path_effects**

[list of *AbstractPathEffect*]

**set_paths**(*segments*)

**set_picker**(*picker*)

Define the picking behavior of the artist.

**Parameters**

**picker**

[None or bool or float or callable] This can be one of the following:

- *None*: Picking is disabled for this artist (default).

- A boolean: If *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist.
- A float: If picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if its data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event
- A function: If picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

to determine the hit test. if the mouse event is over the artist, return *hit=True* and props is a dictionary of properties you want added to the PickEvent attributes.

**set_pickradius**(*pickradius*)

Set the pick radius used for containment tests.

**Parameters**

**pickradius**

[float] Pick radius, in points.

**set_positions**(*positions*)

Set the positions of the events.

**set_rasterized**(*rasterized*)

Force rasterized (bitmap) drawing for vector graphics output.

Rasterized drawing is not supported by all artists. If you try to enable this on an artist that does not support it, the command has no effect and a warning will be issued.

This setting is ignored for pixel-based output.

See also *Rasterization for vector graphics*.

**Parameters**

**rasterized**

[bool]

**set_segments**(*segments*)

**set_sketch_params**(*scale=None*, *length=None*, *randomness=None*)

Set the sketch parameters.

**Parameters**

**scale**

[float, optional] The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is `None`, or not provided, no sketch filter will be provided.

**length**

[float, optional] The length of the wiggle along the line, in pixels (default 128.0)

**randomness**

[float, optional] The scale factor by which the length is shrunken or expanded (default 16.0)

The PGF backend uses this argument as an RNG seed and not as described above. Using the same seed yields the same random shape.

**set_snap**(*snap*)

Set the snapping behavior.

Snapping aligns positions with the pixel grid, which results in clearer images. For example, if a black line of 1px width was defined at a position in between two pixels, the resulting image would contain the interpolated value of that line in the pixel grid, which would be a grey value on both adjacent pixel positions. In contrast, snapping will move the line to the nearest integer pixel value, so that the resulting image will really contain a 1px wide black line.

Snapping is currently only supported by the Agg and MacOSX backends.

**Parameters**

**snap**

[bool or None] Possible values:

- *True*: Snap vertices to the nearest pixel center.
- *False*: Do not modify vertex positions.
- *None*: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center.

**set_transOffset**(*offset_transform*)

Alias for *set_offset_transform*.

**set_transform**(*t*)

Set the artist transform.

**Parameters**

**t**

[*Transform*]

**set_url**(*url*)

Set the url for the artist.

**Parameters**

**url**

[str]

**set_urls**(*urls*)

**Parameters**

**urls**

[list of str or None]

### Notes

URLs are currently only implemented by the SVG backend. They are ignored by all other backends.

**set_verts**(*segments*)

**set_visible**(*b*)

Set the artist's visibility.

**Parameters**

**b**

[bool]

**set_zorder**(*level*)

Set the zorder for the artist. Artists with lower zorder values are drawn first.

**Parameters**

**level**

[float]

**property stale**

Whether the artist is 'stale' and needs to be re-drawn for the output to match the internal state of the artist.

**property sticky_edges**

x and y sticky edge lists for autoscaling.

When performing autoscaling, if a data limit coincides with a value in the corresponding sticky_edges list, then no margin will be added--the view limit "sticks" to the edge. A typical use case is histograms, where one usually expects no margin on the bottom edge (0) of the histogram.

Moreover, margin expansion "bumps" against sticky edges and cannot cross them. For example, if the upper data limit is 1.0, the upper view limit computed by simple margin application is 1.2, but there is a sticky edge at 1.1, then the actual upper view limit will be 1.1.

This attribute cannot be assigned to; however, the `x` and `y` lists can be modified in place as needed.

### Examples

```
>>> artist.sticky_edges.x[:] = (xmin, xmax)
>>> artist.sticky_edges.y[:] = (ymin, ymax)
```

**switch_orientation**()

Switch the orientation of the event line, either from vertical to horizontal or vice versus.

**to_rgba**(*x*, *alpha=None*, *bytes=False*, *norm=True*)

Return a normalized RGBA array corresponding to *x*.

In the normal case, *x* is a 1D or 2D sequence of scalars, and the corresponding `ndarray` of RGBA values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already RGB or RGBA, such as might have been read from an image file. If *x* is an `ndarray` with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an RGB or RGBA array, and no mapping will be done. The array can be `uint8`, or it can be floats with values in the 0-1 range; otherwise a ValueError will be raised. Any NaNs or masked elements will be set to 0 alpha. If the last dimension is 3, the *alpha* kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the *alpha* kwarg is ignored; it does not replace the preexisting alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if *bytes* is *False* (default), the RGBA array will be floats in the 0-1 range; if it is *True*, the returned RGBA array will be `uint8` in the 0 to 255 range.

If norm is False, no normalization of the input data is performed, and it is assumed to be in the range (0-1).

**update**(*props*)

Update this artist's properties from the dict *props*.

**Parameters**

**props**

[dict]

**update_from**(*other*)

Copy properties from other to self.

**update_scalarmappable**()

Update colors from the scalar mappable array, if any.

Assign colors to edges and faces based on the array and/or colors that were directly set, as appropriate.

**zorder = 0**

**class** matplotlib.collections.**LineCollection**(*segments*, *, *zorder=2*, ***kwargs*)

Bases: *Collection*

Represents a sequence of *Line2D*s that should be drawn together.

This class extends *Collection* to represent a sequence of *Line2D*s instead of just a sequence of *Patch*s. Just as in *Collection*, each property of a *LineCollection* may be either a single value or a list of values. This list is then used cyclically for each element of the LineCollection, so the property of the `i`th element of the collection is:

```
prop[i % len(prop)]
```

The properties of each member of a *LineCollection* default to their values in *rcParams["lines.*"]* instead of *rcParams["patch.*"]*, and the property *colors* is added in place of *edgecolors*.

**Parameters**

**segments**

[list of array-like] A sequence (*line0*, *line1*, *line2*) of lines, where each line is a list of points:

```
lineN = [(x0, y0), (x1, y1), ... (xm, ym)]
```

or the equivalent Mx2 numpy array with two columns. Each line can have a different number of segments.

**linewidths**

[float or list of float, default: *rcParams["lines.linewidth"]* (default: `1.5`)] The width of each line in points.

**colors**

[*color* or list of color, default: *rcParams["lines.color"]* (default: `'C0'`)] A sequence of RGBA tuples (e.g., arbitrary color strings, etc, not allowed).

**antialiaseds**

[bool or list of bool, default: *rcParams["lines.antialiased"]* (default: `True`)] Whether to use antialiasing for each line.

**zorder**

[float, default: 2] zorder of the lines once drawn.

**facecolors**

[*color* or list of *color*, default: 'none'] When setting *facecolors*, each line is interpreted as a boundary for an area, implicitly closing the path from the last point to the first point. The enclosed area is filled with *facecolor*. In order to manually specify what should count as the "interior" of each line, please use *PathCollection* instead, where the "interior" can be specified by appropriate usage of *CLOSEPOLY*.

****kwargs**

Forwarded to *Collection*.

**add_callback**(*func*)

Add a callback function that will be called whenever one of the *Artist*'s properties changes.

**Parameters**

**func**

[callable] The callback function. It must have the signature:

```
def func(artist: Artist) -> Any
```

where *artist* is the calling *Artist*. Return values may exist but are ignored.

**Returns**

**int**

The observer id associated with the callback. This id can be used for removing the callback with *remove_callback* later.

**See also:**

***remove_callback***

**autoscale**()

Autoscale the scalar limits on the norm instance using the current array

**autoscale_None**()

Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

**property axes**

The *Axes* instance the artist resides in, or *None*.

**changed**()

Call this whenever the mappable is changed to notify all the callbackSM listeners to the 'changed' signal.

**colorbar**

The last colorbar associated with this ScalarMappable. May be None.

**contains**(*mouseevent*)

Test whether the mouse event occurred in the collection.

Returns `bool, dict(ind=itemlist)`, where every item in itemlist contains the event.

**convert_xunits**(*x*)

Convert *x* using the unit type of the xaxis.

If the artist is not contained in an Axes or if the xaxis does not have units, *x* itself is returned.

**convert_yunits**(*y*)

Convert *y* using the unit type of the yaxis.

If the artist is not contained in an Axes or if the yaxis does not have units, *y* itself is returned.

**draw**(*renderer*)

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

**Parameters**

**renderer**

[*RendererBase* subclass.]

### Notes

This method is overridden in the Artist subclasses.

**findobj**(*match=None*, *include_self=True*)

Find artist objects.

Recursively find all *Artist* instances contained in the artist.

**Parameters**

**match**

A filter criterion for the matches. This can be

- *None*: Return all objects contained in artist.
- A function with signature `def match(artist: Artist) -> bool`. The result will only contain artists for which the function returns *True*.
- A class instance: e.g., *Line2D*. The result will only contain artists of this class or its subclasses (`isinstance` check).

**include_self**

[bool] Include *self* in the list to be checked for a match.

**Returns**

**list of *Artist***

**format_cursor_data**(*data*)

Return a string representation of *data*.

---

**Note:** This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

---

The default implementation converts ints and floats and arrays of ints and floats into a comma-separated string enclosed in square brackets, unless the artist has an associated colorbar, in which case scalar values are formatted using the colorbar's formatter.

**See also:**

*get_cursor_data*

**get_aa**()

Alias for *get_antialiased*.

**get_agg_filter**()

Return filter function to be used for agg filter.

**get_alpha**()

Return the alpha value used for blending - not supported on all backends.

**get_animated**()

Return whether the artist is animated.

**get_antialiased**()

Get the antialiasing state for rendering.

**Returns**

**array of bools**

**get_antialiaseds**()

Alias for *get_antialiased*.

**get_array**()

Return the array of values, that are mapped to colors.

The base class *ScalarMappable* does not make any assumptions on the dimensionality and shape of the array.

**get_capstyle**()

Return the cap style for the collection (for all its elements).

**Returns**

**{'butt', 'projecting', 'round'} or None**

**get_children**()

Return a list of the child *Artist*s of this *Artist*.

**get_clim**()

Return the values (min, max) that are mapped to the colormap limits.

**get_clip_box**()

Return the clipbox.

**get_clip_on**()

Return whether the artist uses clipping.

**get_clip_path**()

Return the clip path.

**get_cmap**()

Return the *Colormap* instance.

**get_color**()

**get_colors**()

**get_cursor_data**(*event*)

Return the cursor data for a given event.

---

**Note:** This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

---

Cursor data can be used by Artists to provide additional context information for a given event. The default implementation just returns *None*.

Subclasses can override the method and return arbitrary data. However, when doing so, they must ensure that *format_cursor_data* can convert the data to a string representation.

The only current use case is displaying the z-value of an *AxesImage* in the status bar of a plot window, while moving the mouse.

**Parameters**

**event**

[*MouseEvent*]

**See also:**

***format_cursor_data***

**get_dashes**()

Alias for *get_linestyle*.

**get_datalim**(*transData*)

**get_ec**()

Alias for *get_edgecolor*.

**get_edgecolor**()

**get_edgecolors**()

Alias for *get_edgecolor*.

**get_facecolor**()

**get_facecolors**()

Alias for *get_facecolor*.

**get_fc**()

Alias for *get_facecolor*.

**get_figure**()

Return the *Figure* instance the artist belongs to.

**get_fill**()

Return whether face is colored.

**get_gapcolor**()

**get_gid**()

Return the group id.

**get_hatch**()

Return the current hatching pattern.

**get_in_layout**()

Return boolean flag, `True` if artist is included in layout calculations.

E.g. *Constrained layout guide*, *`Figure.tight_layout()`*, and `fig.savefig(fname, bbox_inches='tight')`.

**get_joinstyle**()

Return the join style for the collection (for all its elements).

**Returns**

**{'miter', 'round', 'bevel'} or None**

**get_label**()

Return the label used for this artist in the legend.

**get_linestyle**()

**get_linestyles**()

Alias for *get_linestyle*.

**get_linewidth**()

**get_linewidths**()

Alias for *get_linewidth*.

**get_ls**()

Alias for *get_linestyle*.

**get_lw**()

Alias for *get_linewidth*.

**get_mouseover**()

Return whether this artist is queried for custom context information when the mouse cursor moves over it.

**get_offset_transform**()

Return the *Transform* instance used by this artist offset.

**get_offsets**()

Return the offsets for the collection.

**get_path_effects**()

**get_paths**()

**get_picker**()

Return the picking behavior of the artist.

The possible values are described in *Artist.set_picker*.

**See also:**

***Artist.set_picker***, ***Artist.pickable***, ***Artist.pick***

**get_pickradius**()

**get_rasterized**()

Return whether the artist is to be rasterized.

**get_segments**()

**Returns**

**list**

List of segments in the LineCollection. Each list item contains an array of vertices.

**get_sketch_params**()

Return the sketch parameters for the artist.

**Returns**

**tuple or None**

A 3-tuple with the following elements:

- *scale*: The amplitude of the wiggle perpendicular to the source line.
- *length*: The length of the wiggle along the line.
- *randomness*: The scale factor by which the length is shrunken or expanded.

Returns *None* if no sketch parameters were set.

**get_snap**()

Return the snap setting.

See *set_snap* for details.

**get_tightbbox**(*renderer=None*)

Like *Artist.get_window_extent*, but includes any clipping.

**Parameters**

**renderer**

[*RendererBase* subclass, optional] renderer that will be used to draw the figures (i.e. `fig.canvas.get_renderer()`)

**Returns**

***Bbox* or None**

The enclosing bounding box (in figure pixel coordinates). Returns None if clipping results in no intersection.

**get_transOffset**()

Alias for *get_offset_transform*.

**get_transform**()

Return the *Transform* instance used by this artist.

**get_transformed_clip_path_and_affine**()

Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

**get_transforms**()

**get_url**()

Return the url.

**get_urls**()

Return a list of URLs, one for each element of the collection.

The list contains *None* for elements without a URL. See *Hyperlinks* for an example.

**get_visible**()

Return the visibility.

**get_window_extent**(*renderer=None*)

Get the artist's bounding box in display space.

The bounding box' width and height are nonnegative.

Subclasses should override for inclusion in the bounding box "tight" calculation. Default is to return an empty bounding box at 0, 0.

Be careful when using this function, the results will not update if the artist window extent of the artist changes. The extent can change due to any changes in the transform stack, such as changing the Axes limits, the figure size, or the canvas used (as is done when saving a figure). This can lead to unexpected behavior where interactive figures will look fine on the screen, but will save incorrectly.

**get_zorder**()

Return the artist's zorder.

**have_units**()

Return whether units are set on any axis.

**is_transform_set**()

Return whether the Artist has an explicitly set transform.

This is *True* after `set_transform` has been called.

**property mouseover**

Return whether this artist is queried for custom context information when the mouse cursor moves over it.

**property norm**

**pchanged**()

Call all of the registered callbacks.

This function is triggered internally when a property is changed.

**See also:**

***add_callback***
***remove_callback***

**pick**(*mouseevent*)

Process a pick event.

Each child artist will fire a pick event if *mouseevent* is over the artist and the artist has picker set.

**See also:**

*Artist.set_picker*, *Artist.get_picker*, *Artist.pickable*

**pickable**()

Return whether the artist is pickable.

**See also:**

*Artist.set_picker*, *Artist.get_picker*, *Artist.pick*

**properties**()

Return a dictionary of all the properties of the artist.

**remove**()

Remove the artist from the figure if possible.

The effect will not be visible until the figure is redrawn, e.g., with *FigureCanvasBase.draw_idle*. Call *relim* to update the Axes limits if desired.

Note: *relim* will not see collections even if the collection was added to the Axes with *autolim* = True.

Note: there is no support for removing the artist's legend entry.

**remove_callback**(*oid*)

Remove a callback based on its observer id.

**See also:**

*add_callback*

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *antialiased=<UNSET>*, *array=<UNSET>*, *capstyle=<UNSET>*, *clim=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *cmap=<UNSET>*, *color=<UNSET>*, *colors=<UNSET>*, *edgecolor=<UNSET>*, *facecolor=<UNSET>*, *gapcolor=<UNSET>*, *gid=<UNSET>*, *hatch=<UNSET>*, *in_layout=<UNSET>*, *joinstyle=<UNSET>*, *label=<UNSET>*, *linestyle=<UNSET>*, *linewidth=<UNSET>*, *mouseover=<UNSET>*, *norm=<UNSET>*, *offset_transform=<UNSET>*, *offsets=<UNSET>*, *path_effects=<UNSET>*, *paths=<UNSET>*, *picker=<UNSET>*, *pickradius=<UNSET>*, *rasterized=<UNSET>*, *segments=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *urls=<UNSET>*, *verts=<UNSET>*, *visible=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi v |
| *alpha* | unknown |

Table 38 – continued from p

| Property | Description |
|---|---|
| *animated* | bool |
| *antialiased* or aa or antialiaseds | bool or list of bools |
| *array* | array-like or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clim* | (vmin: float, vmax: float) |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *cmap* | *Colormap* or str or None |
| *color* | *color* or list of *color* |
| *colors* | *color* or list of *color* |
| *edgecolor* or ec or edgecolors | *color* or list of *color* or 'face' |
| *facecolor* or facecolors or fc | *color* or list of *color* |
| *figure* | *Figure* |
| *gapcolor* | *color* or list of *color* or None |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or dashes or linestyles or ls | str or tuple or list thereof |
| *linewidth* or linewidths or lw | float or list of floats |
| *mouseover* | bool |
| *norm* | *Normalize* or str or None |
| *offset_transform* or transOffset | *Transform* |
| *offsets* | (N, 2) or (2,) array-like |
| *path_effects* | list of *AbstractPathEffect* |
| *paths* | unknown |
| *picker* | None or bool or float or callable |
| *pickradius* | float |
| *rasterized* | bool |
| *segments* | unknown |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *urls* | list of str or None |
| *verts* | unknown |
| *visible* | bool |
| *zorder* | float |

**set_aa**(*aa*)

Alias for *set_antialiased*.

**set_agg_filter**(*filter_func*)

Set the agg filter.

**Parameters**

**filter_func**

[callable] A filter function, which takes a (m, n, depth) float array and a dpi value, and returns a (m, n, depth) array and two offsets from the bottom left corner of the image

**set_alpha**(*alpha*)

Set the alpha value used for blending - not supported on all backends.

**Parameters**

**alpha**

[array-like or scalar or None] All values must be within the 0-1 range, inclusive. Masked values and nans are not supported.

**set_animated**(*b*)

Set whether the artist is intended to be used in an animation.

If True, the artist is excluded from regular drawing of the figure. You have to call *Figure.draw_artist* / *Axes.draw_artist* explicitly on the artist. This approach is used to speed up animations using blitting.

See also *matplotlib.animation* and *Faster rendering by using blitting*.

**Parameters**

**b**

[bool]

**set_antialiased**(*aa*)

Set the antialiasing state for rendering.

**Parameters**

**aa**

[bool or list of bools]

**set_antialiaseds**(*aa*)

Alias for *set_antialiased*.

**set_array**(*A*)

Set the value array from array-like *A*.

**Parameters**

**A**

[array-like or None] The values that are mapped to colors.

The base class *ScalarMappable* does not make any assumptions on the dimensionality and shape of the value array *A*.

**set_capstyle**(*cs*)

Set the *CapStyle* for the collection (for all its elements).

**Parameters**

**cs**

[*CapStyle* or {'butt', 'projecting', 'round'}]

**set_clim**(*vmin=None*, *vmax=None*)

Set the norm limits for image scaling.

**Parameters**

**vmin, vmax**

[float] The limits.

The limits may also be passed as a tuple (*vmin*, *vmax*) as a single positional argument.

**set_clip_box**(*clipbox*)

Set the artist's clip *Bbox*.

**Parameters**

**clipbox**

[*BboxBase* or None] Will typically be created from a *TransformedBbox*. For instance, `TransformedBbox(Bbox([[0, 0], [1, 1]]), ax.transAxes)` is the default clipping for an artist added to an Axes.

**set_clip_on**(*b*)

Set whether the artist uses clipping.

When False, artists will be visible outside the Axes which can lead to unexpected results.

**Parameters**

**b**

[bool]

**set_clip_path**(*path*, *transform=None*)

Set the artist's clip path.

**Parameters**

**path**

[*Patch* or *Path* or *TransformedPath* or None] The clip path. If given a *Path*, *transform* must be provided as well. If *None*, a previously set clip path is removed.

**transform**

[*Transform*, optional] Only used if *path* is a *Path*, in which case the given *Path* is converted to a *TransformedPath* using *transform*.

### Notes

For efficiency, if *path* is a *Rectangle* this method will set the clipping box to the corresponding rectangle and set the clipping path to `None`.

For technical reasons (support of *set*), a tuple (*path*, *transform*) is also accepted as a single positional parameter.

**set_cmap**(*cmap*)

Set the colormap for luminance data.

**Parameters**

**cmap**

[*Colormap* or str or None]

**set_color**(*c*)

Set the edgecolor(s) of the LineCollection.

**Parameters**

**c**

[*color* or list of *color*] Single color (all lines have same color), or a sequence of RGBA tuples; if it is a sequence the lines will cycle through the sequence.

**set_colors**(*c*)

Set the edgecolor(s) of the LineCollection.

**Parameters**

**c**

[*color* or list of *color*] Single color (all lines have same color), or a sequence of RGBA tuples; if it is a sequence the lines will cycle through the sequence.

**set_dashes**(*ls*)

Alias for *set_linestyle*.

**set_ec**(*c*)

Alias for *set_edgecolor*.

**set_edgecolor**(*c*)

Set the edgecolor(s) of the collection.

**Parameters**

**c**

[*color* or list of *color* or 'face'] The collection edgecolor(s). If a sequence, the patches cycle through it. If 'face', match the facecolor.

**set_edgecolors**(*c*)

Alias for *set_edgecolor*.

**set_facecolor**(*c*)

Set the facecolor(s) of the collection. *c* can be a color (all patches have same color), or a sequence of colors; if it is a sequence the patches will cycle through the sequence.

If *c* is 'none', the patch will not be filled.

**Parameters**

**c**

[*color* or list of *color*]

**set_facecolors**(*c*)

Alias for *set_facecolor*.

**set_fc**(*c*)

Alias for *set_facecolor*.

**set_figure**(*fig*)

Set the *Figure* instance the artist belongs to.

**Parameters**

**fig**

[*Figure*]

**set_gapcolor**(*gapcolor*)

Set a color to fill the gaps in the dashed line style.

---

**Note:** Striped lines are created by drawing two interleaved dashed lines. There can be overlaps between those two, which may result in artifacts when using transparency.

This functionality is experimental and may change.

**Parameters**

**gapcolor**

[*color* or list of *color* or None] The color with which to fill the gaps. If None, the gaps are unfilled.

**set_gid**(*gid*)

Set the (group) id for the artist.

**Parameters**

**gid**

[str]

**set_hatch**(*hatch*)

Set the hatching pattern

*hatch* can be one of:

```
/   - diagonal hatching
\   - back diagonal
|   - vertical
-   - horizontal
+   - crossed
x   - crossed diagonal
o   - small circle
O   - large circle
.   - dots
*   - stars
```

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

**Parameters**

**hatch**

[{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}]

**set_in_layout**(*in_layout*)

Set if artist is to be included in layout calculations, E.g. *Constrained layout guide*, *`Figure.tight_layout()`*, and `fig.savefig(fname, bbox_inches='tight')`.

**Parameters**

**in_layout**

[bool]

**set_joinstyle**(*js*)

Set the *JoinStyle* for the collection (for all its elements).

**Parameters**

**js**

[*JoinStyle* or {'miter', 'round', 'bevel'}]

**set_label**(*s*)

Set a label that will be displayed in the legend.

**Parameters**

**s**

[object] *s* will be converted to a string by calling `str`.

**set_linestyle**(*ls*)

Set the linestyle(s) for the collection.

| linestyle | description |
|---|---|
| `'-'` or `'solid'` | solid line |
| `'--'` or `'dashed'` | dashed line |
| `'-.'` or `'dashdot'` | dash-dotted line |
| `':'` or `'dotted'` | dotted line |

Alternatively a dash tuple of the following form can be provided:

```
(offset, onoffseq),
```

where `onoffseq` is an even length tuple of on and off ink in points.

**Parameters**

**ls**

[str or tuple or list thereof] Valid values for individual linestyles include {'-', '--', '-.', ':', '', (offset, on-off-seq)}. See *Line2D.set_linestyle* for a complete description.

**set_linestyles**(*ls*)

Alias for *set_linestyle*.

**set_linewidth**(*lw*)

Set the linewidth(s) for the collection. *lw* can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

**Parameters**

**lw**

[float or list of floats]

**set_linewidths**(*lw*)

Alias for *set_linewidth*.

**set_ls**(*ls*)

Alias for *set_linestyle*.

**set_lw**(*lw*)

Alias for *set_linewidth*.

**set_mouseover**(*mouseover*)

Set whether this artist is queried for custom context information when the mouse cursor moves over it.

**Parameters**

**mouseover**

[bool]

**See also:**

***get_cursor_data***
***ToolCursorPosition***
***NavigationToolbar2***

**set_norm**(*norm*)

Set the normalization instance.

**Parameters**

**norm**

[*Normalize* or str or None]

### Notes

If there are any colorbars using the mappable for this norm, setting the norm of the mappable will reset the norm, locator, and formatters on the colorbar to default.

**set_offset_transform**(*offset_transform*)

Set the artist offset transform.

**Parameters**

**offset_transform**

[*Transform*]

**set_offsets**(*offsets*)

Set the offsets for the collection.

**Parameters**

**offsets**

[(N, 2) or (2,) array-like]

**set_path_effects**(*path_effects*)

Set the path effects.

**Parameters**

**path_effects**

[list of *AbstractPathEffect*]

**set_paths**(*segments*)

**set_picker**(*picker*)

Define the picking behavior of the artist.

**Parameters**

**picker**

[None or bool or float or callable] This can be one of the following:

- *None*: Picking is disabled for this artist (default).
- A boolean: If *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist.
- A float: If picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if its data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event

- A function: If picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

to determine the hit test. if the mouse event is over the artist, return *hit=True* and props is a dictionary of properties you want added to the PickEvent attributes.

**set_pickradius**(*pickradius*)

Set the pick radius used for containment tests.

**Parameters**

**pickradius**

[float] Pick radius, in points.

**set_rasterized**(*rasterized*)

Force rasterized (bitmap) drawing for vector graphics output.

Rasterized drawing is not supported by all artists. If you try to enable this on an artist that does not support it, the command has no effect and a warning will be issued.

This setting is ignored for pixel-based output.

See also *Rasterization for vector graphics*.

**Parameters**

**rasterized**

[bool]

**set_segments**(*segments*)

**set_sketch_params**(*scale=None*, *length=None*, *randomness=None*)

Set the sketch parameters.

**Parameters**

**scale**

[float, optional] The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is `None`, or not provided, no sketch filter will be provided.

**length**

[float, optional] The length of the wiggle along the line, in pixels (default 128.0)

**randomness**

[float, optional] The scale factor by which the length is shrunken or expanded (default 16.0)

The PGF backend uses this argument as an RNG seed and not as described above. Using the same seed yields the same random shape.

**set_snap**(*snap*)

Set the snapping behavior.

Snapping aligns positions with the pixel grid, which results in clearer images. For example, if a black line of 1px width was defined at a position in between two pixels, the resulting image would contain the interpolated value of that line in the pixel grid, which would be a grey value on both adjacent pixel positions. In contrast, snapping will move the line to the nearest integer pixel value, so that the resulting image will really contain a 1px wide black line.

Snapping is currently only supported by the Agg and MacOSX backends.

**Parameters**

**snap**

[bool or None] Possible values:

- *True*: Snap vertices to the nearest pixel center.
- *False*: Do not modify vertex positions.
- *None*: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center.

**set_transOffset**(*offset_transform*)

Alias for *set_offset_transform*.

**set_transform**(*t*)

Set the artist transform.

**Parameters**

**t**

[*Transform*]

**set_url**(*url*)

Set the url for the artist.

**Parameters**

**url**

[str]

**set_urls**(*urls*)

**Parameters**

**urls**

[list of str or None]

### Notes

URLs are currently only implemented by the SVG backend. They are ignored by all other backends.

**set_verts**(*segments*)

**set_visible**(*b*)

Set the artist's visibility.

**Parameters**

**b**

[bool]

**set_zorder**(*level*)

Set the zorder for the artist. Artists with lower zorder values are drawn first.

**Parameters**

**level**

[float]

**property stale**

Whether the artist is 'stale' and needs to be re-drawn for the output to match the internal state of the artist.

**property sticky_edges**

`x` and `y` sticky edge lists for autoscaling.

When performing autoscaling, if a data limit coincides with a value in the corresponding sticky_edges list, then no margin will be added--the view limit "sticks" to the edge. A typical use case is histograms, where one usually expects no margin on the bottom edge (0) of the histogram.

Moreover, margin expansion "bumps" against sticky edges and cannot cross them. For example, if the upper data limit is 1.0, the upper view limit computed by simple margin application is 1.2, but there is a sticky edge at 1.1, then the actual upper view limit will be 1.1.

This attribute cannot be assigned to; however, the `x` and `y` lists can be modified in place as needed.

### Examples

```
>>> artist.sticky_edges.x[:] = (xmin, xmax)
>>> artist.sticky_edges.y[:] = (ymin, ymax)
```

**to_rgba**(*x*, *alpha=None*, *bytes=False*, *norm=True*)

Return a normalized RGBA array corresponding to *x*.

In the normal case, *x* is a 1D or 2D sequence of scalars, and the corresponding `ndarray` of RGBA values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already RGB or RGBA, such as might have been read from an image file. If *x* is an `ndarray` with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an RGB or RGBA array, and no mapping will be done. The array can be `uint8`, or it can be floats with values in the 0-1 range; otherwise a ValueError will be raised. Any NaNs or masked elements will be set to 0 alpha. If the last dimension is 3, the *alpha* kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the *alpha* kwarg is ignored; it does not replace the preexisting alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if *bytes* is *False* (default), the RGBA array will be floats in the 0-1 range; if it is *True*, the returned RGBA array will be `uint8` in the 0 to 255 range.

If norm is False, no normalization of the input data is performed, and it is assumed to be in the range (0-1).

**update**(*props*)

Update this artist's properties from the dict *props*.

**Parameters**

**props**

[dict]

**update_from**(*other*)

Copy properties from other to self.

**update_scalarmappable**()

Update colors from the scalar mappable array, if any.

Assign colors to edges and faces based on the array and/or colors that were directly set, as appropriate.

**zorder = 0**

**class** matplotlib.collections.**PatchCollection**(*patches*, *, *match_original=False*, ***kwargs*)

Bases: *Collection*

A generic collection of patches.

PatchCollection draws faster than a large number of equivalent individual Patches. It also makes it easier to assign a colormap to a heterogeneous collection of patches.

**Parameters**

**patches**

[list of *Patch*] A sequence of Patch objects. This list may include a heterogeneous assortment of different patch types.

**match_original**

[bool, default: False] If True, use the colors and linewidths of the original patches. If False, new colors may be assigned by providing the standard collection arguments, facecolor, edgecolor, linewidths, norm or cmap.

****kwargs**

All other parameters are forwarded to *Collection*.

If any of *edgecolors*, *facecolors*, *linewidths*, *antialiaseds* are None, they default to their *rcParams* patch setting, in sequence form.

## Notes

The use of *ScalarMappable* functionality is optional. If the *ScalarMappable* matrix _A has been set (via a call to *set_array*), at draw time a call to scalar mappable will be made to set the face colors.

**add_callback**(*func*)

Add a callback function that will be called whenever one of the *Artist*'s properties changes.

**Parameters**

**func**

[callable] The callback function. It must have the signature:

```
def func(artist: Artist) -> Any
```

where *artist* is the calling *Artist*. Return values may exist but are ignored.

**Returns**

**int**

The observer id associated with the callback. This id can be used for removing the callback with *remove_callback* later.

**See also:**

***remove_callback***

**autoscale**()

Autoscale the scalar limits on the norm instance using the current array

**autoscale_None**()

Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

**property axes**

The *Axes* instance the artist resides in, or *None*.

**changed**()

Call this whenever the mappable is changed to notify all the callbackSM listeners to the 'changed' signal.

**colorbar**

The last colorbar associated with this ScalarMappable. May be None.

**contains**(*mouseevent*)

Test whether the mouse event occurred in the collection.

Returns `bool, dict(ind=itemlist)`, where every item in itemlist contains the event.

**convert_xunits**(*x*)

Convert *x* using the unit type of the xaxis.

If the artist is not contained in an Axes or if the xaxis does not have units, *x* itself is returned.

**convert_yunits**(*y*)

Convert *y* using the unit type of the yaxis.

If the artist is not contained in an Axes or if the yaxis does not have units, *y* itself is returned.

**draw**(*renderer*)

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

**Parameters**

**renderer**

[*RendererBase* subclass.]

### Notes

This method is overridden in the Artist subclasses.

**findobj**(*match=None*, *include_self=True*)

Find artist objects.

Recursively find all *Artist* instances contained in the artist.

**Parameters**

**match**

A filter criterion for the matches. This can be

- *None*: Return all objects contained in artist.
- A function with signature `def match(artist: Artist) -> bool`. The result will only contain artists for which the function returns *True*.
- A class instance: e.g., *Line2D*. The result will only contain artists of this class or its subclasses (`isinstance` check).

**include_self**

[bool] Include *self* in the list to be checked for a match.

**Returns**

**list of *Artist***

**format_cursor_data**(*data*)

Return a string representation of *data*.

---

**Note:** This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

---

The default implementation converts ints and floats and arrays of ints and floats into a comma-separated string enclosed in square brackets, unless the artist has an associated colorbar, in which case scalar values are formatted using the colorbar's formatter.

**See also:**

***get_cursor_data***

**get_aa**()

Alias for *get_antialiased*.

**get_agg_filter**()

Return filter function to be used for agg filter.

**get_alpha**()

Return the alpha value used for blending - not supported on all backends.

**get_animated**()

Return whether the artist is animated.

**get_antialiased**()

Get the antialiasing state for rendering.

**Returns**

**array of bools**

**get_antialiaseds**()

Alias for *get_antialiased*.

**get_array**()

Return the array of values, that are mapped to colors.

The base class *ScalarMappable* does not make any assumptions on the dimensionality and shape of the array.

**get_capstyle**()

Return the cap style for the collection (for all its elements).

**Returns**

**{'butt', 'projecting', 'round'} or None**

**get_children**()

Return a list of the child *Artist*s of this *Artist*.

**get_clim**()

Return the values (min, max) that are mapped to the colormap limits.

**get_clip_box**()

Return the clipbox.

**get_clip_on**()

Return whether the artist uses clipping.

**get_clip_path**()

Return the clip path.

**get_cmap**()

Return the *Colormap* instance.

**get_cursor_data**(*event*)

Return the cursor data for a given event.

---

**Note:** This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

---

Cursor data can be used by Artists to provide additional context information for a given event. The default implementation just returns *None*.

Subclasses can override the method and return arbitrary data. However, when doing so, they must ensure that *format_cursor_data* can convert the data to a string representation.

The only current use case is displaying the z-value of an *AxesImage* in the status bar of a plot window, while moving the mouse.

**Parameters**

**event**

[*MouseEvent*]

**See also:**

***format_cursor_data***

**get_dashes**()
Alias for *get_linestyle*.

**get_datalim**(*transData*)

**get_ec**()
Alias for *get_edgecolor*.

**get_edgecolor**()

**get_edgecolors**()
Alias for *get_edgecolor*.

**get_facecolor**()

**get_facecolors**()
Alias for *get_facecolor*.

**get_fc**()
Alias for *get_facecolor*.

**get_figure**()
Return the *Figure* instance the artist belongs to.

**get_fill**()
Return whether face is colored.

**get_gid**()
Return the group id.

**get_hatch**()
Return the current hatching pattern.

**get_in_layout**()
Return boolean flag, `True` if artist is included in layout calculations.

E.g. *Constrained layout guide*, *`Figure.tight_layout()`*, and `fig.savefig(fname, bbox_inches='tight')`.

**get_joinstyle**()
Return the join style for the collection (for all its elements).

**Returns**

**{'miter', 'round', 'bevel'} or None**

**get_label**()

Return the label used for this artist in the legend.

**get_linestyle**()

**get_linestyles**()

Alias for *get_linestyle*.

**get_linewidth**()

**get_linewidths**()

Alias for *get_linewidth*.

**get_ls**()

Alias for *get_linestyle*.

**get_lw**()

Alias for *get_linewidth*.

**get_mouseover**()

Return whether this artist is queried for custom context information when the mouse cursor moves over it.

**get_offset_transform**()

Return the *Transform* instance used by this artist offset.

**get_offsets**()

Return the offsets for the collection.

**get_path_effects**()

**get_paths**()

**get_picker**()

Return the picking behavior of the artist.

The possible values are described in *Artist.set_picker*.

**See also:**

***Artist.set_picker***, ***Artist.pickable***, ***Artist.pick***

**get_pickradius**()

**get_rasterized**()

Return whether the artist is to be rasterized.

**get_sketch_params**()

Return the sketch parameters for the artist.

**Returns**

**tuple or None**

A 3-tuple with the following elements:

- *scale*: The amplitude of the wiggle perpendicular to the source line.
- *length*: The length of the wiggle along the line.
- *randomness*: The scale factor by which the length is shrunken or expanded.

Returns *None* if no sketch parameters were set.

**get_snap**()

Return the snap setting.

See *set_snap* for details.

**get_tightbbox**(*renderer=None*)

Like *Artist.get_window_extent*, but includes any clipping.

**Parameters**

**renderer**

[*RendererBase* subclass, optional] renderer that will be used to draw the figures (i.e. `fig.canvas.get_renderer()`)

**Returns**

***Bbox* or None**

The enclosing bounding box (in figure pixel coordinates). Returns None if clipping results in no intersection.

**get_transOffset**()

Alias for *get_offset_transform*.

**get_transform**()

Return the *Transform* instance used by this artist.

**get_transformed_clip_path_and_affine**()

Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

**get_transforms**()

**get_url**()

Return the url.

**get_urls**()

Return a list of URLs, one for each element of the collection.

The list contains *None* for elements without a URL. See *Hyperlinks* for an example.

**get_visible**()

Return the visibility.

**get_window_extent**(*renderer=None*)

Get the artist's bounding box in display space.

The bounding box' width and height are nonnegative.

Subclasses should override for inclusion in the bounding box "tight" calculation. Default is to return an empty bounding box at 0, 0.

Be careful when using this function, the results will not update if the artist window extent of the artist changes. The extent can change due to any changes in the transform stack, such as changing the Axes limits, the figure size, or the canvas used (as is done when saving a figure). This can lead to unexpected behavior where interactive figures will look fine on the screen, but will save incorrectly.

**get_zorder**()

Return the artist's zorder.

**have_units**()

Return whether units are set on any axis.

**is_transform_set**()

Return whether the Artist has an explicitly set transform.

This is *True* after *set_transform* has been called.

**property mouseover**

Return whether this artist is queried for custom context information when the mouse cursor moves over it.

**property norm**

**pchanged**()

Call all of the registered callbacks.

This function is triggered internally when a property is changed.

**See also:**

***add_callback***
***remove_callback***

**pick**(*mouseevent*)

Process a pick event.

Each child artist will fire a pick event if *mouseevent* is over the artist and the artist has picker set.

**See also:**

***Artist.set_picker***, ***Artist.get_picker***, ***Artist.pickable***

**pickable**()

Return whether the artist is pickable.

**See also:**

*Artist.set_picker*, *Artist.get_picker*, *Artist.pick*

**properties**()

Return a dictionary of all the properties of the artist.

**remove**()

Remove the artist from the figure if possible.

The effect will not be visible until the figure is redrawn, e.g., with *FigureCanvasBase.draw_idle*. Call *relim* to update the Axes limits if desired.

Note: *relim* will not see collections even if the collection was added to the Axes with *autolim* = True.

Note: there is no support for removing the artist's legend entry.

**remove_callback**(*oid*)

Remove a callback based on its observer id.

**See also:**

*add_callback*

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *antialiased=<UNSET>*, *array=<UNSET>*, *capstyle=<UNSET>*, *clim=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *cmap=<UNSET>*, *color=<UNSET>*, *edgecolor=<UNSET>*, *facecolor=<UNSET>*, *gid=<UNSET>*, *hatch=<UNSET>*, *in_layout=<UNSET>*, *joinstyle=<UNSET>*, *label=<UNSET>*, *linestyle=<UNSET>*, *linewidth=<UNSET>*, *mouseover=<UNSET>*, *norm=<UNSET>*, *offset_transform=<UNSET>*, *offsets=<UNSET>*, *path_effects=<UNSET>*, *paths=<UNSET>*, *picker=<UNSET>*, *pickradius=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *urls=<UNSET>*, *visible=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi v |
| *alpha* | array-like or scalar or None |
| *animated* | bool |
| *antialiased* or aa or antialiaseds | bool or list of bools |
| *array* | array-like or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |

Table 39 – continued from p

| Property | Description |
|---|---|
| clim | (vmin: float, vmax: float) |
| clip_box | BboxBase or None |
| clip_on | bool |
| clip_path | Patch or (Path, Transform) or None |
| cmap | Colormap or str or None |
| color | color or list of RGBA tuples |
| edgecolor or ec or edgecolors | color or list of color or 'face' |
| facecolor or facecolors or fc | color or list of color |
| figure | Figure |
| gid | str |
| hatch | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| in_layout | bool |
| joinstyle | JoinStyle or {'miter', 'round', 'bevel'} |
| label | object |
| linestyle or dashes or linestyles or ls | str or tuple or list thereof |
| linewidth or linewidths or lw | float or list of floats |
| mouseover | bool |
| norm | Normalize or str or None |
| offset_transform or transOffset | Transform |
| offsets | (N, 2) or (2,) array-like |
| path_effects | list of AbstractPathEffect |
| paths | unknown |
| picker | None or bool or float or callable |
| pickradius | float |
| rasterized | bool |
| sketch_params | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| transform | Transform |
| url | str |
| urls | list of str or None |
| visible | bool |
| zorder | float |

**set_aa**(*aa*)

Alias for set_antialiased.

**set_agg_filter**(*filter_func*)

Set the agg filter.

**Parameters**

**filter_func**

[callable] A filter function, which takes a (m, n, depth) float array and a dpi value,

and returns a (m, n, depth) array and two offsets from the bottom left corner of the image

**set_alpha**(*alpha*)

Set the alpha value used for blending - not supported on all backends.

**Parameters**

**alpha**

[array-like or scalar or None] All values must be within the 0-1 range, inclusive. Masked values and nans are not supported.

**set_animated**(*b*)

Set whether the artist is intended to be used in an animation.

If True, the artist is excluded from regular drawing of the figure. You have to call *Figure.draw_artist* / *Axes.draw_artist* explicitly on the artist. This approach is used to speed up animations using blitting.

See also *matplotlib.animation* and *Faster rendering by using blitting*.

**Parameters**

**b**

[bool]

**set_antialiased**(*aa*)

Set the antialiasing state for rendering.

**Parameters**

**aa**

[bool or list of bools]

**set_antialiaseds**(*aa*)

Alias for *set_antialiased*.

**set_array**(*A*)

Set the value array from array-like *A*.

**Parameters**

**A**

[array-like or None] The values that are mapped to colors.

The base class *ScalarMappable* does not make any assumptions on the dimensionality and shape of the value array *A*.

**set_capstyle**(*cs*)

Set the *CapStyle* for the collection (for all its elements).

**Parameters**

**cs**

[*CapStyle* or {'butt', 'projecting', 'round'}]

**set_clim**(*vmin=None*, *vmax=None*)

Set the norm limits for image scaling.

**Parameters**

**vmin, vmax**

[float] The limits.

The limits may also be passed as a tuple (*vmin*, *vmax*) as a single positional argument.

**set_clip_box**(*clipbox*)

Set the artist's clip *Bbox*.

**Parameters**

**clipbox**

[*BboxBase* or None] Will typically be created from a *TransformedBbox*. For instance, `TransformedBbox(Bbox([[0, 0], [1, 1]]), ax.transAxes)` is the default clipping for an artist added to an Axes.

**set_clip_on**(*b*)

Set whether the artist uses clipping.

When False, artists will be visible outside the Axes which can lead to unexpected results.

**Parameters**

**b**

[bool]

**set_clip_path**(*path*, *transform=None*)

Set the artist's clip path.

**Parameters**

**path**

[*Patch* or *Path* or *TransformedPath* or None] The clip path. If given a *Path*, *transform* must be provided as well. If *None*, a previously set clip path is removed.

**transform**

[*Transform*, optional] Only used if *path* is a *Path*, in which case the given *Path* is converted to a *TransformedPath* using *transform*.

### Notes

For efficiency, if *path* is a *Rectangle* this method will set the clipping box to the corresponding rectangle and set the clipping path to `None`.

For technical reasons (support of *set*), a tuple (*path*, *transform*) is also accepted as a single positional parameter.

**set_cmap**(*cmap*)

Set the colormap for luminance data.

**Parameters**

**cmap**

[*Colormap* or str or None]

**set_color**(*c*)

Set both the edgecolor and the facecolor.

**Parameters**

**c**

[*color* or list of RGBA tuples]

**See also:**

***Collection.set_facecolor*, *Collection.set_edgecolor***

For setting the edge or face color individually.

**set_dashes**(*ls*)

Alias for *set_linestyle*.

**set_ec**(*c*)

Alias for *set_edgecolor*.

**set_edgecolor**(*c*)

Set the edgecolor(s) of the collection.

**Parameters**

**c**

[*color* or list of *color* or 'face'] The collection edgecolor(s). If a sequence, the patches cycle through it. If 'face', match the facecolor.

**set_edgecolors**(*c*)

Alias for *set_edgecolor*.

**set_facecolor**(*c*)

Set the facecolor(s) of the collection. *c* can be a color (all patches have same color), or a sequence of colors; if it is a sequence the patches will cycle through the sequence.

If *c* is 'none', the patch will not be filled.

**Parameters**

**c**

[*color* or list of *color*]

**set_facecolors**(*c*)

Alias for *set_facecolor*.

**set_fc**(*c*)

Alias for *set_facecolor*.

**set_figure**(*fig*)

Set the *Figure* instance the artist belongs to.

**Parameters**

**fig**

[*Figure*]

**set_gid**(*gid*)

Set the (group) id for the artist.

**Parameters**

**gid**

[str]

**set_hatch**(*hatch*)

Set the hatching pattern

*hatch* can be one of:

```
/   - diagonal hatching
\   - back diagonal
|   - vertical
-   - horizontal
+   - crossed
x   - crossed diagonal
o   - small circle
O   - large circle
```

(continues on next page)

(continued from previous page)

```
.   - dots
*   - stars
```

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

**Parameters**

**hatch**

[{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}]

**set_in_layout**(*in_layout*)

Set if artist is to be included in layout calculations, E.g. *Constrained layout guide*, *Figure.tight_layout()*, and `fig.savefig(fname, bbox_inches='tight')`.

**Parameters**

**in_layout**

[bool]

**set_joinstyle**(*js*)

Set the *JoinStyle* for the collection (for all its elements).

**Parameters**

**js**

[*JoinStyle* or {'miter', 'round', 'bevel'}]

**set_label**(*s*)

Set a label that will be displayed in the legend.

**Parameters**

**s**

[object] *s* will be converted to a string by calling `str`.

**set_linestyle**(*ls*)

Set the linestyle(s) for the collection.

| linestyle | description |
|---|---|
| `'-'` or `'solid'` | solid line |
| `'--'` or `'dashed'` | dashed line |
| `'-.'` or `'dashdot'` | dash-dotted line |
| `':'` or `'dotted'` | dotted line |

Alternatively a dash tuple of the following form can be provided:

```
(offset, onoffseq),
```

where `onoffseq` is an even length tuple of on and off ink in points.

**Parameters**

**ls**

[str or tuple or list thereof] Valid values for individual linestyles include {'-', '--', '-.', ':', '', (offset, on-off-seq)}. See *Line2D.set_linestyle* for a complete description.

**set_linestyles**(*ls*)

Alias for *set_linestyle*.

**set_linewidth**(*lw*)

Set the linewidth(s) for the collection. *lw* can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

**Parameters**

**lw**

[float or list of floats]

**set_linewidths**(*lw*)

Alias for *set_linewidth*.

**set_ls**(*ls*)

Alias for *set_linestyle*.

**set_lw**(*lw*)

Alias for *set_linewidth*.

**set_mouseover**(*mouseover*)

Set whether this artist is queried for custom context information when the mouse cursor moves over it.

**Parameters**

**mouseover**

[bool]

**See also:**

*get_cursor_data*
*ToolCursorPosition*
*NavigationToolbar2*

**set_norm**(*norm*)

Set the normalization instance.

**Parameters**

**norm**

[*Normalize* or str or None]

### Notes

If there are any colorbars using the mappable for this norm, setting the norm of the mappable will reset the norm, locator, and formatters on the colorbar to default.

**set_offset_transform**(*offset_transform*)

Set the artist offset transform.

**Parameters**

**offset_transform**

[*Transform*]

**set_offsets**(*offsets*)

Set the offsets for the collection.

**Parameters**

**offsets**

[(N, 2) or (2,) array-like]

**set_path_effects**(*path_effects*)

Set the path effects.

**Parameters**

**path_effects**

[list of *AbstractPathEffect*]

**set_paths**(*patches*)

**set_picker**(*picker*)

Define the picking behavior of the artist.

**Parameters**

**picker**

[None or bool or float or callable] This can be one of the following:

- *None*: Picking is disabled for this artist (default).
- A boolean: If *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist.
- A float: If picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if its data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event
- A function: If picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

to determine the hit test. if the mouse event is over the artist, return *hit=True* and props is a dictionary of properties you want added to the PickEvent attributes.

**set_pickradius**(*pickradius*)

Set the pick radius used for containment tests.

**Parameters**

**pickradius**

[float] Pick radius, in points.

**set_rasterized**(*rasterized*)

Force rasterized (bitmap) drawing for vector graphics output.

Rasterized drawing is not supported by all artists. If you try to enable this on an artist that does not support it, the command has no effect and a warning will be issued.

This setting is ignored for pixel-based output.

See also *Rasterization for vector graphics*.

**Parameters**

**rasterized**

[bool]

**set_sketch_params**(*scale=None*, *length=None*, *randomness=None*)

Set the sketch parameters.

**Parameters**

**scale**

[float, optional] The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is `None`, or not provided, no sketch filter will be provided.

**length**

[float, optional] The length of the wiggle along the line, in pixels (default 128.0)

**randomness**

[float, optional] The scale factor by which the length is shrunken or expanded (default 16.0)

The PGF backend uses this argument as an RNG seed and not as described above. Using the same seed yields the same random shape.

**set_snap**(*snap*)

Set the snapping behavior.

Snapping aligns positions with the pixel grid, which results in clearer images. For example, if a black line of 1px width was defined at a position in between two pixels, the resulting image would contain the interpolated value of that line in the pixel grid, which would be a grey value on both adjacent pixel positions. In contrast, snapping will move the line to the nearest integer pixel value, so that the resulting image will really contain a 1px wide black line.

Snapping is currently only supported by the Agg and MacOSX backends.

**Parameters**

**snap**

[bool or None] Possible values:

- *True*: Snap vertices to the nearest pixel center.
- *False*: Do not modify vertex positions.
- *None*: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center.

**set_transOffset**(*offset_transform*)

Alias for *set_offset_transform*.

**set_transform**(*t*)

Set the artist transform.

**Parameters**

**t**

[*Transform*]

**set_url**(*url*)

Set the url for the artist.

**Parameters**

**url**

[str]

**set_urls**(*urls*)

**Parameters**

**urls**

[list of str or None]

### Notes

URLs are currently only implemented by the SVG backend. They are ignored by all other backends.

**set_visible**(*b*)

Set the artist's visibility.

**Parameters**

**b**

[bool]

**set_zorder**(*level*)

Set the zorder for the artist. Artists with lower zorder values are drawn first.

**Parameters**

**level**

[float]

**property stale**

Whether the artist is 'stale' and needs to be re-drawn for the output to match the internal state of the artist.

**property sticky_edges**

x and y sticky edge lists for autoscaling.

When performing autoscaling, if a data limit coincides with a value in the corresponding sticky_edges list, then no margin will be added--the view limit "sticks" to the edge. A typical use case is histograms, where one usually expects no margin on the bottom edge (0) of the histogram.

Moreover, margin expansion "bumps" against sticky edges and cannot cross them. For example, if the upper data limit is 1.0, the upper view limit computed by simple margin application is 1.2, but there is a sticky edge at 1.1, then the actual upper view limit will be 1.1.

This attribute cannot be assigned to; however, the x and y lists can be modified in place as needed.

### Examples

```
>>> artist.sticky_edges.x[:] = (xmin, xmax)
>>> artist.sticky_edges.y[:] = (ymin, ymax)
```

**to_rgba**(*x*, *alpha=None*, *bytes=False*, *norm=True*)

Return a normalized RGBA array corresponding to *x*.

In the normal case, *x* is a 1D or 2D sequence of scalars, and the corresponding ndarray of RGBA values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already RGB or RGBA, such as might have been read from an image file. If *x* is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an RGB or RGBA array, and no mapping will be done. The array can be uint8, or it can be floats with values in the 0-1 range; otherwise a ValueError will be raised. Any NaNs or masked elements will be set to 0 alpha. If the last dimension is 3, the *alpha* kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the *alpha* kwarg is ignored; it does not replace the preexisting alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if *bytes* is *False* (default), the RGBA array will be floats in the 0-1 range; if it is *True*, the returned RGBA array will be uint8 in the 0 to 255 range.

If norm is False, no normalization of the input data is performed, and it is assumed to be in the range (0-1).

**update**(*props*)

Update this artist's properties from the dict *props*.

**Parameters**

**props**

[dict]

**update_from**(*other*)

Copy properties from other to self.

**update_scalarmappable**()

Update colors from the scalar mappable array, if any.

Assign colors to edges and faces based on the array and/or colors that were directly set, as appropriate.

**zorder = 0**

**class** matplotlib.collections.**PathCollection**(*paths*, *sizes=None*, ***kwargs*)

Bases: _CollectionWithSizes

A collection of *Path*s, as created by e.g. *scatter*.

**Parameters**

**paths**

[list of *path.Path*] The paths that will make up the *Collection*.

**sizes**

[array-like] The factor by which to scale each drawn *Path*. One unit squared in the Path's data space is scaled to be `sizes**2` points when rendered.

****kwargs**

Forwarded to *Collection*.

**add_callback**(*func*)

Add a callback function that will be called whenever one of the *Artist*'s properties changes.

**Parameters**

**func**

[callable] The callback function. It must have the signature:

```
def func(artist: Artist) -> Any
```

where *artist* is the calling *Artist*. Return values may exist but are ignored.

**Returns**

**int**

The observer id associated with the callback. This id can be used for removing the callback with *remove_callback* later.

**See also:**

***remove_callback***

**autoscale**()

Autoscale the scalar limits on the norm instance using the current array

**autoscale_None**()

Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

**property axes**

The *Axes* instance the artist resides in, or *None*.

**changed**()

Call this whenever the mappable is changed to notify all the callbackSM listeners to the 'changed' signal.

**colorbar**

The last colorbar associated with this ScalarMappable. May be None.

**contains**(*mouseevent*)

Test whether the mouse event occurred in the collection.

Returns `bool, dict(ind=itemlist)`, where every item in itemlist contains the event.

**convert_xunits**(*x*)

Convert *x* using the unit type of the xaxis.

If the artist is not contained in an Axes or if the xaxis does not have units, *x* itself is returned.

**convert_yunits**(*y*)

Convert *y* using the unit type of the yaxis.

If the artist is not contained in an Axes or if the yaxis does not have units, *y* itself is returned.

**draw**(*renderer*)

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

**Parameters**

**renderer**

[*RendererBase* subclass.]

### Notes

This method is overridden in the Artist subclasses.

**findobj**(*match=None*, *include_self=True*)

Find artist objects.

Recursively find all *Artist* instances contained in the artist.

**Parameters**

**match**

A filter criterion for the matches. This can be

- *None*: Return all objects contained in artist.
- A function with signature `def match(artist: Artist) -> bool`. The result will only contain artists for which the function returns *True*.
- A class instance: e.g., *Line2D*. The result will only contain artists of this class or its subclasses (`isinstance` check).

**include_self**

[bool] Include *self* in the list to be checked for a match.

**Returns**

**list of *Artist***

**format_cursor_data**(*data*)

Return a string representation of *data*.

---

**Note:** This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

---

The default implementation converts ints and floats and arrays of ints and floats into a comma-separated string enclosed in square brackets, unless the artist has an associated colorbar, in which case scalar values are formatted using the colorbar's formatter.

**See also:**

***get_cursor_data***

**get_aa**()

Alias for *get_antialiased*.

**get_agg_filter**()

Return filter function to be used for agg filter.

**get_alpha**()

Return the alpha value used for blending - not supported on all backends.

**get_animated**()

Return whether the artist is animated.

**get_antialiased**()

Get the antialiasing state for rendering.

**Returns**

**array of bools**

**get_antialiaseds**()

Alias for *get_antialiased*.

**get_array**()

Return the array of values, that are mapped to colors.

The base class *ScalarMappable* does not make any assumptions on the dimensionality and shape of the array.

**get_capstyle**()

Return the cap style for the collection (for all its elements).

**Returns**

**{'butt', 'projecting', 'round'} or None**

**get_children**()

Return a list of the child *Artist*s of this *Artist*.

**get_clim**()

Return the values (min, max) that are mapped to the colormap limits.

**get_clip_box**()

Return the clipbox.

**get_clip_on**()

Return whether the artist uses clipping.

**get_clip_path**()

Return the clip path.

**get_cmap**()

Return the *Colormap* instance.

**get_cursor_data**(*event*)

Return the cursor data for a given event.

---

**Note:** This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

---

Cursor data can be used by Artists to provide additional context information for a given event. The default implementation just returns *None*.

Subclasses can override the method and return arbitrary data. However, when doing so, they must ensure that *format_cursor_data* can convert the data to a string representation.

The only current use case is displaying the z-value of an *AxesImage* in the status bar of a plot window, while moving the mouse.

**Parameters**

**event**

[*MouseEvent*]

**See also:**

***format_cursor_data***

**get_dashes**()
Alias for *get_linestyle*.

**get_datalim**(*transData*)

**get_ec**()
Alias for *get_edgecolor*.

**get_edgecolor**()

**get_edgecolors**()
Alias for *get_edgecolor*.

**get_facecolor**()

**get_facecolors**()
Alias for *get_facecolor*.

**get_fc**()
Alias for *get_facecolor*.

**get_figure**()
Return the *Figure* instance the artist belongs to.

**get_fill**()
Return whether face is colored.

**get_gid**()
Return the group id.

**get_hatch**()
Return the current hatching pattern.

**get_in_layout**()
Return boolean flag, `True` if artist is included in layout calculations.

E.g. *Constrained layout guide*, *`Figure.tight_layout()`*, and `fig.savefig(fname, bbox_inches='tight')`.

**get_joinstyle**()
Return the join style for the collection (for all its elements).

**Returns**

**{'miter', 'round', 'bevel'} or None**

**get_label**()

Return the label used for this artist in the legend.

**get_linestyle**()

**get_linestyles**()

Alias for *get_linestyle*.

**get_linewidth**()

**get_linewidths**()

Alias for *get_linewidth*.

**get_ls**()

Alias for *get_linestyle*.

**get_lw**()

Alias for *get_linewidth*.

**get_mouseover**()

Return whether this artist is queried for custom context information when the mouse cursor moves over it.

**get_offset_transform**()

Return the *Transform* instance used by this artist offset.

**get_offsets**()

Return the offsets for the collection.

**get_path_effects**()

**get_paths**()

**get_picker**()

Return the picking behavior of the artist.

The possible values are described in *Artist.set_picker*.

**See also:**

***Artist.set_picker***, ***Artist.pickable***, ***Artist.pick***

**get_pickradius**()

**get_rasterized**()

Return whether the artist is to be rasterized.

**get_sizes**()

Return the sizes ('areas') of the elements in the collection.

**Returns**

**array**

The 'area' of each element.

**get_sketch_params**()

Return the sketch parameters for the artist.

**Returns**

**tuple or None**

A 3-tuple with the following elements:

- *scale*: The amplitude of the wiggle perpendicular to the source line.
- *length*: The length of the wiggle along the line.
- *randomness*: The scale factor by which the length is shrunken or expanded.

Returns *None* if no sketch parameters were set.

**get_snap**()

Return the snap setting.

See *set_snap* for details.

**get_tightbbox**(*renderer=None*)

Like *Artist.get_window_extent*, but includes any clipping.

**Parameters**

**renderer**

[*RendererBase* subclass, optional] renderer that will be used to draw the figures (i.e. `fig.canvas.get_renderer()`)

**Returns**

***Bbox* or None**

The enclosing bounding box (in figure pixel coordinates). Returns None if clipping results in no intersection.

**get_transOffset**()

Alias for *get_offset_transform*.

**get_transform**()

Return the *Transform* instance used by this artist.

**get_transformed_clip_path_and_affine**()

Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

**get_transforms**()

**get_url**()

Return the url.

**get_urls**()

Return a list of URLs, one for each element of the collection.

The list contains *None* for elements without a URL. See *Hyperlinks* for an example.

**get_visible**()

Return the visibility.

**get_window_extent**(*renderer=None*)

Get the artist's bounding box in display space.

The bounding box' width and height are nonnegative.

Subclasses should override for inclusion in the bounding box "tight" calculation. Default is to return an empty bounding box at 0, 0.

Be careful when using this function, the results will not update if the artist window extent of the artist changes. The extent can change due to any changes in the transform stack, such as changing the Axes limits, the figure size, or the canvas used (as is done when saving a figure). This can lead to unexpected behavior where interactive figures will look fine on the screen, but will save incorrectly.

**get_zorder**()

Return the artist's zorder.

**have_units**()

Return whether units are set on any axis.

**is_transform_set**()

Return whether the Artist has an explicitly set transform.

This is *True* after *set_transform* has been called.

**legend_elements**(*prop='colors'*, *num='auto'*, *fmt=None*, *func=<function PathCollection.<lambda>>*, ***kwargs*)

Create legend handles and labels for a PathCollection.

Each legend handle is a *Line2D* representing the Path that was drawn, and each label is a string that represents the Path.

This is useful for obtaining a legend for a *scatter* plot; e.g.:

```
scatter = plt.scatter([1, 2, 3],  [4, 5, 6],  c=[7, 2, 3], num=None)
plt.legend(*scatter.legend_elements())
```

creates three legend elements, one for each color with the numerical values passed to *c* as the labels.

Also see the *Automated legend creation* example.

**Parameters**

**prop**

[{"colors", "sizes"}, default: "colors"] If "colors", the legend handles will show the different colors of the collection. If "sizes", the legend will show the different sizes. To set both, use *kwargs* to directly edit the `Line2D` properties.

**num**

[int, None, "auto" (default), array-like, or `Locator`] Target number of elements to create. If None, use all unique elements of the mappable array. If an integer, target to use *num* elements in the normed range. If *"auto"*, try to determine which option better suits the nature of the data. The number of created elements may slightly deviate from *num* due to a `Locator` being used to find useful locations. If a list or array, use exactly those elements for the legend. Finally, a `Locator` can be provided.

**fmt**

[str, `Formatter`, or None (default)] The format or formatter to use for the labels. If a string must be a valid input for a `StrMethodFormatter`. If None (the default), use a `ScalarFormatter`.

**func**

[function, default: `lambda x: x`] Function to calculate the labels. Often the size (or color) argument to `scatter` will have been pre-processed by the user using a function `s = f(x)` to make the markers visible; e.g. `size = np.log10(x)`. Providing the inverse of this function here allows that pre-processing to be inverted, so that the legend labels have the correct values; e.g. `func = lambda x: 10**x`.

****kwargs**

Allowed keyword arguments are *color* and *size*. E.g. it may be useful to set the color of the markers if *prop="sizes"* is used; similarly to set the size of the markers if *prop="colors"* is used. Any further parameters are passed onto the `Line2D` instance. This may be useful to e.g. specify a different *markeredgecolor* or *alpha* for the legend handles.

**Returns**

**handles**

[list of `Line2D`] Visual representation of each element of the legend.

**labels**

[list of str] The string labels for elements of the legend.

**property mouseover**

Return whether this artist is queried for custom context information when the mouse cursor moves over it.

**property norm**

**pchanged**()

Call all of the registered callbacks.

This function is triggered internally when a property is changed.

**See also:**

*add_callback*
*remove_callback*

**pick**(*mouseevent*)

Process a pick event.

Each child artist will fire a pick event if *mouseevent* is over the artist and the artist has picker set.

**See also:**

*Artist.set_picker*, *Artist.get_picker*, *Artist.pickable*

**pickable**()

Return whether the artist is pickable.

**See also:**

*Artist.set_picker*, *Artist.get_picker*, *Artist.pick*

**properties**()

Return a dictionary of all the properties of the artist.

**remove**()

Remove the artist from the figure if possible.

The effect will not be visible until the figure is redrawn, e.g., with *FigureCanvasBase.draw_idle*. Call *relim* to update the Axes limits if desired.

Note: *relim* will not see collections even if the collection was added to the Axes with *autolim* = True.

Note: there is no support for removing the artist's legend entry.

**remove_callback**(*oid*)

Remove a callback based on its observer id.

**See also:**

*add_callback*

**set** (*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *antialiased=<UNSET>*, *array=<UNSET>*, *capstyle=<UNSET>*, *clim=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *cmap=<UNSET>*, *color=<UNSET>*, *edgecolor=<UNSET>*, *facecolor=<UNSET>*, *gid=<UNSET>*, *hatch=<UNSET>*, *in_layout=<UNSET>*, *joinstyle=<UNSET>*, *label=<UNSET>*, *linestyle=<UNSET>*, *linewidth=<UNSET>*, *mouseover=<UNSET>*, *norm=<UNSET>*, *offset_transform=<UNSET>*, *offsets=<UNSET>*, *path_effects=<UNSET>*, *paths=<UNSET>*, *picker=<UNSET>*, *pickradius=<UNSET>*, *rasterized=<UNSET>*, *sizes=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *urls=<UNSET>*, *visible=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi v |
| *alpha* | array-like or scalar or None |
| *animated* | bool |
| *antialiased* or aa or antialiaseds | bool or list of bools |
| *array* | array-like or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clim* | (vmin: float, vmax: float) |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *cmap* | *Colormap* or str or None |
| *color* | *color* or list of RGBA tuples |
| *edgecolor* or ec or edgecolors | *color* or list of *color* or 'face' |
| *facecolor* or facecolors or fc | *color* or list of *color* |
| *figure* | *Figure* |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or dashes or linestyles or ls | str or tuple or list thereof |
| *linewidth* or linewidths or lw | float or list of floats |
| *mouseover* | bool |
| *norm* | *Normalize* or str or None |
| *offset_transform* or transOffset | *Transform* |
| *offsets* | (N, 2) or (2,) array-like |
| *path_effects* | list of *AbstractPathEffect* |
| *paths* | unknown |
| *picker* | None or bool or float or callable |
| *pickradius* | float |
| *rasterized* | bool |

Table 40 – continued from p

| Property | Description |
|---|---|
| sizes | numpy.ndarray or None |
| sketch_params | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| transform | Transform |
| url | str |
| urls | list of str or None |
| visible | bool |
| zorder | float |

**set_aa**(*aa*)

Alias for *set_antialiased*.

**set_agg_filter**(*filter_func*)

Set the agg filter.

**Parameters**

**filter_func**

[callable] A filter function, which takes a (m, n, depth) float array and a dpi value, and returns a (m, n, depth) array and two offsets from the bottom left corner of the image

**set_alpha**(*alpha*)

Set the alpha value used for blending - not supported on all backends.

**Parameters**

**alpha**

[array-like or scalar or None] All values must be within the 0-1 range, inclusive. Masked values and nans are not supported.

**set_animated**(*b*)

Set whether the artist is intended to be used in an animation.

If True, the artist is excluded from regular drawing of the figure. You have to call *Figure.draw_artist* / *Axes.draw_artist* explicitly on the artist. This approach is used to speed up animations using blitting.

See also *matplotlib.animation* and *Faster rendering by using blitting*.

**Parameters**

**b**

[bool]

**set_antialiased**(*aa*)

Set the antialiasing state for rendering.

**Parameters**

**aa**

[bool or list of bools]

**set_antialiaseds**(*aa*)

Alias for *set_antialiased*.

**set_array**(*A*)

Set the value array from array-like *A*.

**Parameters**

**A**

[array-like or None] The values that are mapped to colors.

The base class *ScalarMappable* does not make any assumptions on the dimensionality and shape of the value array *A*.

**set_capstyle**(*cs*)

Set the *CapStyle* for the collection (for all its elements).

**Parameters**

**cs**

[*CapStyle* or {'butt', 'projecting', 'round'}]

**set_clim**(*vmin=None*, *vmax=None*)

Set the norm limits for image scaling.

**Parameters**

**vmin, vmax**

[float] The limits.

The limits may also be passed as a tuple (*vmin*, *vmax*) as a single positional argument.

**set_clip_box**(*clipbox*)

Set the artist's clip *Bbox*.

**Parameters**

**clipbox**

[*BboxBase* or None] Will typically be created from a *TransformedBbox*. For instance, `TransformedBbox(Bbox([[0, 0], [1, 1]]), ax.transAxes)` is the default clipping for an artist added to an Axes.

**set_clip_on**(*b*)

Set whether the artist uses clipping.

When False, artists will be visible outside the Axes which can lead to unexpected results.

**Parameters**

**b**

[bool]

**set_clip_path**(*path*, *transform=None*)

Set the artist's clip path.

**Parameters**

**path**

[*Patch* or *Path* or *TransformedPath* or None] The clip path. If given a *Path*, *transform* must be provided as well. If *None*, a previously set clip path is removed.

**transform**

[*Transform*, optional] Only used if *path* is a *Path*, in which case the given *Path* is converted to a *TransformedPath* using *transform*.

### Notes

For efficiency, if *path* is a *Rectangle* this method will set the clipping box to the corresponding rectangle and set the clipping path to `None`.

For technical reasons (support of *set*), a tuple (*path*, *transform*) is also accepted as a single positional parameter.

**set_cmap**(*cmap*)

Set the colormap for luminance data.

**Parameters**

**cmap**

[*Colormap* or str or None]

**set_color**(*c*)

Set both the edgecolor and the facecolor.

**Parameters**

**c**

[*color* or list of RGBA tuples]

**See also:**

***Collection.set_facecolor*, *Collection.set_edgecolor***

For setting the edge or face color individually.

**set_dashes**(*ls*)

Alias for *set_linestyle*.

**set_ec**(*c*)

Alias for *set_edgecolor*.

**set_edgecolor**(*c*)

Set the edgecolor(s) of the collection.

**Parameters**

**c**

[*color* or list of *color* or 'face'] The collection edgecolor(s). If a sequence, the patches cycle through it. If 'face', match the facecolor.

**set_edgecolors**(*c*)

Alias for *set_edgecolor*.

**set_facecolor**(*c*)

Set the facecolor(s) of the collection. *c* can be a color (all patches have same color), or a sequence of colors; if it is a sequence the patches will cycle through the sequence.

If *c* is 'none', the patch will not be filled.

**Parameters**

**c**

[*color* or list of *color*]

**set_facecolors**(*c*)

Alias for *set_facecolor*.

**set_fc**(*c*)

Alias for *set_facecolor*.

**set_figure**(*fig*)

Set the *Figure* instance the artist belongs to.

**Parameters**

**fig**

[*Figure*]

**set_gid**(*gid*)

Set the (group) id for the artist.

**Parameters**

**gid**

[str]

**set_hatch**(*hatch*)

Set the hatching pattern

*hatch* can be one of:

```
/   - diagonal hatching
\   - back diagonal
|   - vertical
-   - horizontal
+   - crossed
x   - crossed diagonal
o   - small circle
O   - large circle
.   - dots
*   - stars
```

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

**Parameters**

**hatch**

[{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}]

**set_in_layout**(*in_layout*)

Set if artist is to be included in layout calculations, E.g. *Constrained layout guide*, *Figure.tight_layout()*, and `fig.savefig(fname, bbox_inches='tight')`.

**Parameters**

**in_layout**

[bool]

**set_joinstyle**(*js*)

Set the *JoinStyle* for the collection (for all its elements).

**Parameters**

**js**

[*JoinStyle* or {'miter', 'round', 'bevel'}]

**set_label**(*s*)

Set a label that will be displayed in the legend.

**Parameters**

**s**

[object] *s* will be converted to a string by calling `str`.

**set_linestyle**(*ls*)

Set the linestyle(s) for the collection.

| linestyle | description |
|---|---|
| `'-'` or `'solid'` | solid line |
| `'--'` or `'dashed'` | dashed line |
| `'-.'` or `'dashdot'` | dash-dotted line |
| `':'` or `'dotted'` | dotted line |

Alternatively a dash tuple of the following form can be provided:

```
(offset, onoffseq),
```

where `onoffseq` is an even length tuple of on and off ink in points.

**Parameters**

**ls**

[str or tuple or list thereof] Valid values for individual linestyles include {'-', '--', '-.', ':', '', (offset, on-off-seq)}. See *Line2D.set_linestyle* for a complete description.

**set_linestyles**(*ls*)

Alias for *set_linestyle*.

**set_linewidth**(*lw*)

Set the linewidth(s) for the collection. *lw* can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

**Parameters**

**lw**

[float or list of floats]

**set_linewidths**(*lw*)

Alias for *set_linewidth*.

**set_ls**(*ls*)

Alias for *set_linestyle*.

**set_lw**(*lw*)

Alias for *set_linewidth*.

**set_mouseover**(*mouseover*)

Set whether this artist is queried for custom context information when the mouse cursor moves over it.

**Parameters**

**mouseover**

[bool]

**See also:**

***get_cursor_data***
***ToolCursorPosition***
***NavigationToolbar2***

**set_norm**(*norm*)

Set the normalization instance.

**Parameters**

**norm**

[*Normalize* or str or None]

### Notes

If there are any colorbars using the mappable for this norm, setting the norm of the mappable will reset the norm, locator, and formatters on the colorbar to default.

**set_offset_transform**(*offset_transform*)

Set the artist offset transform.

**Parameters**

**offset_transform**

[*Transform*]

**set_offsets**(*offsets*)

Set the offsets for the collection.

**Parameters**

**offsets**

[(N, 2) or (2,) array-like]

**set_path_effects**(*path_effects*)

Set the path effects.

**Parameters**

**path_effects**

[list of *AbstractPathEffect*]

**set_paths**(*paths*)

**set_picker**(*picker*)

Define the picking behavior of the artist.

**Parameters**

**picker**

[None or bool or float or callable] This can be one of the following:

- *None*: Picking is disabled for this artist (default).
- A boolean: If *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist.
- A float: If picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if its data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event

- A function: If picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

to determine the hit test. if the mouse event is over the artist, return *hit=True* and props is a dictionary of properties you want added to the PickEvent attributes.

**set_pickradius**(*pickradius*)

Set the pick radius used for containment tests.

**Parameters**

**pickradius**

[float] Pick radius, in points.

**set_rasterized**(*rasterized*)

Force rasterized (bitmap) drawing for vector graphics output.

Rasterized drawing is not supported by all artists. If you try to enable this on an artist that does not support it, the command has no effect and a warning will be issued.

This setting is ignored for pixel-based output.

See also *Rasterization for vector graphics*.

**Parameters**

**rasterized**

[bool]

**set_sizes**(*sizes*, *dpi=72.0*)

Set the sizes of each member of the collection.

**Parameters**

**sizes**

[`numpy.ndarray` or None] The size to set for each element of the collection. The value is the 'area' of the element.

**dpi**

[float, default: 72] The dpi of the canvas.

**set_sketch_params**(*scale=None*, *length=None*, *randomness=None*)

Set the sketch parameters.

**Parameters**

**scale**

[float, optional] The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is `None`, or not provided, no sketch filter will be provided.

**length**

[float, optional] The length of the wiggle along the line, in pixels (default 128.0)

**randomness**

[float, optional] The scale factor by which the length is shrunken or expanded (default 16.0)

The PGF backend uses this argument as an RNG seed and not as described above. Using the same seed yields the same random shape.

**set_snap**(*snap*)

Set the snapping behavior.

Snapping aligns positions with the pixel grid, which results in clearer images. For example, if a black line of 1px width was defined at a position in between two pixels, the resulting image would contain the interpolated value of that line in the pixel grid, which would be a grey value on both adjacent pixel positions. In contrast, snapping will move the line to the nearest integer pixel value, so that the resulting image will really contain a 1px wide black line.

Snapping is currently only supported by the Agg and MacOSX backends.

**Parameters**

**snap**

[bool or None] Possible values:

- *True*: Snap vertices to the nearest pixel center.
- *False*: Do not modify vertex positions.
- *None*: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center.

**set_transOffset**(*offset_transform*)

Alias for *set_offset_transform*.

**set_transform**(*t*)

Set the artist transform.

**Parameters**

**t**

[*Transform*]

**set_url**(*url*)

Set the url for the artist.

**Parameters**

**url**

[str]

**set_urls**(*urls*)

**Parameters**

**urls**

[list of str or None]

### Notes

URLs are currently only implemented by the SVG backend. They are ignored by all other backends.

**set_visible**(*b*)

Set the artist's visibility.

**Parameters**

**b**

[bool]

**set_zorder**(*level*)

Set the zorder for the artist. Artists with lower zorder values are drawn first.

**Parameters**

**level**

[float]

**property stale**

Whether the artist is 'stale' and needs to be re-drawn for the output to match the internal state of the artist.

**property sticky_edges**

`x` and `y` sticky edge lists for autoscaling.

When performing autoscaling, if a data limit coincides with a value in the corresponding sticky_edges list, then no margin will be added--the view limit "sticks" to the edge. A typical use case is histograms, where one usually expects no margin on the bottom edge (0) of the histogram.

Moreover, margin expansion "bumps" against sticky edges and cannot cross them. For example, if the upper data limit is 1.0, the upper view limit computed by simple margin application is 1.2, but there is a sticky edge at 1.1, then the actual upper view limit will be 1.1.

This attribute cannot be assigned to; however, the `x` and `y` lists can be modified in place as needed.

### Examples

```
>>> artist.sticky_edges.x[:] = (xmin, xmax)
>>> artist.sticky_edges.y[:] = (ymin, ymax)
```

**to_rgba**(*x*, *alpha=None*, *bytes=False*, *norm=True*)

Return a normalized RGBA array corresponding to *x*.

In the normal case, *x* is a 1D or 2D sequence of scalars, and the corresponding `ndarray` of RGBA values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already RGB or RGBA, such as might have been read from an image file. If *x* is an `ndarray` with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an RGB or RGBA array, and no mapping will be done. The array can be `uint8`, or it can be floats with values in the 0-1 range; otherwise a ValueError will be raised. Any NaNs or masked elements will be set to 0 alpha. If the last dimension is 3, the *alpha* kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the *alpha* kwarg is ignored; it does not replace the preexisting alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if *bytes* is *False* (default), the RGBA array will be floats in the 0-1 range; if it is *True*, the returned RGBA array will be `uint8` in the 0 to 255 range.

If norm is False, no normalization of the input data is performed, and it is assumed to be in the range (0-1).

**update**(*props*)

Update this artist's properties from the dict *props*.

**Parameters**

**props**

[dict]

**update_from**(*other*)

Copy properties from other to self.

**update_scalarmappable**()

Update colors from the scalar mappable array, if any.

Assign colors to edges and faces based on the array and/or colors that were directly set, as appropriate.

**zorder = 0**

**class** matplotlib.collections.**PolyCollection**(*verts*, *sizes=None*, *, *closed=True*, ***kwargs*)

Bases: _CollectionWithSizes

**Parameters**

**verts**

[list of array-like] The sequence of polygons [*verts0*, *verts1*, ...] where each element *verts_i* defines the vertices of polygon *i* as a 2D array-like of shape (M, 2).

**sizes**

[array-like, default: None] Squared scaling factors for the polygons. The coordinates of each polygon *verts_i* are multiplied by the square-root of the corresponding entry in *sizes* (i.e., *sizes* specify the scaling of areas). The scaling is applied before the Artist master transform.

**closed**

[bool, default: True] Whether the polygon should be closed by adding a CLOSEPOLY connection at the end.

****kwargs**

Forwarded to *Collection*.

**add_callback**(*func*)

Add a callback function that will be called whenever one of the *Artist*'s properties changes.

**Parameters**

**func**

[callable] The callback function. It must have the signature:

```
def func(artist: Artist) -> Any
```

where *artist* is the calling *Artist*. Return values may exist but are ignored.

**Returns**

**int**

The observer id associated with the callback. This id can be used for removing the callback with *remove_callback* later.

**See also:**

***remove_callback***

**autoscale**()

Autoscale the scalar limits on the norm instance using the current array

**autoscale_None**()

Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

**property axes**

The *Axes* instance the artist resides in, or *None*.

**changed**()

Call this whenever the mappable is changed to notify all the callbackSM listeners to the 'changed' signal.

**colorbar**

The last colorbar associated with this ScalarMappable. May be None.

**contains**(*mouseevent*)

Test whether the mouse event occurred in the collection.

Returns `bool, dict(ind=itemlist)`, where every item in itemlist contains the event.

**convert_xunits**(*x*)

Convert *x* using the unit type of the xaxis.

If the artist is not contained in an Axes or if the xaxis does not have units, *x* itself is returned.

**convert_yunits**(*y*)

Convert *y* using the unit type of the yaxis.

If the artist is not contained in an Axes or if the yaxis does not have units, *y* itself is returned.

**draw**(*renderer*)

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

**Parameters**

**renderer**

[*RendererBase* subclass.]

### Notes

This method is overridden in the Artist subclasses.

**findobj**(*match=None*, *include_self=True*)

Find artist objects.

Recursively find all *Artist* instances contained in the artist.

**Parameters**

**match**

A filter criterion for the matches. This can be

- *None*: Return all objects contained in artist.
- A function with signature `def match(artist: Artist) -> bool`. The result will only contain artists for which the function returns *True*.
- A class instance: e.g., *Line2D*. The result will only contain artists of this class or its subclasses (`isinstance` check).

**include_self**

[bool] Include *self* in the list to be checked for a match.

**Returns**

**list of *Artist***

**format_cursor_data**(*data*)

Return a string representation of *data*.

---

**Note:** This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

---

The default implementation converts ints and floats and arrays of ints and floats into a comma-separated string enclosed in square brackets, unless the artist has an associated colorbar, in which case scalar values are formatted using the colorbar's formatter.

**See also:**

***get_cursor_data***

**get_aa**()

Alias for *get_antialiased*.

**get_agg_filter**()

Return filter function to be used for agg filter.

**get_alpha**()

Return the alpha value used for blending - not supported on all backends.

**get_animated**()

Return whether the artist is animated.

**get_antialiased**()

Get the antialiasing state for rendering.

**Returns**

**array of bools**

**get_antialiaseds**()

Alias for *get_antialiased*.

**get_array**()

Return the array of values, that are mapped to colors.

The base class *ScalarMappable* does not make any assumptions on the dimensionality and shape of the array.

**get_capstyle**()

Return the cap style for the collection (for all its elements).

**Returns**

**{'butt', 'projecting', 'round'} or None**

**get_children**()

Return a list of the child *Artist*s of this *Artist*.

**get_clim**()

Return the values (min, max) that are mapped to the colormap limits.

**get_clip_box**()

Return the clipbox.

**get_clip_on**()

Return whether the artist uses clipping.

**get_clip_path**()

Return the clip path.

**get_cmap**()

Return the *Colormap* instance.

**get_cursor_data**(*event*)

Return the cursor data for a given event.

---

**Note:** This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

---

Cursor data can be used by Artists to provide additional context information for a given event. The default implementation just returns *None*.

Subclasses can override the method and return arbitrary data. However, when doing so, they must ensure that *format_cursor_data* can convert the data to a string representation.

The only current use case is displaying the z-value of an *AxesImage* in the status bar of a plot window, while moving the mouse.

**Parameters**

**event**

[*MouseEvent*]

**See also:**

***format_cursor_data***

**get_dashes**()
Alias for *get_linestyle*.

**get_datalim**(*transData*)

**get_ec**()
Alias for *get_edgecolor*.

**get_edgecolor**()

**get_edgecolors**()
Alias for *get_edgecolor*.

**get_facecolor**()

**get_facecolors**()
Alias for *get_facecolor*.

**get_fc**()
Alias for *get_facecolor*.

**get_figure**()
Return the *Figure* instance the artist belongs to.

**get_fill**()
Return whether face is colored.

**get_gid**()
Return the group id.

**get_hatch**()
Return the current hatching pattern.

**get_in_layout**()
Return boolean flag, `True` if artist is included in layout calculations.

E.g. *Constrained layout guide*, *`Figure.tight_layout()`*, and `fig.savefig(fname, bbox_inches='tight')`.

**get_joinstyle**()
Return the join style for the collection (for all its elements).

**Returns**

**{'miter', 'round', 'bevel'} or None**

**get_label**()

Return the label used for this artist in the legend.

**get_linestyle**()

**get_linestyles**()

Alias for *get_linestyle*.

**get_linewidth**()

**get_linewidths**()

Alias for *get_linewidth*.

**get_ls**()

Alias for *get_linestyle*.

**get_lw**()

Alias for *get_linewidth*.

**get_mouseover**()

Return whether this artist is queried for custom context information when the mouse cursor moves over it.

**get_offset_transform**()

Return the *Transform* instance used by this artist offset.

**get_offsets**()

Return the offsets for the collection.

**get_path_effects**()

**get_paths**()

**get_picker**()

Return the picking behavior of the artist.

The possible values are described in *Artist.set_picker*.

**See also:**

***Artist.set_picker***, ***Artist.pickable***, ***Artist.pick***

**get_pickradius**()

**get_rasterized**()

Return whether the artist is to be rasterized.

**get_sizes**()

Return the sizes ('areas') of the elements in the collection.

**Returns**

**array**

The 'area' of each element.

**get_sketch_params**()

Return the sketch parameters for the artist.

**Returns**

**tuple or None**

A 3-tuple with the following elements:

- *scale*: The amplitude of the wiggle perpendicular to the source line.
- *length*: The length of the wiggle along the line.
- *randomness*: The scale factor by which the length is shrunken or expanded.

Returns *None* if no sketch parameters were set.

**get_snap**()

Return the snap setting.

See *set_snap* for details.

**get_tightbbox**(*renderer=None*)

Like *Artist.get_window_extent*, but includes any clipping.

**Parameters**

**renderer**

[*RendererBase* subclass, optional] renderer that will be used to draw the figures (i.e. `fig.canvas.get_renderer()`)

**Returns**

***Bbox* or None**

The enclosing bounding box (in figure pixel coordinates). Returns None if clipping results in no intersection.

**get_transOffset**()

Alias for *get_offset_transform*.

**get_transform**()

Return the *Transform* instance used by this artist.

**get_transformed_clip_path_and_affine**()

Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

**get_transforms**()

**get_url**()

Return the url.

**get_urls**()

Return a list of URLs, one for each element of the collection.

The list contains *None* for elements without a URL. See *Hyperlinks* for an example.

**get_visible**()

Return the visibility.

**get_window_extent**(*renderer=None*)

Get the artist's bounding box in display space.

The bounding box' width and height are nonnegative.

Subclasses should override for inclusion in the bounding box "tight" calculation. Default is to return an empty bounding box at 0, 0.

Be careful when using this function, the results will not update if the artist window extent of the artist changes. The extent can change due to any changes in the transform stack, such as changing the Axes limits, the figure size, or the canvas used (as is done when saving a figure). This can lead to unexpected behavior where interactive figures will look fine on the screen, but will save incorrectly.

**get_zorder**()

Return the artist's zorder.

**have_units**()

Return whether units are set on any axis.

**is_transform_set**()

Return whether the Artist has an explicitly set transform.

This is *True* after *set_transform* has been called.

**property mouseover**

Return whether this artist is queried for custom context information when the mouse cursor moves over it.

**property norm**

**pchanged**()

Call all of the registered callbacks.

This function is triggered internally when a property is changed.

**See also:**

***add_callback***
***remove_callback***

**pick**(*mouseevent*)

Process a pick event.

Each child artist will fire a pick event if *mouseevent* is over the artist and the artist has picker set.

**See also:**

***Artist.set_picker*, *Artist.get_picker*, *Artist.pickable***

**pickable**()

Return whether the artist is pickable.

**See also:**

***Artist.set_picker*, *Artist.get_picker*, *Artist.pick***

**properties**()

Return a dictionary of all the properties of the artist.

**remove**()

Remove the artist from the figure if possible.

The effect will not be visible until the figure is redrawn, e.g., with *FigureCanvasBase.draw_idle*. Call *relim* to update the Axes limits if desired.

Note: *relim* will not see collections even if the collection was added to the Axes with *autolim* = True.

Note: there is no support for removing the artist's legend entry.

**remove_callback**(*oid*)

Remove a callback based on its observer id.

**See also:**

***add_callback***

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *antialiased=<UNSET>*, *array=<UNSET>*, *capstyle=<UNSET>*, *clim=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *cmap=<UNSET>*, *color=<UNSET>*, *edgecolor=<UNSET>*, *facecolor=<UNSET>*, *gid=<UNSET>*, *hatch=<UNSET>*, *in_layout=<UNSET>*, *joinstyle=<UNSET>*, *label=<UNSET>*, *linestyle=<UNSET>*, *linewidth=<UNSET>*, *mouseover=<UNSET>*, *norm=<UNSET>*, *offset_transform=<UNSET>*, *offsets=<UNSET>*, *path_effects=<UNSET>*, *paths=<UNSET>*, *picker=<UNSET>*, *pickradius=<UNSET>*, *rasterized=<UNSET>*, *sizes=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *urls=<UNSET>*, *verts=<UNSET>*, *verts_and_codes=<UNSET>*, *visible=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi v |
| *alpha* | array-like or scalar or None |
| *animated* | bool |
| *antialiased* or aa or antialiaseds | bool or list of bools |
| *array* | array-like or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clim* | (vmin: float, vmax: float) |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *cmap* | *Colormap* or str or None |
| *color* | *color* or list of RGBA tuples |
| *edgecolor* or ec or edgecolors | *color* or list of *color* or 'face' |
| *facecolor* or facecolors or fc | *color* or list of *color* |
| *figure* | *Figure* |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or dashes or linestyles or ls | str or tuple or list thereof |
| *linewidth* or linewidths or lw | float or list of floats |
| *mouseover* | bool |
| *norm* | *Normalize* or str or None |
| *offset_transform* or transOffset | *Transform* |
| *offsets* | (N, 2) or (2,) array-like |
| *path_effects* | list of *AbstractPathEffect* |
| *paths* | list of array-like |
| *picker* | None or bool or float or callable |
| *pickradius* | float |
| *rasterized* | bool |
| sizes | numpy.ndarray or None |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *urls* | list of str or None |
| *verts* | list of array-like |
| *verts_and_codes* | unknown |
| *visible* | bool |
| *zorder* | float |

**set_aa**(*aa*)

Alias for *set_antialiased*.

**set_agg_filter**(*filter_func*)

Set the agg filter.

**Parameters**

**filter_func**

[callable] A filter function, which takes a (m, n, depth) float array and a dpi value, and returns a (m, n, depth) array and two offsets from the bottom left corner of the image

**set_alpha**(*alpha*)

Set the alpha value used for blending - not supported on all backends.

**Parameters**

**alpha**

[array-like or scalar or None] All values must be within the 0-1 range, inclusive. Masked values and nans are not supported.

**set_animated**(*b*)

Set whether the artist is intended to be used in an animation.

If True, the artist is excluded from regular drawing of the figure. You have to call *Figure.draw_artist* / *Axes.draw_artist* explicitly on the artist. This approach is used to speed up animations using blitting.

See also *matplotlib.animation* and *Faster rendering by using blitting*.

**Parameters**

**b**

[bool]

**set_antialiased**(*aa*)

Set the antialiasing state for rendering.

**Parameters**

**aa**

[bool or list of bools]

**set_antialiaseds**(*aa*)

Alias for *set_antialiased*.

**set_array**(*A*)

Set the value array from array-like *A*.

**Parameters**

**A**

[array-like or None] The values that are mapped to colors.

The base class *ScalarMappable* does not make any assumptions on the dimensionality and shape of the value array *A*.

**set_capstyle**(*cs*)

Set the *CapStyle* for the collection (for all its elements).

**Parameters**

**cs**

[*CapStyle* or {'butt', 'projecting', 'round'}]

**set_clim**(*vmin=None*, *vmax=None*)

Set the norm limits for image scaling.

**Parameters**

**vmin, vmax**

[float] The limits.

The limits may also be passed as a tuple (*vmin*, *vmax*) as a single positional argument.

**set_clip_box**(*clipbox*)

Set the artist's clip *Bbox*.

**Parameters**

**clipbox**

[*BboxBase* or None] Will typically be created from a *TransformedBbox*. For instance, `TransformedBbox(Bbox([[0, 0], [1, 1]]), ax.transAxes)` is the default clipping for an artist added to an Axes.

**set_clip_on**(*b*)

Set whether the artist uses clipping.

When False, artists will be visible outside the Axes which can lead to unexpected results.

**Parameters**

**b**

[bool]

**set_clip_path**(*path*, *transform=None*)

Set the artist's clip path.

**Parameters**

**path**

[*Patch* or *Path* or *TransformedPath* or None] The clip path. If given a *Path*, *transform* must be provided as well. If *None*, a previously set clip path is removed.

**transform**

[*Transform*, optional] Only used if *path* is a *Path*, in which case the given *Path* is converted to a *TransformedPath* using *transform*.

### Notes

For efficiency, if *path* is a *Rectangle* this method will set the clipping box to the corresponding rectangle and set the clipping path to `None`.

For technical reasons (support of *set*), a tuple (*path*, *transform*) is also accepted as a single positional parameter.

**set_cmap**(*cmap*)

Set the colormap for luminance data.

**Parameters**

**cmap**

[*Colormap* or str or None]

**set_color**(*c*)

Set both the edgecolor and the facecolor.

**Parameters**

**c**

[*color* or list of RGBA tuples]

**See also:**

***Collection.set_facecolor*, *Collection.set_edgecolor***

For setting the edge or face color individually.

**set_dashes**(*ls*)

Alias for *set_linestyle*.

**set_ec**(*c*)

Alias for *set_edgecolor*.

**set_edgecolor**(*c*)

Set the edgecolor(s) of the collection.

**Parameters**

**c**

[*color* or list of *color* or 'face'] The collection edgecolor(s). If a sequence, the patches cycle through it. If 'face', match the facecolor.

**set_edgecolors**(*c*)

Alias for *set_edgecolor*.

**set_facecolor**(*c*)

Set the facecolor(s) of the collection. *c* can be a color (all patches have same color), or a sequence of colors; if it is a sequence the patches will cycle through the sequence.

If *c* is 'none', the patch will not be filled.

**Parameters**

**c**

[*color* or list of *color*]

**set_facecolors**(*c*)

Alias for *set_facecolor*.

**set_fc**(*c*)

Alias for *set_facecolor*.

**set_figure**(*fig*)

Set the *Figure* instance the artist belongs to.

**Parameters**

**fig**

[*Figure*]

**set_gid**(*gid*)

Set the (group) id for the artist.

**Parameters**

**gid**

[str]

**set_hatch**(*hatch*)

Set the hatching pattern

*hatch* can be one of:

```
/   - diagonal hatching
\   - back diagonal
|   - vertical
-   - horizontal
+   - crossed
x   - crossed diagonal
o   - small circle
O   - large circle
.   - dots
*   - stars
```

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

**Parameters**

**hatch**

[{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}]

**set_in_layout**(*in_layout*)

Set if artist is to be included in layout calculations, E.g. *Constrained layout guide*, *Figure.tight_layout()*, and `fig.savefig(fname, bbox_inches='tight')`.

**Parameters**

**in_layout**

[bool]

**set_joinstyle**(*js*)

Set the *JoinStyle* for the collection (for all its elements).

**Parameters**

**js**

[*JoinStyle* or {'miter', 'round', 'bevel'}]

**set_label**(*s*)

Set a label that will be displayed in the legend.

**Parameters**

**s**

[object] *s* will be converted to a string by calling `str`.

**set_linestyle**(*ls*)

Set the linestyle(s) for the collection.

| linestyle | description |
|---|---|
| `'-'` or `'solid'` | solid line |
| `'--'` or `'dashed'` | dashed line |
| `'-.'` or `'dashdot'` | dash-dotted line |
| `':'` or `'dotted'` | dotted line |

Alternatively a dash tuple of the following form can be provided:

```
(offset, onoffseq),
```

where `onoffseq` is an even length tuple of on and off ink in points.

**Parameters**

**ls**

[str or tuple or list thereof] Valid values for individual linestyles include {'-', '--', '-.', ':', '', (offset, on-off-seq)}. See *Line2D.set_linestyle* for a complete description.

**set_linestyles**(*ls*)

Alias for *set_linestyle*.

**set_linewidth**(*lw*)

Set the linewidth(s) for the collection. *lw* can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

**Parameters**

**lw**

[float or list of floats]

**set_linewidths**(*lw*)

Alias for *set_linewidth*.

**set_ls**(*ls*)

Alias for *set_linestyle*.

**set_lw**(*lw*)

Alias for *set_linewidth*.

**set_mouseover**(*mouseover*)

Set whether this artist is queried for custom context information when the mouse cursor moves over it.

**Parameters**

**mouseover**

[bool]

**See also:**

*get_cursor_data*
*ToolCursorPosition*
*NavigationToolbar2*

**set_norm**(*norm*)

Set the normalization instance.

**Parameters**

**norm**

[*Normalize* or str or None]

### Notes

If there are any colorbars using the mappable for this norm, setting the norm of the mappable will reset the norm, locator, and formatters on the colorbar to default.

**set_offset_transform**(*offset_transform*)

Set the artist offset transform.

**Parameters**

**offset_transform**

[*Transform*]

**set_offsets**(*offsets*)

Set the offsets for the collection.

**Parameters**

**offsets**

[(N, 2) or (2,) array-like]

**set_path_effects**(*path_effects*)

Set the path effects.

**Parameters**

**path_effects**

[list of *AbstractPathEffect*]

**set_paths**(*verts*, *closed=True*)

Set the vertices of the polygons.

**Parameters**

**verts**

[list of array-like] The sequence of polygons [*verts0*, *verts1*, ...] where each element *verts_i* defines the vertices of polygon *i* as a 2D array-like of shape (M, 2).

**closed**

[bool, default: True] Whether the polygon should be closed by adding a CLOSEPOLY connection at the end.

**set_picker**(*picker*)

Define the picking behavior of the artist.

**Parameters**

**picker**

[None or bool or float or callable] This can be one of the following:

- *None*: Picking is disabled for this artist (default).
- A boolean: If *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist.
- A float: If picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if its data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event
- A function: If picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

to determine the hit test. if the mouse event is over the artist, return *hit=True* and props is a dictionary of properties you want added to the PickEvent attributes.

**set_pickradius**(*pickradius*)

Set the pick radius used for containment tests.

**Parameters**

**pickradius**

[float] Pick radius, in points.

**set_rasterized**(*rasterized*)

Force rasterized (bitmap) drawing for vector graphics output.

Rasterized drawing is not supported by all artists. If you try to enable this on an artist that does not support it, the command has no effect and a warning will be issued.

This setting is ignored for pixel-based output.

See also *Rasterization for vector graphics*.

**Parameters**

**rasterized**

[bool]

**set_sizes**(*sizes*, *dpi=72.0*)

Set the sizes of each member of the collection.

**Parameters**

**sizes**

[`numpy.ndarray` or None] The size to set for each element of the collection. The value is the 'area' of the element.

**dpi**

[float, default: 72] The dpi of the canvas.

**set_sketch_params**(*scale=None*, *length=None*, *randomness=None*)

Set the sketch parameters.

**Parameters**

**scale**

[float, optional] The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is `None`, or not provided, no sketch filter will be provided.

**length**

[float, optional] The length of the wiggle along the line, in pixels (default 128.0)

**randomness**

[float, optional] The scale factor by which the length is shrunken or expanded (default 16.0)

The PGF backend uses this argument as an RNG seed and not as described above. Using the same seed yields the same random shape.

**set_snap**(*snap*)

Set the snapping behavior.

Snapping aligns positions with the pixel grid, which results in clearer images. For example, if a black line of 1px width was defined at a position in between two pixels, the resulting image would contain the interpolated value of that line in the pixel grid, which would be a grey value on both adjacent pixel positions. In contrast, snapping will move the line to the nearest integer pixel value, so that the resulting image will really contain a 1px wide black line.

Snapping is currently only supported by the Agg and MacOSX backends.

**Parameters**

**snap**

[bool or None] Possible values:

- *True*: Snap vertices to the nearest pixel center.
- *False*: Do not modify vertex positions.
- *None*: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center.

**set_transOffset**(*offset_transform*)

Alias for *set_offset_transform*.

**set_transform**(*t*)

Set the artist transform.

**Parameters**

**t**

[*Transform*]

**set_url**(*url*)

Set the url for the artist.

**Parameters**

**url**

[str]

**set_urls**(*urls*)

**Parameters**

**urls**

[list of str or None]

### Notes

URLs are currently only implemented by the SVG backend. They are ignored by all other backends.

**set_verts**(*verts*, *closed=True*)

Set the vertices of the polygons.

**Parameters**

**verts**

[list of array-like] The sequence of polygons [*verts0*, *verts1*, ...] where each element *verts_i* defines the vertices of polygon *i* as a 2D array-like of shape (M, 2).

**closed**

[bool, default: True] Whether the polygon should be closed by adding a CLOSEPOLY connection at the end.

**set_verts_and_codes**(*verts*, *codes*)

Initialize vertices with path codes.

**set_visible**(*b*)

Set the artist's visibility.

**Parameters**

**b**

[bool]

**set_zorder**(*level*)

Set the zorder for the artist. Artists with lower zorder values are drawn first.

**Parameters**

**level**

[float]

**property stale**

Whether the artist is 'stale' and needs to be re-drawn for the output to match the internal state of the artist.

**property sticky_edges**

`x` and `y` sticky edge lists for autoscaling.

When performing autoscaling, if a data limit coincides with a value in the corresponding sticky_edges list, then no margin will be added--the view limit "sticks" to the edge. A typical use case is histograms, where one usually expects no margin on the bottom edge (0) of the histogram.

Moreover, margin expansion "bumps" against sticky edges and cannot cross them. For example, if the upper data limit is 1.0, the upper view limit computed by simple margin application is 1.2, but there is a sticky edge at 1.1, then the actual upper view limit will be 1.1.

This attribute cannot be assigned to; however, the `x` and `y` lists can be modified in place as needed.

### Examples

```
>>> artist.sticky_edges.x[:] = (xmin, xmax)
>>> artist.sticky_edges.y[:] = (ymin, ymax)
```

**to_rgba**(*x*, *alpha=None*, *bytes=False*, *norm=True*)

Return a normalized RGBA array corresponding to *x*.

In the normal case, *x* is a 1D or 2D sequence of scalars, and the corresponding `ndarray` of RGBA values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already RGB or RGBA, such as might have been read from an image file. If *x* is an `ndarray` with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an RGB or RGBA array, and no mapping will be done. The array can be `uint8`, or it can be floats with values in the 0-1 range; otherwise a ValueError will be raised. Any NaNs or masked elements will be set to 0 alpha. If the last dimension is 3, the *alpha* kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the *alpha* kwarg is ignored; it does not replace the preexisting alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if *bytes* is *False* (default), the RGBA array will be floats in the 0-1 range; if it is *True*, the returned RGBA array will be `uint8` in the 0 to 255 range.

If norm is False, no normalization of the input data is performed, and it is assumed to be in the range (0-1).

**update**(*props*)

Update this artist's properties from the dict *props*.

**Parameters**

**props**

[dict]

**update_from**(*other*)

Copy properties from other to self.

**update_scalarmappable**()

Update colors from the scalar mappable array, if any.

Assign colors to edges and faces based on the array and/or colors that were directly set, as appropriate.

**zorder = 0**

**class** matplotlib.collections.**PolyQuadMesh**(*coordinates*, ***kwargs*)

Bases: _MeshData, *PolyCollection*

Class for drawing a quadrilateral mesh as individual Polygons.

A quadrilateral mesh is a grid of M by N adjacent quadrilaterals that are defined via a (M+1, N+1) grid of vertices. The quadrilateral (m, n) is defined by the vertices

```
    (m+1, n) ----------- (m+1, n+1)
       /                   /
      /                   /
     /                   /
(m, n) -------- (m, n+1)
```

The mesh need not be regular and the polygons need not be convex.

**Parameters**

**coordinates**

[(M+1, N+1, 2) array-like] The vertices. `coordinates[m, n]` specifies the (x, y) coordinates of vertex (m, n).

## Notes

Unlike *QuadMesh*, this class will draw each cell as an individual Polygon. This is significantly slower, but allows for more flexibility when wanting to add additional properties to the cells, such as hatching.

Another difference from *QuadMesh* is that if any of the vertices or data of a cell are masked, that Polygon will **not** be drawn and it won't be in the list of paths returned.

**Parameters**

**verts**

[list of array-like] The sequence of polygons [*verts0*, *verts1*, ...] where each element *verts_i* defines the vertices of polygon *i* as a 2D array-like of shape (M, 2).

**sizes**

[array-like, default: None] Squared scaling factors for the polygons. The coordinates of each polygon *verts_i* are multiplied by the square-root of the corresponding entry in *sizes* (i.e., *sizes* specify the scaling of areas). The scaling is applied before the Artist master transform.

**closed**

[bool, default: True] Whether the polygon should be closed by adding a CLOSEPOLY connection at the end.

****kwargs**

Forwarded to *Collection*.

**add_callback**(*func*)

Add a callback function that will be called whenever one of the *Artist*'s properties changes.

**Parameters**

**func**

[callable] The callback function. It must have the signature:

```
def func(artist: Artist) -> Any
```

where *artist* is the calling *Artist*. Return values may exist but are ignored.

**Returns**

**int**

The observer id associated with the callback. This id can be used for removing the callback with *remove_callback* later.

**See also:**

***remove_callback***

**autoscale**()

Autoscale the scalar limits on the norm instance using the current array

**autoscale_None**()

Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

**property axes**

The *Axes* instance the artist resides in, or *None*.

**changed**()

Call this whenever the mappable is changed to notify all the callbackSM listeners to the 'changed' signal.

**colorbar**

The last colorbar associated with this ScalarMappable. May be None.

**contains**(*mouseevent*)

Test whether the mouse event occurred in the collection.

Returns `bool, dict(ind=itemlist)`, where every item in itemlist contains the event.

**convert_xunits**(*x*)

Convert *x* using the unit type of the xaxis.

If the artist is not contained in an Axes or if the xaxis does not have units, *x* itself is returned.

**convert_yunits**(*y*)

Convert *y* using the unit type of the yaxis.

If the artist is not contained in an Axes or if the yaxis does not have units, *y* itself is returned.

**draw**(*renderer*)

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

**Parameters**

**renderer**

[*RendererBase* subclass.]

### Notes

This method is overridden in the Artist subclasses.

**findobj**(*match=None*, *include_self=True*)

Find artist objects.

Recursively find all *Artist* instances contained in the artist.

**Parameters**

**match**

A filter criterion for the matches. This can be

- *None*: Return all objects contained in artist.
- A function with signature `def match(artist: Artist) -> bool`. The result will only contain artists for which the function returns *True*.
- A class instance: e.g., *Line2D*. The result will only contain artists of this class or its subclasses (`isinstance` check).

**include_self**

[bool] Include *self* in the list to be checked for a match.

**Returns**

**list of *Artist***

**format_cursor_data**(*data*)

Return a string representation of *data*.

---

**Note:** This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

---

The default implementation converts ints and floats and arrays of ints and floats into a comma-separated string enclosed in square brackets, unless the artist has an associated colorbar, in which case scalar values are formatted using the colorbar's formatter.

**See also:**

*get_cursor_data*

**get_aa**()

Alias for *get_antialiased*.

**get_agg_filter**()

Return filter function to be used for agg filter.

**get_alpha**()

Return the alpha value used for blending - not supported on all backends.

**get_animated**()

Return whether the artist is animated.

**get_antialiased**()

Get the antialiasing state for rendering.

**Returns**

**array of bools**

**get_antialiaseds**()

Alias for *get_antialiased*.

**get_array**()

Return the array of values, that are mapped to colors.

The base class *ScalarMappable* does not make any assumptions on the dimensionality and shape of the array.

**get_capstyle**()

Return the cap style for the collection (for all its elements).

**Returns**

**{'butt', 'projecting', 'round'} or None**

**get_children**()

Return a list of the child *Artist*s of this *Artist*.

**get_clim**()

Return the values (min, max) that are mapped to the colormap limits.

**get_clip_box**()

Return the clipbox.

**get_clip_on**()

Return whether the artist uses clipping.

**get_clip_path**()

Return the clip path.

**get_cmap**()

Return the *Colormap* instance.

**get_coordinates**()

Return the vertices of the mesh as an (M+1, N+1, 2) array.

M, N are the number of quadrilaterals in the rows / columns of the mesh, corresponding to (M+1, N+1) vertices. The last dimension specifies the components (x, y).

**get_cursor_data**(*event*)

Return the cursor data for a given event.

---

**Note:** This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

---

Cursor data can be used by Artists to provide additional context information for a given event. The default implementation just returns *None*.

Subclasses can override the method and return arbitrary data. However, when doing so, they must ensure that *format_cursor_data* can convert the data to a string representation.

The only current use case is displaying the z-value of an *AxesImage* in the status bar of a plot window, while moving the mouse.

**Parameters**

**event**

[*MouseEvent*]

**See also:**

*format_cursor_data*

**get_dashes**()
Alias for *get_linestyle*.

**get_datalim**(*transData*)

**get_ec**()
Alias for *get_edgecolor*.

**get_edgecolor**()

**get_edgecolors**()
Alias for *get_edgecolor*.

**get_facecolor**()

**get_facecolors**()
Alias for *get_facecolor*.

**get_fc**()
Alias for *get_facecolor*.

**get_figure**()
Return the *Figure* instance the artist belongs to.

**get_fill**()
Return whether face is colored.

**get_gid**()
Return the group id.

**get_hatch**()
Return the current hatching pattern.

**get_in_layout**()
Return boolean flag, `True` if artist is included in layout calculations.

E.g. *Constrained layout guide*, *Figure.tight_layout()*, and `fig.savefig(fname, bbox_inches='tight')`.

**get_joinstyle**()
Return the join style for the collection (for all its elements).

**Returns**

**{'miter', 'round', 'bevel'} or None**

**get_label**()
Return the label used for this artist in the legend.

**get_linestyle**()

**get_linestyles**()

Alias for *get_linestyle*.

**get_linewidth**()

**get_linewidths**()

Alias for *get_linewidth*.

**get_ls**()

Alias for *get_linestyle*.

**get_lw**()

Alias for *get_linewidth*.

**get_mouseover**()

Return whether this artist is queried for custom context information when the mouse cursor moves over it.

**get_offset_transform**()

Return the *Transform* instance used by this artist offset.

**get_offsets**()

Return the offsets for the collection.

**get_path_effects**()

**get_paths**()

**get_picker**()

Return the picking behavior of the artist.

The possible values are described in *Artist.set_picker*.

**See also:**

***Artist.set_picker***, ***Artist.pickable***, ***Artist.pick***

**get_pickradius**()

**get_rasterized**()

Return whether the artist is to be rasterized.

**get_sizes**()

Return the sizes ('areas') of the elements in the collection.

**Returns**

**array**

The 'area' of each element.

**get_sketch_params**()

Return the sketch parameters for the artist.

**Returns**

**tuple or None**

A 3-tuple with the following elements:

- *scale*: The amplitude of the wiggle perpendicular to the source line.
- *length*: The length of the wiggle along the line.
- *randomness*: The scale factor by which the length is shrunken or expanded.

Returns *None* if no sketch parameters were set.

**get_snap**()

Return the snap setting.

See *set_snap* for details.

**get_tightbbox**(*renderer=None*)

Like *Artist.get_window_extent*, but includes any clipping.

**Parameters**

**renderer**

[*RendererBase* subclass, optional] renderer that will be used to draw the figures (i.e. `fig.canvas.get_renderer()`)

**Returns**

***Bbox* or None**

The enclosing bounding box (in figure pixel coordinates). Returns None if clipping results in no intersection.

**get_transOffset**()

Alias for *get_offset_transform*.

**get_transform**()

Return the *Transform* instance used by this artist.

**get_transformed_clip_path_and_affine**()

Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

**get_transforms**()

**get_url**()

Return the url.

**get_urls**()

Return a list of URLs, one for each element of the collection.

The list contains *None* for elements without a URL. See *Hyperlinks* for an example.

**get_visible**()

Return the visibility.

**get_window_extent**(*renderer=None*)

Get the artist's bounding box in display space.

The bounding box' width and height are nonnegative.

Subclasses should override for inclusion in the bounding box "tight" calculation. Default is to return an empty bounding box at 0, 0.

Be careful when using this function, the results will not update if the artist window extent of the artist changes. The extent can change due to any changes in the transform stack, such as changing the Axes limits, the figure size, or the canvas used (as is done when saving a figure). This can lead to unexpected behavior where interactive figures will look fine on the screen, but will save incorrectly.

**get_zorder**()

Return the artist's zorder.

**have_units**()

Return whether units are set on any axis.

**is_transform_set**()

Return whether the Artist has an explicitly set transform.

This is *True* after `set_transform` has been called.

**property mouseover**

Return whether this artist is queried for custom context information when the mouse cursor moves over it.

**property norm**

**pchanged**()

Call all of the registered callbacks.

This function is triggered internally when a property is changed.

**See also:**

***add_callback***
***remove_callback***

**pick**(*mouseevent*)

Process a pick event.

Each child artist will fire a pick event if *mouseevent* is over the artist and the artist has picker set.

**See also:**

*Artist.set_picker*, *Artist.get_picker*, *Artist.pickable*

**pickable**()

Return whether the artist is pickable.

**See also:**

*Artist.set_picker*, *Artist.get_picker*, *Artist.pick*

**properties**()

Return a dictionary of all the properties of the artist.

**remove**()

Remove the artist from the figure if possible.

The effect will not be visible until the figure is redrawn, e.g., with *FigureCanvasBase.draw_idle*. Call *relim* to update the Axes limits if desired.

Note: *relim* will not see collections even if the collection was added to the Axes with *autolim* = True.

Note: there is no support for removing the artist's legend entry.

**remove_callback**(*oid*)

Remove a callback based on its observer id.

**See also:**

*add_callback*

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *antialiased=<UNSET>*, *array=<UNSET>*, *capstyle=<UNSET>*, *clim=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *cmap=<UNSET>*, *color=<UNSET>*, *edgecolor=<UNSET>*, *facecolor=<UNSET>*, *gid=<UNSET>*, *hatch=<UNSET>*, *in_layout=<UNSET>*, *joinstyle=<UNSET>*, *label=<UNSET>*, *linestyle=<UNSET>*, *linewidth=<UNSET>*, *mouseover=<UNSET>*, *norm=<UNSET>*, *offset_transform=<UNSET>*, *offsets=<UNSET>*, *path_effects=<UNSET>*, *paths=<UNSET>*, *picker=<UNSET>*, *pickradius=<UNSET>*, *rasterized=<UNSET>*, *sizes=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *urls=<UNSET>*, *verts=<UNSET>*, *verts_and_codes=<UNSET>*, *visible=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi v |
| *alpha* | array-like or scalar or None |
| *animated* | bool |
| *antialiased* or aa or antialiaseds | bool or list of bools |
| array | unknown |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clim* | (vmin: float, vmax: float) |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *cmap* | *Colormap* or str or None |
| *color* | *color* or list of RGBA tuples |
| *edgecolor* or ec or edgecolors | *color* or list of *color* or 'face' |
| *facecolor* or facecolors or fc | *color* or list of *color* |
| *figure* | *Figure* |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or dashes or linestyles or ls | str or tuple or list thereof |
| *linewidth* or linewidths or lw | float or list of floats |
| *mouseover* | bool |
| *norm* | *Normalize* or str or None |
| *offset_transform* or transOffset | *Transform* |
| *offsets* | (N, 2) or (2,) array-like |
| *path_effects* | list of *AbstractPathEffect* |
| *paths* | list of array-like |
| *picker* | None or bool or float or callable |
| *pickradius* | float |
| *rasterized* | bool |
| sizes | numpy.ndarray or None |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *urls* | list of str or None |
| *verts* | list of array-like |
| *verts_and_codes* | unknown |
| *visible* | bool |
| *zorder* | float |

**set_aa**(*aa*)

Alias for *set_antialiased*.

**set_agg_filter**(*filter_func*)

Set the agg filter.

**Parameters**

**filter_func**

[callable] A filter function, which takes a (m, n, depth) float array and a dpi value, and returns a (m, n, depth) array and two offsets from the bottom left corner of the image

**set_alpha**(*alpha*)

Set the alpha value used for blending - not supported on all backends.

**Parameters**

**alpha**

[array-like or scalar or None] All values must be within the 0-1 range, inclusive. Masked values and nans are not supported.

**set_animated**(*b*)

Set whether the artist is intended to be used in an animation.

If True, the artist is excluded from regular drawing of the figure. You have to call *Figure.draw_artist* / *Axes.draw_artist* explicitly on the artist. This approach is used to speed up animations using blitting.

See also *matplotlib.animation* and *Faster rendering by using blitting*.

**Parameters**

**b**

[bool]

**set_antialiased**(*aa*)

Set the antialiasing state for rendering.

**Parameters**

**aa**

[bool or list of bools]

**set_antialiaseds**(*aa*)

Alias for *set_antialiased*.

**set_array**(*A*)

Set the data values.

**Parameters**

**A**

[array-like] The mesh data. Supported array shapes are:

- (M, N) or (M*N,): a mesh with scalar data. The values are mapped to colors using normalization and a colormap. See parameters *norm*, *cmap*, *vmin*, *vmax*.
- (M, N, 3): an image with RGB values (0-1 float or 0-255 int).
- (M, N, 4): an image with RGBA values (0-1 float or 0-255 int), i.e. including transparency.

If the values are provided as a 2D grid, the shape must match the coordinates grid. If the values are 1D, they are reshaped to 2D. M, N follow from the coordinates grid, where the coordinates grid shape is (M, N) for 'gouraud' *shading* and (M+1, N+1) for 'flat' shading.

**set_capstyle**(*cs*)

Set the *CapStyle* for the collection (for all its elements).

**Parameters**

**cs**

[*CapStyle* or {'butt', 'projecting', 'round'}]

**set_clim**(*vmin=None*, *vmax=None*)

Set the norm limits for image scaling.

**Parameters**

**vmin, vmax**

[float] The limits.

The limits may also be passed as a tuple (*vmin*, *vmax*) as a single positional argument.

**set_clip_box**(*clipbox*)

Set the artist's clip *Bbox*.

**Parameters**

**clipbox**

[*BboxBase* or None] Will typically be created from a *TransformedBbox*. For instance, `TransformedBbox(Bbox([[0, 0], [1, 1]]), ax.transAxes)` is the default clipping for an artist added to an Axes.

**set_clip_on**(*b*)

Set whether the artist uses clipping.

When False, artists will be visible outside the Axes which can lead to unexpected results.

**Parameters**

**b**

[bool]

**set_clip_path**(*path*, *transform=None*)

Set the artist's clip path.

**Parameters**

**path**

[*Patch* or *Path* or *TransformedPath* or None] The clip path. If given a *Path*, *transform* must be provided as well. If *None*, a previously set clip path is removed.

**transform**

[*Transform*, optional] Only used if *path* is a *Path*, in which case the given *Path* is converted to a *TransformedPath* using *transform*.

### Notes

For efficiency, if *path* is a *Rectangle* this method will set the clipping box to the corresponding rectangle and set the clipping path to `None`.

For technical reasons (support of *set*), a tuple (*path*, *transform*) is also accepted as a single positional parameter.

**set_cmap**(*cmap*)

Set the colormap for luminance data.

**Parameters**

**cmap**

[*Colormap* or str or None]

**set_color**(*c*)

Set both the edgecolor and the facecolor.

**Parameters**

**c**

[*color* or list of RGBA tuples]

**See also:**

*Collection.set_facecolor*, *Collection.set_edgecolor*

For setting the edge or face color individually.

**set_dashes**(*ls*)

Alias for *set_linestyle*.

**set_ec**(*c*)

Alias for *set_edgecolor*.

**set_edgecolor**(*c*)

Set the edgecolor(s) of the collection.

**Parameters**

**c**

[*color* or list of *color* or 'face'] The collection edgecolor(s). If a sequence, the patches cycle through it. If 'face', match the facecolor.

**set_edgecolors**(*c*)

Alias for *set_edgecolor*.

**set_facecolor**(*c*)

Set the facecolor(s) of the collection. *c* can be a color (all patches have same color), or a sequence of colors; if it is a sequence the patches will cycle through the sequence.

If *c* is 'none', the patch will not be filled.

**Parameters**

**c**

[*color* or list of *color*]

**set_facecolors**(*c*)

Alias for *set_facecolor*.

**set_fc**(*c*)

Alias for *set_facecolor*.

**set_figure**(*fig*)

Set the *Figure* instance the artist belongs to.

**Parameters**

**fig**

[*Figure*]

**set_gid**(*gid*)

Set the (group) id for the artist.

**Parameters**

**gid**

[str]

**set_hatch**(*hatch*)

Set the hatching pattern

*hatch* can be one of:

```
/   - diagonal hatching
\   - back diagonal
|   - vertical
-   - horizontal
+   - crossed
x   - crossed diagonal
o   - small circle
O   - large circle
.   - dots
*   - stars
```

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

**Parameters**

**hatch**

[{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}]

**set_in_layout**(*in_layout*)

Set if artist is to be included in layout calculations, E.g. *Constrained layout guide*, *Figure.tight_layout()*, and `fig.savefig(fname, bbox_inches='tight')`.

**Parameters**

**in_layout**

[bool]

**set_joinstyle**(*js*)

Set the *JoinStyle* for the collection (for all its elements).

**Parameters**

**js**

[*JoinStyle* or {'miter', 'round', 'bevel'}]

**set_label**(*s*)

Set a label that will be displayed in the legend.

**Parameters**

**s**

[object] *s* will be converted to a string by calling `str`.

**set_linestyle**(*ls*)

Set the linestyle(s) for the collection.

| linestyle | description |
|---|---|
| `'-'` or `'solid'` | solid line |
| `'--'` or `'dashed'` | dashed line |
| `'-.'` or `'dashdot'` | dash-dotted line |
| `':'` or `'dotted'` | dotted line |

Alternatively a dash tuple of the following form can be provided:

```
(offset, onoffseq),
```

where `onoffseq` is an even length tuple of on and off ink in points.

**Parameters**

**ls**

[str or tuple or list thereof] Valid values for individual linestyles include {'-', '--', '-.', ':', '', (offset, on-off-seq)}. See *Line2D.set_linestyle* for a complete description.

**set_linestyles**(*ls*)

Alias for *set_linestyle*.

**set_linewidth**(*lw*)

Set the linewidth(s) for the collection. *lw* can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

**Parameters**

**lw**

[float or list of floats]

**set_linewidths**(*lw*)

Alias for *set_linewidth*.

**set_ls**(*ls*)

Alias for *set_linestyle*.

**set_lw**(*lw*)

Alias for *set_linewidth*.

**set_mouseover**(*mouseover*)

Set whether this artist is queried for custom context information when the mouse cursor moves over it.

**Parameters**

**mouseover**

[bool]

**See also:**

***get_cursor_data***
***ToolCursorPosition***
***NavigationToolbar2***

**set_norm**(*norm*)

Set the normalization instance.

**Parameters**

**norm**

[*Normalize* or str or None]

## Notes

If there are any colorbars using the mappable for this norm, setting the norm of the mappable will reset the norm, locator, and formatters on the colorbar to default.

**set_offset_transform**(*offset_transform*)

Set the artist offset transform.

**Parameters**

**offset_transform**

[*Transform*]

**set_offsets**(*offsets*)

Set the offsets for the collection.

**Parameters**

**offsets**

[(N, 2) or (2,) array-like]

**set_path_effects**(*path_effects*)

Set the path effects.

**Parameters**

**path_effects**

[list of *AbstractPathEffect*]

**set_paths**(*verts*, *closed=True*)

Set the vertices of the polygons.

**Parameters**

**verts**

[list of array-like] The sequence of polygons [*verts0*, *verts1*, ...] where each element *verts_i* defines the vertices of polygon *i* as a 2D array-like of shape (M, 2).

**closed**

[bool, default: True] Whether the polygon should be closed by adding a CLOSEPOLY connection at the end.

**set_picker**(*picker*)

Define the picking behavior of the artist.

**Parameters**

**picker**

[None or bool or float or callable] This can be one of the following:

- *None*: Picking is disabled for this artist (default).
- A boolean: If *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist.
- A float: If picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if its data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event

- A function: If picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

to determine the hit test. if the mouse event is over the artist, return *hit=True* and props is a dictionary of properties you want added to the PickEvent attributes.

**set_pickradius**(*pickradius*)

Set the pick radius used for containment tests.

**Parameters**

**pickradius**

[float] Pick radius, in points.

**set_rasterized**(*rasterized*)

Force rasterized (bitmap) drawing for vector graphics output.

Rasterized drawing is not supported by all artists. If you try to enable this on an artist that does not support it, the command has no effect and a warning will be issued.

This setting is ignored for pixel-based output.

See also *Rasterization for vector graphics*.

**Parameters**

**rasterized**

[bool]

**set_sizes**(*sizes*, *dpi=72.0*)

Set the sizes of each member of the collection.

**Parameters**

**sizes**

[numpy.ndarray or None] The size to set for each element of the collection. The value is the 'area' of the element.

**dpi**

[float, default: 72] The dpi of the canvas.

**set_sketch_params**(*scale=None*, *length=None*, *randomness=None*)

Set the sketch parameters.

**Parameters**

**scale**

[float, optional] The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is `None`, or not provided, no sketch filter will be provided.

**length**

[float, optional] The length of the wiggle along the line, in pixels (default 128.0)

**randomness**

[float, optional] The scale factor by which the length is shrunken or expanded (default 16.0)

The PGF backend uses this argument as an RNG seed and not as described above. Using the same seed yields the same random shape.

**set_snap**(*snap*)

Set the snapping behavior.

Snapping aligns positions with the pixel grid, which results in clearer images. For example, if a black line of 1px width was defined at a position in between two pixels, the resulting image would contain the interpolated value of that line in the pixel grid, which would be a grey value on both adjacent pixel positions. In contrast, snapping will move the line to the nearest integer pixel value, so that the resulting image will really contain a 1px wide black line.

Snapping is currently only supported by the Agg and MacOSX backends.

**Parameters**

**snap**

[bool or None] Possible values:

- *True*: Snap vertices to the nearest pixel center.
- *False*: Do not modify vertex positions.
- *None*: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center.

**set_transOffset**(*offset_transform*)

Alias for *set_offset_transform*.

**set_transform**(*t*)

Set the artist transform.

**Parameters**

**t**

[*Transform*]

**set_url**(*url*)

Set the url for the artist.

**Parameters**

**url**

[str]

**set_urls**(*urls*)

**Parameters**

**urls**

[list of str or None]

### Notes

URLs are currently only implemented by the SVG backend. They are ignored by all other backends.

**set_verts**(*verts*, *closed=True*)

Set the vertices of the polygons.

**Parameters**

**verts**

[list of array-like] The sequence of polygons [*verts0*, *verts1*, ...] where each element *verts_i* defines the vertices of polygon *i* as a 2D array-like of shape (M, 2).

**closed**

[bool, default: True] Whether the polygon should be closed by adding a CLOSEPOLY connection at the end.

**set_verts_and_codes**(*verts*, *codes*)

Initialize vertices with path codes.

**set_visible**(*b*)

Set the artist's visibility.

**Parameters**

**b**

[bool]

**set_zorder**(*level*)

Set the zorder for the artist. Artists with lower zorder values are drawn first.

**Parameters**

**level**

[float]

**property stale**

Whether the artist is 'stale' and needs to be re-drawn for the output to match the internal state of the artist.

**property sticky_edges**

`x` and `y` sticky edge lists for autoscaling.

When performing autoscaling, if a data limit coincides with a value in the corresponding sticky_edges list, then no margin will be added--the view limit "sticks" to the edge. A typical use case is histograms, where one usually expects no margin on the bottom edge (0) of the histogram.

Moreover, margin expansion "bumps" against sticky edges and cannot cross them. For example, if the upper data limit is 1.0, the upper view limit computed by simple margin application is 1.2, but there is a sticky edge at 1.1, then the actual upper view limit will be 1.1.

This attribute cannot be assigned to; however, the `x` and `y` lists can be modified in place as needed.

### Examples

```
>>> artist.sticky_edges.x[:] = (xmin, xmax)
>>> artist.sticky_edges.y[:] = (ymin, ymax)
```

**to_rgba**(*x*, *alpha=None*, *bytes=False*, *norm=True*)

Return a normalized RGBA array corresponding to *x*.

In the normal case, *x* is a 1D or 2D sequence of scalars, and the corresponding `ndarray` of RGBA values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already RGB or RGBA, such as might have been read from an image file. If *x* is an `ndarray` with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an RGB or RGBA array, and no mapping will be done. The array can be `uint8`, or it can be floats with values in the 0-1 range; otherwise a ValueError will be raised. Any NaNs or masked elements will be set to 0 alpha. If the last dimension is 3, the *alpha* kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the *alpha* kwarg is ignored; it does not replace the preexisting alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if *bytes* is *False* (default), the RGBA array will be floats in the 0-1 range; if it is *True*, the returned RGBA array will be `uint8` in the 0 to 255 range.

If norm is False, no normalization of the input data is performed, and it is assumed to be in the range (0-1).

**update**(*props*)

Update this artist's properties from the dict *props*.

**Parameters**

**props**

[dict]

**update_from**(*other*)

Copy properties from other to self.

**update_scalarmappable**()

Update colors from the scalar mappable array, if any.

Assign colors to edges and faces based on the array and/or colors that were directly set, as appropriate.

**zorder = 0**

**class** matplotlib.collections.**QuadMesh**(*coordinates*, *, *antialiased=True*, *shading='flat'*, ***kwargs*)

Bases: _MeshData, *Collection*

Class for the efficient drawing of a quadrilateral mesh.

A quadrilateral mesh is a grid of M by N adjacent quadrilaterals that are defined via a (M+1, N+1) grid of vertices. The quadrilateral (m, n) is defined by the vertices

```
    (m+1, n) ----------- (m+1, n+1)
       /                   /
      /                   /
     /                   /
(m, n) -------- (m, n+1)
```

The mesh need not be regular and the polygons need not be convex.

**Parameters**

**coordinates**

[(M+1, N+1, 2) array-like] The vertices. `coordinates[m, n]` specifies the (x, y) coordinates of vertex (m, n).

**antialiased**

[bool, default: True]

**shading**

[{'flat', 'gouraud'}, default: 'flat']

### Notes

Unlike other *Collection*s, the default *pickradius* of *QuadMesh* is 0, i.e. *contains* checks whether the test point is within any of the mesh quadrilaterals.

**Parameters**

**edgecolors**

[*color* or list of colors, default: *rcParams["patch.edgecolor"]* (default: `'black'`)] Edge color for each patch making up the collection. The special value 'face' can be passed to make the edgecolor match the facecolor.

**facecolors**

[*color* or list of colors, default: *rcParams["patch.facecolor"]* (default: `'C0'`)] Face color for each patch making up the collection.

**linewidths**

[float or list of floats, default: *rcParams["patch.linewidth"]* (default: `1.0`)] Line width for each patch making up the collection.

**linestyles**

[str or tuple or list thereof, default: 'solid'] Valid strings are ['solid', 'dashed', 'dashdot', 'dotted', '-', '--', '-.', ':']. Dash tuples should be of the form:

```
(offset, onoffseq),
```

where *onoffseq* is an even length tuple of on and off ink lengths in points. For examples, see *Linestyles*.

**capstyle**

[*CapStyle*-like, default: *rcParams["patch.capstyle"]*] Style to use for capping lines for all paths in the collection. Allowed values are {'butt', 'projecting', 'round'}.

**joinstyle**

[*JoinStyle*-like, default: *rcParams["patch.joinstyle"]*] Style to use for joining lines for all paths in the collection. Allowed values are {'miter', 'round', 'bevel'}.

**antialiaseds**

[bool or list of bool, default: *rcParams["patch.antialiased"]* (default: `True`)] Whether each patch in the collection should be drawn with antialiasing.

**offsets**

[(float, float) or list thereof, default: (0, 0)] A vector by which to translate each patch after rendering (default is no translation). The translation is performed in screen (pixel) coordinates (i.e. after the Artist's transform is applied).

**offset_transform**

[*Transform*, default: *IdentityTransform*] A single transform which will be applied to each *offsets* vector before it is used.

**cmap, norm**

Data normalization and colormapping parameters. See *ScalarMappable* for a detailed description.

**hatch**

[str, optional] Hatching pattern to use in filled paths, if any. Valid strings are ['/', '', '|', '-', '+', 'x', 'o', 'O', '.', '*']. See *Hatch style reference* for the meaning of each hatch type.

**pickradius**

[float, default: 5.0] If `pickradius <= 0`, then *Collection.contains* will return `True` whenever the test point is inside of one of the polygons formed by the control points of a Path in the Collection. On the other hand, if it is greater than 0, then we instead check if the test point is contained in a stroke of width `2*pickradius` following any of the Paths in the Collection.

**urls**

[list of str, default: None] A URL for each patch to link to once drawn. Currently only works for the SVG backend. See *Hyperlinks* for examples.

**zorder**

[float, default: 1] The drawing order, shared by all Patches in the Collection. See *Zorder Demo* for all defaults and examples.

****kwargs**

Remaining keyword arguments will be used to set properties as `Collection.set_{key}(val)` for each key-value pair in *kwargs*.

**add_callback**(*func*)

Add a callback function that will be called whenever one of the *Artist*'s properties changes.

**Parameters**

**func**

[callable] The callback function. It must have the signature:

```
def func(artist: Artist) -> Any
```

where *artist* is the calling *Artist*. Return values may exist but are ignored.

**Returns**

**int**

The observer id associated with the callback. This id can be used for removing the callback with *remove_callback* later.

**See also:**

***remove_callback***

**autoscale**()

Autoscale the scalar limits on the norm instance using the current array

**autoscale_None**()

Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

**property axes**

The *Axes* instance the artist resides in, or *None*.

**changed**()

Call this whenever the mappable is changed to notify all the callbackSM listeners to the 'changed' signal.

**colorbar**

The last colorbar associated with this ScalarMappable. May be None.

**contains**(*mouseevent*)

Test whether the mouse event occurred in the collection.

Returns `bool, dict(ind=itemlist)`, where every item in itemlist contains the event.

**convert_xunits**(*x*)

Convert *x* using the unit type of the xaxis.

If the artist is not contained in an Axes or if the xaxis does not have units, *x* itself is returned.

**convert_yunits**(*y*)

Convert *y* using the unit type of the yaxis.

If the artist is not contained in an Axes or if the yaxis does not have units, *y* itself is returned.

**draw**(*renderer*)

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

**Parameters**

**renderer**

[*RendererBase* subclass.]

### Notes

This method is overridden in the Artist subclasses.

**findobj**(*match=None*, *include_self=True*)

Find artist objects.

Recursively find all *Artist* instances contained in the artist.

**Parameters**

**match**

A filter criterion for the matches. This can be

- *None*: Return all objects contained in artist.
- A function with signature `def match(artist: Artist) -> bool`. The result will only contain artists for which the function returns *True*.
- A class instance: e.g., *Line2D*. The result will only contain artists of this class or its subclasses (`isinstance` check).

**include_self**

[bool] Include *self* in the list to be checked for a match.

**Returns**

**list of *Artist***

**format_cursor_data**(*data*)

Return a string representation of *data*.

---

**Note:** This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

---

The default implementation converts ints and floats and arrays of ints and floats into a comma-separated string enclosed in square brackets, unless the artist has an associated colorbar, in which case scalar values are formatted using the colorbar's formatter.

**See also:**

***get_cursor_data***

**get_aa**()

Alias for *get_antialiased*.

**get_agg_filter**()

Return filter function to be used for agg filter.

**get_alpha**()

Return the alpha value used for blending - not supported on all backends.

**get_animated**()

Return whether the artist is animated.

**get_antialiased**()

Get the antialiasing state for rendering.

**Returns**

**array of bools**

**get_antialiaseds**()

Alias for *get_antialiased*.

**get_array**()

Return the array of values, that are mapped to colors.

The base class *ScalarMappable* does not make any assumptions on the dimensionality and shape of the array.

**get_capstyle**()

Return the cap style for the collection (for all its elements).

**Returns**

**{'butt', 'projecting', 'round'} or None**

**get_children**()

Return a list of the child *Artist*s of this *Artist*.

**get_clim**()

Return the values (min, max) that are mapped to the colormap limits.

**get_clip_box**()

Return the clipbox.

**get_clip_on**()

Return whether the artist uses clipping.

**get_clip_path**()

Return the clip path.

**get_cmap**()

Return the *Colormap* instance.

**get_coordinates**()

Return the vertices of the mesh as an (M+1, N+1, 2) array.

M, N are the number of quadrilaterals in the rows / columns of the mesh, corresponding to (M+1, N+1) vertices. The last dimension specifies the components (x, y).

**get_cursor_data**(*event*)

Return the cursor data for a given event.

---

**Note:** This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

---

Cursor data can be used by Artists to provide additional context information for a given event. The default implementation just returns *None*.

Subclasses can override the method and return arbitrary data. However, when doing so, they must ensure that *format_cursor_data* can convert the data to a string representation.

The only current use case is displaying the z-value of an *AxesImage* in the status bar of a plot window, while moving the mouse.

**Parameters**

**event**

[*MouseEvent*]

**See also:**

***format_cursor_data***

**get_dashes**()

Alias for *get_linestyle*.

**get_datalim**(*transData*)

**get_ec**()

Alias for *get_edgecolor*.

**get_edgecolor**()

**get_edgecolors**()

Alias for *get_edgecolor*.

**get_facecolor**()

**get_facecolors**()

Alias for *get_facecolor*.

**get_fc**()

Alias for *get_facecolor*.

**get_figure**()

Return the *Figure* instance the artist belongs to.

**get_fill**()

Return whether face is colored.

**get_gid**()

Return the group id.

**get_hatch**()

Return the current hatching pattern.

**get_in_layout**()

Return boolean flag, `True` if artist is included in layout calculations.

E.g. *Constrained layout guide*, *`Figure.tight_layout()`*, and `fig.savefig(fname, bbox_inches='tight')`.

**get_joinstyle**()

Return the join style for the collection (for all its elements).

**Returns**

**{'miter', 'round', 'bevel'} or None**

**get_label**()

Return the label used for this artist in the legend.

**get_linestyle**()

**get_linestyles**()

Alias for *`get_linestyle`*.

**get_linewidth**()

**get_linewidths**()

Alias for *`get_linewidth`*.

**get_ls**()

Alias for *`get_linestyle`*.

**get_lw**()

Alias for *`get_linewidth`*.

**get_mouseover**()

Return whether this artist is queried for custom context information when the mouse cursor moves over it.

**get_offset_transform**()

Return the *`Transform`* instance used by this artist offset.

**get_offsets**()

Return the offsets for the collection.

**get_path_effects**()

**get_paths**()

**get_picker**()

Return the picking behavior of the artist.

The possible values are described in *Artist.set_picker*.

**See also:**

***Artist.set_picker***, ***Artist.pickable***, ***Artist.pick***

**get_pickradius**()

**get_rasterized**()

Return whether the artist is to be rasterized.

**get_sketch_params**()

Return the sketch parameters for the artist.

**Returns**

**tuple or None**

A 3-tuple with the following elements:

- *scale*: The amplitude of the wiggle perpendicular to the source line.
- *length*: The length of the wiggle along the line.
- *randomness*: The scale factor by which the length is shrunken or expanded.

Returns *None* if no sketch parameters were set.

**get_snap**()

Return the snap setting.

See *set_snap* for details.

**get_tightbbox**(*renderer=None*)

Like *Artist.get_window_extent*, but includes any clipping.

**Parameters**

**renderer**

[*RendererBase* subclass, optional] renderer that will be used to draw the figures (i.e. `fig.canvas.get_renderer()`)

**Returns**

***Bbox*** **or None**

The enclosing bounding box (in figure pixel coordinates). Returns None if clipping results in no intersection.

**get_transOffset**()

Alias for *get_offset_transform*.

**get_transform**()

Return the *Transform* instance used by this artist.

**get_transformed_clip_path_and_affine**()

Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

**get_transforms**()

**get_url**()

Return the url.

**get_urls**()

Return a list of URLs, one for each element of the collection.

The list contains *None* for elements without a URL. See *Hyperlinks* for an example.

**get_visible**()

Return the visibility.

**get_window_extent**(*renderer=None*)

Get the artist's bounding box in display space.

The bounding box' width and height are nonnegative.

Subclasses should override for inclusion in the bounding box "tight" calculation. Default is to return an empty bounding box at 0, 0.

Be careful when using this function, the results will not update if the artist window extent of the artist changes. The extent can change due to any changes in the transform stack, such as changing the Axes limits, the figure size, or the canvas used (as is done when saving a figure). This can lead to unexpected behavior where interactive figures will look fine on the screen, but will save incorrectly.

**get_zorder**()

Return the artist's zorder.

**have_units**()

Return whether units are set on any axis.

**is_transform_set**()

Return whether the Artist has an explicitly set transform.

This is *True* after *set_transform* has been called.

**property mouseover**

Return whether this artist is queried for custom context information when the mouse cursor moves over it.

**property norm**

**pchanged**()

Call all of the registered callbacks.

This function is triggered internally when a property is changed.

**See also:**

*add_callback*
*remove_callback*

**pick**(*mouseevent*)

Process a pick event.

Each child artist will fire a pick event if *mouseevent* is over the artist and the artist has picker set.

**See also:**

*Artist.set_picker*, *Artist.get_picker*, *Artist.pickable*

**pickable**()

Return whether the artist is pickable.

**See also:**

*Artist.set_picker*, *Artist.get_picker*, *Artist.pick*

**properties**()

Return a dictionary of all the properties of the artist.

**remove**()

Remove the artist from the figure if possible.

The effect will not be visible until the figure is redrawn, e.g., with *FigureCanvasBase.draw_idle*. Call *relim* to update the Axes limits if desired.

Note: *relim* will not see collections even if the collection was added to the Axes with *autolim* = True.

Note: there is no support for removing the artist's legend entry.

**remove_callback**(*oid*)

Remove a callback based on its observer id.

**See also:**

*add_callback*

**set** (*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *antialiased=<UNSET>*, *array=<UNSET>*, *capstyle=<UNSET>*, *clim=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *cmap=<UNSET>*, *color=<UNSET>*, *edgecolor=<UNSET>*, *facecolor=<UNSET>*, *gid=<UNSET>*, *hatch=<UNSET>*, *in_layout=<UNSET>*, *joinstyle=<UNSET>*, *label=<UNSET>*, *linestyle=<UNSET>*, *linewidth=<UNSET>*, *mouseover=<UNSET>*, *norm=<UNSET>*, *offset_transform=<UNSET>*, *offsets=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *pickradius=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *urls=<UNSET>*, *visible=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi v |
| *alpha* | array-like or scalar or None |
| *animated* | bool |
| *antialiased* or aa or antialiaseds | bool or list of bools |
| array | array-like |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clim* | (vmin: float, vmax: float) |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *cmap* | *Colormap* or str or None |
| *color* | *color* or list of RGBA tuples |
| *edgecolor* or ec or edgecolors | *color* or list of *color* or 'face' |
| *facecolor* or facecolors or fc | *color* or list of *color* |
| *figure* | *Figure* |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or dashes or linestyles or ls | str or tuple or list thereof |
| *linewidth* or linewidths or lw | float or list of floats |
| *mouseover* | bool |
| *norm* | *Normalize* or str or None |
| *offset_transform* or transOffset | *Transform* |
| *offsets* | (N, 2) or (2,) array-like |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *pickradius* | float |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |

Table 43 – continued from p

| Property | Description |
|---|---|
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *urls* | list of str or None |
| *visible* | bool |
| *zorder* | float |

**set_aa**(*aa*)

Alias for *set_antialiased*.

**set_agg_filter**(*filter_func*)

Set the agg filter.

**Parameters**

**filter_func**

[callable] A filter function, which takes a (m, n, depth) float array and a dpi value, and returns a (m, n, depth) array and two offsets from the bottom left corner of the image

**set_alpha**(*alpha*)

Set the alpha value used for blending - not supported on all backends.

**Parameters**

**alpha**

[array-like or scalar or None] All values must be within the 0-1 range, inclusive. Masked values and nans are not supported.

**set_animated**(*b*)

Set whether the artist is intended to be used in an animation.

If True, the artist is excluded from regular drawing of the figure. You have to call *Figure.draw_artist* / *Axes.draw_artist* explicitly on the artist. This approach is used to speed up animations using blitting.

See also *matplotlib.animation* and *Faster rendering by using blitting*.

**Parameters**

**b**

[bool]

**set_antialiased**(*aa*)

Set the antialiasing state for rendering.

**Parameters**

**aa**

[bool or list of bools]

**set_antialiaseds**(*aa*)

Alias for *set_antialiased*.

**set_array**(*A*)

Set the data values.

**Parameters**

**A**

[array-like] The mesh data. Supported array shapes are:

- (M, N) or (M*N,): a mesh with scalar data. The values are mapped to colors using normalization and a colormap. See parameters *norm*, *cmap*, *vmin*, *vmax*.
- (M, N, 3): an image with RGB values (0-1 float or 0-255 int).
- (M, N, 4): an image with RGBA values (0-1 float or 0-255 int), i.e. including transparency.

If the values are provided as a 2D grid, the shape must match the coordinates grid. If the values are 1D, they are reshaped to 2D. M, N follow from the coordinates grid, where the coordinates grid shape is (M, N) for 'gouraud' *shading* and (M+1, N+1) for 'flat' shading.

**set_capstyle**(*cs*)

Set the *CapStyle* for the collection (for all its elements).

**Parameters**

**cs**

[*CapStyle* or {'butt', 'projecting', 'round'}]

**set_clim**(*vmin=None*, *vmax=None*)

Set the norm limits for image scaling.

**Parameters**

**vmin, vmax**

[float] The limits.

The limits may also be passed as a tuple (*vmin*, *vmax*) as a single positional argument.

**set_clip_box**(*clipbox*)

Set the artist's clip *Bbox*.

**Parameters**

**clipbox**

[*BboxBase* or None] Will typically be created from a *TransformedBbox*. For instance, `TransformedBbox(Bbox([[0, 0], [1, 1]]), ax.transAxes)` is the default clipping for an artist added to an Axes.

**set_clip_on**(*b*)

Set whether the artist uses clipping.

When False, artists will be visible outside the Axes which can lead to unexpected results.

**Parameters**

**b**

[bool]

**set_clip_path**(*path*, *transform=None*)

Set the artist's clip path.

**Parameters**

**path**

[*Patch* or *Path* or *TransformedPath* or None] The clip path. If given a *Path*, *transform* must be provided as well. If *None*, a previously set clip path is removed.

**transform**

[*Transform*, optional] Only used if *path* is a *Path*, in which case the given *Path* is converted to a *TransformedPath* using *transform*.

### Notes

For efficiency, if *path* is a *Rectangle* this method will set the clipping box to the corresponding rectangle and set the clipping path to `None`.

For technical reasons (support of *set*), a tuple (*path*, *transform*) is also accepted as a single positional parameter.

**set_cmap**(*cmap*)

Set the colormap for luminance data.

**Parameters**

**cmap**

[*Colormap* or str or None]

**set_color**(*c*)

Set both the edgecolor and the facecolor.

**Parameters**

**c**

[*color* or list of RGBA tuples]

**See also:**

***Collection.set_facecolor*, *Collection.set_edgecolor***

For setting the edge or face color individually.

**set_dashes**(*ls*)

Alias for *set_linestyle*.

**set_ec**(*c*)

Alias for *set_edgecolor*.

**set_edgecolor**(*c*)

Set the edgecolor(s) of the collection.

**Parameters**

**c**

[*color* or list of *color* or 'face'] The collection edgecolor(s). If a sequence, the patches cycle through it. If 'face', match the facecolor.

**set_edgecolors**(*c*)

Alias for *set_edgecolor*.

**set_facecolor**(*c*)

Set the facecolor(s) of the collection. *c* can be a color (all patches have same color), or a sequence of colors; if it is a sequence the patches will cycle through the sequence.

If *c* is 'none', the patch will not be filled.

**Parameters**

**c**

[*color* or list of *color*]

**set_facecolors**(*c*)

Alias for *set_facecolor*.

**set_fc**(*c*)

Alias for *set_facecolor*.

**set_figure**(*fig*)

Set the *Figure* instance the artist belongs to.

**Parameters**

**fig**

[*Figure*]

**set_gid**(*gid*)

Set the (group) id for the artist.

**Parameters**

**gid**

[str]

**set_hatch**(*hatch*)

Set the hatching pattern

*hatch* can be one of:

```
/   - diagonal hatching
\   - back diagonal
|   - vertical
-   - horizontal
+   - crossed
x   - crossed diagonal
o   - small circle
O   - large circle
.   - dots
*   - stars
```

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

**Parameters**

**hatch**

[{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}]

**set_in_layout**(*in_layout*)

Set if artist is to be included in layout calculations, E.g. *Constrained layout guide*, *Figure.tight_layout()*, and `fig.savefig(fname, bbox_inches='tight')`.

**Parameters**

**in_layout**

[bool]

**set_joinstyle**(*js*)

Set the *JoinStyle* for the collection (for all its elements).

**Parameters**

**js**

[*JoinStyle* or {'miter', 'round', 'bevel'}]

**set_label**(*s*)

Set a label that will be displayed in the legend.

**Parameters**

**s**

[object] *s* will be converted to a string by calling `str`.

**set_linestyle**(*ls*)

Set the linestyle(s) for the collection.

| linestyle | description |
|---|---|
| `'-'` or `'solid'` | solid line |
| `'--'` or `'dashed'` | dashed line |
| `'-.'` or `'dashdot'` | dash-dotted line |
| `':'` or `'dotted'` | dotted line |

Alternatively a dash tuple of the following form can be provided:

```
(offset, onoffseq),
```

where `onoffseq` is an even length tuple of on and off ink in points.

**Parameters**

**ls**

[str or tuple or list thereof] Valid values for individual linestyles include {'-', '--', '-.', ':', '', (offset, on-off-seq)}. See *Line2D.set_linestyle* for a complete description.

**set_linestyles**(*ls*)

Alias for *set_linestyle*.

**set_linewidth**(*lw*)

Set the linewidth(s) for the collection. *lw* can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

**Parameters**

**lw**

[float or list of floats]

**set_linewidths**(*lw*)

Alias for *set_linewidth*.

**set_ls**(*ls*)

Alias for *set_linestyle*.

**set_lw**(*lw*)

Alias for *set_linewidth*.

**set_mouseover**(*mouseover*)

Set whether this artist is queried for custom context information when the mouse cursor moves over it.

**Parameters**

**mouseover**

[bool]

**See also:**

***get_cursor_data***
***ToolCursorPosition***
***NavigationToolbar2***

**set_norm**(*norm*)

Set the normalization instance.

**Parameters**

**norm**

[*Normalize* or str or None]

### Notes

If there are any colorbars using the mappable for this norm, setting the norm of the mappable will reset the norm, locator, and formatters on the colorbar to default.

**set_offset_transform**(*offset_transform*)

Set the artist offset transform.

**Parameters**

**offset_transform**

[*Transform*]

**set_offsets**(*offsets*)

Set the offsets for the collection.

**Parameters**

**offsets**

[(N, 2) or (2,) array-like]

**set_path_effects**(*path_effects*)

Set the path effects.

**Parameters**

**path_effects**

[list of *AbstractPathEffect*]

**set_paths**()

**set_picker**(*picker*)

Define the picking behavior of the artist.

**Parameters**

**picker**

[None or bool or float or callable] This can be one of the following:

- *None*: Picking is disabled for this artist (default).
- A boolean: If *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist.
- A float: If picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if its data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event

- A function: If picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

to determine the hit test. if the mouse event is over the artist, return *hit=True* and props is a dictionary of properties you want added to the PickEvent attributes.

**set_pickradius**(*pickradius*)

Set the pick radius used for containment tests.

**Parameters**

**pickradius**

[float] Pick radius, in points.

**set_rasterized**(*rasterized*)

Force rasterized (bitmap) drawing for vector graphics output.

Rasterized drawing is not supported by all artists. If you try to enable this on an artist that does not support it, the command has no effect and a warning will be issued.

This setting is ignored for pixel-based output.

See also *Rasterization for vector graphics*.

**Parameters**

**rasterized**

[bool]

**set_sketch_params**(*scale=None*, *length=None*, *randomness=None*)

Set the sketch parameters.

**Parameters**

**scale**

[float, optional] The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is `None`, or not provided, no sketch filter will be provided.

**length**

[float, optional] The length of the wiggle along the line, in pixels (default 128.0)

**randomness**

[float, optional] The scale factor by which the length is shrunken or expanded (default 16.0)

The PGF backend uses this argument as an RNG seed and not as described above. Using the same seed yields the same random shape.

**set_snap**(*snap*)

Set the snapping behavior.

Snapping aligns positions with the pixel grid, which results in clearer images. For example, if a black line of 1px width was defined at a position in between two pixels, the resulting image would contain the interpolated value of that line in the pixel grid, which would be a grey value on both adjacent pixel positions. In contrast, snapping will move the line to the nearest integer pixel value, so that the resulting image will really contain a 1px wide black line.

Snapping is currently only supported by the Agg and MacOSX backends.

**Parameters**

**snap**

[bool or None] Possible values:

- *True*: Snap vertices to the nearest pixel center.
- *False*: Do not modify vertex positions.
- *None*: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center.

**set_transOffset**(*offset_transform*)

Alias for *set_offset_transform*.

**set_transform**(*t*)

Set the artist transform.

**Parameters**

**t**

[*Transform*]

**set_url**(*url*)

Set the url for the artist.

**Parameters**

**url**

[str]

**set_urls**(*urls*)

**Parameters**

**urls**

[list of str or None]

### Notes

URLs are currently only implemented by the SVG backend. They are ignored by all other backends.

**set_visible**(*b*)

Set the artist's visibility.

**Parameters**

**b**

[bool]

**set_zorder**(*level*)

Set the zorder for the artist. Artists with lower zorder values are drawn first.

**Parameters**

**level**

[float]

**property stale**

Whether the artist is 'stale' and needs to be re-drawn for the output to match the internal state of the artist.

**property sticky_edges**

`x` and `y` sticky edge lists for autoscaling.

When performing autoscaling, if a data limit coincides with a value in the corresponding sticky_edges list, then no margin will be added--the view limit "sticks" to the edge. A typical use case is histograms, where one usually expects no margin on the bottom edge (0) of the histogram.

Moreover, margin expansion "bumps" against sticky edges and cannot cross them. For example, if the upper data limit is 1.0, the upper view limit computed by simple margin application is 1.2, but there is a sticky edge at 1.1, then the actual upper view limit will be 1.1.

This attribute cannot be assigned to; however, the `x` and `y` lists can be modified in place as needed.

### Examples

```
>>> artist.sticky_edges.x[:] = (xmin, xmax)
>>> artist.sticky_edges.y[:] = (ymin, ymax)
```

**to_rgba**(*x*, *alpha=None*, *bytes=False*, *norm=True*)

Return a normalized RGBA array corresponding to *x*.

In the normal case, *x* is a 1D or 2D sequence of scalars, and the corresponding `ndarray` of RGBA values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already RGB or RGBA, such as might have been read from an image file. If *x* is an `ndarray` with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an RGB or RGBA array, and no mapping will be done. The array can be `uint8`, or it can be floats with values in the 0-1 range; otherwise a ValueError will be raised. Any NaNs or masked elements will be set to 0 alpha. If the last dimension is 3, the *alpha* kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the *alpha* kwarg is ignored; it does not replace the preexisting alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if *bytes* is *False* (default), the RGBA array will be floats in the 0-1 range; if it is *True*, the returned RGBA array will be `uint8` in the 0 to 255 range.

If norm is False, no normalization of the input data is performed, and it is assumed to be in the range (0-1).

**update**(*props*)

Update this artist's properties from the dict *props*.

**Parameters**

**props**

[dict]

**update_from**(*other*)

Copy properties from other to self.

**update_scalarmappable**()

Update colors from the scalar mappable array, if any.

Assign colors to edges and faces based on the array and/or colors that were directly set, as appropriate.

**zorder = 0**

**class** matplotlib.collections.**RegularPolyCollection**(*numsides*, *, *rotation=0*, *sizes=(1,)*, ***kwargs*)

Bases: `_CollectionWithSizes`

A collection of n-sided regular polygons.

**Parameters**

**numsides**

[int] The number of sides of the polygon.

**rotation**

[float] The rotation of the polygon in radians.

**sizes**

[tuple of float] The area of the circle circumscribing the polygon in points^2.

****kwargs**

Forwarded to *Collection*.

### Examples

See *Lasso Demo* for a complete example:

```
offsets = np.random.rand(20, 2)
facecolors = [cm.jet(x) for x in np.random.rand(20)]

collection = RegularPolyCollection(
    numsides=5, # a pentagon
    rotation=0, sizes=(50,),
    facecolors=facecolors,
    edgecolors=("black",),
    linewidths=(1,),
    offsets=offsets,
    offset_transform=ax.transData,
    )
```

**add_callback**(*func*)

Add a callback function that will be called whenever one of the *Artist*'s properties changes.

**Parameters**

**func**

[callable] The callback function. It must have the signature:

```
def func(artist: Artist) -> Any
```

where *artist* is the calling *Artist*. Return values may exist but are ignored.

**Returns**

**int**

The observer id associated with the callback. This id can be used for removing the callback with *remove_callback* later.

**See also:**

*remove_callback*

**autoscale**()

Autoscale the scalar limits on the norm instance using the current array

**autoscale_None**()

Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

**property axes**

The *Axes* instance the artist resides in, or *None*.

**changed**()

Call this whenever the mappable is changed to notify all the callbackSM listeners to the 'changed' signal.

**colorbar**

The last colorbar associated with this ScalarMappable. May be None.

**contains**(*mouseevent*)

Test whether the mouse event occurred in the collection.

Returns `bool, dict(ind=itemlist)`, where every item in itemlist contains the event.

**convert_xunits**(*x*)

Convert *x* using the unit type of the xaxis.

If the artist is not contained in an Axes or if the xaxis does not have units, *x* itself is returned.

**convert_yunits**(*y*)

Convert *y* using the unit type of the yaxis.

If the artist is not contained in an Axes or if the yaxis does not have units, *y* itself is returned.

**draw**(*renderer*)

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

**Parameters**

**renderer**

[*RendererBase* subclass.]

### Notes

This method is overridden in the Artist subclasses.

**findobj**(*match=None*, *include_self=True*)

Find artist objects.

Recursively find all *Artist* instances contained in the artist.

**Parameters**

**match**

A filter criterion for the matches. This can be

- *None*: Return all objects contained in artist.
- A function with signature `def match(artist: Artist) -> bool`. The result will only contain artists for which the function returns *True*.
- A class instance: e.g., *Line2D*. The result will only contain artists of this class or its subclasses (`isinstance` check).

**include_self**

[bool] Include *self* in the list to be checked for a match.

**Returns**

**list of *Artist***

**format_cursor_data**(*data*)

Return a string representation of *data*.

---

**Note:** This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

---

The default implementation converts ints and floats and arrays of ints and floats into a comma-separated string enclosed in square brackets, unless the artist has an associated colorbar, in which case scalar values are formatted using the colorbar's formatter.

**See also:**

***get_cursor_data***

**get_aa**()

Alias for *get_antialiased*.

**get_agg_filter**()

Return filter function to be used for agg filter.

**get_alpha**()

Return the alpha value used for blending - not supported on all backends.

**get_animated**()

Return whether the artist is animated.

**get_antialiased**()

Get the antialiasing state for rendering.

**Returns**

**array of bools**

**get_antialiaseds**()

Alias for *get_antialiased*.

**get_array**()

Return the array of values, that are mapped to colors.

The base class *ScalarMappable* does not make any assumptions on the dimensionality and shape of the array.

**get_capstyle**()

Return the cap style for the collection (for all its elements).

**Returns**

**{'butt', 'projecting', 'round'} or None**

**get_children**()

Return a list of the child *Artist*s of this *Artist*.

**get_clim**()

Return the values (min, max) that are mapped to the colormap limits.

**get_clip_box**()

Return the clipbox.

**get_clip_on**()

Return whether the artist uses clipping.

**get_clip_path**()

Return the clip path.

**get_cmap**()

Return the *Colormap* instance.

**get_cursor_data**(*event*)

Return the cursor data for a given event.

**Note:** This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

Cursor data can be used by Artists to provide additional context information for a given event. The default implementation just returns *None*.

Subclasses can override the method and return arbitrary data. However, when doing so, they must ensure that *format_cursor_data* can convert the data to a string representation.

The only current use case is displaying the z-value of an *AxesImage* in the status bar of a plot window, while moving the mouse.

**Parameters**

**event**

[*MouseEvent*]

**See also:**

***format_cursor_data***

**get_dashes**()
Alias for *get_linestyle*.

**get_datalim**(*transData*)

**get_ec**()
Alias for *get_edgecolor*.

**get_edgecolor**()

**get_edgecolors**()
Alias for *get_edgecolor*.

**get_facecolor**()

**get_facecolors**()
Alias for *get_facecolor*.

**get_fc**()
Alias for *get_facecolor*.

**get_figure**()
Return the *Figure* instance the artist belongs to.

**get_fill**()
Return whether face is colored.

**get_gid**()

Return the group id.

**get_hatch**()

Return the current hatching pattern.

**get_in_layout**()

Return boolean flag, `True` if artist is included in layout calculations.

E.g. *Constrained layout guide*, *`Figure.tight_layout()`*, and `fig.savefig(fname, bbox_inches='tight')`.

**get_joinstyle**()

Return the join style for the collection (for all its elements).

**Returns**

**{'miter', 'round', 'bevel'} or None**

**get_label**()

Return the label used for this artist in the legend.

**get_linestyle**()

**get_linestyles**()

Alias for *`get_linestyle`*.

**get_linewidth**()

**get_linewidths**()

Alias for *`get_linewidth`*.

**get_ls**()

Alias for *`get_linestyle`*.

**get_lw**()

Alias for *`get_linewidth`*.

**get_mouseover**()

Return whether this artist is queried for custom context information when the mouse cursor moves over it.

**get_numsides**()

**get_offset_transform**()

Return the *`Transform`* instance used by this artist offset.

**get_offsets**()

Return the offsets for the collection.

**get_path_effects**()

**get_paths**()

**get_picker**()

Return the picking behavior of the artist.

The possible values are described in *Artist.set_picker*.

**See also:**

***Artist.set_picker***, ***Artist.pickable***, ***Artist.pick***

**get_pickradius**()

**get_rasterized**()

Return whether the artist is to be rasterized.

**get_rotation**()

**get_sizes**()

Return the sizes ('areas') of the elements in the collection.

**Returns**

**array**

The 'area' of each element.

**get_sketch_params**()

Return the sketch parameters for the artist.

**Returns**

**tuple or None**

A 3-tuple with the following elements:

- *scale*: The amplitude of the wiggle perpendicular to the source line.
- *length*: The length of the wiggle along the line.
- *randomness*: The scale factor by which the length is shrunken or expanded.

Returns *None* if no sketch parameters were set.

**get_snap**()

Return the snap setting.

See *set_snap* for details.

**get_tightbbox**(*renderer=None*)

Like *Artist.get_window_extent*, but includes any clipping.

**Parameters**

**renderer**

[*RendererBase* subclass, optional] renderer that will be used to draw the figures (i.e. `fig.canvas.get_renderer()`)

**Returns**

***Bbox* or None**

The enclosing bounding box (in figure pixel coordinates). Returns None if clipping results in no intersection.

**get_transOffset**()

Alias for *get_offset_transform*.

**get_transform**()

Return the *Transform* instance used by this artist.

**get_transformed_clip_path_and_affine**()

Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

**get_transforms**()

**get_url**()

Return the url.

**get_urls**()

Return a list of URLs, one for each element of the collection.

The list contains *None* for elements without a URL. See *Hyperlinks* for an example.

**get_visible**()

Return the visibility.

**get_window_extent**(*renderer=None*)

Get the artist's bounding box in display space.

The bounding box' width and height are nonnegative.

Subclasses should override for inclusion in the bounding box "tight" calculation. Default is to return an empty bounding box at 0, 0.

Be careful when using this function, the results will not update if the artist window extent of the artist changes. The extent can change due to any changes in the transform stack, such as changing the Axes limits, the figure size, or the canvas used (as is done when saving a figure). This can lead to unexpected behavior where interactive figures will look fine on the screen, but will save incorrectly.

**get_zorder**()

Return the artist's zorder.

**have_units**()

Return whether units are set on any axis.

**is_transform_set**()

Return whether the Artist has an explicitly set transform.

This is *True* after *set_transform* has been called.

**property mouseover**

Return whether this artist is queried for custom context information when the mouse cursor moves over it.

**property norm**

**pchanged**()

Call all of the registered callbacks.

This function is triggered internally when a property is changed.

**See also:**

***add_callback***
***remove_callback***

**pick**(*mouseevent*)

Process a pick event.

Each child artist will fire a pick event if *mouseevent* is over the artist and the artist has picker set.

**See also:**

***Artist.set_picker***, ***Artist.get_picker***, ***Artist.pickable***

**pickable**()

Return whether the artist is pickable.

**See also:**

***Artist.set_picker***, ***Artist.get_picker***, ***Artist.pick***

**properties**()

Return a dictionary of all the properties of the artist.

**remove**()

Remove the artist from the figure if possible.

The effect will not be visible until the figure is redrawn, e.g., with *FigureCanvasBase.draw_idle*. Call *relim* to update the Axes limits if desired.

Note: *relim* will not see collections even if the collection was added to the Axes with *autolim* = True.

Note: there is no support for removing the artist's legend entry.

**remove_callback**(*oid*)

Remove a callback based on its observer id.

**See also:**

*add_callback*

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *antialiased=<UNSET>*, *array=<UNSET>*, *capstyle=<UNSET>*, *clim=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *cmap=<UNSET>*, *color=<UNSET>*, *edgecolor=<UNSET>*, *facecolor=<UNSET>*, *gid=<UNSET>*, *hatch=<UNSET>*, *in_layout=<UNSET>*, *joinstyle=<UNSET>*, *label=<UNSET>*, *linestyle=<UNSET>*, *linewidth=<UNSET>*, *mouseover=<UNSET>*, *norm=<UNSET>*, *offset_transform=<UNSET>*, *offsets=<UNSET>*, *path_effects=<UNSET>*, *paths=<UNSET>*, *picker=<UNSET>*, *pickradius=<UNSET>*, *rasterized=<UNSET>*, *sizes=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *urls=<UNSET>*, *visible=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi v |
| *alpha* | array-like or scalar or None |
| *animated* | bool |
| *antialiased* or aa or antialiaseds | bool or list of bools |
| *array* | array-like or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clim* | (vmin: float, vmax: float) |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *cmap* | *Colormap* or str or None |
| *color* | *color* or list of RGBA tuples |
| *edgecolor* or ec or edgecolors | *color* or list of *color* or 'face' |
| *facecolor* or facecolors or fc | *color* or list of *color* |
| *figure* | *Figure* |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or dashes or linestyles or ls | str or tuple or list thereof |
| *linewidth* or linewidths or lw | float or list of floats |
| *mouseover* | bool |
| *norm* | *Normalize* or str or None |

Table 44 – continued from p

| Property | Description |
|---|---|
| *offset_transform* or transOffset | *Transform* |
| *offsets* | (N, 2) or (2,) array-like |
| *path_effects* | list of *AbstractPathEffect* |
| *paths* | unknown |
| *picker* | None or bool or float or callable |
| *pickradius* | float |
| *rasterized* | bool |
| sizes | numpy.ndarray or None |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *urls* | list of str or None |
| *visible* | bool |
| *zorder* | float |

**set_aa**(*aa*)

Alias for *set_antialiased*.

**set_agg_filter**(*filter_func*)

Set the agg filter.

**Parameters**

**filter_func**

[callable] A filter function, which takes a (m, n, depth) float array and a dpi value, and returns a (m, n, depth) array and two offsets from the bottom left corner of the image

**set_alpha**(*alpha*)

Set the alpha value used for blending - not supported on all backends.

**Parameters**

**alpha**

[array-like or scalar or None] All values must be within the 0-1 range, inclusive. Masked values and nans are not supported.

**set_animated**(*b*)

Set whether the artist is intended to be used in an animation.

If True, the artist is excluded from regular drawing of the figure. You have to call *Figure.draw_artist* / *Axes.draw_artist* explicitly on the artist. This approach is used to speed up animations using blitting.

See also *matplotlib.animation* and *Faster rendering by using blitting*.

**Parameters**

**b**

[bool]

**set_antialiased**(*aa*)

Set the antialiasing state for rendering.

**Parameters**

**aa**

[bool or list of bools]

**set_antialiaseds**(*aa*)

Alias for *set_antialiased*.

**set_array**(*A*)

Set the value array from array-like *A*.

**Parameters**

**A**

[array-like or None] The values that are mapped to colors.

The base class *ScalarMappable* does not make any assumptions on the dimensionality and shape of the value array *A*.

**set_capstyle**(*cs*)

Set the *CapStyle* for the collection (for all its elements).

**Parameters**

**cs**

[*CapStyle* or {'butt', 'projecting', 'round'}]

**set_clim**(*vmin=None*, *vmax=None*)

Set the norm limits for image scaling.

**Parameters**

**vmin, vmax**

[float] The limits.

The limits may also be passed as a tuple (*vmin*, *vmax*) as a single positional argument.

**set_clip_box**(*clipbox*)

Set the artist's clip *Bbox*.

**Parameters**

**clipbox**

[*BboxBase* or None] Will typically be created from a *TransformedBbox*. For instance, `TransformedBbox(Bbox([[0, 0], [1, 1]]), ax.transAxes)` is the default clipping for an artist added to an Axes.

**set_clip_on**(*b*)

Set whether the artist uses clipping.

When False, artists will be visible outside the Axes which can lead to unexpected results.

**Parameters**

**b**

[bool]

**set_clip_path**(*path*, *transform=None*)

Set the artist's clip path.

**Parameters**

**path**

[*Patch* or *Path* or *TransformedPath* or None] The clip path. If given a *Path*, *transform* must be provided as well. If *None*, a previously set clip path is removed.

**transform**

[*Transform*, optional] Only used if *path* is a *Path*, in which case the given *Path* is converted to a *TransformedPath* using *transform*.

### Notes

For efficiency, if *path* is a *Rectangle* this method will set the clipping box to the corresponding rectangle and set the clipping path to `None`.

For technical reasons (support of *set*), a tuple (*path*, *transform*) is also accepted as a single positional parameter.

**set_cmap**(*cmap*)

Set the colormap for luminance data.

**Parameters**

**cmap**

[*Colormap* or str or None]

**set_color**(*c*)

Set both the edgecolor and the facecolor.

**Parameters**

**c**

[*color* or list of RGBA tuples]

**See also:**

***Collection.set_facecolor*, *Collection.set_edgecolor***

For setting the edge or face color individually.

**set_dashes**(*ls*)

Alias for *set_linestyle*.

**set_ec**(*c*)

Alias for *set_edgecolor*.

**set_edgecolor**(*c*)

Set the edgecolor(s) of the collection.

**Parameters**

**c**

[*color* or list of *color* or 'face'] The collection edgecolor(s). If a sequence, the patches cycle through it. If 'face', match the facecolor.

**set_edgecolors**(*c*)

Alias for *set_edgecolor*.

**set_facecolor**(*c*)

Set the facecolor(s) of the collection. *c* can be a color (all patches have same color), or a sequence of colors; if it is a sequence the patches will cycle through the sequence.

If *c* is 'none', the patch will not be filled.

**Parameters**

**c**

[*color* or list of *color*]

**set_facecolors**(*c*)

Alias for *set_facecolor*.

**set_fc**(*c*)

Alias for *set_facecolor*.

**set_figure**(*fig*)

Set the *Figure* instance the artist belongs to.

**Parameters**

**fig**

[*Figure*]

**set_gid**(*gid*)

Set the (group) id for the artist.

**Parameters**

**gid**

[str]

**set_hatch**(*hatch*)

Set the hatching pattern

*hatch* can be one of:

```
/   - diagonal hatching
\   - back diagonal
|   - vertical
-   - horizontal
+   - crossed
x   - crossed diagonal
o   - small circle
O   - large circle
.   - dots
*   - stars
```

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

**Parameters**

**hatch**

[{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}]

**set_in_layout**(*in_layout*)

Set if artist is to be included in layout calculations, E.g. *Constrained layout guide*, *Figure.tight_layout()*, and `fig.savefig(fname, bbox_inches='tight')`.

**Parameters**

**in_layout**

[bool]

**set_joinstyle**(*js*)

Set the *JoinStyle* for the collection (for all its elements).

**Parameters**

**js**

[*JoinStyle* or {'miter', 'round', 'bevel'}]

**set_label**(*s*)

Set a label that will be displayed in the legend.

**Parameters**

**s**

[object] *s* will be converted to a string by calling `str`.

**set_linestyle**(*ls*)

Set the linestyle(s) for the collection.

| linestyle | description |
|---|---|
| `'-'` or `'solid'` | solid line |
| `'--'` or `'dashed'` | dashed line |
| `'-.'` or `'dashdot'` | dash-dotted line |
| `':'` or `'dotted'` | dotted line |

Alternatively a dash tuple of the following form can be provided:

```
(offset, onoffseq),
```

where `onoffseq` is an even length tuple of on and off ink in points.

**Parameters**

**ls**

[str or tuple or list thereof] Valid values for individual linestyles include {'-', '--', '-.', ':', '', (offset, on-off-seq)}. See *Line2D.set_linestyle* for a complete description.

**set_linestyles**(*ls*)

Alias for *set_linestyle*.

**set_linewidth**(*lw*)

Set the linewidth(s) for the collection. *lw* can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

**Parameters**

**lw**

[float or list of floats]

**set_linewidths**(*lw*)

Alias for *set_linewidth*.

**set_ls**(*ls*)

Alias for *set_linestyle*.

**set_lw**(*lw*)

Alias for *set_linewidth*.

**set_mouseover**(*mouseover*)

Set whether this artist is queried for custom context information when the mouse cursor moves over it.

**Parameters**

**mouseover**

[bool]

**See also:**

***get_cursor_data***
***ToolCursorPosition***
***NavigationToolbar2***

**set_norm**(*norm*)

Set the normalization instance.

**Parameters**

**norm**

[*Normalize* or str or None]

### Notes

If there are any colorbars using the mappable for this norm, setting the norm of the mappable will reset the norm, locator, and formatters on the colorbar to default.

**set_offset_transform**(*offset_transform*)

Set the artist offset transform.

**Parameters**

**offset_transform**

[*Transform*]

**set_offsets**(*offsets*)

Set the offsets for the collection.

**Parameters**

**offsets**

[(N, 2) or (2,) array-like]

**set_path_effects**(*path_effects*)

Set the path effects.

**Parameters**

**path_effects**

[list of *AbstractPathEffect*]

**set_paths**(*paths*)

**set_picker**(*picker*)

Define the picking behavior of the artist.

**Parameters**

**picker**

[None or bool or float or callable] This can be one of the following:

- *None*: Picking is disabled for this artist (default).
- A boolean: If *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist.
- A float: If picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if its data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event

- A function: If picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

to determine the hit test. if the mouse event is over the artist, return *hit=True* and props is a dictionary of properties you want added to the PickEvent attributes.

**set_pickradius**(*pickradius*)

Set the pick radius used for containment tests.

**Parameters**

**pickradius**

[float] Pick radius, in points.

**set_rasterized**(*rasterized*)

Force rasterized (bitmap) drawing for vector graphics output.

Rasterized drawing is not supported by all artists. If you try to enable this on an artist that does not support it, the command has no effect and a warning will be issued.

This setting is ignored for pixel-based output.

See also *Rasterization for vector graphics*.

**Parameters**

**rasterized**

[bool]

**set_sizes**(*sizes*, *dpi=72.0*)

Set the sizes of each member of the collection.

**Parameters**

**sizes**

[numpy.ndarray or None] The size to set for each element of the collection. The value is the 'area' of the element.

**dpi**

[float, default: 72] The dpi of the canvas.

**set_sketch_params**(*scale=None*, *length=None*, *randomness=None*)

Set the sketch parameters.

**Parameters**

**scale**

[float, optional] The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is `None`, or not provided, no sketch filter will be provided.

**length**

[float, optional] The length of the wiggle along the line, in pixels (default 128.0)

**randomness**

[float, optional] The scale factor by which the length is shrunken or expanded (default 16.0)

The PGF backend uses this argument as an RNG seed and not as described above. Using the same seed yields the same random shape.

**set_snap**(*snap*)

Set the snapping behavior.

Snapping aligns positions with the pixel grid, which results in clearer images. For example, if a black line of 1px width was defined at a position in between two pixels, the resulting image would contain the interpolated value of that line in the pixel grid, which would be a grey value on both adjacent pixel positions. In contrast, snapping will move the line to the nearest integer pixel value, so that the resulting image will really contain a 1px wide black line.

Snapping is currently only supported by the Agg and MacOSX backends.

**Parameters**

**snap**

[bool or None] Possible values:

- *True*: Snap vertices to the nearest pixel center.
- *False*: Do not modify vertex positions.
- *None*: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center.

**set_transOffset**(*offset_transform*)

Alias for *set_offset_transform*.

**set_transform**(*t*)

Set the artist transform.

**Parameters**

**t**

[*Transform*]

**set_url**(*url*)

Set the url for the artist.

**Parameters**

**url**

[str]

**set_urls**(*urls*)

**Parameters**

**urls**

[list of str or None]

### Notes

URLs are currently only implemented by the SVG backend. They are ignored by all other backends.

**set_visible**(*b*)

Set the artist's visibility.

**Parameters**

**b**

[bool]

**set_zorder**(*level*)

Set the zorder for the artist. Artists with lower zorder values are drawn first.

**Parameters**

**level**

[float]

**property stale**

Whether the artist is 'stale' and needs to be re-drawn for the output to match the internal state of the artist.

**property sticky_edges**

`x` and `y` sticky edge lists for autoscaling.

When performing autoscaling, if a data limit coincides with a value in the corresponding sticky_edges list, then no margin will be added--the view limit "sticks" to the edge. A typical use case is histograms, where one usually expects no margin on the bottom edge (0) of the histogram.

Moreover, margin expansion "bumps" against sticky edges and cannot cross them. For example, if the upper data limit is 1.0, the upper view limit computed by simple margin application is 1.2, but there is a sticky edge at 1.1, then the actual upper view limit will be 1.1.

This attribute cannot be assigned to; however, the `x` and `y` lists can be modified in place as needed.

### Examples

```
>>> artist.sticky_edges.x[:] = (xmin, xmax)
>>> artist.sticky_edges.y[:] = (ymin, ymax)
```

**to_rgba**(*x*, *alpha=None*, *bytes=False*, *norm=True*)

Return a normalized RGBA array corresponding to *x*.

In the normal case, *x* is a 1D or 2D sequence of scalars, and the corresponding `ndarray` of RGBA values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already RGB or RGBA, such as might have been read from an image file. If *x* is an `ndarray` with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an RGB or RGBA array, and no mapping will be done. The array can be `uint8`, or it can be floats with values in the 0-1 range; otherwise a ValueError will be raised. Any NaNs or masked elements will be set to 0 alpha. If the last dimension is 3, the *alpha* kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the *alpha* kwarg is ignored; it does not replace the preexisting alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if *bytes* is *False* (default), the RGBA array will be floats in the 0-1 range; if it is *True*, the returned RGBA array will be `uint8` in the 0 to 255 range.

If norm is False, no normalization of the input data is performed, and it is assumed to be in the range (0-1).

**update**(*props*)

Update this artist's properties from the dict *props*.

**Parameters**

**props**

[dict]

**update_from**(*other*)

Copy properties from other to self.

**update_scalarmappable**()

Update colors from the scalar mappable array, if any.

Assign colors to edges and faces based on the array and/or colors that were directly set, as appropriate.

**zorder = 0**

**class** matplotlib.collections.**StarPolygonCollection**(*numsides*, *, *rotation=0*, *sizes=(1,)*, ***kwargs*)

Bases: *RegularPolyCollection*

Draw a collection of regular stars with *numsides* points.

**Parameters**

**numsides**

[int] The number of sides of the polygon.

**rotation**

[float] The rotation of the polygon in radians.

**sizes**

[tuple of float] The area of the circle circumscribing the polygon in points^2.

****kwargs**

Forwarded to *Collection*.

## Examples

See *Lasso Demo* for a complete example:

```
offsets = np.random.rand(20, 2)
facecolors = [cm.jet(x) for x in np.random.rand(20)]

collection = RegularPolyCollection(
    numsides=5, # a pentagon
    rotation=0, sizes=(50,),
    facecolors=facecolors,
    edgecolors=("black",),
    linewidths=(1,),
    offsets=offsets,
    offset_transform=ax.transData,
    )
```

**add_callback**(*func*)

Add a callback function that will be called whenever one of the *Artist*'s properties changes.

**Parameters**

**func**

[callable] The callback function. It must have the signature:

```
def func(artist: Artist) -> Any
```

where *artist* is the calling *Artist*. Return values may exist but are ignored.

**Returns**

**int**

The observer id associated with the callback. This id can be used for removing the callback with *remove_callback* later.

**See also:**

***remove_callback***

**autoscale**()

Autoscale the scalar limits on the norm instance using the current array

**autoscale_None**()

Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

**property axes**

The *Axes* instance the artist resides in, or *None*.

**changed**()

Call this whenever the mappable is changed to notify all the callbackSM listeners to the 'changed' signal.

**colorbar**

The last colorbar associated with this ScalarMappable. May be None.

**contains**(*mouseevent*)

Test whether the mouse event occurred in the collection.

Returns `bool, dict(ind=itemlist)`, where every item in itemlist contains the event.

**convert_xunits**(*x*)

Convert *x* using the unit type of the xaxis.

If the artist is not contained in an Axes or if the xaxis does not have units, *x* itself is returned.

**convert_yunits**(*y*)

Convert *y* using the unit type of the yaxis.

If the artist is not contained in an Axes or if the yaxis does not have units, *y* itself is returned.

**draw**(*renderer*)

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

**Parameters**

**renderer**

[*RendererBase* subclass.]

### Notes

This method is overridden in the Artist subclasses.

**findobj**(*match=None*, *include_self=True*)

Find artist objects.

Recursively find all *Artist* instances contained in the artist.

**Parameters**

**match**

A filter criterion for the matches. This can be

- *None*: Return all objects contained in artist.
- A function with signature `def match(artist: Artist) -> bool`. The result will only contain artists for which the function returns *True*.
- A class instance: e.g., *Line2D*. The result will only contain artists of this class or its subclasses (`isinstance` check).

**include_self**

[bool] Include *self* in the list to be checked for a match.

**Returns**

**list of *Artist***

**format_cursor_data**(*data*)

Return a string representation of *data*.

---

**Note:** This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

---

The default implementation converts ints and floats and arrays of ints and floats into a comma-separated string enclosed in square brackets, unless the artist has an associated colorbar, in which case scalar values are formatted using the colorbar's formatter.

**See also:**

***get_cursor_data***

**get_aa**()

Alias for *get_antialiased*.

**get_agg_filter**()

Return filter function to be used for agg filter.

**get_alpha**()

Return the alpha value used for blending - not supported on all backends.

**get_animated**()

Return whether the artist is animated.

**get_antialiased**()

Get the antialiasing state for rendering.

**Returns**

**array of bools**

**get_antialiaseds**()

Alias for *get_antialiased*.

**get_array**()

Return the array of values, that are mapped to colors.

The base class *ScalarMappable* does not make any assumptions on the dimensionality and shape of the array.

**get_capstyle**()

Return the cap style for the collection (for all its elements).

**Returns**

**{'butt', 'projecting', 'round'} or None**

**get_children**()

Return a list of the child *Artist*s of this *Artist*.

**get_clim**()

Return the values (min, max) that are mapped to the colormap limits.

**get_clip_box**()

Return the clipbox.

**get_clip_on**()

Return whether the artist uses clipping.

**get_clip_path**()

Return the clip path.

**get_cmap**()

Return the *Colormap* instance.

**get_cursor_data**(*event*)

Return the cursor data for a given event.

**Note:** This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

Cursor data can be used by Artists to provide additional context information for a given event. The default implementation just returns *None*.

Subclasses can override the method and return arbitrary data. However, when doing so, they must ensure that *format_cursor_data* can convert the data to a string representation.

The only current use case is displaying the z-value of an *AxesImage* in the status bar of a plot window, while moving the mouse.

**Parameters**

**event**

[*MouseEvent*]

**See also:**

***format_cursor_data***

**get_dashes**()
Alias for *get_linestyle*.

**get_datalim**(*transData*)

**get_ec**()
Alias for *get_edgecolor*.

**get_edgecolor**()

**get_edgecolors**()
Alias for *get_edgecolor*.

**get_facecolor**()

**get_facecolors**()
Alias for *get_facecolor*.

**get_fc**()
Alias for *get_facecolor*.

**get_figure**()
Return the *Figure* instance the artist belongs to.

**get_fill**()
Return whether face is colored.

**get_gid**()

Return the group id.

**get_hatch**()

Return the current hatching pattern.

**get_in_layout**()

Return boolean flag, `True` if artist is included in layout calculations.

E.g. *Constrained layout guide*, *`Figure.tight_layout()`*, and `fig.savefig(fname, bbox_inches='tight')`.

**get_joinstyle**()

Return the join style for the collection (for all its elements).

**Returns**

**{'miter', 'round', 'bevel'} or None**

**get_label**()

Return the label used for this artist in the legend.

**get_linestyle**()

**get_linestyles**()

Alias for *`get_linestyle`*.

**get_linewidth**()

**get_linewidths**()

Alias for *`get_linewidth`*.

**get_ls**()

Alias for *`get_linestyle`*.

**get_lw**()

Alias for *`get_linewidth`*.

**get_mouseover**()

Return whether this artist is queried for custom context information when the mouse cursor moves over it.

**get_numsides**()

**get_offset_transform**()

Return the *`Transform`* instance used by this artist offset.

**get_offsets**()

Return the offsets for the collection.

**get_path_effects**()

**get_paths**()

**get_picker**()

Return the picking behavior of the artist.

The possible values are described in *Artist.set_picker*.

**See also:**

***Artist.set_picker***, ***Artist.pickable***, ***Artist.pick***

**get_pickradius**()

**get_rasterized**()

Return whether the artist is to be rasterized.

**get_rotation**()

**get_sizes**()

Return the sizes ('areas') of the elements in the collection.

**Returns**

**array**

The 'area' of each element.

**get_sketch_params**()

Return the sketch parameters for the artist.

**Returns**

**tuple or None**

A 3-tuple with the following elements:

- *scale*: The amplitude of the wiggle perpendicular to the source line.
- *length*: The length of the wiggle along the line.
- *randomness*: The scale factor by which the length is shrunken or expanded.

Returns *None* if no sketch parameters were set.

**get_snap**()

Return the snap setting.

See *set_snap* for details.

**get_tightbbox**(*renderer=None*)

Like *Artist.get_window_extent*, but includes any clipping.

**Parameters**

**renderer**

[*RendererBase* subclass, optional] renderer that will be used to draw the figures (i.e. `fig.canvas.get_renderer()`)

**Returns**

***Bbox* or None**

The enclosing bounding box (in figure pixel coordinates). Returns None if clipping results in no intersection.

**get_transOffset**()

Alias for *get_offset_transform*.

**get_transform**()

Return the *Transform* instance used by this artist.

**get_transformed_clip_path_and_affine**()

Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

**get_transforms**()

**get_url**()

Return the url.

**get_urls**()

Return a list of URLs, one for each element of the collection.

The list contains *None* for elements without a URL. See *Hyperlinks* for an example.

**get_visible**()

Return the visibility.

**get_window_extent**(*renderer=None*)

Get the artist's bounding box in display space.

The bounding box' width and height are nonnegative.

Subclasses should override for inclusion in the bounding box "tight" calculation. Default is to return an empty bounding box at 0, 0.

Be careful when using this function, the results will not update if the artist window extent of the artist changes. The extent can change due to any changes in the transform stack, such as changing the Axes limits, the figure size, or the canvas used (as is done when saving a figure). This can lead to unexpected behavior where interactive figures will look fine on the screen, but will save incorrectly.

**get_zorder**()

Return the artist's zorder.

**have_units**()

Return whether units are set on any axis.

**is_transform_set**()

Return whether the Artist has an explicitly set transform.

This is *True* after *set_transform* has been called.

**property mouseover**

Return whether this artist is queried for custom context information when the mouse cursor moves over it.

**property norm**

**pchanged**()

Call all of the registered callbacks.

This function is triggered internally when a property is changed.

**See also:**

***add_callback***
***remove_callback***

**pick**(*mouseevent*)

Process a pick event.

Each child artist will fire a pick event if *mouseevent* is over the artist and the artist has picker set.

**See also:**

***Artist.set_picker***, ***Artist.get_picker***, ***Artist.pickable***

**pickable**()

Return whether the artist is pickable.

**See also:**

***Artist.set_picker***, ***Artist.get_picker***, ***Artist.pick***

**properties**()

Return a dictionary of all the properties of the artist.

**remove**()

Remove the artist from the figure if possible.

The effect will not be visible until the figure is redrawn, e.g., with *FigureCanvasBase.draw_idle*. Call *relim* to update the Axes limits if desired.

Note: *relim* will not see collections even if the collection was added to the Axes with *autolim* = True.

Note: there is no support for removing the artist's legend entry.

**remove_callback**(*oid*)

Remove a callback based on its observer id.

**See also:**

*add_callback*

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *antialiased=<UNSET>*, *array=<UNSET>*, *capstyle=<UNSET>*, *clim=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *cmap=<UNSET>*, *color=<UNSET>*, *edgecolor=<UNSET>*, *facecolor=<UNSET>*, *gid=<UNSET>*, *hatch=<UNSET>*, *in_layout=<UNSET>*, *joinstyle=<UNSET>*, *label=<UNSET>*, *linestyle=<UNSET>*, *linewidth=<UNSET>*, *mouseover=<UNSET>*, *norm=<UNSET>*, *offset_transform=<UNSET>*, *offsets=<UNSET>*, *path_effects=<UNSET>*, *paths=<UNSET>*, *picker=<UNSET>*, *pickradius=<UNSET>*, *rasterized=<UNSET>*, *sizes=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *urls=<UNSET>*, *visible=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi v |
| *alpha* | array-like or scalar or None |
| *animated* | bool |
| *antialiased* or aa or antialiaseds | bool or list of bools |
| *array* | array-like or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clim* | (vmin: float, vmax: float) |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *cmap* | *Colormap* or str or None |
| *color* | *color* or list of RGBA tuples |
| *edgecolor* or ec or edgecolors | *color* or list of *color* or 'face' |
| *facecolor* or facecolors or fc | *color* or list of *color* |
| *figure* | *Figure* |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or dashes or linestyles or ls | str or tuple or list thereof |
| *linewidth* or linewidths or lw | float or list of floats |
| *mouseover* | bool |
| *norm* | *Normalize* or str or None |

Table 45 – continued from p

| Property | Description |
|---|---|
| *offset_transform* or transOffset | *Transform* |
| *offsets* | (N, 2) or (2,) array-like |
| *path_effects* | list of *AbstractPathEffect* |
| *paths* | unknown |
| *picker* | None or bool or float or callable |
| *pickradius* | float |
| *rasterized* | bool |
| sizes | numpy.ndarray or None |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *urls* | list of str or None |
| *visible* | bool |
| *zorder* | float |

**set_aa**(*aa*)

Alias for *set_antialiased*.

**set_agg_filter**(*filter_func*)

Set the agg filter.

**Parameters**

**filter_func**

[callable] A filter function, which takes a (m, n, depth) float array and a dpi value, and returns a (m, n, depth) array and two offsets from the bottom left corner of the image

**set_alpha**(*alpha*)

Set the alpha value used for blending - not supported on all backends.

**Parameters**

**alpha**

[array-like or scalar or None] All values must be within the 0-1 range, inclusive. Masked values and nans are not supported.

**set_animated**(*b*)

Set whether the artist is intended to be used in an animation.

If True, the artist is excluded from regular drawing of the figure. You have to call *Figure.draw_artist* / *Axes.draw_artist* explicitly on the artist. This approach is used to speed up animations using blitting.

See also *matplotlib.animation* and *Faster rendering by using blitting*.

**Parameters**

**b**

[bool]

**set_antialiased**(*aa*)

Set the antialiasing state for rendering.

**Parameters**

**aa**

[bool or list of bools]

**set_antialiaseds**(*aa*)

Alias for *set_antialiased*.

**set_array**(*A*)

Set the value array from array-like *A*.

**Parameters**

**A**

[array-like or None] The values that are mapped to colors.

The base class *ScalarMappable* does not make any assumptions on the dimensionality and shape of the value array *A*.

**set_capstyle**(*cs*)

Set the *CapStyle* for the collection (for all its elements).

**Parameters**

**cs**

[*CapStyle* or {'butt', 'projecting', 'round'}]

**set_clim**(*vmin=None*, *vmax=None*)

Set the norm limits for image scaling.

**Parameters**

**vmin, vmax**

[float] The limits.

The limits may also be passed as a tuple (*vmin*, *vmax*) as a single positional argument.

**set_clip_box**(*clipbox*)

Set the artist's clip *Bbox*.

**Parameters**

**clipbox**

[*BboxBase* or None] Will typically be created from a *TransformedBbox*. For instance, `TransformedBbox(Bbox([[0, 0], [1, 1]]), ax.transAxes)` is the default clipping for an artist added to an Axes.

**set_clip_on**(*b*)

Set whether the artist uses clipping.

When False, artists will be visible outside the Axes which can lead to unexpected results.

**Parameters**

**b**

[bool]

**set_clip_path**(*path*, *transform=None*)

Set the artist's clip path.

**Parameters**

**path**

[*Patch* or *Path* or *TransformedPath* or None] The clip path. If given a *Path*, *transform* must be provided as well. If *None*, a previously set clip path is removed.

**transform**

[*Transform*, optional] Only used if *path* is a *Path*, in which case the given *Path* is converted to a *TransformedPath* using *transform*.

### Notes

For efficiency, if *path* is a *Rectangle* this method will set the clipping box to the corresponding rectangle and set the clipping path to `None`.

For technical reasons (support of *set*), a tuple (*path*, *transform*) is also accepted as a single positional parameter.

**set_cmap**(*cmap*)

Set the colormap for luminance data.

**Parameters**

**cmap**

[*Colormap* or str or None]

**set_color**(*c*)

Set both the edgecolor and the facecolor.

**Parameters**

**c**

[*color* or list of RGBA tuples]

**See also:**

***Collection.set_facecolor*, *Collection.set_edgecolor***

For setting the edge or face color individually.

**set_dashes**(*ls*)

Alias for *set_linestyle*.

**set_ec**(*c*)

Alias for *set_edgecolor*.

**set_edgecolor**(*c*)

Set the edgecolor(s) of the collection.

**Parameters**

**c**

[*color* or list of *color* or 'face'] The collection edgecolor(s). If a sequence, the patches cycle through it. If 'face', match the facecolor.

**set_edgecolors**(*c*)

Alias for *set_edgecolor*.

**set_facecolor**(*c*)

Set the facecolor(s) of the collection. *c* can be a color (all patches have same color), or a sequence of colors; if it is a sequence the patches will cycle through the sequence.

If *c* is 'none', the patch will not be filled.

**Parameters**

**c**

[*color* or list of *color*]

**set_facecolors**(*c*)

Alias for *set_facecolor*.

**set_fc**(*c*)

Alias for *set_facecolor*.

**set_figure**(*fig*)

Set the *Figure* instance the artist belongs to.

**Parameters**

**fig**

[*Figure*]

**set_gid**(*gid*)

Set the (group) id for the artist.

**Parameters**

**gid**

[str]

**set_hatch**(*hatch*)

Set the hatching pattern

*hatch* can be one of:

```
/   - diagonal hatching
\   - back diagonal
|   - vertical
-   - horizontal
+   - crossed
x   - crossed diagonal
o   - small circle
O   - large circle
.   - dots
*   - stars
```

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

**Parameters**

**hatch**

[{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}]

**set_in_layout**(*in_layout*)

Set if artist is to be included in layout calculations, E.g. *Constrained layout guide*, *Figure.tight_layout()*, and `fig.savefig(fname, bbox_inches='tight')`.

**Parameters**

**in_layout**

[bool]

**set_joinstyle**(*js*)

Set the *JoinStyle* for the collection (for all its elements).

**Parameters**

**js**

[*JoinStyle* or {'miter', 'round', 'bevel'}]

**set_label**(*s*)

Set a label that will be displayed in the legend.

**Parameters**

**s**

[object] *s* will be converted to a string by calling `str`.

**set_linestyle**(*ls*)

Set the linestyle(s) for the collection.

| linestyle | description |
|---|---|
| `'-'` or `'solid'` | solid line |
| `'--'` or `'dashed'` | dashed line |
| `'-.'` or `'dashdot'` | dash-dotted line |
| `':'` or `'dotted'` | dotted line |

Alternatively a dash tuple of the following form can be provided:

```
(offset, onoffseq),
```

where `onoffseq` is an even length tuple of on and off ink in points.

**Parameters**

**ls**

[str or tuple or list thereof] Valid values for individual linestyles include {'-', '--', '-.', ':', '', (offset, on-off-seq)}. See *Line2D.set_linestyle* for a complete description.

**set_linestyles**(*ls*)

Alias for *set_linestyle*.

**set_linewidth**(*lw*)

Set the linewidth(s) for the collection. *lw* can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

**Parameters**

**lw**

[float or list of floats]

**set_linewidths**(*lw*)

Alias for *set_linewidth*.

**set_ls**(*ls*)

Alias for *set_linestyle*.

**set_lw**(*lw*)

Alias for *set_linewidth*.

**set_mouseover**(*mouseover*)

Set whether this artist is queried for custom context information when the mouse cursor moves over it.

**Parameters**

**mouseover**

[bool]

**See also:**

***get_cursor_data***
***ToolCursorPosition***
***NavigationToolbar2***

**set_norm**(*norm*)

Set the normalization instance.

**Parameters**

**norm**

[*Normalize* or str or None]

### Notes

If there are any colorbars using the mappable for this norm, setting the norm of the mappable will reset the norm, locator, and formatters on the colorbar to default.

**set_offset_transform**(*offset_transform*)

Set the artist offset transform.

**Parameters**

**offset_transform**

[*Transform*]

**set_offsets**(*offsets*)

Set the offsets for the collection.

**Parameters**

**offsets**

[(N, 2) or (2,) array-like]

**set_path_effects**(*path_effects*)

Set the path effects.

**Parameters**

**path_effects**

[list of *AbstractPathEffect*]

**set_paths**(*paths*)

**set_picker**(*picker*)

Define the picking behavior of the artist.

**Parameters**

**picker**

[None or bool or float or callable] This can be one of the following:

- *None*: Picking is disabled for this artist (default).
- A boolean: If *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist.
- A float: If picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if its data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event

- A function: If picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

to determine the hit test. if the mouse event is over the artist, return *hit=True* and props is a dictionary of properties you want added to the PickEvent attributes.

**set_pickradius** (*pickradius*)

Set the pick radius used for containment tests.

**Parameters**

**pickradius**

[float] Pick radius, in points.

**set_rasterized** (*rasterized*)

Force rasterized (bitmap) drawing for vector graphics output.

Rasterized drawing is not supported by all artists. If you try to enable this on an artist that does not support it, the command has no effect and a warning will be issued.

This setting is ignored for pixel-based output.

See also *Rasterization for vector graphics*.

**Parameters**

**rasterized**

[bool]

**set_sizes** (*sizes*, *dpi=72.0*)

Set the sizes of each member of the collection.

**Parameters**

**sizes**

[numpy.ndarray or None] The size to set for each element of the collection. The value is the 'area' of the element.

**dpi**

[float, default: 72] The dpi of the canvas.

**set_sketch_params** (*scale=None*, *length=None*, *randomness=None*)

Set the sketch parameters.

**Parameters**

**scale**

[float, optional] The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is `None`, or not provided, no sketch filter will be provided.

**length**

[float, optional] The length of the wiggle along the line, in pixels (default 128.0)

**randomness**

[float, optional] The scale factor by which the length is shrunken or expanded (default 16.0)

The PGF backend uses this argument as an RNG seed and not as described above. Using the same seed yields the same random shape.

**set_snap**(*snap*)

Set the snapping behavior.

Snapping aligns positions with the pixel grid, which results in clearer images. For example, if a black line of 1px width was defined at a position in between two pixels, the resulting image would contain the interpolated value of that line in the pixel grid, which would be a grey value on both adjacent pixel positions. In contrast, snapping will move the line to the nearest integer pixel value, so that the resulting image will really contain a 1px wide black line.

Snapping is currently only supported by the Agg and MacOSX backends.

**Parameters**

**snap**

[bool or None] Possible values:

- *True*: Snap vertices to the nearest pixel center.
- *False*: Do not modify vertex positions.
- *None*: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center.

**set_transOffset**(*offset_transform*)

Alias for *set_offset_transform*.

**set_transform**(*t*)

Set the artist transform.

**Parameters**

**t**

[*Transform*]

**set_url**(*url*)

Set the url for the artist.

**Parameters**

**url**

[str]

**set_urls**(*urls*)

**Parameters**

**urls**

[list of str or None]

### Notes

URLs are currently only implemented by the SVG backend. They are ignored by all other backends.

**set_visible**(*b*)

Set the artist's visibility.

**Parameters**

**b**

[bool]

**set_zorder**(*level*)

Set the zorder for the artist. Artists with lower zorder values are drawn first.

**Parameters**

**level**

[float]

**property stale**

Whether the artist is 'stale' and needs to be re-drawn for the output to match the internal state of the artist.

**property sticky_edges**

`x` and `y` sticky edge lists for autoscaling.

When performing autoscaling, if a data limit coincides with a value in the corresponding sticky_edges list, then no margin will be added--the view limit "sticks" to the edge. A typical use case is histograms, where one usually expects no margin on the bottom edge (0) of the histogram.

Moreover, margin expansion "bumps" against sticky edges and cannot cross them. For example, if the upper data limit is 1.0, the upper view limit computed by simple margin application is 1.2, but there is a sticky edge at 1.1, then the actual upper view limit will be 1.1.

This attribute cannot be assigned to; however, the `x` and `y` lists can be modified in place as needed.

### Examples

```
>>> artist.sticky_edges.x[:] = (xmin, xmax)
>>> artist.sticky_edges.y[:] = (ymin, ymax)
```

**to_rgba**(*x*, *alpha=None*, *bytes=False*, *norm=True*)

Return a normalized RGBA array corresponding to *x*.

In the normal case, *x* is a 1D or 2D sequence of scalars, and the corresponding `ndarray` of RGBA values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already RGB or RGBA, such as might have been read from an image file. If *x* is an `ndarray` with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an RGB or RGBA array, and no mapping will be done. The array can be `uint8`, or it can be floats with values in the 0-1 range; otherwise a ValueError will be raised. Any NaNs or masked elements will be set to 0 alpha. If the last dimension is 3, the *alpha* kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the *alpha* kwarg is ignored; it does not replace the preexisting alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if *bytes* is *False* (default), the RGBA array will be floats in the 0-1 range; if it is *True*, the returned RGBA array will be `uint8` in the 0 to 255 range.

If norm is False, no normalization of the input data is performed, and it is assumed to be in the range (0-1).

**update**(*props*)

Update this artist's properties from the dict *props*.

**Parameters**

**props**

[dict]

**update_from**(*other*)

Copy properties from other to self.

**update_scalarmappable**()

Update colors from the scalar mappable array, if any.

Assign colors to edges and faces based on the array and/or colors that were directly set, as appropriate.

**zorder = 0**

**class** matplotlib.collections.**TriMesh**(*triangulation*, ***kwargs*)

Bases: *Collection*

Class for the efficient drawing of a triangular mesh using Gouraud shading.

A triangular mesh is a *Triangulation* object.

**Parameters**

**edgecolors**

[*color* or list of colors, default: *rcParams["patch.edgecolor"]* (default: `'black'`)] Edge color for each patch making up the collection. The special value 'face' can be passed to make the edgecolor match the facecolor.

**facecolors**

[*color* or list of colors, default: *rcParams["patch.facecolor"]* (default: `'C0'`)] Face color for each patch making up the collection.

**linewidths**

[float or list of floats, default: *rcParams["patch.linewidth"]* (default: `1.0`)] Line width for each patch making up the collection.

**linestyles**

[str or tuple or list thereof, default: 'solid'] Valid strings are ['solid', 'dashed', 'dashdot', 'dotted', '-', '--', '-.', ':']. Dash tuples should be of the form:

```
(offset, onoffseq),
```

where *onoffseq* is an even length tuple of on and off ink lengths in points. For examples, see *Linestyles*.

**capstyle**

[*CapStyle*-like, default: *rcParams["patch.capstyle"]*] Style to use for capping lines for all paths in the collection. Allowed values are {'butt', 'projecting', 'round'}.

**joinstyle**

[*JoinStyle*-like, default: *rcParams["patch.joinstyle"]*] Style to use for joining lines for all paths in the collection. Allowed values are {'miter', 'round', 'bevel'}.

**antialiaseds**

[bool or list of bool, default: *rcParams["patch.antialiased"]* (default: `True`)] Whether each patch in the collection should be drawn with antialiasing.

**offsets**

[(float, float) or list thereof, default: (0, 0)] A vector by which to translate each patch after rendering (default is no translation). The translation is performed in screen (pixel) coordinates (i.e. after the Artist's transform is applied).

**offset_transform**

[*Transform*, default: *IdentityTransform*] A single transform which will be applied to each *offsets* vector before it is used.

**cmap, norm**

Data normalization and colormapping parameters. See *ScalarMappable* for a detailed description.

**hatch**

[str, optional] Hatching pattern to use in filled paths, if any. Valid strings are ['/', '', '|', '-', '+', 'x', 'o', 'O', '.', '*']. See *Hatch style reference* for the meaning of each hatch type.

**pickradius**

[float, default: 5.0] If `pickradius <= 0`, then *Collection.contains* will return `True` whenever the test point is inside of one of the polygons formed by the control points of a Path in the Collection. On the other hand, if it is greater than 0, then we instead check if the test point is contained in a stroke of width `2*pickradius` following any of the Paths in the Collection.

**urls**

[list of str, default: None] A URL for each patch to link to once drawn. Currently only works for the SVG backend. See *Hyperlinks* for examples.

**zorder**

[float, default: 1] The drawing order, shared by all Patches in the Collection. See *Zorder Demo* for all defaults and examples.

****kwargs**

Remaining keyword arguments will be used to set properties as `Collection.set_{key}(val)` for each key-value pair in *kwargs*.

**add_callback**(*func*)

Add a callback function that will be called whenever one of the *Artist*'s properties changes.

**Parameters**

**func**

[callable] The callback function. It must have the signature:

```
def func(artist: Artist) -> Any
```

where *artist* is the calling *Artist*. Return values may exist but are ignored.

**Returns**

**int**

The observer id associated with the callback. This id can be used for removing the callback with *remove_callback* later.

**See also:**

***remove_callback***

**autoscale**()

Autoscale the scalar limits on the norm instance using the current array

**autoscale_None**()

Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

**property axes**

The *Axes* instance the artist resides in, or *None*.

**changed**()

Call this whenever the mappable is changed to notify all the callbackSM listeners to the 'changed' signal.

**colorbar**

The last colorbar associated with this ScalarMappable. May be None.

**contains**(*mouseevent*)

Test whether the mouse event occurred in the collection.

Returns `bool, dict(ind=itemlist)`, where every item in itemlist contains the event.

**static convert_mesh_to_paths**(*tri*)

Convert a given mesh into a sequence of *Path* objects.

This function is primarily of use to implementers of backends that do not directly support meshes.

**convert_xunits**(*x*)

Convert *x* using the unit type of the xaxis.

If the artist is not contained in an Axes or if the xaxis does not have units, *x* itself is returned.

**convert_yunits**(*y*)

Convert *y* using the unit type of the yaxis.

If the artist is not contained in an Axes or if the yaxis does not have units, *y* itself is returned.

**draw**(*renderer*)

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

**Parameters**

**renderer**

[*RendererBase* subclass.]

### Notes

This method is overridden in the Artist subclasses.

**findobj**(*match=None*, *include_self=True*)

Find artist objects.

Recursively find all *Artist* instances contained in the artist.

**Parameters**

**match**

A filter criterion for the matches. This can be

- *None*: Return all objects contained in artist.
- A function with signature `def match(artist: Artist) -> bool`. The result will only contain artists for which the function returns *True*.
- A class instance: e.g., *Line2D*. The result will only contain artists of this class or its subclasses (`isinstance` check).

**include_self**

[bool] Include *self* in the list to be checked for a match.

**Returns**

**list of *Artist***

**format_cursor_data**(*data*)

Return a string representation of *data*.

---

**Note:** This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

---

The default implementation converts ints and floats and arrays of ints and floats into a comma-separated string enclosed in square brackets, unless the artist has an associated colorbar, in which case scalar values are formatted using the colorbar's formatter.

**See also:**

***get_cursor_data***

**get_aa**()

Alias for *get_antialiased*.

**get_agg_filter**()

Return filter function to be used for agg filter.

**get_alpha**()

Return the alpha value used for blending - not supported on all backends.

**get_animated**()

Return whether the artist is animated.

**get_antialiased**()

Get the antialiasing state for rendering.

**Returns**

**array of bools**

**get_antialiaseds**()

Alias for *get_antialiased*.

**get_array**()

Return the array of values, that are mapped to colors.

The base class *ScalarMappable* does not make any assumptions on the dimensionality and shape of the array.

**get_capstyle**()

Return the cap style for the collection (for all its elements).

**Returns**

**{'butt', 'projecting', 'round'} or None**

**get_children**()

Return a list of the child *Artist*s of this *Artist*.

**get_clim**()

Return the values (min, max) that are mapped to the colormap limits.

**get_clip_box**()

Return the clipbox.

**get_clip_on**()

Return whether the artist uses clipping.

**get_clip_path**()

Return the clip path.

**get_cmap**()

Return the *Colormap* instance.

**get_cursor_data**(*event*)

Return the cursor data for a given event.

---

**Note:** This method is intended to be overridden by artist subclasses. As an end-user of Matplotlib you will most likely not call this method yourself.

---

Cursor data can be used by Artists to provide additional context information for a given event. The default implementation just returns *None*.

Subclasses can override the method and return arbitrary data. However, when doing so, they must ensure that *format_cursor_data* can convert the data to a string representation.

The only current use case is displaying the z-value of an *AxesImage* in the status bar of a plot window, while moving the mouse.

**Parameters**

**event**

[*MouseEvent*]

**See also:**

***format_cursor_data***

**get_dashes**()

Alias for *get_linestyle*.

**get_datalim**(*transData*)

**get_ec**()

Alias for *get_edgecolor*.

**get_edgecolor**()

**get_edgecolors**()

Alias for *get_edgecolor*.

**get_facecolor**()

**get_facecolors**()

Alias for *get_facecolor*.

**get_fc**()

Alias for *get_facecolor*.

**get_figure**()

Return the *Figure* instance the artist belongs to.

**get_fill**()

Return whether face is colored.

**get_gid**()

Return the group id.

**get_hatch**()

Return the current hatching pattern.

**get_in_layout**()

Return boolean flag, `True` if artist is included in layout calculations.

E.g. *Constrained layout guide*, *Figure.tight_layout()*, and `fig.savefig(fname, bbox_inches='tight')`.

**get_joinstyle**()

Return the join style for the collection (for all its elements).

**Returns**

**{'miter', 'round', 'bevel'} or None**

**get_label**()

Return the label used for this artist in the legend.

**get_linestyle**()

**get_linestyles**()

Alias for *get_linestyle*.

**get_linewidth**()

**get_linewidths**()

Alias for *get_linewidth*.

**get_ls**()

Alias for *get_linestyle*.

**get_lw**()

Alias for *get_linewidth*.

**get_mouseover**()

Return whether this artist is queried for custom context information when the mouse cursor moves over it.

**get_offset_transform**()

Return the *Transform* instance used by this artist offset.

**get_offsets**()

Return the offsets for the collection.

**get_path_effects**()

**get_paths**()

**get_picker**()

Return the picking behavior of the artist.

The possible values are described in *Artist.set_picker*.

**See also:**

***Artist.set_picker***, ***Artist.pickable***, ***Artist.pick***

**get_pickradius**()

**get_rasterized**()

Return whether the artist is to be rasterized.

**get_sketch_params**()

Return the sketch parameters for the artist.

**Returns**

**tuple or None**

A 3-tuple with the following elements:

- *scale*: The amplitude of the wiggle perpendicular to the source line.
- *length*: The length of the wiggle along the line.
- *randomness*: The scale factor by which the length is shrunken or expanded.

Returns *None* if no sketch parameters were set.

**get_snap**()

Return the snap setting.

See *set_snap* for details.

**get_tightbbox**(*renderer=None*)

Like *Artist.get_window_extent*, but includes any clipping.

**Parameters**

**renderer**

[*RendererBase* subclass, optional] renderer that will be used to draw the figures (i.e. `fig.canvas.get_renderer()`)

**Returns**

*Bbox* or None

The enclosing bounding box (in figure pixel coordinates). Returns None if clipping results in no intersection.

**get_transOffset**()

Alias for *get_offset_transform*.

**get_transform**()

Return the *Transform* instance used by this artist.

**get_transformed_clip_path_and_affine**()

Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

**get_transforms**()

**get_url**()

Return the url.

**get_urls**()

Return a list of URLs, one for each element of the collection.

The list contains *None* for elements without a URL. See *Hyperlinks* for an example.

**get_visible**()

Return the visibility.

**get_window_extent**(*renderer=None*)

Get the artist's bounding box in display space.

The bounding box' width and height are nonnegative.

Subclasses should override for inclusion in the bounding box "tight" calculation. Default is to return an empty bounding box at 0, 0.

Be careful when using this function, the results will not update if the artist window extent of the artist changes. The extent can change due to any changes in the transform stack, such as changing the Axes limits, the figure size, or the canvas used (as is done when saving a figure). This can lead to unexpected behavior where interactive figures will look fine on the screen, but will save incorrectly.

**get_zorder**()

Return the artist's zorder.

**have_units**()

Return whether units are set on any axis.

**is_transform_set**()

Return whether the Artist has an explicitly set transform.

This is *True* after *set_transform* has been called.

**property mouseover**

Return whether this artist is queried for custom context information when the mouse cursor moves over it.

**property norm**

**pchanged**()

Call all of the registered callbacks.

This function is triggered internally when a property is changed.

**See also:**

*add_callback*
*remove_callback*

**pick**(*mouseevent*)

Process a pick event.

Each child artist will fire a pick event if *mouseevent* is over the artist and the artist has picker set.

**See also:**

*Artist.set_picker*, *Artist.get_picker*, *Artist.pickable*

**pickable**()

Return whether the artist is pickable.

**See also:**

*Artist.set_picker*, *Artist.get_picker*, *Artist.pick*

**properties**()

Return a dictionary of all the properties of the artist.

**remove**()

Remove the artist from the figure if possible.

The effect will not be visible until the figure is redrawn, e.g., with *FigureCanvasBase.draw_idle*. Call *relim* to update the Axes limits if desired.

Note: *relim* will not see collections even if the collection was added to the Axes with *autolim* = True.

Note: there is no support for removing the artist's legend entry.

**remove_callback**(*oid*)

Remove a callback based on its observer id.

**See also:**

*add_callback*

**set** (*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *antialiased=<UNSET>*, *array=<UNSET>*, *capstyle=<UNSET>*, *clim=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *cmap=<UNSET>*, *color=<UNSET>*, *edgecolor=<UNSET>*, *facecolor=<UNSET>*, *gid=<UNSET>*, *hatch=<UNSET>*, *in_layout=<UNSET>*, *joinstyle=<UNSET>*, *label=<UNSET>*, *linestyle=<UNSET>*, *linewidth=<UNSET>*, *mouseover=<UNSET>*, *norm=<UNSET>*, *offset_transform=<UNSET>*, *offsets=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *pickradius=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *urls=<UNSET>*, *visible=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi v |
| *alpha* | array-like or scalar or None |
| *animated* | bool |
| *antialiased* or aa or antialiaseds | bool or list of bools |
| *array* | array-like or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clim* | (vmin: float, vmax: float) |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *cmap* | *Colormap* or str or None |
| *color* | *color* or list of RGBA tuples |
| *edgecolor* or ec or edgecolors | *color* or list of *color* or 'face' |
| *facecolor* or facecolors or fc | *color* or list of *color* |
| *figure* | *Figure* |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or dashes or linestyles or ls | str or tuple or list thereof |
| *linewidth* or linewidths or lw | float or list of floats |
| *mouseover* | bool |
| *norm* | *Normalize* or str or None |
| *offset_transform* or transOffset | *Transform* |
| *offsets* | (N, 2) or (2,) array-like |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *pickradius* | float |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |

Table 46 – continued from p

| Property | Description |
|---|---|
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *urls* | list of str or None |
| *visible* | bool |
| *zorder* | float |

**set_aa**(*aa*)

Alias for *set_antialiased*.

**set_agg_filter**(*filter_func*)

Set the agg filter.

**Parameters**

**filter_func**

[callable] A filter function, which takes a (m, n, depth) float array and a dpi value, and returns a (m, n, depth) array and two offsets from the bottom left corner of the image

**set_alpha**(*alpha*)

Set the alpha value used for blending - not supported on all backends.

**Parameters**

**alpha**

[array-like or scalar or None] All values must be within the 0-1 range, inclusive. Masked values and nans are not supported.

**set_animated**(*b*)

Set whether the artist is intended to be used in an animation.

If True, the artist is excluded from regular drawing of the figure. You have to call *Figure.draw_artist* / *Axes.draw_artist* explicitly on the artist. This approach is used to speed up animations using blitting.

See also *matplotlib.animation* and *Faster rendering by using blitting*.

**Parameters**

**b**

[bool]

**set_antialiased**(*aa*)

Set the antialiasing state for rendering.

**Parameters**

**aa**

[bool or list of bools]

**set_antialiaseds**(*aa*)

Alias for *set_antialiased*.

**set_array**(*A*)

Set the value array from array-like *A*.

**Parameters**

**A**

[array-like or None] The values that are mapped to colors.

The base class *ScalarMappable* does not make any assumptions on the dimensionality and shape of the value array *A*.

**set_capstyle**(*cs*)

Set the *CapStyle* for the collection (for all its elements).

**Parameters**

**cs**

[*CapStyle* or {'butt', 'projecting', 'round'}]

**set_clim**(*vmin=None*, *vmax=None*)

Set the norm limits for image scaling.

**Parameters**

**vmin, vmax**

[float] The limits.

The limits may also be passed as a tuple (*vmin*, *vmax*) as a single positional argument.

**set_clip_box**(*clipbox*)

Set the artist's clip *Bbox*.

**Parameters**

**clipbox**

[*BboxBase* or None] Will typically be created from a *TransformedBbox*. For instance, `TransformedBbox(Bbox([[0, 0], [1, 1]]), ax.transAxes)` is the default clipping for an artist added to an Axes.

**set_clip_on**(*b*)

Set whether the artist uses clipping.

When False, artists will be visible outside the Axes which can lead to unexpected results.

**Parameters**

**b**

[bool]

**set_clip_path**(*path*, *transform=None*)

Set the artist's clip path.

**Parameters**

**path**

[*Patch* or *Path* or *TransformedPath* or None] The clip path. If given a *Path*, *transform* must be provided as well. If *None*, a previously set clip path is removed.

**transform**

[*Transform*, optional] Only used if *path* is a *Path*, in which case the given *Path* is converted to a *TransformedPath* using *transform*.

### Notes

For efficiency, if *path* is a *Rectangle* this method will set the clipping box to the corresponding rectangle and set the clipping path to `None`.

For technical reasons (support of *set*), a tuple (*path*, *transform*) is also accepted as a single positional parameter.

**set_cmap**(*cmap*)

Set the colormap for luminance data.

**Parameters**

**cmap**

[*Colormap* or str or None]

**set_color**(*c*)

Set both the edgecolor and the facecolor.

**Parameters**

**c**

[*color* or list of RGBA tuples]

**See also:**

***Collection.set_facecolor*, *Collection.set_edgecolor***

For setting the edge or face color individually.

**set_dashes**(*ls*)

Alias for *set_linestyle*.

**set_ec**(*c*)

Alias for *set_edgecolor*.

**set_edgecolor**(*c*)

Set the edgecolor(s) of the collection.

**Parameters**

**c**

[*color* or list of *color* or 'face'] The collection edgecolor(s). If a sequence, the patches cycle through it. If 'face', match the facecolor.

**set_edgecolors**(*c*)

Alias for *set_edgecolor*.

**set_facecolor**(*c*)

Set the facecolor(s) of the collection. *c* can be a color (all patches have same color), or a sequence of colors; if it is a sequence the patches will cycle through the sequence.

If *c* is 'none', the patch will not be filled.

**Parameters**

**c**

[*color* or list of *color*]

**set_facecolors**(*c*)

Alias for *set_facecolor*.

**set_fc**(*c*)

Alias for *set_facecolor*.

**set_figure**(*fig*)

Set the *Figure* instance the artist belongs to.

**Parameters**

**fig**

[*Figure*]

**set_gid**(*gid*)

Set the (group) id for the artist.

**Parameters**

**gid**

[str]

**set_hatch**(*hatch*)

Set the hatching pattern

*hatch* can be one of:

```
/   - diagonal hatching
\   - back diagonal
|   - vertical
-   - horizontal
+   - crossed
x   - crossed diagonal
o   - small circle
O   - large circle
.   - dots
*   - stars
```

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

**Parameters**

**hatch**

[{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}]

**set_in_layout**(*in_layout*)

Set if artist is to be included in layout calculations, E.g. *Constrained layout guide*, *Figure.tight_layout()*, and `fig.savefig(fname, bbox_inches='tight')`.

**Parameters**

**in_layout**

[bool]

**set_joinstyle**(*js*)

Set the *JoinStyle* for the collection (for all its elements).

**Parameters**

**js**

[*JoinStyle* or {'miter', 'round', 'bevel'}]

**set_label**(*s*)

Set a label that will be displayed in the legend.

**Parameters**

**s**

[object] *s* will be converted to a string by calling `str`.

**set_linestyle**(*ls*)

Set the linestyle(s) for the collection.

| linestyle | description |
|---|---|
| `'-'` or `'solid'` | solid line |
| `'--'` or `'dashed'` | dashed line |
| `'-.'` or `'dashdot'` | dash-dotted line |
| `':'` or `'dotted'` | dotted line |

Alternatively a dash tuple of the following form can be provided:

```
(offset, onoffseq),
```

where `onoffseq` is an even length tuple of on and off ink in points.

**Parameters**

**ls**

[str or tuple or list thereof] Valid values for individual linestyles include {'-', '--', '-.', ':', '', (offset, on-off-seq)}. See *Line2D.set_linestyle* for a complete description.

**set_linestyles**(*ls*)

Alias for *set_linestyle*.

**set_linewidth**(*lw*)

Set the linewidth(s) for the collection. *lw* can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

**Parameters**

**lw**

[float or list of floats]

**set_linewidths**(*lw*)

Alias for *set_linewidth*.

**set_ls**(*ls*)

Alias for *set_linestyle*.

**set_lw**(*lw*)

Alias for *set_linewidth*.

**set_mouseover**(*mouseover*)

Set whether this artist is queried for custom context information when the mouse cursor moves over it.

**Parameters**

**mouseover**

[bool]

**See also:**

***get_cursor_data***
***ToolCursorPosition***
***NavigationToolbar2***

**set_norm**(*norm*)

Set the normalization instance.

**Parameters**

**norm**

[*Normalize* or str or None]

### Notes

If there are any colorbars using the mappable for this norm, setting the norm of the mappable will reset the norm, locator, and formatters on the colorbar to default.

**set_offset_transform**(*offset_transform*)

Set the artist offset transform.

**Parameters**

**offset_transform**

[*Transform*]

**set_offsets**(*offsets*)

Set the offsets for the collection.

**Parameters**

**offsets**

[(N, 2) or (2,) array-like]

**set_path_effects**(*path_effects*)

Set the path effects.

**Parameters**

**path_effects**

[list of *AbstractPathEffect*]

**set_paths**()

**set_picker**(*picker*)

Define the picking behavior of the artist.

**Parameters**

**picker**

[None or bool or float or callable] This can be one of the following:

- *None*: Picking is disabled for this artist (default).
- A boolean: If *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist.
- A float: If picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if its data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event

- A function: If picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

to determine the hit test. if the mouse event is over the artist, return *hit=True* and props is a dictionary of properties you want added to the PickEvent attributes.

**set_pickradius**(*pickradius*)

Set the pick radius used for containment tests.

**Parameters**

**pickradius**

[float] Pick radius, in points.

**set_rasterized**(*rasterized*)

Force rasterized (bitmap) drawing for vector graphics output.

Rasterized drawing is not supported by all artists. If you try to enable this on an artist that does not support it, the command has no effect and a warning will be issued.

This setting is ignored for pixel-based output.

See also *Rasterization for vector graphics*.

**Parameters**

**rasterized**

[bool]

**set_sketch_params**(*scale=None*, *length=None*, *randomness=None*)

Set the sketch parameters.

**Parameters**

**scale**

[float, optional] The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is `None`, or not provided, no sketch filter will be provided.

**length**

[float, optional] The length of the wiggle along the line, in pixels (default 128.0)

**randomness**

[float, optional] The scale factor by which the length is shrunken or expanded (default 16.0)

The PGF backend uses this argument as an RNG seed and not as described above. Using the same seed yields the same random shape.

**set_snap**(*snap*)

Set the snapping behavior.

Snapping aligns positions with the pixel grid, which results in clearer images. For example, if a black line of 1px width was defined at a position in between two pixels, the resulting image would contain the interpolated value of that line in the pixel grid, which would be a grey value on both adjacent pixel positions. In contrast, snapping will move the line to the nearest integer pixel value, so that the resulting image will really contain a 1px wide black line.

Snapping is currently only supported by the Agg and MacOSX backends.

**Parameters**

**snap**

[bool or None] Possible values:

- *True*: Snap vertices to the nearest pixel center.
- *False*: Do not modify vertex positions.
- *None*: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center.

**set_transOffset**(*offset_transform*)

Alias for *set_offset_transform*.

**set_transform**(*t*)

Set the artist transform.

**Parameters**

**t**

[*Transform*]

**set_url**(*url*)

Set the url for the artist.

**Parameters**

**url**

[str]

**set_urls**(*urls*)

**Parameters**

**urls**

[list of str or None]

### Notes

URLs are currently only implemented by the SVG backend. They are ignored by all other backends.

**set_visible**(*b*)

Set the artist's visibility.

**Parameters**

**b**

[bool]

**set_zorder**(*level*)

Set the zorder for the artist. Artists with lower zorder values are drawn first.

**Parameters**

**level**

[float]

**property stale**

Whether the artist is 'stale' and needs to be re-drawn for the output to match the internal state of the artist.

**property sticky_edges**

`x` and `y` sticky edge lists for autoscaling.

When performing autoscaling, if a data limit coincides with a value in the corresponding sticky_edges list, then no margin will be added--the view limit "sticks" to the edge. A typical use case is histograms, where one usually expects no margin on the bottom edge (0) of the histogram.

Moreover, margin expansion "bumps" against sticky edges and cannot cross them. For example, if the upper data limit is 1.0, the upper view limit computed by simple margin application is 1.2, but there is a sticky edge at 1.1, then the actual upper view limit will be 1.1.

This attribute cannot be assigned to; however, the `x` and `y` lists can be modified in place as needed.

### Examples

```
>>> artist.sticky_edges.x[:] = (xmin, xmax)
>>> artist.sticky_edges.y[:] = (ymin, ymax)
```

**to_rgba**(*x*, *alpha=None*, *bytes=False*, *norm=True*)

Return a normalized RGBA array corresponding to *x*.

In the normal case, *x* is a 1D or 2D sequence of scalars, and the corresponding `ndarray` of RGBA values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already RGB or RGBA, such as might have been read from an image file. If *x* is an `ndarray` with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an RGB or RGBA array, and no mapping will be done. The array can be `uint8`, or it can be floats with values in the 0-1 range; otherwise a ValueError will be raised. Any NaNs or masked elements will be set to 0 alpha. If the last dimension is 3, the *alpha* kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the *alpha* kwarg is ignored; it does not replace the preexisting alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if *bytes* is *False* (default), the RGBA array will be floats in the 0-1 range; if it is *True*, the returned RGBA array will be `uint8` in the 0 to 255 range.

If norm is False, no normalization of the input data is performed, and it is assumed to be in the range (0-1).

**update**(*props*)

Update this artist's properties from the dict *props*.

**Parameters**

**props**

[dict]

**update_from**(*other*)

Copy properties from other to self.

**update_scalarmappable**()

Update colors from the scalar mappable array, if any.

Assign colors to edges and faces based on the array and/or colors that were directly set, as appropriate.

**zorder = 0**

### 7.2.16 matplotlib.colorbar

Colorbars are a visualization of the mapping from scalar values to colors. In Matplotlib they are drawn into a dedicated *Axes*.

---

**Note:** Colorbars are typically created through *Figure.colorbar* or its pyplot wrapper *pyplot.colorbar*, which internally use *Colorbar* together with *make_axes_gridspec* (for *GridSpec*-positioned Axes) or *make_axes* (for non-*GridSpec*-positioned Axes).

End-users most likely won't need to directly use this module's API.

---

**class** matplotlib.colorbar.**Colorbar**(*ax*, *mappable=None*, *, *cmap=None*, *norm=None*, *alpha=None*, *values=None*, *boundaries=None*, *orientation=None*, *ticklocation='auto'*, *extend=None*, *spacing='uniform'*, *ticks=None*, *format=None*, *drawedges=False*, *extendfrac=None*, *extendrect=False*, *label=''*, *location=None*)

Bases: object

Draw a colorbar in an existing Axes.

Typically, colorbars are created using *Figure.colorbar* or *pyplot.colorbar* and associated with *ScalarMappable*s (such as an *AxesImage* generated via *imshow*).

In order to draw a colorbar not associated with other elements in the figure, e.g. when showing a colormap by itself, one can create an empty *ScalarMappable*, or directly pass *cmap* and *norm* instead of *mappable* to *Colorbar*.

Useful public methods are *set_label()* and *add_lines()*.

**Parameters**

**ax**

[*Axes*] The *Axes* instance in which the colorbar is drawn.

**mappable**

[*ScalarMappable*] The mappable whose colormap and norm will be used.

To show the under- and over- value colors, the mappable's norm should be specified as

```
norm = colors.Normalize(clip=False)
```

To show the colors versus index instead of on a 0-1 scale, use:

```
norm=colors.NoNorm()
```

**cmap**

[*Colormap*, default: *rcParams["image.cmap"]* (default: 'viridis')] The colormap to use. This parameter is ignored, unless *mappable* is None.

**norm**

[*Normalize*] The normalization to use. This parameter is ignored, unless *mappable* is None.

**alpha**

[float] The colorbar transparency between 0 (transparent) and 1 (opaque).

**orientation**

[None or {'vertical', 'horizontal'}] If None, use the value determined by *location*. If both *orientation* and *location* are None then defaults to 'vertical'.

**ticklocation**

[{'auto', 'left', 'right', 'top', 'bottom'}] The location of the colorbar ticks. The *ticklocation* must match *orientation*. For example, a horizontal colorbar can only have ticks at the top or the bottom. If 'auto', the ticks will be the same as *location*, so a colorbar to the left will have ticks to the left. If *location* is None, the ticks will be at the bottom for a horizontal colorbar and at the right for a vertical.

**drawedges**

[bool] Whether to draw lines at color boundaries.

**extend**

[{'neither', 'both', 'min', 'max'}] Make pointed end(s) for out-of-range values (unless 'neither'). These are set for a given colormap using the colormap set_under and set_over methods.

**extendfrac**

[{*None*, 'auto', length, lengths}] If set to *None*, both the minimum and maximum triangular colorbar extensions will have a length of 5% of the interior colorbar length (this is the default setting).

If set to 'auto', makes the triangular colorbar extensions the same lengths as the interior boxes (when *spacing* is set to 'uniform') or the same lengths as the respective adjacent interior boxes (when *spacing* is set to 'proportional').

If a scalar, indicates the length of both the minimum and maximum triangular colorbar extensions as a fraction of the interior colorbar length. A two-element sequence of fractions may also be given, indicating the lengths of the minimum and maximum colorbar extensions respectively as a fraction of the interior colorbar length.

**extendrect**

[bool] If *False* the minimum and maximum colorbar extensions will be triangular (the default). If *True* the extensions will be rectangular.

**spacing**

[{'uniform', 'proportional'}] For discrete colorbars (*BoundaryNorm* or contours), 'uniform' gives each color the same space; 'proportional' makes the space proportional to the data interval.

**ticks**

[None or list of ticks or Locator] If None, ticks are determined automatically from the input.

**format**

[None or str or Formatter] If None, *ScalarFormatter* is used. Format strings, e.g., `"%4.2e"` or `"{x:.2e}"`, are supported. An alternative *Formatter* may be given instead.

**drawedges**

[bool] Whether to draw lines at color boundaries.

**label**

[str] The label on the colorbar's long axis.

**boundaries, values**

[None or a sequence] If unset, the colormap will be displayed on a 0-1 scale. If sequences, *values* must have a length 1 less than *boundaries*. For each region delimited by adjacent entries in *boundaries*, the color mapped to the corresponding value in values will be used. Normally only useful for indexed colors (i.e. `norm=NoNorm()`) or other unusual circumstances.

**location**

[None or {'left', 'right', 'top', 'bottom'}] Set the *orientation* and *ticklocation* of the colorbar using a single argument. Colorbars on the left and right are vertical, colorbars at the top and bottom are horizontal. The *ticklocation* is the same as *location*, so if *location* is 'top', the ticks are on the top. *orientation* and/or *ticklocation* can be provided as well and overrides the value set by *location*, but there will be an error for incompatible combinations.

New in version 3.7.

### Attributes

**ax**

[*Axes*] The *Axes* instance in which the colorbar is drawn.

**lines**

[list] A list of *LineCollection* (empty if no lines were drawn).

**dividers**

[*LineCollection*] A LineCollection (empty if *drawedges* is `False`).

**add_lines**(**args*, ***kwargs*)

Draw lines on the colorbar.

The lines are appended to the list `lines`.

**Parameters**

**levels**

[array-like] The positions of the lines.

**colors**

[*color* or list of *color*] Either a single color applying to all lines or one color value for each line.

**linewidths**

[float or array-like] Either a single linewidth applying to all lines or one linewidth for each line.

**erase**

[bool, default: True] Whether to remove any previously added lines.

### Notes

Alternatively, this method can also be called with the signature `colorbar.add_lines(contour_set, erase=True)`, in which case *levels*, *colors*, and *linewidths* are taken from *contour_set*.

**drag_pan**(*button*, *key*, *x*, *y*)

**property formatter**

Major tick label *Formatter* for the colorbar.

**get_ticks**(*minor=False*)

Return the ticks as a list of locations.

**Parameters**

**minor**

[boolean, default: False] if True return the minor ticks.

**property locator**

Major tick *Locator* for the colorbar.

**property minorformatter**

Minor tick *Formatter* for the colorbar.

**property minorlocator**

Minor tick *Locator* for the colorbar.

**minorticks_off**()

Turn the minor ticks of the colorbar off.

**minorticks_on**()

Turn on colorbar minor ticks.

**n_rasterize = 50**

**remove**()

Remove this colorbar from the figure.

If the colorbar was created with `use_gridspec=True` the previous gridspec is restored.

**set_alpha**(*alpha*)

Set the transparency between 0 (transparent) and 1 (opaque).

If an array is provided, *alpha* will be set to None to use the transparency values associated with the colormap.

**set_label**(*label*, *, *loc=None*, ***kwargs*)

Add a label to the long axis of the colorbar.

**Parameters**

**label**

[str] The label text.

**loc**

[str, optional] The location of the label.

- For horizontal orientation one of {'left', 'center', 'right'}
- For vertical orientation one of {'bottom', 'center', 'top'}

Defaults to *rcParams["xaxis.labellocation"]* (default: `'center'`) or *rcParams["yaxis.labellocation"]* (default: `'center'`) depending on the orientation.

****kwargs**

Keyword arguments are passed to *set_xlabel* / *set_ylabel*. Supported keywords are *labelpad* and *Text* properties.

**set_ticklabels**(*ticklabels*, *, *minor=False*, ***kwargs*)

[*Discouraged*] Set tick labels.

---

**Discouraged**

The use of this method is discouraged, because of the dependency on tick positions. In most cases, you'll want to use `set_ticks(positions, labels=labels)` instead.

If you are using this method, you should always fix the tick positions before, e.g. by using *Colorbar.set_ticks* or by explicitly setting a *FixedLocator* on the long axis of the colorbar. Otherwise, ticks are free to move and the labels may end up in unexpected positions.

---

**Parameters**

**ticklabels**

[sequence of str or of *Text*] Texts for labeling each tick location in the sequence set by *Colorbar.set_ticks*; the number of labels must match the number of locations.

**update_ticks**

[bool, default: True] This keyword argument is ignored and will be removed. Deprecated

**minor**

[bool] If True, set minor ticks instead of major ticks.

****kwargs**

*Text* properties for the labels.

**set_ticks**(*ticks*, *, *labels=None*, *minor=False*, ***kwargs*)

Set tick locations.

**Parameters**

**ticks**

[1D array-like] List of tick locations.

**labels**

[list of str, optional] List of tick labels. If not set, the labels show the data value.

**minor**

[bool, default: False] If `False`, set the major ticks; if `True`, the minor ticks.

****kwargs**

*Text* properties for the labels. These take effect only if you pass *labels*. In other cases, please use *tick_params*.

**update_normal**(*mappable*)

Update solid patches, lines, etc.

This is meant to be called when the norm of the image or contour plot to which this colorbar belongs changes.

If the norm on the mappable is different than before, this resets the locator and formatter for the axis, so if these have been customized, they will need to be customized again. However, if

the norm only changes values of *vmin*, *vmax* or *cmap* then the old formatter and locator will be preserved.

**update_ticks**()

Set up the ticks and ticklabels. This should not be needed by users.

matplotlib.colorbar.**ColorbarBase**

alias of *Colorbar*

matplotlib.colorbar.**make_axes**(*parents*, *location=None*, *orientation=None*, *fraction=0.15*, *shrink=1.0*, *aspect=20*, ***kwargs*)

Create an *Axes* suitable for a colorbar.

The Axes is placed in the figure of the *parents* Axes, by resizing and repositioning *parents*.

**Parameters**

**parents**

[*Axes* or iterable or numpy.ndarray of *Axes*] The Axes to use as parents for placing the colorbar.

**location**

[None or {'left', 'right', 'top', 'bottom'}] The location, relative to the parent Axes, where the colorbar Axes is created. It also determines the *orientation* of the colorbar (colorbars on the left and right are vertical, colorbars at the top and bottom are horizontal). If None, the location will come from the *orientation* if it is set (vertical colorbars on the right, horizontal ones at the bottom), or default to 'right' if *orientation* is unset.

**orientation**

[None or {'vertical', 'horizontal'}] The orientation of the colorbar. It is preferable to set the *location* of the colorbar, as that also determines the *orientation*; passing incompatible values for *location* and *orientation* raises an exception.

**fraction**

[float, default: 0.15] Fraction of original Axes to use for colorbar.

**shrink**

[float, default: 1.0] Fraction by which to multiply the size of the colorbar.

**aspect**

[float, default: 20] Ratio of long to short dimensions.

**pad**

[float, default: 0.05 if vertical, 0.15 if horizontal] Fraction of original Axes between colorbar and new image Axes.

**anchor**

[(float, float), optional] The anchor point of the colorbar Axes. Defaults to (0.0, 0.5) if vertical; (0.5, 1.0) if horizontal.

**panchor**

[(float, float), or *False*, optional] The anchor point of the colorbar parent Axes. If *False*, the parent axes' anchor will be unchanged. Defaults to (1.0, 0.5) if vertical; (0.5, 0.0) if horizontal.

**Returns**

**cax**

[*Axes*] The child Axes.

**kwargs**

[dict] The reduced keyword dictionary to be passed when creating the colorbar instance.

matplotlib.colorbar.**make_axes_gridspec**(*parent*, *, *location=None*, *orientation=None*, *fraction=0.15*, *shrink=1.0*, *aspect=20*, ***kwargs*)

Create an *Axes* suitable for a colorbar.

The Axes is placed in the figure of the *parent* Axes, by resizing and repositioning *parent*.

This function is similar to *make_axes* and mostly compatible with it. Primary differences are

- *make_axes_gridspec* requires the *parent* to have a subplotspec.
- *make_axes* positions the Axes in figure coordinates; *make_axes_gridspec* positions it using a subplotspec.
- *make_axes* updates the position of the parent. *make_axes_gridspec* replaces the parent gridspec with a new one.

**Parameters**

**parent**

[*Axes*] The Axes to use as parent for placing the colorbar.

**location**

[None or {'left', 'right', 'top', 'bottom'}] The location, relative to the parent Axes, where the colorbar Axes is created. It also determines the *orientation* of the colorbar (colorbars on the left and right are vertical, colorbars at the top and bottom are horizontal). If None, the location will come from the *orientation* if it is set (vertical colorbars on the right, horizontal ones at the bottom), or default to 'right' if *orientation* is unset.

**orientation**

[None or {'vertical', 'horizontal'}] The orientation of the colorbar. It is preferable to set the *location* of the colorbar, as that also determines the *orientation*; passing incompatible values for *location* and *orientation* raises an exception.

**fraction**

[float, default: 0.15] Fraction of original Axes to use for colorbar.

**shrink**

[float, default: 1.0] Fraction by which to multiply the size of the colorbar.

**aspect**

[float, default: 20] Ratio of long to short dimensions.

**pad**

[float, default: 0.05 if vertical, 0.15 if horizontal] Fraction of original Axes between colorbar and new image Axes.

**anchor**

[(float, float), optional] The anchor point of the colorbar Axes. Defaults to (0.0, 0.5) if vertical; (0.5, 1.0) if horizontal.

**panchor**

[(float, float), or *False*, optional] The anchor point of the colorbar parent Axes. If *False*, the parent axes' anchor will be unchanged. Defaults to (1.0, 0.5) if vertical; (0.5, 0.0) if horizontal.

**Returns**

**cax**

[*Axes*] The child Axes.

**kwargs**

[dict] The reduced keyword dictionary to be passed when creating the colorbar instance.

### 7.2.17 matplotlib.colors

**Note:** The Color *tutorials* and *examples* demonstrate how to set colors and colormaps. You may want to read those instead.

A module for converting numbers or color arguments to *RGB* or *RGBA*.

*RGB* and *RGBA* are sequences of, respectively, 3 or 4 floats in the range 0-1.

This module includes functions and classes for color specification conversions, and for mapping numbers to colors in a 1-D array of colors called a colormap.

Mapping data onto colors using a colormap typically involves two steps: a data array is first mapped onto the range 0-1 using a subclass of *Normalize*, then this number is mapped to a color using a subclass of *Colormap*. Two subclasses of *Colormap* provided here: *LinearSegmentedColormap*, which uses piecewise-linear interpolation to define colormaps, and *ListedColormap*, which makes a colormap from a list of colors.

**See also:**

*Creating Colormaps in Matplotlib* for examples of how to make colormaps and

*Choosing Colormaps in Matplotlib* for a list of built-in colormaps.

*Colormap normalization* for more details about data normalization

More colormaps are available at palettable.

The module also provides functions for checking whether an object can be interpreted as a color (*is_color_like*), for converting such an object to an RGBA tuple (*to_rgba*) or to an HTML-like hex string in the "#rrggbb" format (*to_hex*), and a sequence of colors to an (n, 4) RGBA array (*to_rgba_array*). Caching is used for efficiency.

Colors that Matplotlib recognizes are listed at *Specifying colors*.

### Color norms

| | |
|---|---|
| *Normalize*([vmin, vmax, clip]) | A class which, when called, maps values within the interval `[vmin, vmax]` linearly to the interval `[0.0, 1.0]`. |
| *NoNorm*([vmin, vmax, clip]) | Dummy replacement for *Normalize*, for the case where we want to use indices directly in a *ScalarMappable*. |
| *AsinhNorm*([linear_width, vmin, vmax, clip]) | The inverse hyperbolic sine scale is approximately linear near the origin, but becomes logarithmic for larger positive or negative values. |
| *BoundaryNorm*(boundaries, ncolors[, clip, extend]) | Generate a colormap index based on discrete intervals. |
| *CenteredNorm*([vcenter, halfrange, clip]) | Normalize symmetrical data around a center (0 by default). |
| *FuncNorm*(functions[, vmin, vmax, clip]) | Arbitrary normalization using functions for the forward and inverse. |
| *LogNorm*([vmin, vmax, clip]) | Normalize a given value to the 0-1 range on a log scale. |
| *PowerNorm*(gamma[, vmin, vmax, clip]) | Linearly map a given value to the 0-1 range and then apply a power-law normalization over that range. |
| *SymLogNorm*(linthresh[, linscale, vmin, ...]) | The symmetrical logarithmic scale is logarithmic in both the positive and negative directions from the origin. |
| *TwoSlopeNorm*(vcenter[, vmin, vmax]) | Normalize data with a set center. |

### matplotlib.colors.Normalize

**class** matplotlib.colors.**Normalize**(*vmin=None*, *vmax=None*, *clip=False*)

Bases: object

A class which, when called, maps values within the interval `[vmin, vmax]` linearly to the interval `[0.0, 1.0]`. The mapping of values outside `[vmin, vmax]` depends on *clip*.

**See also:**

*Colormap normalization*

#### Examples

```
x = [-2, -1, 0, 1, 2]

norm = mpl.colors.Normalize(vmin=-1, vmax=1, clip=False)
norm(x)  # [-0.5, 0., 0.5, 1., 1.5]
norm = mpl.colors.Normalize(vmin=-1, vmax=1, clip=True)
norm(x)  # [0., 0., 0.5, 1., 1.]
```

**Parameters**

**vmin, vmax**

[float or None] Values within the range `[vmin, vmax]` from the input data will be linearly mapped to `[0, 1]`. If either *vmin* or *vmax* is not provided, they default to the minimum and maximum values of the input, respectively.

**clip**

[bool, default: False] Determines the behavior for mapping values outside the range `[vmin, vmax]`.

If clipping is off, values outside the range `[vmin, vmax]` are also transformed, resulting in values outside `[0, 1]`. This behavior is usually desirable, as colormaps can mark these *under* and *over* values with specific colors.

If clipping is on, values below *vmin* are mapped to 0 and values above *vmax* are mapped to 1. Such values become indistinguishable from regular boundary values, which may cause misinterpretation of the data.

### Notes

If `vmin == vmax`, input data will be mapped to 0.

**__call__**(*value*, *clip=None*)

Normalize the data and return the normalized data.

**Parameters**

**value**

Data to normalize.

**clip**

[bool, optional] See the description of the parameter *clip* in *Normalize*.

If `None`, defaults to `self.clip` (which defaults to `False`).

#### Notes

If not already initialized, `self.vmin` and `self.vmax` are initialized using `self.autoscale_None(value)`.

**autoscale**(*A*)

Set *vmin*, *vmax* to min, max of *A*.

**autoscale_None**(*A*)

If *vmin* or *vmax* are not set, use the min/max of *A* to set them.

**property clip**

**inverse**(*value*)

Maps the normalized value (i.e., index in the colormap) back to image data value.

**Parameters**

**value**

Normalized value.

**static process_value**(*value*)

Homogenize the input *value* for easy and efficient normalization.

*value* can be a scalar or sequence.

**Parameters**

**value**

Data to normalize.

**Returns**

**result**

[masked array] Masked array with the same shape as *value*.

**is_scalar**

[bool] Whether *value* is a scalar.

### Notes

Float dtypes are preserved; integer types with two bytes or smaller are converted to np.float32, and larger types are converted to np.float64. Preserving float32 when possible, and using in-place operations, greatly improves speed for large arrays.

**scaled**()

Return whether *vmin* and *vmax* are both set.

**property vmax**

**property vmin**

## Examples using `matplotlib.colors.Normalize`

- *Multicolored lines*
- *Mapping marker properties to multivariate data*
- *Colormap normalizations*
- *Colormap normalizations SymLogNorm*
- *Contour Image*
- *Creating annotated heatmaps*
- *Image Masked*
- *Blend transparency with color in 2D images*
- *Multiple images*
- *pcolor images*
- *pcolormesh*
- *Histograms*
- *Shaded & power normalized rendering*
- *Exploring normalizations*
- *Hillshading*
- *Left ventricle bullseye*
- *Quick start guide*

- *Constrained layout guide*
- *Customized Colorbars Tutorial*
- *Colormap normalization*

### matplotlib.colors.NoNorm

**class** matplotlib.colors.**NoNorm**(*vmin=None*, *vmax=None*, *clip=False*)

Bases: *Normalize*

Dummy replacement for *Normalize*, for the case where we want to use indices directly in a *ScalarMappable*.

**Parameters**

**vmin, vmax**

[float or None] Values within the range `[vmin, vmax]` from the input data will be linearly mapped to `[0, 1]`. If either *vmin* or *vmax* is not provided, they default to the minimum and maximum values of the input, respectively.

**clip**

[bool, default: False] Determines the behavior for mapping values outside the range `[vmin, vmax]`.

If clipping is off, values outside the range `[vmin, vmax]` are also transformed, resulting in values outside `[0, 1]`. This behavior is usually desirable, as colormaps can mark these *under* and *over* values with specific colors.

If clipping is on, values below *vmin* are mapped to 0 and values above *vmax* are mapped to 1. Such values become indistinguishable from regular boundary values, which may cause misinterpretation of the data.

#### Notes

If `vmin == vmax`, input data will be mapped to 0.

**__call__**(*value*, *clip=None*)

Normalize the data and return the normalized data.

**Parameters**

**value**

Data to normalize.

**clip**

[bool, optional] See the description of the parameter *clip* in *Normalize*.

If `None`, defaults to `self.clip` (which defaults to `False`).

### Notes

If not already initialized, `self.vmin` and `self.vmax` are initialized using `self.autoscale_None(value)`.

**inverse**(*value*)

Maps the normalized value (i.e., index in the colormap) back to image data value.

**Parameters**

**value**

Normalized value.

## matplotlib.colors.AsinhNorm

**class** matplotlib.colors.**AsinhNorm**(*linear_width=1*, *vmin=None*, *vmax=None*, *clip=False*)

Bases: *AsinhNorm*

The inverse hyperbolic sine scale is approximately linear near the origin, but becomes logarithmic for larger positive or negative values. Unlike the *SymLogNorm*, the transition between these linear and logarithmic regions is smooth, which may reduce the risk of visual artifacts.

**Note:** This API is provisional and may be revised in the future based on early user feedback.

**Parameters**

**linear_width**

[float, default: 1] The effective width of the linear region, beyond which the transformation becomes asymptotically logarithmic

**Parameters**

**vmin, vmax**

[float or None] Values within the range `[vmin, vmax]` from the input data will be linearly mapped to `[0, 1]`. If either *vmin* or *vmax* is not provided, they default to the minimum and maximum values of the input, respectively.

**clip**

[bool, default: False] Determines the behavior for mapping values outside the range `[vmin, vmax]`.

If clipping is off, values outside the range `[vmin, vmax]` are also transformed, resulting in values outside `[0, 1]`. This behavior is usually desirable, as colormaps can mark these *under* and *over* values with specific colors.

If clipping is on, values below *vmin* are mapped to 0 and values above *vmax* are mapped to 1. Such values become indistinguishable from regular boundary values, which may cause misinterpretation of the data.

### Notes

If `vmin == vmax`, input data will be mapped to 0.

**__call__**(*value*, *clip=None*)

Normalize the data and return the normalized data.

**Parameters**

**value**

Data to normalize.

**clip**

[bool, optional] See the description of the parameter *clip* in *Normalize*.

If `None`, defaults to `self.clip` (which defaults to `False`).

#### Notes

If not already initialized, `self.vmin` and `self.vmax` are initialized using `self.autoscale_None(value)`.

**autoscale_None**(*A*)

If *vmin* or *vmax* are not set, use the min/max of *A* to set them.

**inverse**(*value*)

Maps the normalized value (i.e., index in the colormap) back to image data value.

**Parameters**

**value**

Normalized value.

## Examples using `matplotlib.colors.AsinhNorm`

- *Colormap normalizations SymLogNorm*

### matplotlib.colors.BoundaryNorm

**class** matplotlib.colors.**BoundaryNorm**(*boundaries*, *ncolors*, *clip=False*, ***, *extend='neither'*)

Bases: *Normalize*

Generate a colormap index based on discrete intervals.

Unlike *Normalize* or *LogNorm*, *BoundaryNorm* maps values to integers instead of to the interval 0-1.

**Parameters**

**boundaries**

[array-like] Monotonically increasing sequence of at least 2 bin edges: data falling in the n-th bin will be mapped to the n-th color.

**ncolors**

[int] Number of colors in the colormap to be used.

**clip**

[bool, optional] If clip is `True`, out of range values are mapped to 0 if they are below `boundaries[0]` or mapped to `ncolors - 1` if they are above `boundaries[-1]`.

If clip is `False`, out of range values are mapped to -1 if they are below `boundaries[0]` or mapped to *ncolors* if they are above `boundaries[-1]`. These are then converted to valid indices by *Colormap.__call__*.

**extend**

[{'neither', 'both', 'min', 'max'}, default: 'neither'] Extend the number of bins to include one or both of the regions beyond the boundaries. For example, if `extend` is 'min', then the color to which the region between the first pair of boundaries is mapped will be distinct from the first color in the colormap, and by default a *Colorbar* will be drawn with the triangle extension on the left or lower end.

#### Notes

If there are fewer bins (including extensions) than colors, then the color index is chosen by linearly interpolating the `[0, nbins - 1]` range onto the `[0, ncolors - 1]` range, effectively skipping some colors in the middle of the colormap.

**__call__**(*value*, *clip=None*)

This method behaves similarly to *Normalize.__call__*, except that it returns integers or arrays of int16.

**inverse**(*value*)

**Raises**

**ValueError**

BoundaryNorm is not invertible, so calling this method will always raise an error

### Examples using `matplotlib.colors.BoundaryNorm`

- *Multicolored lines*
- *Colormap normalizations*
- *Creating annotated heatmaps*
- *Image Masked*
- *pcolormesh*
- *Left ventricle bullseye*
- *Customized Colorbars Tutorial*
- *Colormap normalization*

## matplotlib.colors.CenteredNorm

**class** matplotlib.colors.**CenteredNorm**(*vcenter=0*, *halfrange=None*, *clip=False*)

Bases: *Normalize*

Normalize symmetrical data around a center (0 by default).

Unlike *TwoSlopeNorm*, *CenteredNorm* applies an equal rate of change around the center.

Useful when mapping symmetrical data around a conceptual center e.g., data that range from -2 to 4, with 0 as the midpoint, and with equal rates of change around that midpoint.

**Parameters**

**vcenter**

[float, default: 0] The data value that defines `0.5` in the normalization.

**halfrange**

[float, optional] The range of data values that defines a range of `0.5` in the normalization, so that *vcenter* - *halfrange* is `0.0` and *vcenter* + *halfrange* is `1.0` in the normalization. Defaults to the largest absolute difference to *vcenter* for the values in the dataset.

**clip**

[bool, default: False] Determines the behavior for mapping values outside the range `[vmin, vmax]`.

If clipping is off, values outside the range `[vmin, vmax]` are also transformed, resulting in values outside `[0, 1]`. This behavior is usually desirable, as colormaps can mark these *under* and *over* values with specific colors.

If clipping is on, values below *vmin* are mapped to 0 and values above *vmax* are mapped to 1. Such values become indistinguishable from regular boundary values, which may cause misinterpretation of the data.

### Examples

This maps data values -2 to 0.25, 0 to 0.5, and 4 to 1.0 (assuming equal rates of change above and below 0.0):

```
>>> import matplotlib.colors as mcolors
>>> norm = mcolors.CenteredNorm(halfrange=4.0)
>>> data = [-2., 0., 4.]
>>> norm(data)
array([0.25, 0.5 , 1.  ])
```

**autoscale**(*A*)

Set *halfrange* to `max(abs(A-vcenter))`, then set *vmin* and *vmax*.

**autoscale_None**(*A*)

Set *vmin* and *vmax*.

**property halfrange**

**property vcenter**

**property vmax**

**property vmin**

## Examples using `matplotlib.colors.CenteredNorm`

- *Colormap normalization*

## matplotlib.colors.FuncNorm

**class** matplotlib.colors.**FuncNorm**(*functions*, *vmin=None*, *vmax=None*, *clip=False*)

Bases: *FuncNorm*

Arbitrary normalization using functions for the forward and inverse.

**Parameters**

**functions**

[(callable, callable)] two-tuple of the forward and inverse functions for the normalization. The forward function must be monotonic.

Both functions must have the signature

```
def forward(values: array-like) -> array-like
```

**vmin, vmax**

[float or None] If *vmin* and/or *vmax* is not given, they are initialized from the minimum and maximum value, respectively, of the first input processed; i.e., `__call__(A)` calls `autoscale_None(A)`.

**clip**

[bool, default: False] Determines the behavior for mapping values outside the range `[vmin, vmax]`.

If clipping is off, values outside the range `[vmin, vmax]` are also transformed by the function, resulting in values outside `[0, 1]`. This behavior is usually desirable, as colormaps can mark these *under* and *over* values with specific colors.

If clipping is on, values below *vmin* are mapped to 0 and values above *vmax* are mapped to 1. Such values become indistinguishable from regular boundary values, which may cause misinterpretation of the data.

**Parameters**

**vmin, vmax**

[float or None] Values within the range `[vmin, vmax]` from the input data will be linearly mapped to `[0, 1]`. If either *vmin* or *vmax* is not provided, they default to the minimum and maximum values of the input, respectively.

**clip**

[bool, default: False] Determines the behavior for mapping values outside the range `[vmin, vmax]`.

If clipping is off, values outside the range `[vmin, vmax]` are also transformed, resulting in values outside `[0, 1]`. This behavior is usually desirable, as colormaps can mark these *under* and *over* values with specific colors.

If clipping is on, values below *vmin* are mapped to 0 and values above *vmax* are mapped to 1. Such values become indistinguishable from regular boundary values, which may cause misinterpretation of the data.

## Notes

If `vmin == vmax`, input data will be mapped to 0.

**__call__**(*value*, *clip=None*)

Normalize the data and return the normalized data.

**Parameters**

**value**

Data to normalize.

**clip**

[bool, optional] See the description of the parameter *clip* in *Normalize*.

If `None`, defaults to `self.clip` (which defaults to `False`).

### Notes

If not already initialized, `self.vmin` and `self.vmax` are initialized using `self.autoscale_None(value)`.

**autoscale_None**(*A*)

If *vmin* or *vmax* are not set, use the min/max of *A* to set them.

**inverse**(*value*)

Maps the normalized value (i.e., index in the colormap) back to image data value.

**Parameters**

**value**

Normalized value.

## Examples using `matplotlib.colors.FuncNorm`

- *Colormap normalization*

## matplotlib.colors.LogNorm

**class** matplotlib.colors.**LogNorm**(*vmin=None*, *vmax=None*, *clip=False*)

Bases: *Normalize*

Normalize a given value to the 0-1 range on a log scale.

**Parameters**

**vmin, vmax**

[float or None] Values within the range `[vmin, vmax]` from the input data will be linearly mapped to `[0, 1]`. If either *vmin* or *vmax* is not provided, they default to the minimum and maximum values of the input, respectively.

**clip**

[bool, default: False] Determines the behavior for mapping values outside the range `[vmin, vmax]`.

If clipping is off, values outside the range `[vmin, vmax]` are also transformed, resulting in values outside `[0, 1]`. This behavior is usually desirable, as colormaps can mark these *under* and *over* values with specific colors.

If clipping is on, values below *vmin* are mapped to 0 and values above *vmax* are mapped to 1. Such values become indistinguishable from regular boundary values, which may cause misinterpretation of the data.

### Notes

If `vmin == vmax`, input data will be mapped to 0.

**__call__**(*value*, *clip=None*)

Normalize the data and return the normalized data.

**Parameters**

**value**

Data to normalize.

**clip**

[bool, optional] See the description of the parameter *clip* in *Normalize*.

If `None`, defaults to `self.clip` (which defaults to `False`).

#### Notes

If not already initialized, `self.vmin` and `self.vmax` are initialized using `self.autoscale_None(value)`.

**autoscale_None**(*A*)

If *vmin* or *vmax* are not set, use the min/max of *A* to set them.

**inverse**(*value*)

Maps the normalized value (i.e., index in the colormap) back to image data value.

**Parameters**

**value**

Normalized value.

### Examples using `matplotlib.colors.LogNorm`

- *Colormap normalizations*
- *pcolor images*
- *Histograms*
- *Quick start guide*
- *Colormap normalization*

## matplotlib.colors.PowerNorm

**class** matplotlib.colors.**PowerNorm**(*gamma*, *vmin=None*, *vmax=None*, *clip=False*)

Bases: *Normalize*

Linearly map a given value to the 0-1 range and then apply a power-law normalization over that range.

**Parameters**

**gamma**

[float] Power law exponent.

**vmin, vmax**

[float or None] If *vmin* and/or *vmax* is not given, they are initialized from the minimum and maximum value, respectively, of the first input processed; i.e., `__call__(A)` calls `autoscale_None(A)`.

**clip**

[bool, default: False] Determines the behavior for mapping values outside the range `[vmin, vmax]`.

If clipping is off, values above *vmax* are transformed by the power function, resulting in values above 1, and values below *vmin* are linearly transformed resulting in values below 0. This behavior is usually desirable, as colormaps can mark these *under* and *over* values with specific colors.

If clipping is on, values below *vmin* are mapped to 0 and values above *vmax* are mapped to 1. Such values become indistinguishable from regular boundary values, which may cause misinterpretation of the data.

### Notes

The normalization formula is

$$\left(\frac{x - v_{min}}{v_{max} - v_{min}}\right)^{\gamma}$$

For input values below *vmin*, gamma is set to one.

**Parameters**

**vmin, vmax**

[float or None] Values within the range `[vmin, vmax]` from the input data will be linearly mapped to `[0, 1]`. If either *vmin* or *vmax* is not provided, they default to the minimum and maximum values of the input, respectively.

**clip**

[bool, default: False] Determines the behavior for mapping values outside the range `[vmin, vmax]`.

If clipping is off, values outside the range `[vmin, vmax]` are also transformed, resulting in values outside `[0, 1]`. This behavior is usually desirable, as colormaps can mark these *under* and *over* values with specific colors.

If clipping is on, values below *vmin* are mapped to 0 and values above *vmax* are mapped to 1. Such values become indistinguishable from regular boundary values, which may cause misinterpretation of the data.

### Notes

If `vmin == vmax`, input data will be mapped to 0.

**`__call__`**(*value*, *clip=None*)

Normalize the data and return the normalized data.

**Parameters**

**value**

Data to normalize.

**clip**

[bool, optional] See the description of the parameter *clip* in *Normalize*.

If `None`, defaults to `self.clip` (which defaults to `False`).

#### Notes

If not already initialized, `self.vmin` and `self.vmax` are initialized using `self.autoscale_None(value)`.

**inverse**(*value*)

Maps the normalized value (i.e., index in the colormap) back to image data value.

**Parameters**

**value**

Normalized value.

### Examples using `matplotlib.colors.PowerNorm`

- *Colormap normalizations*
- *Shaded & power normalized rendering*
- *Exploring normalizations*
- *Colormap normalization*

## matplotlib.colors.SymLogNorm

**class** matplotlib.colors.**SymLogNorm**(*linthresh*, *linscale=1.0*, *vmin=None*, *vmax=None*, *clip=False*, ***, *base=10*)

Bases: *SymLogNorm*

The symmetrical logarithmic scale is logarithmic in both the positive and negative directions from the origin.

Since the values close to zero tend toward infinity, there is a need to have a range around zero that is linear. The parameter *linthresh* allows the user to specify the size of this range (*-linthresh*, *linthresh*).

**Parameters**

**linthresh**

[float] The range within which the plot is linear (to avoid having the plot go to infinity around zero).

**linscale**

[float, default: 1] This allows the linear range (*-linthresh* to *linthresh*) to be stretched relative to the logarithmic range. Its value is the number of decades to use for each half of the linear range. For example, when *linscale* == 1.0 (the default), the space used for the positive and negative halves of the linear range will be equal to one decade in the logarithmic range.

**base**

[float, default: 10]

**Parameters**

**vmin, vmax**

[float or None] Values within the range `[vmin, vmax]` from the input data will be linearly mapped to `[0, 1]`. If either *vmin* or *vmax* is not provided, they default to the minimum and maximum values of the input, respectively.

**clip**

[bool, default: False] Determines the behavior for mapping values outside the range `[vmin, vmax]`.

If clipping is off, values outside the range `[vmin, vmax]` are also transformed, resulting in values outside `[0, 1]`. This behavior is usually desirable, as colormaps can mark these *under* and *over* values with specific colors.

If clipping is on, values below *vmin* are mapped to 0 and values above *vmax* are mapped to 1. Such values become indistinguishable from regular boundary values, which may cause misinterpretation of the data.

## Notes

If `vmin == vmax`, input data will be mapped to 0.

**__call__**(*value*, *clip=None*)

Normalize the data and return the normalized data.

**Parameters**

**value**

Data to normalize.

**clip**

[bool, optional] See the description of the parameter *clip* in *Normalize*.

If `None`, defaults to `self.clip` (which defaults to `False`).

#### Notes

If not already initialized, `self.vmin` and `self.vmax` are initialized using `self.autoscale_None(value)`.

**autoscale_None**(*A*)

If *vmin* or *vmax* are not set, use the min/max of *A* to set them.

**inverse**(*value*)

Maps the normalized value (i.e., index in the colormap) back to image data value.

**Parameters**

**value**

Normalized value.

### Examples using `matplotlib.colors.SymLogNorm`

- *Colormap normalizations*
- *Colormap normalizations SymLogNorm*
- *Colormap normalization*

## matplotlib.colors.TwoSlopeNorm

**class** matplotlib.colors.**TwoSlopeNorm**(*vcenter*, *vmin=None*, *vmax=None*)

Bases: *Normalize*

Normalize data with a set center.

Useful when mapping data with an unequal rates of change around a conceptual center, e.g., data that range from -2 to 4, with 0 as the midpoint.

**Parameters**

**vcenter**

[float] The data value that defines `0.5` in the normalization.

**vmin**

[float, optional] The data value that defines `0.0` in the normalization. Defaults to the min value of the dataset.

**vmax**

[float, optional] The data value that defines `1.0` in the normalization. Defaults to the max value of the dataset.

### Examples

This maps data value -4000 to 0., 0 to 0.5, and +10000 to 1.0; data between is linearly interpolated:

```
>>> import matplotlib.colors as mcolors
>>> offset = mcolors.TwoSlopeNorm(vmin=-4000.,
...                               vcenter=0., vmax=10000)
>>> data = [-4000., -2000., 0., 2500., 5000., 7500., 10000.]
>>> offset(data)
array([0., 0.25, 0.5, 0.625, 0.75, 0.875, 1.0])
```

**__call__**(*value*, *clip=None*)

Map value to the interval [0, 1]. The *clip* argument is unused.

**autoscale_None**(*A*)

Get vmin and vmax.

If vcenter isn't in the range [vmin, vmax], either vmin or vmax is expanded so that vcenter lies in the middle of the modified range [vmin, vmax].

**inverse**(*value*)

Maps the normalized value (i.e., index in the colormap) back to image data value.

**Parameters**

**value**

Normalized value.

**property vcenter**

### Examples using `matplotlib.colors.TwoSlopeNorm`

- *Colormap normalization*

### Colormaps

| | |
|---|---|
| *Colormap*(name[, N]) | Baseclass for all scalar to RGBA mappings. |
| *LinearSegmentedColormap*(name, segmentdata[, ...]) | Colormap objects based on lookup tables using linear segments. |
| *ListedColormap*(colors[, name, N]) | Colormap object generated from a list of colors. |

### matplotlib.colors.Colormap

**class** matplotlib.colors.**Colormap**(*name*, *N=256*)

Bases: object

Baseclass for all scalar to RGBA mappings.

Typically, Colormap instances are used to convert data values (floats) from the interval `[0, 1]` to the RGBA color that the respective Colormap represents. For scaling of data into the `[0, 1]` interval see *matplotlib.colors.Normalize*. Subclasses of *matplotlib.cm.ScalarMappable* make heavy use of this `data -> normalize -> map-to-color` processing chain.

**Parameters**

**name**

[str] The name of the colormap.

**N**

[int] The number of RGB quantization levels.

**__call__**(*X*, *alpha=None*, *bytes=False*)

**Parameters**

**X**

[float or int, `ndarray` or scalar] The data value(s) to convert to RGBA. For floats, *X* should be in the interval `[0.0, 1.0]` to return the RGBA values `X*100` percent along the Colormap line. For integers, *X* should be in the interval `[0, Colormap.N)` to return RGBA values *indexed* from the Colormap with index `X`.

**alpha**

[float or array-like or None] Alpha must be a scalar between 0 and 1, a sequence of such floats with shape matching X, or None.

**bytes**

[bool] If False (default), the returned RGBA values will be floats in the interval `[0, 1]` otherwise they will be `numpy.uint8`s in the interval `[0, 255]`.

**Returns**

**Tuple of RGBA values if X is scalar, otherwise an array of RGBA values with a shape of `X.shape + (4, )`.**

**colorbar_extend**

When this colormap exists on a scalar mappable and colorbar_extend is not False, colorbar creation will pick up `colorbar_extend` as the default value for the `extend` keyword in the *matplotlib.colorbar.Colorbar* constructor.

**copy**()

Return a copy of the colormap.

**get_bad**()

Get the color for masked values.

**get_over**()

Get the color for high out-of-range values.

**get_under**()

Get the color for low out-of-range values.

**is_gray**()

Return whether the colormap is grayscale.

**resampled**(*lutsize*)

Return a new colormap with *lutsize* entries.

**reversed**(*name=None*)

Return a reversed instance of the Colormap.

---

**Note:** This function is not implemented for the base class.

---

**Parameters**

**name**

[str, optional] The name for the reversed colormap. If None, the name is set to `self.name + "_r"`.

**See also:**

***LinearSegmentedColormap.reversed***
***ListedColormap.reversed***

**set_bad**(*color='k'*, *alpha=None*)

Set the color for masked values.

**set_extremes**(*, *bad=None*, *under=None*, *over=None*)

Set the colors for masked (*bad*) values and, when `norm.clip = False`, low (*under*) and high (*over*) out-of-range values.

**set_over**(*color='k'*, *alpha=None*)

Set the color for high out-of-range values.

**set_under**(*color='k'*, *alpha=None*)

Set the color for low out-of-range values.

**with_extremes**(*, *bad=None*, *under=None*, *over=None*)

Return a copy of the colormap, for which the colors for masked (*bad*) values and, when `norm.clip = False`, low (*under*) and high (*over*) out-of-range values, have been set accordingly.

### Examples using `matplotlib.colors.Colormap`

- *Multicolored lines*
- *Contourf demo*
- *Creating a colormap from a list of colors*
- *Selecting individual colors from a colormap*
- *Lasso Demo*
- *Left ventricle bullseye*
- *Customized Colorbars Tutorial*
- *Creating Colormaps in Matplotlib*

## matplotlib.colors.LinearSegmentedColormap

**class** matplotlib.colors.**LinearSegmentedColormap**(*name*, *segmentdata*, *N=256*, *gamma=1.0*)

Bases: *Colormap*

Colormap objects based on lookup tables using linear segments.

The lookup table is generated using linear interpolation for each primary color, with the 0-1 domain divided into any number of segments.

Create colormap from linear mapping segments

segmentdata argument is a dictionary with a red, green and blue entries. Each entry should be a list of *x*, *y0*, *y1* tuples, forming rows in a table. Entries for alpha are optional.

Example: suppose you want red to increase from 0 to 1 over the bottom half, green to do the same over the middle half, and blue over the top half. Then you would use:

```
cdict = {'red':   [(0.0,  0.0, 0.0),
                   (0.5,  1.0, 1.0),
                   (1.0,  1.0, 1.0)],

         'green': [(0.0,  0.0, 0.0),
                   (0.25, 0.0, 0.0),
                   (0.75, 1.0, 1.0),
                   (1.0,  1.0, 1.0)],

         'blue':  [(0.0,  0.0, 0.0),
```

(continues on next page)

(continued from previous page)

```
                   (0.5,  0.0, 0.0),
                   (1.0,  1.0, 1.0)]}
```

Each row in the table for a given color is a sequence of *x*, *y0*, *y1* tuples. In each sequence, *x* must increase monotonically from 0 to 1. For any input value *z* falling between *x[i]* and *x[i+1]*, the output value of a given color will be linearly interpolated between *y1[i]* and *y0[i+1]*:

```
row i:   x  y0  y1
                /
               /
row i+1: x  y0  y1
```

Hence y0 in the first row and y1 in the last row are never used.

**See also:**

***LinearSegmentedColormap.from_list***

Static method; factory function for generating a smoothly-varying LinearSegmentedColormap.

**static from_list**(*name*, *colors*, *N=256*, *gamma=1.0*)

Create a *LinearSegmentedColormap* from a list of colors.

**Parameters**

**name**

[str] The name of the colormap.

**colors**

[list of *color* or list of (value, color)] If only colors are given, they are equidistantly mapped from the range [0, 1]; i.e. 0 maps to `colors[0]` and 1 maps to `colors[-1]`. If (value, color) pairs are given, the mapping is from *value* to *color*. This can be used to divide the range unevenly.

**N**

[int] The number of RGB quantization levels.

**gamma**

[float]

**resampled**(*lutsize*)

Return a new colormap with *lutsize* entries.

**reversed**(*name=None*)

Return a reversed instance of the Colormap.

**Parameters**

**name**

[str, optional] The name for the reversed colormap. If None, the name is set to `self.name + "_r"`.

**Returns**

**LinearSegmentedColormap**

The reversed colormap.

**set_gamma**(*gamma*)

Set a new gamma value and regenerate colormap.

## Examples using `matplotlib.colors.LinearSegmentedColormap`

- *Bar chart with gradients*
- *Scatter plots with a legend*
- *Contour Demo*
- *Contour Image*
- *Contourf demo*
- *Contourf and log color scale*
- *Many ways to plot images*
- *Image Masked*
- *Image nonuniform*
- *Layer Images*
- *pcolormesh*
- *Shading example*
- *Axes box aspect*
- *Composing Custom Legends*
- *Using a text as a Path*
- *Creating a colormap from a list of colors*
- *Dolphins*
- *Demo CurveLinear Grid2*
- *Shaded & power normalized rendering*
- *Manual Contour*
- *AGG filter*

- *Matplotlib logo*
- *Table Demo*
- *Plot contour (level) curves in 3D*
- *Plot contour (level) curves in 3D using the extend3d option*
- *Filled contours*
- *Custom hillshading in a 3D surface plot*
- *3D plots as subplots*
- *3D surface (colormap)*
- *3D surface with polar coordinates*
- *Triangular 3D contour plot*
- *Triangular 3D filled contour plot*
- *More triangular 3D surfaces*
- *Hillshading*
- *Left ventricle bullseye*
- *Topographic hillshading*
- *plot_surface(X, Y, Z)*
- *plot_trisurf(x, y, z)*
- *Customized Colorbars Tutorial*
- *Creating Colormaps in Matplotlib*
- *Colormap normalization*

### matplotlib.colors.ListedColormap

**class** matplotlib.colors.**ListedColormap**(*colors*, *name='from_list'*, *N=None*)

Bases: *Colormap*

Colormap object generated from a list of colors.

This may be most useful when indexing directly into a colormap, but it can also be used to generate special colormaps for ordinary mapping.

**Parameters**

**colors**

[list, array] Sequence of Matplotlib color specifications (color names or RGB(A) values).

**name**

[str, optional] String to identify the colormap.

**N**

[int, optional] Number of entries in the map. The default is *None*, in which case there is one colormap entry for each element in the list of colors. If

```
N < len(colors)
```

the list will be truncated at *N*. If

```
N > len(colors)
```

the list will be extended by repetition.

**Parameters**

**name**

[str] The name of the colormap.

**N**

[int] The number of RGB quantization levels.

**resampled**(*lutsize*)

Return a new colormap with *lutsize* entries.

**reversed**(*name=None*)

Return a reversed instance of the Colormap.

**Parameters**

**name**

[str, optional] The name for the reversed colormap. If None, the name is set to `self.name + "_r"`.

**Returns**

**ListedColormap**

A reversed instance of the colormap.

### Examples using `matplotlib.colors.ListedColormap`

- *Multicolored lines*
- *Mapping marker properties to multivariate data*
- *Layer Images*
- *QuadMesh Demo*
- *Histograms*
- *Time Series Histogram*
- *Nested pie charts*
- *Bar chart on polar axis*
- *Selecting individual colors from a colormap*
- *Lasso Demo*
- *Left ventricle bullseye*
- *Customized Colorbars Tutorial*
- *Creating Colormaps in Matplotlib*

### Other classes

| | |
|---|---|
| *ColorSequenceRegistry*() | Container for sequences of colors that are known to Matplotlib by name. |
| *LightSource*([azdeg, altdeg, hsv_min_val, ...]) | Create a light source coming from the specified azimuth and elevation. |

### matplotlib.colors.ColorSequenceRegistry

**class** matplotlib.colors.**ColorSequenceRegistry**

Bases: Mapping

Container for sequences of colors that are known to Matplotlib by name.

The universal registry instance is *matplotlib.color_sequences*. There should be no need for users to instantiate *ColorSequenceRegistry* themselves.

Read access uses a dict-like interface mapping names to lists of colors:

```
import matplotlib as mpl
cmap = mpl.color_sequences['tab10']
```

The returned lists are copies, so that their modification does not change the global definition of the color sequence.

Additional color sequences can be added via *ColorSequenceRegistry.register*:

```
mpl.color_sequences.register('rgb', ['r', 'g', 'b'])
```

**register**(*name*, *color_list*)

Register a new color sequence.

The color sequence registry stores a copy of the given *color_list*, so that future changes to the original list do not affect the registered color sequence. Think of this as the registry taking a snapshot of *color_list* at registration.

**Parameters**

**name**

[str] The name for the color sequence.

**color_list**

[list of *color*] An iterable returning valid Matplotlib colors when iterating over. Note however that the returned color sequence will always be a list regardless of the input type.

**unregister**(*name*)

Remove a sequence from the registry.

You cannot remove built-in color sequences.

If the name is not registered, returns with no error.

## matplotlib.colors.LightSource

**class** matplotlib.colors.**LightSource**(*azdeg=315*, *altdeg=45*, *hsv_min_val=0*, *hsv_max_val=1*, *hsv_min_sat=1*, *hsv_max_sat=0*)

Bases: object

Create a light source coming from the specified azimuth and elevation. Angles are in degrees, with the azimuth measured clockwise from north and elevation up from the zero plane of the surface.

*shade* is used to produce "shaded" RGB values for a data array. *shade_rgb* can be used to combine an RGB image with an elevation map. *hillshade* produces an illumination map of a surface.

Specify the azimuth (measured clockwise from south) and altitude (measured up from the plane of the surface) of the light source in degrees.

**Parameters**

**azdeg**

[float, default: 315 degrees (from the northwest)] The azimuth (0-360, degrees clockwise from North) of the light source.

**altdeg**

[float, default: 45 degrees] The altitude (0-90, degrees up from horizontal) of the light source.

**hsv_min_val**

[number, default: 0] The minimum value ("v" in "hsv") that the *intensity* map can shift the output image to.

**hsv_max_val**

[number, default: 1] The maximum value ("v" in "hsv") that the *intensity* map can shift the output image to.

**hsv_min_sat**

[number, default: 1] The minimum saturation value that the *intensity* map can shift the output image to.

**hsv_max_sat**

[number, default: 0] The maximum saturation value that the *intensity* map can shift the output image to.

## Notes

For backwards compatibility, the parameters *hsv_min_val*, *hsv_max_val*, *hsv_min_sat*, and *hsv_max_sat* may be supplied at initialization as well. However, these parameters will only be used if "blend_mode='hsv'" is passed into `shade` or `shade_rgb`. See the documentation for `blend_hsv` for more details.

**blend_hsv**(*rgb*, *intensity*, *hsv_max_sat=None*, *hsv_max_val=None*, *hsv_min_val=None*, *hsv_min_sat=None*)

Take the input data array, convert to HSV values in the given colormap, then adjust those color values to give the impression of a shaded relief map with a specified light source. RGBA values are returned, which can then be used to plot the shaded image with imshow.

The color of the resulting image will be darkened by moving the (s, v) values (in HSV colorspace) toward (hsv_min_sat, hsv_min_val) in the shaded regions, or lightened by sliding (s, v) toward (hsv_max_sat, hsv_max_val) in regions that are illuminated. The default extremes are chose so that completely shaded points are nearly black (s = 1, v = 0) and completely illuminated points are nearly white (s = 0, v = 1).

**Parameters**

**rgb**

[`ndarray`] An (M, N, 3) RGB array of floats ranging from 0 to 1 (color image).

**intensity**

[ndarray] An (M, N, 1) array of floats ranging from 0 to 1 (grayscale image).

**hsv_max_sat**

[number, optional] The maximum saturation value that the *intensity* map can shift the output image to. If not provided, use the value provided upon initialization.

**hsv_min_sat**

[number, optional] The minimum saturation value that the *intensity* map can shift the output image to. If not provided, use the value provided upon initialization.

**hsv_max_val**

[number, optional] The maximum value ("v" in "hsv") that the *intensity* map can shift the output image to. If not provided, use the value provided upon initialization.

**hsv_min_val**

[number, optional] The minimum value ("v" in "hsv") that the *intensity* map can shift the output image to. If not provided, use the value provided upon initialization.

**Returns**

**ndarray**

An (M, N, 3) RGB array representing the combined images.

**blend_overlay**(*rgb*, *intensity*)

Combine an RGB image with an intensity map using "overlay" blending.

**Parameters**

**rgb**

[ndarray] An (M, N, 3) RGB array of floats ranging from 0 to 1 (color image).

**intensity**

[ndarray] An (M, N, 1) array of floats ranging from 0 to 1 (grayscale image).

**Returns**

**ndarray**

An (M, N, 3) RGB array representing the combined images.

**blend_soft_light**(*rgb*, *intensity*)

Combine an RGB image with an intensity map using "soft light" blending, using the "pegtop" formula.

**Parameters**

**rgb**

[ndarray] An (M, N, 3) RGB array of floats ranging from 0 to 1 (color image).

**intensity**

[ndarray] An (M, N, 1) array of floats ranging from 0 to 1 (grayscale image).

**Returns**

**ndarray**

An (M, N, 3) RGB array representing the combined images.

**property direction**

The unit vector direction towards the light source.

**hillshade**(*elevation*, *vert_exag=1*, *dx=1*, *dy=1*, *fraction=1.0*)

Calculate the illumination intensity for a surface using the defined azimuth and elevation for the light source.

This computes the normal vectors for the surface, and then passes them on to *shade_normals*

**Parameters**

**elevation**

[2D array-like] The height values used to generate an illumination map

**vert_exag**

[number, optional] The amount to exaggerate the elevation values by when calculating illumination. This can be used either to correct for differences in units between the x-y coordinate system and the elevation coordinate system (e.g. decimal degrees vs. meters) or to exaggerate or de-emphasize topographic effects.

**dx**

[number, optional] The x-spacing (columns) of the input *elevation* grid.

**dy**

[number, optional] The y-spacing (rows) of the input *elevation* grid.

**fraction**

[number, optional] Increases or decreases the contrast of the hillshade. Values greater than one will cause intermediate values to move closer to full illumination or shadow (and clipping any values that move beyond 0 or 1). Note that this is not visually or mathematically the same as vertical exaggeration.

**Returns**

**ndarray**

A 2D array of illumination values between 0-1, where 0 is completely in shadow and 1 is completely illuminated.

**shade**(*data*, *cmap*, *norm=None*, *blend_mode='overlay'*, *vmin=None*, *vmax=None*, *vert_exag=1*, *dx=1*, *dy=1*, *fraction=1*, ***kwargs*)

Combine colormapped data values with an illumination intensity map (a.k.a. "hillshade") of the values.

**Parameters**

**data**

[2D array-like] The height values used to generate a shaded map.

**cmap**

[*Colormap*] The colormap used to color the *data* array. Note that this must be a *Colormap* instance. For example, rather than passing in `cmap='gist_earth'`, use `cmap=plt.get_cmap('gist_earth')` instead.

**norm**

[*Normalize* instance, optional] The normalization used to scale values before colormapping. If None, the input will be linearly scaled between its min and max.

**blend_mode**

[{'hsv', 'overlay', 'soft'} or callable, optional] The type of blending used to combine the colormapped data values with the illumination intensity. Default is "overlay". Note that for most topographic surfaces, "overlay" or "soft" appear more visually realistic. If a user-defined function is supplied, it is expected to combine an (M, N, 3) RGB array of floats (ranging 0 to 1) with an (M, N, 1) hillshade array (also 0 to 1). (Call signature `func(rgb, illum, **kwargs)`) Additional kwargs supplied to this function will be passed on to the *blend_mode* function.

**vmin**

[float or None, optional] The minimum value used in colormapping *data*. If *None* the minimum value in *data* is used. If *norm* is specified, then this argument will be ignored.

**vmax**

[float or None, optional] The maximum value used in colormapping *data*. If *None* the maximum value in *data* is used. If *norm* is specified, then this argument will be ignored.

**vert_exag**

[number, optional] The amount to exaggerate the elevation values by when calculating illumination. This can be used either to correct for differences in units between the x-y coordinate system and the elevation coordinate system (e.g. decimal degrees vs. meters) or to exaggerate or de-emphasize topography.

**dx**

[number, optional] The x-spacing (columns) of the input *elevation* grid.

**dy**

[number, optional] The y-spacing (rows) of the input *elevation* grid.

**fraction**

[number, optional] Increases or decreases the contrast of the hillshade. Values greater than one will cause intermediate values to move closer to full illumination or shadow (and clipping any values that move beyond 0 or 1). Note that this is not visually or mathematically the same as vertical exaggeration.

****kwargs**

Additional kwargs are passed on to the *blend_mode* function.

**Returns**

**ndarray**

An (M, N, 4) array of floats ranging between 0-1.

**shade_normals**(*normals*, *fraction=1.0*)

Calculate the illumination intensity for the normal vectors of a surface using the defined azimuth and elevation for the light source.

Imagine an artificial sun placed at infinity in some azimuth and elevation position illuminating our surface. The parts of the surface that slope toward the sun should brighten while those sides facing away should become darker.

**Parameters**

**fraction**

[number, optional] Increases or decreases the contrast of the hillshade. Values greater than one will cause intermediate values to move closer to full illumination or shadow (and clipping any values that move beyond 0 or 1). Note that this is not visually or mathematically the same as vertical exaggeration.

**Returns**

**ndarray**

A 2D array of illumination values between 0-1, where 0 is completely in shadow and 1 is completely illuminated.

**shade_rgb**(*rgb*, *elevation*, *fraction=1.0*, *blend_mode='hsv'*, *vert_exag=1*, *dx=1*, *dy=1*, ***kwargs*)

Use this light source to adjust the colors of the *rgb* input array to give the impression of a shaded relief map with the given *elevation*.

**Parameters**

**rgb**

[array-like] An (M, N, 3) RGB array, assumed to be in the range of 0 to 1.

**elevation**

[array-like] An (M, N) array of the height values used to generate a shaded map.

**fraction**

[number] Increases or decreases the contrast of the hillshade. Values greater than one will cause intermediate values to move closer to full illumination or shadow (and clipping any values that move beyond 0 or 1). Note that this is not visually or mathematically the same as vertical exaggeration.

**blend_mode**

[{'hsv', 'overlay', 'soft'} or callable, optional] The type of blending used to combine the colormapped data values with the illumination intensity. For backwards compatibility, this defaults to "hsv". Note that for most topographic surfaces, "overlay" or "soft" appear more visually realistic. If a user-defined function is supplied, it is expected to combine an (M, N, 3) RGB array of floats (ranging 0 to 1) with an (M, N, 1) hillshade array (also 0 to 1). (Call signature `func(rgb, illum, **kwargs)`) Additional kwargs supplied to this function will be passed on to the *blend_mode* function.

**vert_exag**

[number, optional] The amount to exaggerate the elevation values by when calculating illumination. This can be used either to correct for differences in units between the x-y coordinate system and the elevation coordinate system (e.g. decimal degrees vs. meters) or to exaggerate or de-emphasize topography.

**dx**

[number, optional] The x-spacing (columns) of the input *elevation* grid.

**dy**

[number, optional] The y-spacing (rows) of the input *elevation* grid.

****kwargs**

Additional kwargs are passed on to the *blend_mode* function.

**Returns**

**ndarray**

An (m, n, 3) array of floats ranging between 0-1.

## Examples using `matplotlib.colors.LightSource`

- *Shading example*
- *Shaded & power normalized rendering*
- *AGG filter*
- *Custom hillshading in a 3D surface plot*
- *Hillshading*
- *Topographic hillshading*

## Functions

| | |
|---|---|
| *from_levels_and_colors*(levels, colors[, extend]) | A helper routine to generate a cmap and a norm instance which behave similar to contourf's levels and colors arguments. |
| *hsv_to_rgb*(hsv) | Convert HSV values to RGB. |
| *rgb_to_hsv*(arr) | Convert an array of float RGB values (in the range [0, 1]) to HSV values. |
| *to_hex*(c[, keep_alpha]) | Convert *c* to a hex color. |
| *to_rgb*(c) | Convert *c* to an RGB color, silently dropping the alpha channel. |
| *to_rgba*(c[, alpha]) | Convert *c* to an RGBA color. |
| *to_rgba_array*(c[, alpha]) | Convert *c* to a (n, 4) array of RGBA colors. |
| *is_color_like*(c) | Return whether *c* can be interpreted as an RGB(A) color. |
| *same_color*(c1, c2) | Return whether the colors *c1* and *c2* are the same. |
| *get_named_colors_mapping*() | Return the global mapping of names to named colors. |
| *make_norm_from_scale*(scale_cls[, ...]) | Decorator for building a *Normalize* subclass from a *ScaleBase* subclass. |

## matplotlib.colors.from_levels_and_colors

matplotlib.colors.**from_levels_and_colors**(*levels*, *colors*, *extend='neither'*)

A helper routine to generate a cmap and a norm instance which behave similar to contourf's levels and colors arguments.

**Parameters**

**levels**

[sequence of numbers] The quantization levels used to construct the *BoundaryNorm*. Value `v` is quantized to level `i` if `lev[i] <= v < lev[i+1]`.

**colors**

[sequence of colors] The fill color to use for each level. If *extend* is "neither" there must be `n_level - 1` colors. For an *extend* of "min" or "max" add one extra color, and for an *extend* of "both" add two colors.

**extend**

[{'neither', 'min', 'max', 'both'}, optional] The behaviour when a value falls out of range of the given levels. See *contourf* for details.

**Returns**

**cmap**

[*Colormap*]

**norm**

[*Normalize*]

## matplotlib.colors.hsv_to_rgb

matplotlib.colors.**hsv_to_rgb**(*hsv*)

Convert HSV values to RGB.

**Parameters**

**hsv**

[(..., 3) array-like] All values assumed to be in range [0, 1]

**Returns**

**(..., 3) ndarray**

Colors converted to RGB values in range [0, 1]

### Examples using `matplotlib.colors.hsv_to_rgb`

- *3D voxel / volumetric plot with cylindrical coordinates*

## matplotlib.colors.rgb_to_hsv

matplotlib.colors.**rgb_to_hsv**(*arr*)

Convert an array of float RGB values (in the range [0, 1]) to HSV values.

**Parameters**

**arr**

[(..., 3) array-like] All values must be in the range [0, 1]

**Returns**

**(..., 3) ndarray**

Colors converted to HSV values in range [0, 1]

### Examples using `matplotlib.colors.rgb_to_hsv`

- *List of named colors*
- *Style sheets reference*

## matplotlib.colors.to_hex

matplotlib.colors.**to_hex**(*c*, *keep_alpha=False*)

Convert *c* to a hex color.

**Parameters**

**c**

[*color* or `numpy.ma.masked`]

**keep_alpha**

[bool, default: False] If False, use the `#rrggbb` format, otherwise use `#rrggbbaa`.

**Returns**

**str**

`#rrggbb` or `#rrggbbaa` hex color string

### matplotlib.colors.to_rgb

matplotlib.colors.**to_rgb**(*c*)

Convert *c* to an RGB color, silently dropping the alpha channel.

### Examples using `matplotlib.colors.to_rgb`

- *List of named colors*
- *Style sheets reference*
- *Ribbon Box*

### matplotlib.colors.to_rgba

matplotlib.colors.**to_rgba**(*c*, *alpha=None*)

Convert *c* to an RGBA color.

**Parameters**

**c**

[Matplotlib color or `np.ma.masked`]

**alpha**

[float, optional] If *alpha* is given, force the alpha value of the returned RGBA tuple to *alpha*.

If None, the alpha value from *c* is used. If *c* does not have an alpha channel, then alpha defaults to 1.

*alpha* is ignored for the color value `"none"` (case-insensitive), which always maps to `(0, 0, 0, 0)`.

**Returns**

**tuple**

Tuple of floats `(r, g, b, a)`, where each channel (red, green, blue, alpha) can assume values between 0 and 1.

### Examples using `matplotlib.colors.to_rgba`

- *List of named colors*

### matplotlib.colors.to_rgba_array

`matplotlib.colors.`**`to_rgba_array`**(*c*, *alpha=None*)

Convert *c* to a (n, 4) array of RGBA colors.

**Parameters**

**c**

[Matplotlib color or array of colors] If *c* is a masked array, an `ndarray` is returned with a (0, 0, 0, 0) row for each masked value or row in *c*.

**alpha**

[float or sequence of floats, optional] If *alpha* is given, force the alpha value of the returned RGBA tuple to *alpha*.

If None, the alpha value from *c* is used. If *c* does not have an alpha channel, then alpha defaults to 1.

*alpha* is ignored for the color value `"none"` (case-insensitive), which always maps to `(0, 0, 0, 0)`.

If *alpha* is a sequence and *c* is a single color, *c* will be repeated to match the length of *alpha*.

**Returns**

**array**

(n, 4) array of RGBA colors, where each channel (red, green, blue, alpha) can assume values between 0 and 1.

### Examples using `matplotlib.colors.to_rgba_array`

- *Specifying colors*

### matplotlib.colors.is_color_like

matplotlib.colors.**is_color_like**(*c*)

Return whether *c* can be interpreted as an RGB(A) color.

### matplotlib.colors.same_color

matplotlib.colors.**same_color**(*c1*, *c2*)

Return whether the colors *c1* and *c2* are the same.

*c1*, *c2* can be single colors or lists/arrays of colors.

### matplotlib.colors.get_named_colors_mapping

matplotlib.colors.**get_named_colors_mapping**()

Return the global mapping of names to named colors.

### matplotlib.colors.make_norm_from_scale

matplotlib.colors.**make_norm_from_scale**(*scale_cls*, *base_norm_cls=None*, ***, *init=None*)

Decorator for building a *Normalize* subclass from a *ScaleBase* subclass.

After

```
@make_norm_from_scale(scale_cls)
class norm_cls(Normalize):
    ...
```

*norm_cls* is filled with methods so that normalization computations are forwarded to *scale_cls* (i.e., *scale_cls* is the scale that would be used for the colorbar of a mappable normalized with *norm_cls*).

If *init* is not passed, then the constructor signature of *norm_cls* will be `norm_cls(vmin=None, vmax=None, clip=False)`; these three parameters will be forwarded to the base class (`Normalize.__init__`), and a *scale_cls* object will be initialized with no arguments (other than a dummy axis).

If the *scale_cls* constructor takes additional parameters, then *init* should be passed to *make_norm_from_scale*. It is a callable which is *only* used for its signature. First, this signature will become the signature of *norm_cls*. Second, the *norm_cls* constructor will bind the parameters passed to it using this signature, extract the bound *vmin*, *vmax*, and *clip* values, pass those to `Normalize.__init__`, and forward the remaining bound values (including any defaults defined by the signature) to the *scale_cls* constructor.

### Exported colors

The data used to populate the *List of named colors* are exposed as dictionaries that map color names to hex strings.

matplotlib.colors.**BASE_COLORS**

matplotlib.colors.**TABLEAU_COLORS**

matplotlib.colors.**CSS4_COLORS**

matplotlib.colors.**XKCD_COLORS**

## 7.2.18 matplotlib.container

**class** matplotlib.container.**BarContainer**(**args*, ***kwargs*)

Bases: *Container*

Container for the artists of bar plots (e.g. created by *Axes.bar*).

The container can be treated as a tuple of the *patches* themselves. Additionally, you can access these and further parameters by the attributes.

**Attributes**

**patches**

[list of *Rectangle*] The artists of the bars.

**errorbar**

[None or *ErrorbarContainer*] A container for the error bar artists if error bars are present. *None* otherwise.

**datavalues**

[None or array-like] The underlying data values corresponding to the bars.

**orientation**

[{'vertical', 'horizontal'}, default: None] If 'vertical', the bars are assumed to be vertical. If 'horizontal', the bars are assumed to be horizontal.

**class** matplotlib.container.**Container**(**args*, ***kwargs*)

Bases: tuple

Base class for containers.

Containers are classes that collect semantically related Artists such as the bars of a bar plot.

**add_callback**(*func*)

Add a callback function that will be called whenever one of the *Artist*'s properties changes.

**Parameters**

**func**

[callable] The callback function. It must have the signature:

```
def func(artist: Artist) -> Any
```

where *artist* is the calling *Artist*. Return values may exist but are ignored.

**Returns**

**int**

The observer id associated with the callback. This id can be used for removing the callback with *remove_callback* later.

**See also:**

***remove_callback***

**get_children**()

**get_label**()

Return the label used for this artist in the legend.

**pchanged**()

Call all of the registered callbacks.

This function is triggered internally when a property is changed.

**See also:**

***add_callback***
***remove_callback***

**remove**()

**remove_callback**(*oid*)

Remove a callback based on its observer id.

**See also:**

***add_callback***

**set_label**(*s*)

Set a label that will be displayed in the legend.

**Parameters**

**s**

[object] *s* will be converted to a string by calling `str`.

**class** matplotlib.container.**ErrorbarContainer**(**args*, ***kwargs*)

Bases: *Container*

Container for the artists of error bars (e.g. created by *Axes.errorbar*).

The container can be treated as the *lines* tuple itself. Additionally, you can access these and further parameters by the attributes.

**Attributes**

**lines**

[tuple] Tuple of `(data_line, caplines, barlinecols)`.

- data_line : *Line2D* instance of x, y plot markers and/or line.
- caplines : tuple of *Line2D* instances of the error bar caps.
- barlinecols : list of *LineCollection* with the horizontal and vertical error ranges.

**has_xerr, has_yerr**

[bool] `True` if the errorbar has x/y errors.

**class** matplotlib.container.**StemContainer**(**args*, ***kwargs*)

Bases: *Container*

Container for the artists created in a *Axes.stem()* plot.

The container can be treated like a namedtuple `(markerline, stemlines, baseline)`.

**Attributes**

**markerline**

[*Line2D*] The artist of the markers at the stem heads.

**stemlines**

[list of *Line2D*] The artists of the vertical lines for all stems.

**baseline**

[*Line2D*] The artist of the horizontal baseline.

**Parameters**

**markerline_stemlines_baseline**

[tuple] Tuple of `(markerline, stemlines, baseline)`. `markerline` contains the *LineCollection* of the markers, `stemlines` is a *LineCollection* of the main lines, `baseline` is the *Line2D* of the baseline.

### 7.2.19 matplotlib.contour

Classes to support contour plotting and labelling for the Axes class.

**class** matplotlib.contour.**ContourLabeler**

Bases: object

Mixin to provide labelling capability to *ContourSet*.

**add_label**(*x*, *y*, *rotation*, *lev*, *cvalue*)

Add a contour label, respecting whether *use_clabeltext* was set.

**add_label_clabeltext**(*x*, *y*, *rotation*, *lev*, *cvalue*)

[*Deprecated*] Add contour label with *Text.set_transform_rotates_text*.

#### Notes

Deprecated since version 3.8: Use add_label instead.

**add_label_near**(*x*, *y*, *inline=True*, *inline_spacing=5*, *transform=None*)

Add a label near the point `(x, y)`.

**Parameters**

**x, y**

[float] The approximate location of the label.

**inline**

[bool, default: True] If *True* remove the segment of the contour beneath the label.

**inline_spacing**

[int, default: 5] Space in pixels to leave on each side of label when placing inline. This spacing will be exact for labels at locations where the contour is straight, less so for labels on curved contours.

**transform**

[*Transform* or `False`, default: `self.axes.transData`] A transform applied to `(x, y)` before labeling. The default causes `(x, y)` to be interpreted as data coordinates. `False` is a synonym for *IdentityTransform*; i.e. `(x, y)` should be interpreted as display coordinates.

**calc_label_rot_and_inline**(*slc*, *ind*, *lw*, *lc=None*, *spacing=5*)

[*Deprecated*] Calculate the appropriate label rotation given the linecontour coordinates in screen units, the index of the label location and the label width.

If *lc* is not None or empty, also break contours and compute inlining.

*spacing* is the empty space to leave around the label, in pixels.

Both tasks are done together to avoid calculating path lengths multiple times, which is relatively costly.

The method used here involves computing the path length along the contour in pixel coordinates and then looking approximately (label width / 2) away from central point to determine rotation and then to break contour if desired.

### Notes

Deprecated since version 3.8.

**clabel**(*levels=None*, ***, *fontsize=None*, *inline=True*, *inline_spacing=5*, *fmt=None*, *colors=None*, *use_clabeltext=False*, *manual=False*, *rightside_up=True*, *zorder=None*)

Label a contour plot.

Adds labels to line contours in this *ContourSet* (which inherits from this mixin class).

**Parameters**

**levels**

[array-like, optional] A list of level values, that should be labeled. The list must be a subset of `cs.levels`. If not given, all levels are labeled.

**fontsize**

[str or float, default: *rcParams["font.size"]* (default: `10.0`)] Size in points or relative size e.g., 'smaller', 'x-large'. See *Text.set_size* for accepted string values.

**colors**

[*color* or colors or None, default: None] The label colors:

- If *None*, the color of each label matches the color of the corresponding contour.
- If one string color, e.g., *colors* = 'r' or *colors* = 'red', all labels will be plotted in this color.
- If a tuple of colors (string, float, RGB, etc), different labels will be plotted in different colors in the order specified.

**inline**

[bool, default: True] If `True` the underlying contour is removed where the label is placed.

**inline_spacing**

[float, default: 5] Space in pixels to leave on each side of label when placing inline.

This spacing will be exact for labels at locations where the contour is straight, less so for labels on curved contours.

**fmt**

[*Formatter* or str or callable or dict, optional] How the levels are formatted:

- If a *Formatter*, it is used to format all levels at once, using its *Formatter.format_ticks* method.
- If a str, it is interpreted as a %-style format string.
- If a callable, it is called with one level at a time and should return the corresponding label.
- If a dict, it should directly map levels to labels.

The default is to use a standard *ScalarFormatter*.

**manual**

[bool or iterable, default: False] If `True`, contour labels will be placed manually using mouse clicks. Click the first button near a contour to add a label, click the second button (or potentially both mouse buttons at once) to finish adding labels. The third button can be used to remove the last label added, but only if labels are not inline. Alternatively, the keyboard can be used to select label locations (enter to end label placement, delete or backspace act like the third mouse button, and any other key will select a label location).

*manual* can also be an iterable object of (x, y) tuples. Contour labels will be created as if mouse is clicked at each (x, y) position.

**rightside_up**

[bool, default: True] If `True`, label rotations will always be plus or minus 90 degrees from level.

**use_clabeltext**

[bool, default: False] If `True`, use *Text.set_transform_rotates_text* to ensure that label rotation is updated whenever the Axes aspect changes.

**zorder**

[float or None, default: `(2 + contour.get_zorder())`] zorder of the contour labels.

**Returns**

**labels**

A list of *Text* instances for the labels.

**get_text**(*lev*, *fmt*)

Get the text of the label.

**labels**(*inline*, *inline_spacing*)

**locate_label**(*linecontour*, *labelwidth*)

Find good place to draw a label (relatively flat part of the contour).

**pop_label**(*index=-1*)

Defaults to removing last label, but any index can be supplied

**print_label**(*linecontour*, *labelwidth*)

Return whether a contour is long enough to hold a label.

**remove**()

**too_close**(*x*, *y*, *lw*)

Return whether a label is already near this location.

**class** matplotlib.contour.**ContourSet**(*ax*, **args*, *levels=None*, *filled=False*, *linewidths=None*, *linestyles=None*, *hatches=(None,)*, *alpha=None*, *origin=None*, *extent=None*, *cmap=None*, *colors=None*, *norm=None*, *vmin=None*, *vmax=None*, *extend='neither'*, *antialiased=None*, *nchunk=0*, *locator=None*, *transform=None*, *negative_linestyles=None*, *clip_path=None*, ***kwargs*)

Bases: *ContourLabeler*, *Collection*

Store a set of contour lines or filled regions.

User-callable method: *clabel*

**Parameters**

**ax**

[*Axes*]

**levels**

[[level0, level1, ..., leveln]] A list of floating point numbers indicating the contour levels.

**allsegs**

[[level0segs, level1segs, ...]] List of all the polygon segments for all the *levels*. For contour lines `len(allsegs) == len(levels)`, and for filled contour regions `len(allsegs) = len(levels)-1`. The lists should look like

```
level0segs = [polygon0, polygon1, ...]
polygon0 = [[x0, y0], [x1, y1], ...]
```

**allkinds**

[`None` or [level0kinds, level1kinds, ...]] Optional list of all the polygon vertex kinds (code types), as described and used in Path. This is used to allow multiply- connected paths such as holes within filled polygons. If not `None`, `len(allkinds) == len(allsegs)`. The lists should look like

```
level0kinds = [polygon0kinds, ...]
polygon0kinds = [vertexcode0, vertexcode1, ...]
```

If *allkinds* is not `None`, usually all polygons for a particular contour level are grouped together so that `level0segs = [polygon0]` and `level0kinds = [polygon0kinds]`.

****kwargs**

Keyword arguments are as described in the docstring of *contour*.

**Attributes**

**ax**

[*Axes*] The Axes object in which the contours are drawn.

***collections***

[*silent_list* of *PathCollection*s] [*Deprecated*]

**levels**

[array] The values of the contour levels.

**layers**

[array] Same as levels for line contours; half-way between levels for filled contours. See `ContourSet._process_colors`.

Draw contour lines or filled regions, depending on whether keyword arg *filled* is `False` (default) or `True`.

Call signature:

```
ContourSet(ax, levels, allsegs, [allkinds], **kwargs)
```

**Parameters**

**ax**

[*Axes*] The *Axes* object to draw on.

**levels**

[[level0, level1, ..., leveln]] A list of floating point numbers indicating the contour levels.

**allsegs**

[[level0segs, level1segs, ...]] List of all the polygon segments for all the *levels*. For contour lines `len(allsegs) == len(levels)`, and for filled contour regions `len(allsegs) = len(levels)-1`. The lists should look like

```
level0segs = [polygon0, polygon1, ...]
polygon0 = [[x0, y0], [x1, y1], ...]
```

**allkinds**

[[level0kinds, level1kinds, ...], optional] Optional list of all the polygon vertex kinds (code types), as described and used in Path. This is used to allow multiply- connected paths such as holes within filled polygons. If not `None`, `len(allkinds) == len(allsegs)`. The lists should look like

```
level0kinds = [polygon0kinds, ...]
polygon0kinds = [vertexcode0, vertexcode1, ...]
```

If *allkinds* is not `None`, usually all polygons for a particular contour level are grouped together so that `level0segs = [polygon0]` and `level0kinds = [polygon0kinds]`.

****kwargs**

Keyword arguments are as described in the docstring of *contour*.

**property allkinds**

**property allsegs**

**property alpha**

**property antialiased**

**changed**()

Call this whenever the mappable is changed to notify all the callbackSM listeners to the 'changed' signal.

**property collections**

[*Deprecated*]

### Notes

Deprecated since version 3.8:

**draw**(*renderer*)

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

**Parameters**

**renderer**

[*RendererBase* subclass.]

### Notes

This method is overridden in the Artist subclasses.

**find_nearest_contour**(*x*, *y*, *indices=None*, *pixel=True*)

Find the point in the contour plot that is closest to `(x, y)`.

This method does not support filled contours.

**Parameters**

**x, y**

[float] The reference point.

**indices**

[list of int or None, default: None] Indices of contour levels to consider. If None (the default), all levels are considered.

**pixel**

[bool, default: True] If *True*, measure distance in pixel (screen) space, which is useful for manual contour labeling; else, measure distance in axes space.

**Returns**

**path**

[int] The index of the path that is closest to `(x, y)`. Each path corresponds to one contour level.

**subpath**

[int] The index within that closest path of the subpath that is closest to `(x, y)`. Each subpath corresponds to one unbroken contour line.

**index**

[int] The index of the vertices within that subpath that are closest to `(x, y)`.

**xmin, ymin**

[float] The point in the contour plot that is closest to `(x, y)`.

**d2**

[float] The squared distance from `(xmin, ymin)` to `(x, y)`.

**get_transform**()

Return the *Transform* instance used by this ContourSet.

**legend_elements**(*variable_name='x'*, *str_format=<class 'str'>*)

Return a list of artists and labels suitable for passing through to *legend* which represent this ContourSet.

The labels have the form "0 < x <= 1" stating the data ranges which the artists represent.

**Parameters**

**variable_name**

[str] The string used inside the inequality used on the labels.

**str_format**

[function: float -> str] Function used to format the numbers in the labels.

**Returns**

**artists**

[list[*Artist*]] A list of the artists.

**labels**

[list[str]] A list of the labels.

**property linestyles**

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *antialiased=<UNSET>*, *array=<UNSET>*, *capstyle=<UNSET>*, *clim=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *cmap=<UNSET>*, *color=<UNSET>*, *edgecolor=<UNSET>*, *facecolor=<UNSET>*, *gid=<UNSET>*, *hatch=<UNSET>*, *in_layout=<UNSET>*, *joinstyle=<UNSET>*, *label=<UNSET>*, *linestyle=<UNSET>*, *linewidth=<UNSET>*, *mouseover=<UNSET>*, *norm=<UNSET>*, *offset_transform=<UNSET>*, *offsets=<UNSET>*, *path_effects=<UNSET>*, *paths=<UNSET>*, *picker=<UNSET>*, *pickradius=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *urls=<UNSET>*, *visible=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi v |
| *alpha* | array-like or scalar or None |
| *animated* | bool |
| *antialiased* or aa or antialiaseds | bool or list of bools |
| *array* | array-like or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clim* | (vmin: float, vmax: float) |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *cmap* | *Colormap* or str or None |

Table 47 – continued from p

| Property | Description |
|---|---|
| *color* | *color* or list of RGBA tuples |
| *edgecolor* or ec or edgecolors | *color* or list of *color* or 'face' |
| *facecolor* or facecolors or fc | *color* or list of *color* |
| *figure* | *Figure* |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or dashes or linestyles or ls | str or tuple or list thereof |
| *linewidth* or linewidths or lw | float or list of floats |
| *mouseover* | bool |
| *norm* | *Normalize* or str or None |
| *offset_transform* or transOffset | *Transform* |
| *offsets* | (N, 2) or (2,) array-like |
| *path_effects* | list of *AbstractPathEffect* |
| *paths* | unknown |
| *picker* | None or bool or float or callable |
| *pickradius* | float |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *urls* | list of str or None |
| *visible* | bool |
| *zorder* | float |

**property tcolors**

[*Deprecated*]

### Notes

Deprecated since version 3.8:

**property tlinewidths**

[*Deprecated*]

### Notes

Deprecated since version 3.8:

**class** matplotlib.contour.**QuadContourSet**(*ax*, **args*, *levels=None*, *filled=False*, *linewidths=None*, *linestyles=None*, *hatches=(None,)*, *alpha=None*, *origin=None*, *extent=None*, *cmap=None*, *colors=None*, *norm=None*, *vmin=None*, *vmax=None*, *extend='neither'*, *antialiased=None*, *nchunk=0*, *locator=None*, *transform=None*, *negative_linestyles=None*, *clip_path=None*, ***kwargs*)

Bases: *ContourSet*

Create and store a set of contour lines or filled regions.

This class is typically not instantiated directly by the user but by *contour* and *contourf*.

**Attributes**

**ax**

[*Axes*] The Axes object in which the contours are drawn.

**collections**

[*silent_list* of *PathCollection*s] [*Deprecated*]

**levels**

[array] The values of the contour levels.

**layers**

[array] Same as levels for line contours; half-way between levels for filled contours. See `ContourSet._process_colors`.

Draw contour lines or filled regions, depending on whether keyword arg *filled* is `False` (default) or `True`.

Call signature:

```
ContourSet(ax, levels, allsegs, [allkinds], **kwargs)
```

**Parameters**

**ax**

[*Axes*] The *Axes* object to draw on.

**levels**

[[level0, level1, ..., leveln]] A list of floating point numbers indicating the contour levels.

**allsegs**

[[level0segs, level1segs, ...]] List of all the polygon segments for all the *levels*. For contour lines `len(allsegs) == len(levels)`, and for filled contour regions `len(allsegs) = len(levels)-1`. The lists should look like

```
level0segs = [polygon0, polygon1, ...]
polygon0 = [[x0, y0], [x1, y1], ...]
```

**allkinds**

[[level0kinds, level1kinds, ...], optional] Optional list of all the polygon vertex kinds (code types), as described and used in Path. This is used to allow multiply- connected paths such as holes within filled polygons. If not `None`, `len(allkinds) == len(allsegs)`. The lists should look like

```
level0kinds = [polygon0kinds, ...]
polygon0kinds = [vertexcode0, vertexcode1, ...]
```

If *allkinds* is not `None`, usually all polygons for a particular contour level are grouped together so that `level0segs = [polygon0]` and `level0kinds = [polygon0kinds]`.

****kwargs**

Keyword arguments are as described in the docstring of *contour*.

**set** (*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *antialiased=<UNSET>*, *array=<UNSET>*, *capstyle=<UNSET>*, *clim=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *cmap=<UNSET>*, *color=<UNSET>*, *edgecolor=<UNSET>*, *facecolor=<UNSET>*, *gid=<UNSET>*, *hatch=<UNSET>*, *in_layout=<UNSET>*, *joinstyle=<UNSET>*, *label=<UNSET>*, *linestyle=<UNSET>*, *linewidth=<UNSET>*, *mouseover=<UNSET>*, *norm=<UNSET>*, *offset_transform=<UNSET>*, *offsets=<UNSET>*, *path_effects=<UNSET>*, *paths=<UNSET>*, *picker=<UNSET>*, *pickradius=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *urls=<UNSET>*, *visible=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

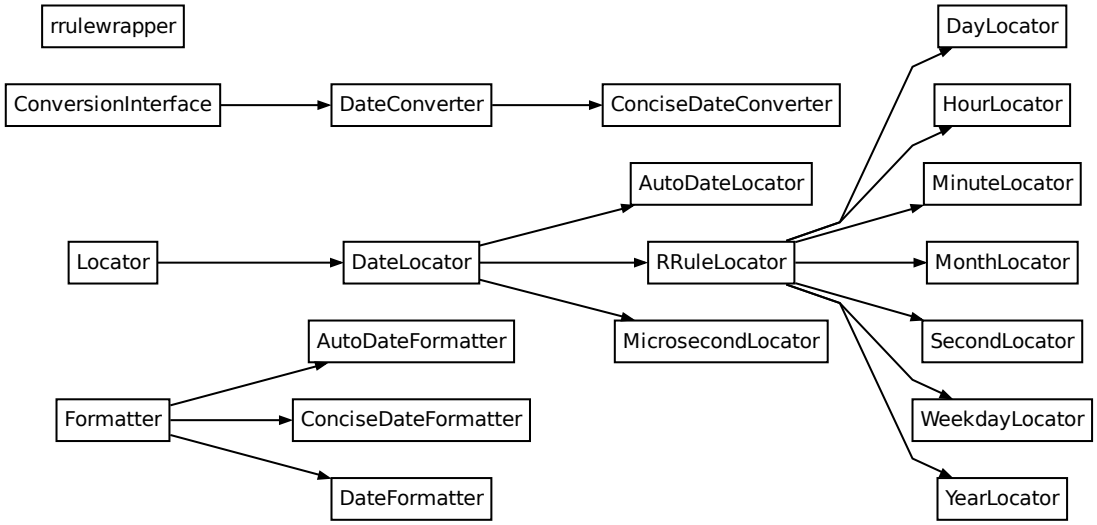
Supported properties are

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi v |
| *alpha* | array-like or scalar or None |
| *animated* | bool |
| *antialiased* or aa or antialiaseds | bool or list of bools |
| *array* | array-like or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clim* | (vmin: float, vmax: float) |

Table 48 – continued from p

| Property | Description |
|---|---|
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *cmap* | *Colormap* or str or None |
| *color* | *color* or list of RGBA tuples |
| *edgecolor* or ec or edgecolors | *color* or list of *color* or 'face' |
| *facecolor* or facecolors or fc | *color* or list of *color* |
| *figure* | *Figure* |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or dashes or linestyles or ls | str or tuple or list thereof |
| *linewidth* or linewidths or lw | float or list of floats |
| *mouseover* | bool |
| *norm* | *Normalize* or str or None |
| *offset_transform* or transOffset | *Transform* |
| *offsets* | (N, 2) or (2,) array-like |
| *path_effects* | list of *AbstractPathEffect* |
| *paths* | unknown |
| *picker* | None or bool or float or callable |
| *pickradius* | float |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *urls* | list of str or None |
| *visible* | bool |
| *zorder* | float |

### 7.2.20 matplotlib.dates




Matplotlib provides sophisticated date plotting capabilities, standing on the shoulders of python `datetime` and the add-on module dateutil.

By default, Matplotlib uses the units machinery described in *units* to convert `datetime.datetime`, and `numpy.datetime64` objects when plotted on an x- or y-axis. The user does not need to do anything for dates to be formatted, but dates often have strict formatting needs, so this module provides many tick locators and formatters. A basic example using `numpy.datetime64` is:

```
import numpy as np

times = np.arange(np.datetime64('2001-01-02'),
                  np.datetime64('2002-02-03'), np.timedelta64(75, 'm'))
y = np.random.randn(len(times))

fig, ax = plt.subplots()
ax.plot(times, y)
```

**See also:**

- *Date tick labels*
- *Formatting date ticks using ConciseDateFormatter*
- *Date Demo Convert*

### Matplotlib date format

Matplotlib represents dates using floating point numbers specifying the number of days since a default epoch of 1970-01-01 UTC; for example, 1970-01-01, 06:00 is the floating point number 0.25. The formatters and locators require the use of `datetime.datetime` objects, so only dates between year 0001 and 9999 can be represented. Microsecond precision is achievable for (approximately) 70 years on either side of the epoch, and 20 microseconds for the rest of the allowable range of dates (year 0001 to 9999). The epoch can be changed at import time via *dates.set_epoch* or *rcParams["dates.epoch"]* to other dates if necessary; see *Date Precision and Epochs* for a discussion.

**Note:** Before Matplotlib 3.3, the epoch was 0000-12-31 which lost modern microsecond precision and also made the default axis limit of 0 an invalid datetime. In 3.3 the epoch was changed as above. To convert old ordinal floats to the new epoch, users can do:

```
new_ordinal = old_ordinal + mdates.date2num(np.datetime64('0000-12-31'))
```

There are a number of helper functions to convert between `datetime` objects and Matplotlib dates:

| | |
|---|---|
| *datestr2num* | Convert a date string to a datenum using `dateutil.parser.parse`. |
| *date2num* | Convert datetime objects to Matplotlib dates. |
| *num2date* | Convert Matplotlib dates to `datetime` objects. |
| *num2timedelta* | Convert number of days to a `timedelta` object. |
| *drange* | Return a sequence of equally spaced Matplotlib dates. |
| *set_epoch* | Set the epoch (origin for dates) for datetime calculations. |
| *get_epoch* | Get the epoch used by *dates*. |

**Note:** Like Python's `datetime.datetime`, Matplotlib uses the Gregorian calendar for all conversions between dates and floating point numbers. This practice is not universal, and calendar differences can cause confusing differences between what Python and Matplotlib give as the number of days since 0001-01-01 and what other software and databases yield. For example, the US Naval Observatory uses a calendar that switches from Julian to Gregorian in October, 1582. Hence, using their calculator, the number of days between 0001-01-01 and 2006-04-01 is 732403, whereas using the Gregorian calendar via the datetime module we find:

```
In [1]: date(2006, 4, 1).toordinal() - date(1, 1, 1).toordinal()
Out[1]: 732401
```

All the Matplotlib date converters, locators and formatters are timezone aware. If no explicit timezone is provided, *rcParams["timezone"]* (default: `'UTC'`) is assumed, provided as a string. If you want to use a different timezone, pass the *tz* keyword argument of *num2date* to any date tick locators or formatters

you create. This can be either a `datetime.tzinfo` instance or a string with the timezone name that can be parsed by `gettz`.

A wide range of specific and general purpose date tick locators and formatters are provided in this module. See *matplotlib.ticker* for general information on tick locators and formatters. These are described below.

The dateutil module provides additional code to handle date ticking, making it easy to place ticks on any kinds of dates. See examples below.

### Date tick locators

Most of the date tick locators can locate single or multiple ticks. For example:

```
# import constants for the days of the week
from matplotlib.dates import MO, TU, WE, TH, FR, SA, SU

# tick on Mondays every week
loc = WeekdayLocator(byweekday=MO, tz=tz)

# tick on Mondays and Saturdays
loc = WeekdayLocator(byweekday=(MO, SA))
```

In addition, most of the constructors take an interval argument:

```
# tick on Mondays every second week
loc = WeekdayLocator(byweekday=MO, interval=2)
```

The rrule locator allows completely general date ticking:

```
# tick every 5th easter
rule = rrulewrapper(YEARLY, byeaster=1, interval=5)
loc = RRuleLocator(rule)
```

The available date tick locators are:

- *MicrosecondLocator*: Locate microseconds.
- *SecondLocator*: Locate seconds.
- *MinuteLocator*: Locate minutes.
- *HourLocator*: Locate hours.
- *DayLocator*: Locate specified days of the month.
- *WeekdayLocator*: Locate days of the week, e.g., MO, TU.
- *MonthLocator*: Locate months, e.g., 7 for July.
- *YearLocator*: Locate years that are multiples of base.
- *RRuleLocator*: Locate using a *rrulewrapper*. *rrulewrapper* is a simple wrapper around dateutil's `dateutil.rrule` which allow almost arbitrary date tick specifications. See *rrule example*.

- *AutoDateLocator*: On autoscale, this class picks the best *DateLocator* (e.g., *RRuleLocator*) to set the view limits and the tick locations. If called with `interval_multiples=True` it will make ticks line up with sensible multiples of the tick intervals. For example, if the interval is 4 hours, it will pick hours 0, 4, 8, etc. as ticks. This behaviour is not guaranteed by default.

### Date formatters

The available date formatters are:

- *AutoDateFormatter*: attempts to figure out the best format to use. This is most useful when used with the *AutoDateLocator*.
- *ConciseDateFormatter*: also attempts to figure out the best format to use, and to make the format as compact as possible while still having complete date information. This is most useful when used with the *AutoDateLocator*.
- *DateFormatter*: use `strftime` format strings.

**class** matplotlib.dates.**AutoDateFormatter**(*locator*, *tz=None*, *defaultfmt='%Y-%m-%d'*, ***, *usetex=None*)

Bases: *Formatter*

A *Formatter* which attempts to figure out the best format to use. This is most useful when used with the *AutoDateLocator*.

*AutoDateFormatter* has a `.scale` dictionary that maps tick scales (the interval in days between one major tick) to format strings; this dictionary defaults to

```
self.scaled = {
    DAYS_PER_YEAR: rcParams['date.autoformatter.year'],
    DAYS_PER_MONTH: rcParams['date.autoformatter.month'],
    1: rcParams['date.autoformatter.day'],
    1 / HOURS_PER_DAY: rcParams['date.autoformatter.hour'],
    1 / MINUTES_PER_DAY: rcParams['date.autoformatter.minute'],
    1 / SEC_PER_DAY: rcParams['date.autoformatter.second'],
    1 / MUSECONDS_PER_DAY: rcParams['date.autoformatter.microsecond'],
}
```

The formatter uses the format string corresponding to the lowest key in the dictionary that is greater or equal to the current scale. Dictionary entries can be customized:

```
locator = AutoDateLocator()
formatter = AutoDateFormatter(locator)
formatter.scaled[1/(24*60)] = '%M:%S' # only show min and sec
```

Custom callables can also be used instead of format strings. The following example shows how to use a custom format function to strip trailing zeros from decimal seconds and adds the date to the first ticklabel:

```
def my_format_function(x, pos=None):
    x = matplotlib.dates.num2date(x)
```

(continues on next page)

(continued from previous page)

```
    if pos == 0:
        fmt = '%D %H:%M:%S.%f'
    else:
        fmt = '%H:%M:%S.%f'
    label = x.strftime(fmt)
    label = label.rstrip("0")
    label = label.rstrip(".")
    return label

formatter.scaled[1/(24*60)] = my_format_function
```

Autoformat the date labels.

**Parameters**

**locator**

[*ticker.Locator*] Locator that this axis is using.

**tz**

[str or `tzinfo`, default: *rcParams["timezone"]* (default: `'UTC'`)] Ticks timezone. If a string, *tz* is passed to `dateutil.tz`.

**defaultfmt**

[str] The default format to use if none of the values in `self.scaled` are greater than the unit returned by `locator._get_unit()`.

**usetex**

[bool, default: *rcParams["text.usetex"]* (default: `False`)] To enable/disable the use of TeX's math mode for rendering the results of the formatter. If any entries in `self.scaled` are set as functions, then it is up to the customized function to enable or disable TeX's math mode itself.

**class** matplotlib.dates.**AutoDateLocator**(*tz=None*, *minticks=5*, *maxticks=None*, *interval_multiples=True*)

Bases: *DateLocator*

On autoscale, this class picks the best *DateLocator* to set the view limits and the tick locations.

**Attributes**

**intervald**

[dict] Mapping of tick frequencies to multiples allowed for that ticking. The default is

```
self.intervald = {
    YEARLY  : [1, 2, 4, 5, 10, 20, 40, 50, 100, 200, 400,␣
↪500,
               1000, 2000, 4000, 5000, 10000],
```

(continues on next page)

(continued from previous page)

```
    MONTHLY : [1, 2, 3, 4, 6],
    DAILY   : [1, 2, 3, 7, 14, 21],
    HOURLY  : [1, 2, 3, 4, 6, 12],
    MINUTELY: [1, 5, 10, 15, 30],
    SECONDLY: [1, 5, 10, 15, 30],
    MICROSECONDLY: [1, 2, 5, 10, 20, 50, 100, 200, 500,
                    1000, 2000, 5000, 10000, 20000, 50000,
                    100000, 200000, 500000, 1000000],
}
```

where the keys are defined in `dateutil.rrule`.

The interval is used to specify multiples that are appropriate for the frequency of ticking. For instance, every 7 days is sensible for daily ticks, but for minutes/seconds, 15 or 30 make sense.

When customizing, you should only modify the values for the existing keys. You should not add or delete entries.

Example for forcing ticks every 3 hours:

```
locator = AutoDateLocator()
locator.intervald[HOURLY] = [3]  # only show every 3 hours
```

**Parameters**

**tz**

[str or `tzinfo`, default: *rcParams["timezone"]* (default: `'UTC'`)] Ticks timezone. If a string, *tz* is passed to `dateutil.tz`.

**minticks**

[int] The minimum number of ticks desired; controls whether ticks occur yearly, monthly, etc.

**maxticks**

[int] The maximum number of ticks desired; controls the interval between ticks (ticking every other, every 3, etc.). For fine-grained control, this can be a dictionary mapping individual rrule frequency constants (YEARLY, MONTHLY, etc.) to their own maximum number of ticks. This can be used to keep the number of ticks appropriate to the format chosen in *AutoDateFormatter*. Any frequency not specified in this dictionary is given a default value.

**interval_multiples**

[bool, default: True] Whether ticks should be chosen to be multiple of the interval, locking them to 'nicer' locations. For example, this will force the ticks to be at hours 0, 6, 12, 18 when hourly ticking is done at 6 hour intervals.

**get_locator**(*dmin*, *dmax*)

Pick the best locator based on a distance.

**nonsingular**(*vmin*, *vmax*)

Given the proposed upper and lower extent, adjust the range if it is too close to being singular (i.e. a range of ~0).

**tick_values**(*vmin*, *vmax*)

Return the values of the located ticks given **vmin** and **vmax**.

---

**Note:** To get tick locations with the vmin and vmax values defined automatically for the associated `axis` simply call the Locator instance:

```
>>> print(type(loc))
<type 'Locator'>
>>> print(loc())
[1, 2, 3, 4]
```

---

**class** matplotlib.dates.**ConciseDateConverter**(*formats=None*, *zero_formats=None*, *offset_formats=None*, *show_offset=True*, ***, *interval_multiples=True*)

Bases: *DateConverter*

**axisinfo**(*unit*, *axis*)

Return the *AxisInfo* for *unit*.

*unit* is a `tzinfo` instance or None. The *axis* argument is required but not used.

**class** matplotlib.dates.**ConciseDateFormatter**(*locator*, *tz=None*, *formats=None*, *offset_formats=None*, *zero_formats=None*, *show_offset=True*, ***, *usetex=None*)

Bases: *Formatter*

A *Formatter* which attempts to figure out the best format to use for the date, and to make it as compact as possible, but still be complete. This is most useful when used with the *AutoDateLocator*:

```
>>> locator = AutoDateLocator()
>>> formatter = ConciseDateFormatter(locator)
```

**Parameters**

**locator**

[*ticker.Locator*] Locator that this axis is using.

**tz**

[str or `tzinfo`, default: *rcParams["timezone"]* (default: `'UTC'`)] Ticks timezone, passed to *dates.num2date*.

**formats**

[list of 6 strings, optional] Format strings for 6 levels of tick labelling: mostly years, months, days, hours, minutes, and seconds. Strings use the same format codes as `strftime`. Default is `['%Y', '%b', '%d', '%H:%M', '%H:%M', '%S.%f']`

**zero_formats**

[list of 6 strings, optional] Format strings for tick labels that are "zeros" for a given tick level. For instance, if most ticks are months, ticks around 1 Jan 2005 will be labeled "Dec", "2005", "Feb". The default is `['', '%Y', '%b', '%b-%d', '%H:%M', '%H:%M']`

**offset_formats**

[list of 6 strings, optional] Format strings for the 6 levels that is applied to the "offset" string found on the right side of an x-axis, or top of a y-axis. Combined with the tick labels this should completely specify the date. The default is:

```
['', '%Y', '%Y-%b', '%Y-%b-%d', '%Y-%b-%d', '%Y-%b-%d %H:%M
 ↪']
```

**show_offset**

[bool, default: True] Whether to show the offset or not.

**usetex**

[bool, default: *rcParams["text.usetex"]* (default: `False`)] To enable/disable the use of TeX's math mode for rendering the results of the formatter.

### Examples

See *Formatting date ticks using ConciseDateFormatter*

Autoformat the date labels. The default format is used to form an initial string, and then redundant elements are removed.

**format_data_short**(*value*)

Return a short string version of the tick value.

Defaults to the position-independent long value.

**format_ticks**(*values*)

Return the tick labels for all the ticks at once.

**get_offset**()

**class** matplotlib.dates.**DateConverter**(*, *interval_multiples=True*)

Bases: *ConversionInterface*

Converter for `datetime.date` and `datetime.datetime` data, or for date/time data represented as it would be converted by *date2num*.

The 'unit' tag for such data is None or a `tzinfo` instance.

**axisinfo**(*unit*, *axis*)

Return the *AxisInfo* for *unit*.

*unit* is a `tzinfo` instance or None. The *axis* argument is required but not used.

**static convert**(*value*, *unit*, *axis*)

If *value* is not already a number or sequence of numbers, convert it with *date2num*.

The *unit* and *axis* arguments are not used.

**static default_units**(*x*, *axis*)

Return the `tzinfo` instance of *x* or of its first element, or None

**class** matplotlib.dates.**DateFormatter**(*fmt*, *tz=None*, *, *usetex=None*)

Bases: *Formatter*

Format a tick (in days since the epoch) with a `strftime` format string.

**Parameters**

**fmt**

[str] `strftime` format string

**tz**

[str or `tzinfo`, default: *rcParams["timezone"]* (default: `'UTC'`)] Ticks timezone. If a string, *tz* is passed to `dateutil.tz`.

**usetex**

[bool, default: *rcParams["text.usetex"]* (default: `False`)] To enable/disable the use of TeX's math mode for rendering the results of the formatter.

**set_tzinfo**(*tz*)

**class** matplotlib.dates.**DateLocator**(*tz=None*)

Bases: *Locator*

Determines the tick locations when plotting dates.

This class is subclassed by other Locators and is not meant to be used on its own.

**Parameters**

**tz**

[str or `tzinfo`, default: *rcParams["timezone"]* (default: `'UTC'`)] Ticks timezone. If a string, *tz* is passed to `dateutil.tz`.

**datalim_to_dt**()

Convert axis data interval to datetime objects.

**hms0d = {'byhour': 0, 'byminute': 0, 'bysecond': 0}**

**nonsingular**(*vmin*, *vmax*)

Given the proposed upper and lower extent, adjust the range if it is too close to being singular (i.e. a range of ~0).

**set_tzinfo**(*tz*)

Set timezone info.

**Parameters**

**tz**

[str or `tzinfo`, default: *rcParams["timezone"]* (default: `'UTC'`)] Ticks timezone. If a string, *tz* is passed to `dateutil.tz`.

**viewlim_to_dt**()

Convert the view interval to datetime objects.

**class** matplotlib.dates.**DayLocator**(*bymonthday=None*, *interval=1*, *tz=None*)

Bases: *RRuleLocator*

Make ticks on occurrences of each day of the month. For example, 1, 15, 30.

**Parameters**

**bymonthday**

[int or list of int, default: all days] Ticks will be placed on every day in *bymonthday*. Default is `bymonthday=range(1, 32)`, i.e., every day of the month.

**interval**

[int, default: 1] The interval between each iteration. For example, if `interval=2`, mark every second occurrence.

**tz**

[str or `tzinfo`, default: *rcParams["timezone"]* (default: `'UTC'`)] Ticks timezone. If a string, *tz* is passed to `dateutil.tz`.

**class** matplotlib.dates.**HourLocator**(*byhour=None*, *interval=1*, *tz=None*)

Bases: *RRuleLocator*

Make ticks on occurrences of each hour.

**Parameters**

**byhour**

[int or list of int, default: all hours] Ticks will be placed on every hour in *byhour*. Default is `byhour=range(24)`, i.e., every hour.

**interval**

[int, default: 1] The interval between each iteration. For example, if `interval=2`, mark every second occurrence.

**tz**

[str or `tzinfo`, default: *rcParams["timezone"]* (default: `'UTC'`)] Ticks timezone. If a string, *tz* is passed to `dateutil.tz`.

**class** matplotlib.dates.**MicrosecondLocator**(*interval=1*, *tz=None*)

Bases: *DateLocator*

Make ticks on regular intervals of one or more microsecond(s).

---

**Note:** By default, Matplotlib uses a floating point representation of time in days since the epoch, so plotting data with microsecond time resolution does not work well for dates that are far (about 70 years) from the epoch (check with *get_epoch*).

If you want sub-microsecond resolution time plots, it is strongly recommended to use floating point seconds, not datetime-like time representation.

If you really must use datetime.datetime() or similar and still need microsecond precision, change the time origin via *dates.set_epoch* to something closer to the dates being plotted. See *Date Precision and Epochs*.

---

**Parameters**

**interval**

[int, default: 1] The interval between each iteration. For example, if `interval=2`, mark every second occurrence.

**tz**

[str or `tzinfo`, default: *rcParams["timezone"]* (default: `'UTC'`)] Ticks timezone. If a string, *tz* is passed to `dateutil.tz`.

**set_axis**(*axis*)

**tick_values**(*vmin*, *vmax*)

Return the values of the located ticks given **vmin** and **vmax**.

---

**Note:** To get tick locations with the vmin and vmax values defined automatically for the associated `axis` simply call the Locator instance:

```
>>> print(type(loc))
<type 'Locator'>
>>> print(loc())
[1, 2, 3, 4]
```

---

**class** matplotlib.dates.**MinuteLocator**(*byminute=None*, *interval=1*, *tz=None*)

Bases: *RRuleLocator*

Make ticks on occurrences of each minute.

**Parameters**

**byminute**

[int or list of int, default: all minutes] Ticks will be placed on every minute in *byminute*. Default is `byminute=range(60)`, i.e., every minute.

**interval**

[int, default: 1] The interval between each iteration. For example, if `interval=2`, mark every second occurrence.

**tz**

[str or `tzinfo`, default: *rcParams["timezone"]* (default: `'UTC'`)] Ticks timezone. If a string, *tz* is passed to `dateutil.tz`.

**class** matplotlib.dates.**MonthLocator**(*bymonth=None*, *bymonthday=1*, *interval=1*, *tz=None*)

Bases: *RRuleLocator*

Make ticks on occurrences of each month, e.g., 1, 3, 12.

**Parameters**

**bymonth**

[int or list of int, default: all months] Ticks will be placed on every month in *bymonth*. Default is `range(1, 13)`, i.e. every month.

**bymonthday**

[int, default: 1] The day on which to place the ticks.

**interval**

[int, default: 1] The interval between each iteration. For example, if `interval=2`, mark every second occurrence.

**tz**

[str or `tzinfo`, default: *rcParams["timezone"]* (default: `'UTC'`)] Ticks timezone. If a string, *tz* is passed to `dateutil.tz`.

**class** matplotlib.dates.**RRuleLocator**(*o*, *tz=None*)

Bases: *DateLocator*

**Parameters**

**tz**

[str or tzinfo, default: *rcParams["timezone"]* (default: 'UTC')] Ticks timezone. If a string, *tz* is passed to dateutil.tz.

**static get_unit_generic**(*freq*)

**tick_values**(*vmin*, *vmax*)

Return the values of the located ticks given **vmin** and **vmax**.

---

**Note:** To get tick locations with the vmin and vmax values defined automatically for the associated axis simply call the Locator instance:

```
>>> print(type(loc))
<type 'Locator'>
>>> print(loc())
[1, 2, 3, 4]
```

---

**class** matplotlib.dates.**SecondLocator**(*bysecond=None*, *interval=1*, *tz=None*)

Bases: *RRuleLocator*

Make ticks on occurrences of each second.

**Parameters**

**bysecond**

[int or list of int, default: all seconds] Ticks will be placed on every second in *bysecond*. Default is bysecond = range(60), i.e., every second.

**interval**

[int, default: 1] The interval between each iteration. For example, if interval=2, mark every second occurrence.

**tz**

[str or tzinfo, default: *rcParams["timezone"]* (default: 'UTC')] Ticks timezone. If a string, *tz* is passed to dateutil.tz.

**class** matplotlib.dates.**WeekdayLocator**(*byweekday=1*, *interval=1*, *tz=None*)

Bases: *RRuleLocator*

Make ticks on occurrences of each weekday.

**Parameters**

**byweekday**

[int or list of int, default: all days] Ticks will be placed on every weekday in *byweekday*. Default is every day.

Elements of *byweekday* must be one of MO, TU, WE, TH, FR, SA, SU, the constants from `dateutil.rrule`, which have been imported into the *matplotlib.dates* namespace.

**interval**

[int, default: 1] The interval between each iteration. For example, if `interval=2`, mark every second occurrence.

**tz**

[str or `tzinfo`, default: *rcParams["timezone"]* (default: `'UTC'`)] Ticks timezone. If a string, *tz* is passed to `dateutil.tz`.

**class** matplotlib.dates.**YearLocator**(*base=1*, *month=1*, *day=1*, *tz=None*)

Bases: *RRuleLocator*

Make ticks on a given day of each year that is a multiple of base.

Examples:

```
# Tick every year on Jan 1st
locator = YearLocator()

# Tick every 5 years on July 4th
locator = YearLocator(5, month=7, day=4)
```

**Parameters**

**base**

[int, default: 1] Mark ticks every *base* years.

**month**

[int, default: 1] The month on which to place the ticks, starting from 1. Default is January.

**day**

[int, default: 1] The day on which to place the ticks.

**tz**

[str or `tzinfo`, default: *rcParams["timezone"]* (default: `'UTC'`)] Ticks timezone. If a string, *tz* is passed to `dateutil.tz`.

matplotlib.dates.**date2num**(*d*)

Convert datetime objects to Matplotlib dates.

**Parameters**

**d**

[datetime.datetime or numpy.datetime64 or sequences of these]

**Returns**

**float or sequence of floats**

Number of days since the epoch. See *get_epoch* for the epoch, which can be changed by *rcParams["date.epoch"]* (default: '1970-01-01T00:00:00') or *set_epoch*. If the epoch is "1970-01-01T00:00:00" (default) then noon Jan 1 1970 ("1970-01-01T12:00:00") returns 0.5.

## Notes

The Gregorian calendar is assumed; this is not universal practice. For details see the module docstring.

matplotlib.dates.**datestr2num**(*d*, *default=None*)

Convert a date string to a datenum using dateutil.parser.parse.

**Parameters**

**d**

[str or sequence of str] The dates to convert.

**default**

[datetime.datetime, optional] The default date to use when fields are missing in *d*.

matplotlib.dates.**drange**(*dstart*, *dend*, *delta*)

Return a sequence of equally spaced Matplotlib dates.

The dates start at *dstart* and reach up to, but not including *dend*. They are spaced by *delta*.

**Parameters**

**dstart, dend**

[datetime] The date limits.

**delta**

[datetime.timedelta] Spacing of the dates.

**Returns**

**numpy.array**

A list floats representing Matplotlib dates.

matplotlib.dates.**get_epoch**()

Get the epoch used by *dates*.

**Returns**

**epoch**

[str] String for the epoch (parsable by numpy.datetime64).

matplotlib.dates.**num2date**(*x*, *tz=None*)

Convert Matplotlib dates to datetime objects.

**Parameters**

**x**

[float or sequence of floats] Number of days (fraction part represents hours, minutes, seconds) since the epoch. See *get_epoch* for the epoch, which can be changed by *rcParams["date.epoch"]* (default: '1970-01-01T00:00:00') or *set_epoch*.

**tz**

[str or tzinfo, default: *rcParams["timezone"]* (default: 'UTC')] Timezone of *x*. If a string, *tz* is passed to dateutil.tz.

**Returns**

**datetime or sequence of datetime**

Dates are returned in timezone *tz*.

If *x* is a sequence, a sequence of datetime objects will be returned.

## Notes

The Gregorian calendar is assumed; this is not universal practice. For details, see the module docstring.

matplotlib.dates.**num2timedelta**(*x*)

Convert number of days to a timedelta object.

If *x* is a sequence, a sequence of timedelta objects will be returned.

**Parameters**

**x**

[float, sequence of floats] Number of days. The fraction part represents hours, minutes, seconds.

**Returns**

**datetime.timedelta or list[datetime.timedelta]**

**class** matplotlib.dates.**relativedelta**(*dt1=None*, *dt2=None*, *years=0*, *months=0*, *days=0*, *leapdays=0*, *weeks=0*, *hours=0*, *minutes=0*, *seconds=0*, *microseconds=0*, *year=None*, *month=None*, *day=None*, *weekday=None*, *yearday=None*, *nlyearday=None*, *hour=None*, *minute=None*, *second=None*, *microsecond=None*)

Bases: object

The relativedelta type is designed to be applied to an existing datetime and can replace specific components of that datetime, or represents an interval of time.

It is based on the specification of the excellent work done by M.-A. Lemburg in his mx.DateTime extension. However, notice that this type does *NOT* implement the same algorithm as his work. Do *NOT* expect it to behave like mx.DateTime's counterpart.

There are two different ways to build a relativedelta instance. The first one is passing it two date/datetime classes:

```
relativedelta(datetime1, datetime2)
```

The second one is passing it any number of the following keyword arguments:

```
relativedelta(arg1=x,arg2=y,arg3=z...)

year, month, day, hour, minute, second, microsecond:
    Absolute information (argument is singular); adding or subtracting a
    relativedelta with absolute information does not perform an␣
 ↪arithmetic
    operation, but rather REPLACES the corresponding value in the
    original datetime with the value(s) in relativedelta.

years, months, weeks, days, hours, minutes, seconds, microseconds:
    Relative information, may be negative (argument is plural); adding
    or subtracting a relativedelta with relative information performs
    the corresponding arithmetic operation on the original datetime value
    with the information in the relativedelta.

weekday:
    One of the weekday instances (MO, TU, etc) available in the
    relativedelta module. These instances may receive a parameter N,
    specifying the Nth weekday, which could be positive or negative
    (like MO(+1) or MO(-2)). Not specifying it is the same as specifying
    +1. You can also use an integer, where 0=MO. This argument is always
    relative e.g. if the calculated date is already Monday, using MO(1)
    or MO(-1) won't change the day. To effectively make it absolute, use
    it in combination with the day argument (e.g. day=1, MO(1) for first
    Monday of the month).

leapdays:
    Will add given days to the date found, if year is a leap
```

(continues on next page)

(continued from previous page)

```
    year, and the date found is post 28 of february.

yearday, nlyearday:
    Set the yearday or the non-leap year day (jump leap days).
    These are converted to day/month/leapdays information.
```

There are relative and absolute forms of the keyword arguments. The plural is relative, and the singular is absolute. For each argument in the order below, the absolute form is applied first (by setting each attribute to that value) and then the relative form (by adding the value to the attribute).

The order of attributes considered when this relativedelta is added to a datetime is:

1. Year
2. Month
3. Day
4. Hours
5. Minutes
6. Seconds
7. Microseconds

Finally, weekday is applied, using the rule described above.

For example

```
>>> from datetime import datetime
>>> from dateutil.relativedelta import relativedelta, MO
>>> dt = datetime(2018, 4, 9, 13, 37, 0)
>>> delta = relativedelta(hours=25, day=1, weekday=MO(1))
>>> dt + delta
datetime.datetime(2018, 4, 2, 14, 37)
```

First, the day is set to 1 (the first of the month), then 25 hours are added, to get to the 2nd day and 14th hour, finally the weekday is applied, but since the 2nd is already a Monday there is no effect.

**normalized**()

Return a version of this object represented entirely using integer values for the relative attributes.

```
>>> relativedelta(days=1.5, hours=2).normalized()
relativedelta(days=+1, hours=+14)
```

**Returns**

Returns a *dateutil.relativedelta.relativedelta* object.

**property weeks**

**class** matplotlib.dates.**rrulewrapper**(*freq*, *tzinfo=None*, ***kwargs*)

Bases: object

A simple wrapper around a dateutil.rrule allowing flexible date tick specifications.

**Parameters**

**freq**

[{YEARLY, MONTHLY, WEEKLY, DAILY, HOURLY, MINUTELY, SECONDLY}] Tick frequency. These constants are defined in dateutil.rrule, but they are accessible from *matplotlib.dates* as well.

**tzinfo**

[datetime.tzinfo, optional] Time zone information. The default is None.

****kwargs**

Additional keyword arguments are passed to the dateutil.rrule.

**set**(***kwargs*)

Set parameters for an existing wrapper.

matplotlib.dates.**set_epoch**(*epoch*)

Set the epoch (origin for dates) for datetime calculations.

The default epoch is *rcParams["dates.epoch"]* (by default 1970-01-01T00:00).

If microsecond accuracy is desired, the date being plotted needs to be within approximately 70 years of the epoch. Matplotlib internally represents dates as days since the epoch, so floating point dynamic range needs to be within a factor of 2^52.

*set_epoch* must be called before any dates are converted (i.e. near the import section) or a RuntimeError will be raised.

See also *Date Precision and Epochs*.

**Parameters**

**epoch**

[str] valid UTC date parsable by numpy.datetime64 (do not include timezone).

### 7.2.21 matplotlib.docstring

**Attention:** This module is considered internal.

Its use is deprecated and it will be removed in a future version.

**class** matplotlib._docstring.**Substitution**(**args*, ***kwargs*)

Bases: object

A decorator that performs %-substitution on an object's docstring.

This decorator should be robust even if obj.__doc__ is None (for example, if -OO was passed to the interpreter).

Usage: construct a docstring.Substitution with a sequence or dictionary suitable for performing substitution; then decorate a suitable function with the constructed object, e.g.:

```
sub_author_name = Substitution(author='Jason')

@sub_author_name
def some_function(x):
    "%(author)s wrote this function"

# note that some_function.__doc__ is now "Jason wrote this function"
```

One can also use positional arguments:

```
sub_first_last_names = Substitution('Edgar Allen', 'Poe')

@sub_first_last_names
def some_function(x):
    "%s %s wrote the Raven"
```

**update**(**args*, ***kwargs*)

Update self.params (which must be a dict) with the supplied args.

matplotlib._docstring.**copy**(*source*)

Copy a docstring from another source function (if present).

matplotlib._docstring.**kwarg_doc**(*text*)

Decorator for defining the kwdoc documentation of artist properties.

This decorator can be applied to artist property setter methods. The given text is stored in a private attribute _kwarg_doc on the method. It is used to overwrite auto-generated documentation in the *kwdoc list* for artists. The kwdoc list is used to document **kwargs when they are properties of an artist. See e.g. the **kwargs section in *Axes.text*.

The text should contain the supported types, as well as the default value if applicable, e.g.:

@_docstring.kwarg_doc("bool, default: *rcParams["text.usetex"]* (default: False)") def set_usetex(self, usetex):

**See also:**

*matplotlib.artist.kwdoc*

### 7.2.22 matplotlib.dviread

A module for reading dvi files output by TeX. Several limitations make this not (currently) useful as a general-purpose dvi preprocessor, but it is currently used by the pdf backend for processing usetex text.

Interface:

```
with Dvi(filename, 72) as dvi:
    # iterate over pages:
    for page in dvi:
        w, h, d = page.width, page.height, page.descent
        for x, y, font, glyph, width in page.text:
            fontname = font.texname
            pointsize = font.size
            ...
        for x, y, height, width in page.boxes:
            ...
```

**class** matplotlib.dviread.**Dvi**(*filename*, *dpi*)

Bases: object

A reader for a dvi ("device-independent") file, as produced by TeX.

The current implementation can only iterate through pages in order, and does not even attempt to verify the postamble.

This class can be used as a context manager to close the underlying file upon exit. Pages can be read via iteration. Here is an overly simple way to extract text without trying to detect whitespace:

```
>>> with matplotlib.dviread.Dvi('input.dvi', 72) as dvi:
...     for page in dvi:
...         print(''.join(chr(t.glyph) for t in page.text))
```

Read the data from the file named *filename* and convert TeX's internal units to units of *dpi* per inch. *dpi* only sets the units and does not limit the resolution. Use None to return TeX's internal units.

**close**()

Close the underlying file if it is open.

**class** matplotlib.dviread.**DviFont**(*scale*, *tfm*, *texname*, *vf*)

Bases: object

Encapsulation of a font that a DVI file can refer to.

This class holds a font's texname and size, supports comparison, and knows the widths of glyphs in the same units as the AFM file. There are also internal attributes (for use by dviread.py) that are *not* used for comparison.

The size is in Adobe points (converted from TeX points).

**Parameters**

**scale**

[float] Factor by which the font is scaled from its natural size.

**tfm**

[Tfm] TeX font metrics for this font

**texname**

[bytes] Name of the font as used internally by TeX and friends, as an ASCII bytestring. This is usually very different from any external font names; *PsfontsMap* can be used to find the external name of the font.

**vf**

[Vf] A TeX "virtual font" file, or None if this font is not virtual.

**Attributes**

**texname**

[bytes]

**size**

[float] Size of the font in Adobe points, converted from the slightly smaller TeX points.

**widths**

[list] Widths of glyphs in glyph-space units, typically 1/1000ths of the point size.

**size**

**texname**

**widths**

**class** matplotlib.dviread.**PsFont**(*texname*, *psname*, *effects*, *encoding*, *filename*)

Bases: tuple

Create new instance of PsFont(texname, psname, effects, encoding, filename)

**effects**

Alias for field number 2

**encoding**

Alias for field number 3

**filename**

Alias for field number 4

**psname**

Alias for field number 1

**texname**

Alias for field number 0

**class** matplotlib.dviread.**PsfontsMap**(*filename*)

Bases: object

A psfonts.map formatted file, mapping TeX fonts to PS fonts.

**Parameters**

**filename**

[str or path-like]

### Notes

For historical reasons, TeX knows many Type-1 fonts by different names than the outside world. (For one thing, the names have to fit in eight characters.) Also, TeX's native fonts are not Type-1 but Metafont, which is nontrivial to convert to PostScript except as a bitmap. While high-quality conversions to Type-1 format exist and are shipped with modern TeX distributions, we need to know which Type-1 fonts are the counterparts of which native fonts. For these reasons a mapping is needed from internal font names to font file names.

A texmf tree typically includes mapping files called e.g. `psfonts.map`, `pdftex.map`, or `dvipdfm.map`. The file `psfonts.map` is used by **dvips**, `pdftex.map` by **pdfTeX**, and `dvipdfm.map` by **dvipdfm**. `psfonts.map` might avoid embedding the 35 PostScript fonts (i.e., have no filename for them, as in the Times-Bold example above), while the pdf-related files perhaps only avoid the "Base 14" pdf fonts. But the user may have configured these files differently.

### Examples

```
>>> map = PsfontsMap(find_tex_file('pdftex.map'))
>>> entry = map[b'ptmbo8r']
>>> entry.texname
b'ptmbo8r'
>>> entry.psname
b'Times-Bold'
>>> entry.encoding
'/usr/local/texlive/2008/texmf-dist/fonts/enc/dvips/base/8r.enc'
>>> entry.effects
{'slant': 0.16700000000000001}
>>> entry.filename
```

**class** matplotlib.dviread.**Tfm**(*filename*)

Bases: object

A TeX Font Metric file.

This implementation covers only the bare minimum needed by the Dvi class.

**Parameters**

**filename**

[str or path-like]

**Attributes**

**checksum**

[int] Used for verifying against the dvi file.

**design_size**

[int] Design size of the font (unknown units)

**width, height, depth**

[dict] Dimensions of each character, need to be scaled by the factor specified in the dvi file. These are dicts because indexing may not start from 0.

**checksum**

**depth**

**design_size**

**height**

**width**

**class** matplotlib.dviread.**Vf**(*filename*)

Bases: *Dvi*

A virtual font (*.vf file) containing subroutines for dvi files.

**Parameters**

**filename**

[str or path-like]

## Notes

The virtual font format is a derivative of dvi: http://mirrors.ctan.org/info/knuth/virtual-fonts This class reuses some of the machinery of *Dvi* but replaces the `_read` loop and dispatch mechanism.

#### Examples

```
vf = Vf(filename)
glyph = vf[code]
glyph.text, glyph.boxes, glyph.width
```

Read the data from the file named *filename* and convert TeX's internal units to units of *dpi* per inch. *dpi* only sets the units and does not limit the resolution. Use None to return TeX's internal units.

matplotlib.dviread.**find_tex_file**(*filename*)

Find a file in the texmf tree using kpathsea.

The kpathsea library, provided by most existing TeX distributions, both on Unix-like systems and on Windows (MikTeX), is invoked via a long-lived luatex process if luatex is installed, or via kpsewhich otherwise.

**Parameters**

**filename**

[str or path-like]

**Raises**

**FileNotFoundError**

If the file is not found.

### 7.2.23 matplotlib.figure

*matplotlib.figure* implements the following classes:

***Figure***

Top level *Artist*, which holds all plot elements. Many methods are implemented in *FigureBase*.

***SubFigure***

A logical figure inside a figure, usually added to a figure (or parent *SubFigure*) with *Figure.add_subfigure* or *Figure.subfigures* methods (provisional API v3.4).

Figures are typically created using pyplot methods *figure*, *subplots*, and *subplot_mosaic*.

```
fig, ax = plt.subplots(figsize=(2, 2), facecolor='lightskyblue',
                       layout='constrained')
fig.suptitle('Figure')
ax.set_title('Axes', loc='left', fontstyle='oblique', fontsize='medium')
```

Some situations call for directly instantiating a *Figure* class, usually inside an application of some sort (see *Embedding Matplotlib in graphical user interfaces* for a list of examples) . More information about Figures can be found at *Introduction to Figures*.

## Figure

## Figure class

| | |
|---|---|
| *Figure* | The top level container for all the plot elements. |

### matplotlib.figure.Figure

**class** matplotlib.figure.**Figure**(*figsize=None*, *dpi=None*, *, *facecolor=None*, *edgecolor=None*, *linewidth=0.0*, *frameon=None*, *subplotpars=None*, *tight_layout=None*, *constrained_layout=None*, *layout=None*, ***kwargs*)

The top level container for all the plot elements.

See *matplotlib.figure* for an index of class methods.

**Attributes**

**patch**

The *Rectangle* instance representing the figure background patch.

**suppressComposite**

For multiple images, the figure will make composite images depending on the renderer option_image_nocomposite function. If *suppressComposite* is a boolean, this will override the renderer.

**Parameters**

**figsize**

[2-tuple of floats, default: *rcParams["figure.figsize"]* (default: `[6.4, 4.8]`)] Figure dimension `(width, height)` in inches.

**dpi**

[float, default: *rcParams["figure.dpi"]* (default: `100.0`)] Dots per inch.

**facecolor**

[default: *rcParams["figure.facecolor"]* (default: `'white'`)] The figure patch facecolor.

**edgecolor**

[default: *rcParams["figure.edgecolor"]* (default: `'white'`)] The figure patch edge color.

**linewidth**

[float] The linewidth of the frame (i.e. the edge linewidth of the figure patch).

**frameon**

[bool, default: *rcParams["figure.frameon"]* (default: `True`)] If `False`, suppress drawing the figure background patch.

**subplotpars**

[*SubplotParams*] Subplot parameters. If not given, the default subplot parameters *rcParams["figure.subplot.*"]* are used.

**tight_layout**

[bool or dict, default: *rcParams["figure.autolayout"]* (default: `False`)] Whether to use the tight layout mechanism. See *set_tight_layout*.

---

**Discouraged**

The use of this parameter is discouraged. Please use `layout='tight'` instead for the common case of `tight_layout=True` and use *set_tight_layout* otherwise.

---

**constrained_layout**

[bool, default: *rcParams["figure.constrained_layout.use"]* (default: `False`)] This is equal to `layout='constrained'`.

---

**Discouraged**

The use of this parameter is discouraged. Please use `layout='constrained'` instead.

---

**layout**

[{'constrained', 'compressed', 'tight', 'none', *LayoutEngine*, None}, default: None] The layout mechanism for positioning of plot elements to avoid overlapping Axes decorations (labels, ticks, etc). Note that layout managers can have significant performance penalties.

- 'constrained': The constrained layout solver adjusts Axes sizes to avoid overlapping Axes decorations. Can handle complex plot layouts and colorbars, and is thus recommended.

  See *Constrained layout guide* for examples.

- 'compressed': uses the same algorithm as 'constrained', but removes extra space between fixed-aspect-ratio Axes. Best for simple grids of Axes.

- 'tight': Use the tight layout mechanism. This is a relatively simple algorithm that adjusts the subplot parameters so that decorations do not overlap.

  See *Tight layout guide* for examples.

- 'none': Do not use a layout engine.

- A *LayoutEngine* instance. Builtin layout classes are *ConstrainedLayoutEngine* and *TightLayoutEngine*, more easily accessible by 'constrained' and 'tight'. Passing an instance allows third parties to provide their own layout engine.

If not given, fall back to using the parameters *tight_layout* and *constrained_layout*, including their config defaults *rcParams["figure.autolayout"]* (default: `False`) and *rcParams["figure.constrained_layout.use"]* (default: `False`).

### Other Parameters

#### **kwargs

[*Figure* properties, optional]

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi valu |
| *alpha* | scalar or None |
| *animated* | bool |
| *canvas* | FigureCanvas |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *constrained_layout* | unknown |
| *constrained_layout_pads* | unknown |
| *dpi* | float |
| *edgecolor* | *color* |
| *facecolor* | *color* |
| *figheight* | float |
| *figure* | *Figure* |
| *figwidth* | float |
| *frameon* | bool |
| *gid* | str |
| *in_layout* | bool |
| *label* | object |
| *layout_engine* | {'constrained', 'compressed', 'tight', 'none', *LayoutEngine*, No |
| *linewidth* | number |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |

Table 49 – continued from

| Property | Description |
|---|---|
| *rasterized* | bool |
| *size_inches* | (float, float) or float |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *tight_layout* | unknown |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *zorder* | float |

### Adding Axes and SubFigures

| | |
|---|---|
| *Figure.add_axes* | Add an *Axes* to the figure. |
| *Figure.add_subplot* | Add an *Axes* to the figure as part of a subplot arrangement. |
| *Figure.subplots* | Add a set of subplots to this figure. |
| *Figure.subplot_mosaic* | Build a layout of Axes based on ASCII art or nested lists. |
| *Figure.add_gridspec* | Low-level API for creating a *GridSpec* that has this figure as a parent. |
| *Figure.get_axes* | List of Axes in the Figure. |
| *Figure.axes* | List of Axes in the Figure. |
| *Figure.delaxes* | Remove the *Axes* *ax* from the figure; update the current Axes. |
| *Figure.subfigures* | Add a set of subfigures to this figure or subfigure. |
| *Figure.add_subfigure* | Add a *SubFigure* to the figure as part of a subplot arrangement. |

### matplotlib.figure.Figure.add_axes

Figure.**add_axes**(**args*, ***kwargs*)

Add an *Axes* to the figure.

Call signatures:

```
add_axes(rect, projection=None, polar=False, **kwargs)
add_axes(ax)
```

**Parameters**

**rect**

[tuple (left, bottom, width, height)] The dimensions (left, bottom, width, height) of the new *Axes*. All quantities are in fractions of figure width and height.

**projection**

[{None, 'aitoff', 'hammer', 'lambert', 'mollweide', 'polar', 'rectilinear', str}, optional] The projection type of the *Axes*. *str* is the name of a custom projection, see *projections*. The default None results in a 'rectilinear' projection.

**polar**

[bool, default: False] If True, equivalent to projection='polar'.

**axes_class**

[subclass type of *Axes*, optional] The *axes.Axes* subclass that is instantiated. This parameter is incompatible with *projection* and *polar*. See *axisartist* for examples.

**sharex, sharey**

[*Axes*, optional] Share the x or y *axis* with sharex and/or sharey. The axis will have the same limits, ticks, and scale as the axis of the shared Axes.

**label**

[str] A label for the returned Axes.

### Returns

***Axes*, or a subclass of *Axes***

The returned Axes class depends on the projection used. It is *Axes* if rectilinear projection is used and *projections.polar.PolarAxes* if polar projection is used.

### Other Parameters

****kwargs**

This method also takes the keyword arguments for the returned Axes class. The keyword arguments for the rectilinear Axes class *Axes* can be found in the following table but there might also be other keyword arguments if another projection is used, see the actual Axes class.

| Property | Description |
|---|---|
| *adjustable* | {'box', 'datalim'} |
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi v |
| *alpha* | scalar or None |
| *anchor* | (float, float) or {'C', 'SW', 'S', 'SE', 'E', 'NE', ...} |
| *animated* | bool |
| *aspect* | {'auto', 'equal'} or float |

Table 50 – continued from

| Property | Description |
|---|---|
| *autoscale_on* | bool |
| *autoscalex_on* | unknown |
| *autoscaley_on* | unknown |
| *axes_locator* | Callable[[Axes, Renderer], Bbox] |
| *axisbelow* | bool or 'line' |
| *box_aspect* | float or None |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *facecolor* or fc | *color* |
| *figure* | *Figure* |
| *forward_navigation_events* | bool or "auto" |
| *frame_on* | bool |
| *gid* | str |
| *in_layout* | bool |
| *label* | object |
| *mouseover* | bool |
| *navigate* | bool |
| *navigate_mode* | unknown |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *position* | [left, bottom, width, height] or *Bbox* |
| *prop_cycle* | *Cycler* |
| *rasterization_zorder* | float or None |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *subplotspec* | unknown |
| *title* | str |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *xbound* | (lower: float, upper: float) |
| *xlabel* | str |
| *xlim* | (left: float, right: float) |
| *xmargin* | float greater than -0.5 |
| *xscale* | unknown |
| *xticklabels* | unknown |
| *xticks* | unknown |
| *ybound* | (lower: float, upper: float) |
| *ylabel* | str |
| *ylim* | (bottom: float, top: float) |
| *ymargin* | float greater than -0.5 |
| *yscale* | unknown |

Table 50 – continued from

| Property | Description |
|---|---|
| *yticklabels* | unknown |
| *yticks* | unknown |
| *zorder* | float |

**See also:**

***Figure.add_subplot***
***pyplot.subplot***
***pyplot.axes***
***Figure.subplots***
***pyplot.subplots***

### Notes

In rare circumstances, *add_axes* may be called with a single argument, an Axes instance already created in the present figure but not in the figure's list of Axes.

### Examples

Some simple examples:

```
rect = l, b, w, h
fig = plt.figure()
fig.add_axes(rect)
fig.add_axes(rect, frameon=False, facecolor='g')
fig.add_axes(rect, polar=True)
ax = fig.add_axes(rect, projection='polar')
fig.delaxes(ax)
fig.add_axes(ax)
```

## Examples using `matplotlib.figure.Figure.add_axes`

- *Barcode*
- *Axes Demo*
- *Axes with a fixed physical size*
- *Make room for ylabel using axes_grid*
- *Simple Axes Divider 1*
- *Simple axes divider 3*
- *Parasite Axes demo*

- *Anatomy of a figure*
- *Firefox*
- *Shaded & power normalized rendering*
- *Pan/zoom events of overlapping axes*
- *XKCD*
- *Rain simulation*
- *SVG Filter Line*
- *SVG filter pie*
- *Fig Axes Customize Simple*
- *Buttons*
- *Thresholding an Image with RangeSlider*
- *Slider*
- *Snapping Sliders to Discrete Values*
- *Textbox*
- *Artist tutorial*
- *Arranging multiple Axes in a Figure*
- *Specifying colors*
- *Text properties and layout*

## matplotlib.figure.Figure.add_subplot

`Figure.`**`add_subplot`**`(*args, **kwargs)`

Add an `Axes` to the figure as part of a subplot arrangement.

Call signatures:

```
add_subplot(nrows, ncols, index, **kwargs)
add_subplot(pos, **kwargs)
add_subplot(ax)
add_subplot()
```

**Parameters**

***args**

[int, (int, int, *index*), or `SubplotSpec`, default: (1, 1, 1)] The position of the subplot described by one of

- Three integers (*nrows*, *ncols*, *index*). The subplot will take the *index* position on a grid with *nrows* rows and *ncols* columns. *index* starts at 1 in the upper left corner and increases to the right. *index* can also be a two-tuple specifying the (*first*, *last*) indices (1-based, and including *last*) of the subplot, e.g., `fig.add_subplot(3, 1, (1, 2))` makes a subplot that spans the upper 2/3 of the figure.
- A 3-digit integer. The digits are interpreted as if given separately as three single-digit integers, i.e. `fig.add_subplot(235)` is the same as `fig.add_subplot(2, 3, 5)`. Note that this can only be used if there are no more than 9 subplots.
- A *SubplotSpec*.

In rare circumstances, *add_subplot* may be called with a single argument, a subplot Axes instance already created in the present figure but not in the figure's list of Axes.

**projection**

[{None, 'aitoff', 'hammer', 'lambert', 'mollweide', 'polar', 'rectilinear', str}, optional] The projection type of the subplot (*Axes*). *str* is the name of a custom projection, see *projections*. The default None results in a 'rectilinear' projection.

**polar**

[bool, default: False] If True, equivalent to projection='polar'.

**axes_class**

[subclass type of *Axes*, optional] The *axes.Axes* subclass that is instantiated. This parameter is incompatible with *projection* and *polar*. See *axisartist* for examples.

**sharex, sharey**

[*Axes*, optional] Share the x or y *axis* with sharex and/or sharey. The axis will have the same limits, ticks, and scale as the axis of the shared Axes.

**label**

[str] A label for the returned Axes.

**Returns**

***Axes***

The Axes of the subplot. The returned Axes can actually be an instance of a subclass, such as *projections.polar.PolarAxes* for polar projections.

**Other Parameters**

****kwargs**

This method also takes the keyword arguments for the returned Axes base class; except for the *figure* argument. The keyword arguments for the rectilinear base class *Axes* can be found in the following table but there might also be other keyword arguments if another projection is used.

| Property | Description |
|---|---|
| *adjustable* | {'box', 'datalim'} |
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi v |
| *alpha* | scalar or None |
| *anchor* | (float, float) or {'C', 'SW', 'S', 'SE', 'E', 'NE', ...} |
| *animated* | bool |
| *aspect* | {'auto', 'equal'} or float |
| *autoscale_on* | bool |
| *autoscalex_on* | unknown |
| *autoscaley_on* | unknown |
| *axes_locator* | Callable[[Axes, Renderer], Bbox] |
| *axisbelow* | bool or 'line' |
| *box_aspect* | float or None |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *facecolor* or fc | *color* |
| *figure* | *Figure* |
| *forward_navigation_events* | bool or "auto" |
| *frame_on* | bool |
| *gid* | str |
| *in_layout* | bool |
| *label* | object |
| *mouseover* | bool |
| *navigate* | bool |
| *navigate_mode* | unknown |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *position* | [left, bottom, width, height] or *Bbox* |
| *prop_cycle* | *Cycler* |
| *rasterization_zorder* | float or None |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *subplotspec* | unknown |
| *title* | str |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *xbound* | (lower: float, upper: float) |
| *xlabel* | str |

Table 51 – continued from

| Property | Description |
|---|---|
| *xlim* | (left: float, right: float) |
| *xmargin* | float greater than -0.5 |
| *xscale* | unknown |
| *xticklabels* | unknown |
| *xticks* | unknown |
| *ybound* | (lower: float, upper: float) |
| *ylabel* | str |
| *ylim* | (bottom: float, top: float) |
| *ymargin* | float greater than -0.5 |
| *yscale* | unknown |
| *yticklabels* | unknown |
| *yticks* | unknown |
| *zorder* | float |

**See also:**

***Figure.add_axes***
***pyplot.subplot***
***pyplot.axes***
***Figure.subplots***
***pyplot.subplots***

### Examples

```
fig = plt.figure()

fig.add_subplot(231)
ax1 = fig.add_subplot(2, 3, 1)  # equivalent but more general

fig.add_subplot(232, frameon=False)  # subplot with no frame
fig.add_subplot(233, projection='polar')  # polar subplot
fig.add_subplot(234, sharex=ax1)  # subplot sharing x-axis with ax1
fig.add_subplot(235, facecolor="red")  # red subplot

ax1.remove()  # delete ax1 from the figure
fig.add_subplot(ax1)  # add ax1 back to the figure
```

### Examples using `matplotlib.figure.Figure.add_subplot`

- *EventCollection Demo*
- *Scatter plot with histograms*
- *Resizing Axes with constrained layout*
- *Resizing Axes with tight layout*
- *Combining two subplots using subplots and GridSpec*
- *Using Gridspec to make multi-column/row subplot layouts*
- *Nested Gridspecs*
- *Figure subfigures*
- *Error bar rendering on polar axis*
- *Polar legend*
- *Scatter plot on polar axis*
- *Annotation Polar*
- *Text Commands*
- *Inset locator demo*
- *Parasite Simple2*
- *Axis Direction*
- *axis_direction demo*
- *Axis line styles*
- *Curvilinear grid demo*
- *Demo CurveLinear Grid2*
- *floating_axes features*
- *floating_axis demo*
- *Ticklabel alignment*
- *Ticklabel direction*
- *Simple axis direction*
- *Simple axis tick label and tick directions*
- *Simple Axis Pad*
- *Custom spines with axisartist*
- *Simple Axisline*
- *Simple Axisline3*

- *Pan/zoom events of overlapping axes*
- *The double pendulum problem*
- *Animated 3D random walk*
- *Demo of 3D bar charts*
- *Create 2D bar graphs in different planes*
- *3D box surface plot*
- *Create 3D histogram of 2D data*
- *2D and 3D Axes in same figure*
- *Draw flat objects in 3D plot*
- *3D scatterplot*
- *3D plots as subplots*
- *3D surface (solid color)*
- *3D surface with polar coordinates*
- *More triangular 3D surfaces*
- *3D wireframe plot*
- *The Sankey class*
- *Long chain of connections using Sankey*
- *Rankine power cycle*
- *SkewT-logP diagram: using transforms and custom projections*
- *Ellipse with units*
- *CanvasAgg demo*
- *GridSpec demo*
- *Simple Legend01*
- *Artist tutorial*
- *Transformations Tutorial*
- *Constrained layout guide*
- *Tight layout guide*
- *Arranging multiple Axes in a Figure*
- *Text in Matplotlib*
- *Annotations*

### matplotlib.figure.Figure.subplots

Figure.**subplots**(*nrows=1*, *ncols=1*, *, *sharex=False*, *sharey=False*, *squeeze=True*, *width_ratios=None*, *height_ratios=None*, *subplot_kw=None*, *gridspec_kw=None*)

Add a set of subplots to this figure.

This utility wrapper makes it convenient to create common layouts of subplots in a single call.

**Parameters**

**nrows, ncols**

[int, default: 1] Number of rows/columns of the subplot grid.

**sharex, sharey**

[bool or {'none', 'all', 'row', 'col'}, default: False] Controls sharing of x-axis (*sharex*) or y-axis (*sharey*):

- True or 'all': x- or y-axis will be shared among all subplots.
- False or 'none': each subplot x- or y-axis will be independent.
- 'row': each subplot row will share an x- or y-axis.
- 'col': each subplot column will share an x- or y-axis.

When subplots have a shared x-axis along a column, only the x tick labels of the bottom subplot are created. Similarly, when subplots have a shared y-axis along a row, only the y tick labels of the first column subplot are created. To later turn other subplots' ticklabels on, use *tick_params*.

When subplots have a shared axis that has units, calling *Axis.set_units* will update each axis with the new units.

**squeeze**

[bool, default: True]

- If True, extra dimensions are squeezed out from the returned array of Axes:
  - if only one subplot is constructed (nrows=ncols=1), the resulting single Axes object is returned as a scalar.
  - for Nx1 or 1xM subplots, the returned object is a 1D numpy object array of Axes objects.
  - for NxM, subplots with N>1 and M>1 are returned as a 2D array.
- If False, no squeezing at all is done: the returned Axes object is always a 2D array containing Axes instances, even if it ends up being 1x1.

**width_ratios**

[array-like of length *ncols*, optional] Defines the relative widths of the columns. Each column gets a relative width of `width_ratios[i] /`

`sum(width_ratios)`. If not given, all columns will have the same width. Equivalent to `gridspec_kw={'width_ratios': [...]}`.

**height_ratios**

[array-like of length *nrows*, optional] Defines the relative heights of the rows. Each row gets a relative height of `height_ratios[i] / sum(height_ratios)`. If not given, all rows will have the same height. Equivalent to `gridspec_kw={'height_ratios': [...]}`.

**subplot_kw**

[dict, optional] Dict with keywords passed to the *Figure.add_subplot* call used to create each subplot.

**gridspec_kw**

[dict, optional] Dict with keywords passed to the *GridSpec* constructor used to create the grid the subplots are placed on.

**Returns**

***Axes* or array of Axes**

Either a single *Axes* object or an array of Axes objects if more than one subplot was created. The dimensions of the resulting array can be controlled with the *squeeze* keyword, see above.

**See also:**

***pyplot.subplots***
***Figure.add_subplot***
***pyplot.subplot***

## Examples

```
# First create some toy data:
x = np.linspace(0, 2*np.pi, 400)
y = np.sin(x**2)

# Create a figure
fig = plt.figure()

# Create a subplot
ax = fig.subplots()
ax.plot(x, y)
ax.set_title('Simple plot')

# Create two subplots and unpack the output array immediately
ax1, ax2 = fig.subplots(1, 2, sharey=True)
ax1.plot(x, y)
```

(continues on next page)

(continued from previous page)

```
ax1.set_title('Sharing Y axis')
ax2.scatter(x, y)

# Create four polar Axes and access them through the returned array
axes = fig.subplots(2, 2, subplot_kw=dict(projection='polar'))
axes[0, 0].plot(x, y)
axes[1, 1].scatter(x, y)

# Share an X-axis with each column of subplots
fig.subplots(2, 2, sharex='col')

# Share a Y-axis with each row of subplots
fig.subplots(2, 2, sharey='row')

# Share both X- and Y-axes with all subplots
fig.subplots(2, 2, sharex='all', sharey='all')

# Note that this is the same as
fig.subplots(2, 2, sharex=True, sharey=True)
```

### Examples using `matplotlib.figure.Figure.subplots`

- *Plotting cumulative distributions*
- *Text Rotation Mode*
- *Asinh Demo*
- *Left ventricle bullseye*
- *Rectangle and ellipse selectors*
- *Complex and semantic figure composition (subplot_mosaic)*

## matplotlib.figure.Figure.subplot_mosaic

Figure.**subplot_mosaic**(*mosaic*, *, *sharex=False*, *sharey=False*, *width_ratios=None*, *height_ratios=None*, *empty_sentinel='.'*, *subplot_kw=None*, *per_subplot_kw=None*, *gridspec_kw=None*)

Build a layout of Axes based on ASCII art or nested lists.

This is a helper function to build complex GridSpec layouts visually.

See *Complex and semantic figure composition (subplot_mosaic)* for an example and full API documentation

**Parameters**

**mosaic**

[list of list of {hashable or nested} or str] A visual layout of how you want your Axes to be arranged labeled as strings. For example

```
x = [['A panel', 'A panel', 'edge'],
     ['C panel', '.',       'edge']]
```

produces 4 Axes:

- 'A panel' which is 1 row high and spans the first two columns
- 'edge' which is 2 rows high and is on the right edge
- 'C panel' which in 1 row and 1 column wide in the bottom left
- a blank space 1 row and 1 column wide in the bottom center

Any of the entries in the layout can be a list of lists of the same form to create nested layouts.

If input is a str, then it can either be a multi-line string of the form

```
'''
AAE
C.E
'''
```

where each character is a column and each line is a row. Or it can be a single-line string where rows are separated by `;`:

```
'AB;CC'
```

The string notation allows only single character Axes labels and does not support nesting but is very terse.

The Axes identifiers may be `str` or a non-iterable hashable object (e.g. `tuple` s may not be used).

**sharex, sharey**

[bool, default: False] If True, the x-axis (*sharex*) or y-axis (*sharey*) will be shared among all subplots. In that case, tick label visibility and axis units behave as for *`subplots`*. If False, each subplot's x- or y-axis will be independent.

**width_ratios**

[array-like of length *ncols*, optional] Defines the relative widths of the columns. Each column gets a relative width of `width_ratios[i] / sum(width_ratios)`. If not given, all columns will have the same width. Equivalent to `gridspec_kw={'width_ratios': [...]}`. In the case of nested layouts, this argument applies only to the outer layout.

**height_ratios**

[array-like of length *nrows*, optional] Defines the relative heights of the rows. Each row gets a relative height of `height_ratios[i] / sum(height_ratios)`. If not given, all rows will have the same height.

Equivalent to `gridspec_kw={'height_ratios': [...]}`. In the case of nested layouts, this argument applies only to the outer layout.

**subplot_kw**

[dict, optional] Dictionary with keywords passed to the *Figure.add_subplot* call used to create each subplot. These values may be overridden by values in *per_subplot_kw*.

**per_subplot_kw**

[dict, optional] A dictionary mapping the Axes identifiers or tuples of identifiers to a dictionary of keyword arguments to be passed to the *Figure.add_subplot* call used to create each subplot. The values in these dictionaries have precedence over the values in *subplot_kw*.

If *mosaic* is a string, and thus all keys are single characters, it is possible to use a single string instead of a tuple as keys; i.e. `"AB"` is equivalent to `("A", "B")`.

New in version 3.7.

**gridspec_kw**

[dict, optional] Dictionary with keywords passed to the *GridSpec* constructor used to create the grid the subplots are placed on. In the case of nested layouts, this argument applies only to the outer layout. For more complex layouts, users should use *Figure.subfigures* to create the nesting.

**empty_sentinel**

[object, optional] Entry in the layout to mean "leave this space empty". Defaults to `'.'`. Note, if *layout* is a string, it is processed via `inspect.cleandoc` to remove leading white space, which may interfere with using white-space as the empty sentinel.

**Returns**

**dict[label, Axes]**

A dictionary mapping the labels to the Axes objects. The order of the Axes is left-to-right and top-to-bottom of their position in the total layout.

## Examples using `matplotlib.figure.Figure.subplot_mosaic`

- *Spectrum representations*
- *Arrow Demo*
- *Labelling subplots*
- *Hatch demo*
- *Complex and semantic figure composition (subplot_mosaic)*

### matplotlib.figure.Figure.add_gridspec

Figure.**add_gridspec**(*nrows=1*, *ncols=1*, ***kwargs*)

Low-level API for creating a *GridSpec* that has this figure as a parent.

This is a low-level API, allowing you to create a gridspec and subsequently add subplots based on the gridspec. Most users do not need that freedom and should use the higher-level methods *subplots* or *subplot_mosaic*.

**Parameters**

**nrows**

[int, default: 1] Number of rows in grid.

**ncols**

[int, default: 1] Number of columns in grid.

**Returns**

***GridSpec***

**Other Parameters**

****kwargs**

Keyword arguments are passed to *GridSpec*.

**See also:**

***matplotlib.pyplot.subplots***

#### Examples

Adding a subplot that spans two rows:

```
fig = plt.figure()
gs = fig.add_gridspec(2, 2)
ax1 = fig.add_subplot(gs[0, 0])
ax2 = fig.add_subplot(gs[1, 0])
# spans two rows:
ax3 = fig.add_subplot(gs[:, 1])
```

### Examples using `matplotlib.figure.Figure.add_gridspec`

- *Scatter plot with histograms*
- *Resizing Axes with tight layout*
- *Creating multiple subplots using plt.subplots*
- *Custom spines with axisartist*
- *Nested GridSpecs*
- *Constrained layout guide*
- *Arranging multiple Axes in a Figure*

## matplotlib.figure.Figure.get_axes

Figure.**get_axes**()

List of Axes in the Figure. You can access and modify the Axes in the Figure through this list.

Do not modify the list itself. Instead, use *add_axes*, *add_subplot* or *delaxes* to add or remove an Axes.

Note: The *Figure.axes* property and *get_axes* method are equivalent.

### Examples using `matplotlib.figure.Figure.get_axes`

- *Creating multiple subplots using plt.subplots*
- *Nested GridSpecs*
- *Arranging multiple Axes in a Figure*

## matplotlib.figure.Figure.axes

**property** Figure.**axes**

List of Axes in the Figure. You can access and modify the Axes in the Figure through this list.

Do not modify the list itself. Instead, use *add_axes*, *add_subplot* or *delaxes* to add or remove an Axes.

Note: The *Figure.axes* property and *get_axes* method are equivalent.

### matplotlib.figure.Figure.delaxes

Figure.**delaxes**(*ax*)

Remove the *Axes* *ax* from the figure; update the current Axes.

### matplotlib.figure.Figure.subfigures

Figure.**subfigures**(*nrows=1*, *ncols=1*, *squeeze=True*, *wspace=None*, *hspace=None*, *width_ratios=None*, *height_ratios=None*, ***kwargs*)

Add a set of subfigures to this figure or subfigure.

A subfigure has the same artist methods as a figure, and is logically the same as a figure, but cannot print itself. See *Figure subfigures*.

---

**Note:** The *subfigure* concept is new in v3.4, and the API is still provisional.

---

#### Parameters

**nrows, ncols**

[int, default: 1] Number of rows/columns of the subfigure grid.

**squeeze**

[bool, default: True] If True, extra dimensions are squeezed out from the returned array of subfigures.

**wspace, hspace**

[float, default: None] The amount of width/height reserved for space between subfigures, expressed as a fraction of the average subfigure width/height. If not given, the values will be inferred from rcParams if using constrained layout (see *ConstrainedLayoutEngine*), or zero if not using a layout engine.

**width_ratios**

[array-like of length *ncols*, optional] Defines the relative widths of the columns. Each column gets a relative width of `width_ratios[i] / sum(width_ratios)`. If not given, all columns will have the same width.

**height_ratios**

[array-like of length *nrows*, optional] Defines the relative heights of the rows. Each row gets a relative height of `height_ratios[i] / sum(height_ratios)`. If not given, all rows will have the same height.

### Examples using `matplotlib.figure.Figure.subfigures`

- *Figure subfigures*
- *Text Rotation Mode*
- *Tick formatters*
- *Constrained layout guide*
- *Arranging multiple Axes in a Figure*
- *Axis ticks*
- *Complex and semantic figure composition (subplot_mosaic)*

## matplotlib.figure.Figure.add_subfigure

`Figure.`**`add_subfigure`**(*subplotspec*, ***kwargs*)

Add a *SubFigure* to the figure as part of a subplot arrangement.

**Parameters**

**subplotspec**

[*gridspec.SubplotSpec*] Defines the region in a parent gridspec where the subfigure will be placed.

**Returns**

***SubFigure***

**Other Parameters**

****kwargs**

Are passed to the *SubFigure* object.

**See also:**

***Figure.subfigures***

### Examples using `matplotlib.figure.Figure.add_subfigure`

- *Figure subfigures*

### Saving

| | |
|---|---|
| *Figure.savefig* | Save the current figure as an image or vector graphic to a file. |

## matplotlib.figure.Figure.savefig

Figure.**savefig**(*fname*, *, *transparent=None*, ***kwargs*)

Save the current figure as an image or vector graphic to a file.

Call signature:

```
savefig(fname, *, transparent=None, dpi='figure', format=None,
        metadata=None, bbox_inches=None, pad_inches=0.1,
        facecolor='auto', edgecolor='auto', backend=None,
        **kwargs
       )
```

The available output formats depend on the backend being used.

**Parameters**

**fname**

[str or path-like or binary file-like] A path, or a Python file-like object, or possibly some backend-dependent object such as *matplotlib.backends.backend_pdf.PdfPages*.

If *format* is set, it determines the output format, and the file is saved as *fname*. Note that *fname* is used verbatim, and there is no attempt to make the extension, if any, of *fname* match *format*, and no extension is appended.

If *format* is not set, then the format is inferred from the extension of *fname*, if there is one. If *format* is not set and *fname* has no extension, then the file is saved with *rcParams["savefig.format"]* (default: `'png'`) and the appropriate extension is appended to *fname*.

**Other Parameters**

**transparent**

[bool, default: *rcParams["savefig.transparent"]* (default: `False`)] If *True*, the Axes patches will all be transparent; the Figure patch will also be transparent unless *facecolor* and/or *edgecolor* are specified via kwargs.

If *False* has no effect and the color of the Axes and Figure patches are unchanged (unless the Figure patch is specified via the *facecolor* and/or *edgecolor* keyword arguments in which case those colors are used).

The transparency of these patches will be restored to their original values upon exit of this function.

This is useful, for example, for displaying a plot on top of a colored background on a web page.

**dpi**

[float or 'figure', default: *rcParams["savefig.dpi"]* (default: `'figure'`)] The resolution in dots per inch. If 'figure', use the figure's dpi value.

**format**

[str] The file format, e.g. 'png', 'pdf', 'svg', ... The behavior when this is unset is documented under *fname*.

**metadata**

[dict, optional] Key/value pairs to store in the image metadata. The supported keys and defaults depend on the image format and backend:

- 'png' with Agg backend: See the parameter `metadata` of *print_png*.
- 'pdf' with pdf backend: See the parameter `metadata` of *PdfPages*.
- 'svg' with svg backend: See the parameter `metadata` of *print_svg*.
- 'eps' and 'ps' with PS backend: Only 'Creator' is supported.

Not supported for 'pgf', 'raw', and 'rgba' as those formats do not support embedding metadata. Does not currently support 'jpg', 'tiff', or 'webp', but may include embedding EXIF metadata in the future.

**bbox_inches**

[str or *Bbox*, default: *rcParams["savefig.bbox"]* (default: `None`)] Bounding box in inches: only the given portion of the figure is saved. If 'tight', try to figure out the tight bbox of the figure.

**pad_inches**

[float or 'layout', default: *rcParams["savefig.pad_inches"]* (default: `0.1`)] Amount of padding in inches around the figure when bbox_inches is 'tight'. If 'layout' use the padding from the constrained or compressed layout engine; ignored if one of those engines is not in use.

**facecolor**

[*color* or 'auto', default: *rcParams["savefig.facecolor"]* (default: `'auto'`)] The facecolor of the figure. If 'auto', use the current figure facecolor.

**edgecolor**

[*color* or 'auto', default: *rcParams["savefig.edgecolor"]* (default: `'auto'`)] The edgecolor of the figure. If 'auto', use the current figure edgecolor.

**backend**

[str, optional] Use a non-default backend to render the file, e.g. to render a png file with the "cairo" backend rather than the default "agg", or a pdf file with the "pgf" backend rather than the default "pdf". Note that the default backend is normally sufficient. See *The builtin backends* for a list of valid backends for each file format. Custom backends can be referenced as "module://...".

**orientation**

[{'landscape', 'portrait'}] Currently only supported by the postscript backend.

**papertype**

[str] One of 'letter', 'legal', 'executive', 'ledger', 'a0' through 'a10', 'b0' through 'b10'. Only supported for postscript output.

**bbox_extra_artists**

[list of *Artist*, optional] A list of extra artists that will be considered when the tight bbox is calculated.

**pil_kwargs**

[dict, optional] Additional keyword arguments that are passed to `PIL.Image.Image.save` when saving the figure.

### Examples using `matplotlib.figure.Figure.savefig`

- *Simple Plot*
- *Convert texts to images*
- *Ellipse with units*
- *CanvasAgg demo*
- *PGF fonts*
- *PGF texsystem*
- *Constrained layout guide*

### Annotating

| | |
|---|---|
| *Figure.colorbar* | Add a colorbar to a plot. |
| *Figure.legend* | Place a legend on the figure. |
| *Figure.text* | Add text to figure. |
| *Figure.suptitle* | Add a centered suptitle to the figure. |
| *Figure.get_suptitle* | Return the suptitle as string or an empty string if not set. |
| *Figure.supxlabel* | Add a centered supxlabel to the figure. |
| *Figure.get_supxlabel* | Return the supxlabel as string or an empty string if not set. |
| *Figure.supylabel* | Add a centered supylabel to the figure. |
| *Figure.get_supylabel* | Return the supylabel as string or an empty string if not set. |
| *Figure.align_labels* | Align the xlabels and ylabels of subplots with the same subplots row or column (respectively) if label alignment is being done automatically (i.e. the label position is not manually set). |
| *Figure.align_xlabels* | Align the xlabels of subplots in the same subplot row if label alignment is being done automatically (i.e. the label position is not manually set). |
| *Figure.align_ylabels* | Align the ylabels of subplots in the same subplot column if label alignment is being done automatically (i.e. the label position is not manually set). |
| *Figure.align_titles* | Align the titles of subplots in the same subplot row if title alignment is being done automatically (i.e. the title position is not manually set). |
| *Figure.autofmt_xdate* | Date ticklabels often overlap, so it is useful to rotate them and right align them. |

### matplotlib.figure.Figure.colorbar

Figure.**colorbar**(*mappable*, *cax=None*, *ax=None*, *use_gridspec=True*, ***kwargs*)

Add a colorbar to a plot.

**Parameters**

**mappable**

The *matplotlib.cm.ScalarMappable* (i.e., *AxesImage*, *ContourSet*, etc.) described by this colorbar. This argument is mandatory for the *Figure.colorbar* method but optional for the *pyplot.colorbar* function, which sets the default to the current image.

Note that one can create a *ScalarMappable* "on-the-fly" to generate colorbars not attached to a previously drawn artist, e.g.

```
fig.colorbar(cm.ScalarMappable(norm=norm, cmap=cmap),␣
↪ax=ax)
```

**cax**

[*Axes*, optional] Axes into which the colorbar will be drawn. If `None`, then a new Axes is created and the space for it will be stolen from the Axes(s) specified in *ax*.

**ax**

[*Axes* or iterable or `numpy.ndarray` of Axes, optional] The one or more parent Axes from which space for a new colorbar Axes will be stolen. This parameter is only used if *cax* is not set.

Defaults to the Axes that contains the mappable used to create the colorbar.

**use_gridspec**

[bool, optional] If *cax* is `None`, a new *cax* is created as an instance of Axes. If *ax* is positioned with a subplotspec and *use_gridspec* is `True`, then *cax* is also positioned with a subplotspec.

**Returns**

**colorbar**

[*Colorbar*]

**Other Parameters**

**location**

[None or {'left', 'right', 'top', 'bottom'}] The location, relative to the parent Axes, where the colorbar Axes is created. It also determines the *orientation* of the colorbar (colorbars on the left and right are vertical, colorbars at the top and bottom are horizontal). If None, the location will come from the *orientation* if it is set (vertical colorbars on the right, horizontal ones at the bottom), or default to 'right' if *orientation* is unset.

**orientation**

[None or {'vertical', 'horizontal'}] The orientation of the colorbar. It is preferable to set the *location* of the colorbar, as that also determines the *orientation*; passing incompatible values for *location* and *orientation* raises an exception.

**fraction**

[float, default: 0.15] Fraction of original Axes to use for colorbar.

**shrink**

[float, default: 1.0] Fraction by which to multiply the size of the colorbar.

**aspect**

[float, default: 20] Ratio of long to short dimensions.

**pad**

[float, default: 0.05 if vertical, 0.15 if horizontal] Fraction of original Axes between colorbar and new image Axes.

**anchor**

[(float, float), optional] The anchor point of the colorbar Axes. Defaults to (0.0, 0.5) if vertical; (0.5, 1.0) if horizontal.

**panchor**

[(float, float), or *False*, optional] The anchor point of the colorbar parent Axes. If *False*, the parent axes' anchor will be unchanged. Defaults to (1.0, 0.5) if vertical; (0.5, 0.0) if horizontal.

**extend**

[{'neither', 'both', 'min', 'max'}] Make pointed end(s) for out-of-range values (unless 'neither'). These are set for a given colormap using the colormap set_under and set_over methods.

**extendfrac**

[{*None*, 'auto', length, lengths}] If set to *None*, both the minimum and maximum triangular colorbar extensions will have a length of 5% of the interior colorbar length (this is the default setting).

If set to 'auto', makes the triangular colorbar extensions the same lengths as the interior boxes (when *spacing* is set to 'uniform') or the same lengths as the respective adjacent interior boxes (when *spacing* is set to 'proportional').

If a scalar, indicates the length of both the minimum and maximum triangular colorbar extensions as a fraction of the interior colorbar length. A two-element sequence of fractions may also be given, indicating the lengths of the minimum and maximum colorbar extensions respectively as a fraction of the interior colorbar length.

**extendrect**

[bool] If *False* the minimum and maximum colorbar extensions will be triangular (the default). If *True* the extensions will be rectangular.

**spacing**

[{'uniform', 'proportional'}] For discrete colorbars (`BoundaryNorm` or contours), 'uniform' gives each color the same space; 'proportional' makes the space proportional to the data interval.

**ticks**

[None or list of ticks or Locator] If None, ticks are determined automatically from the input.

**format**

[None or str or Formatter] If None, *ScalarFormatter* is used. Format strings, e.g., `"%4.2e"` or `"{x:.2e}"`, are supported. An alternative *Formatter* may be given instead.

**drawedges**

[bool] Whether to draw lines at color boundaries.

**label**

[str] The label on the colorbar's long axis.

**boundaries, values**

[None or a sequence] If unset, the colormap will be displayed on a 0-1 scale. If sequences, *values* must have a length 1 less than *boundaries*. For each region delimited by adjacent entries in *boundaries*, the color mapped to the corresponding value in values will be used. Normally only useful for indexed colors (i.e. `norm=NoNorm()`) or other unusual circumstances.

### Notes

If *mappable* is a *ContourSet*, its *extend* kwarg is included automatically.

The *shrink* kwarg provides a simple way to scale the colorbar with respect to the Axes. Note that if *cax* is specified, it determines the size of the colorbar, and *shrink* and *aspect* are ignored.

For more precise control, you can manually specify the positions of the axes objects in which the mappable and the colorbar are drawn. In this case, do not use any of the Axes properties kwargs.

It is known that some vector graphics viewers (svg and pdf) render white gaps between segments of the colorbar. This is due to bugs in the viewers, not Matplotlib. As a workaround, the colorbar can be rendered with overlapping segments:

```
cbar = colorbar()
cbar.solids.set_edgecolor("face")
draw()
```

However, this has negative consequences in other circumstances, e.g. with semi-transparent images (alpha < 1) and colorbar extensions; therefore, this workaround is not used by default (see issue #1188).

### Examples using `matplotlib.figure.Figure.colorbar`

- *Multicolored lines*
- *Mapping marker properties to multivariate data*
- *Interactive Adjustment of Colormap Range*
- *Colormap normalizations*

- *Colormap normalizations SymLogNorm*
- *Contour Demo*
- *Contour Image*
- *Contourf demo*
- *Contourf Hatching*
- *Contourf and log color scale*
- *Creating annotated heatmaps*
- *Image Masked*
- *Contour plot of irregularly spaced data*
- *Multiple images*
- *pcolor images*
- *pcolormesh*
- *Streamplot*
- *Tricontour Demo*
- *Tripcolor Demo*
- *Axis Label Position*
- *Hexagonal binned plot*
- *Time Series Histogram*
- *Colorbar*
- *Creating a colormap from a list of colors*
- *Plotting multiple lines with a LineCollection*
- *Circles, Wedges and Polygons*
- *Colorbar with AxesDivider*
- *Controlling the position and size of colorbars with Inset Axes*
- *Manual Contour*
- *3D box surface plot*
- *3D plots as subplots*
- *3D surface (colormap)*
- *Left ventricle bullseye*
- *Colorbar Tick Labelling*
- *mplcvd -- an example of figure hook*
- *Quick start guide*

- *Constrained layout guide*
- *Placing colorbars*
- *Customized Colorbars Tutorial*
- *Creating Colormaps in Matplotlib*
- *Colormap normalization*

### matplotlib.figure.Figure.legend

`Figure.`**`legend`**`(*args, **kwargs)`

Place a legend on the figure.

Call signatures:

```
legend()
legend(handles, labels)
legend(handles=handles)
legend(labels)
```

The call signatures correspond to the following different ways to use this method:

**1. Automatic detection of elements to be shown in the legend**

The elements to be added to the legend are automatically determined, when you do not pass in any extra arguments.

In this case, the labels are taken from the artist. You can specify them either at artist creation or by calling the *`set_label()`* method on the artist:

```
ax.plot([1, 2, 3], label='Inline label')
fig.legend()
```

or:

```
line, = ax.plot([1, 2, 3])
line.set_label('Label via method')
fig.legend()
```

Specific lines can be excluded from the automatic legend element selection by defining a label starting with an underscore. This is default for all artists, so calling *`Figure.legend`* without any arguments and without setting the labels manually will result in no legend being drawn.

**2. Explicitly listing the artists and labels in the legend**

For full control of which artists have a legend entry, it is possible to pass an iterable of legend artists followed by an iterable of legend labels respectively:

```
fig.legend([line1, line2, line3], ['label1', 'label2', 'label3'])
```

**3. Explicitly listing the artists in the legend**

This is similar to 2, but the labels are taken from the artists' label properties. Example:

```
line1, = ax1.plot([1, 2, 3], label='label1')
line2, = ax2.plot([1, 2, 3], label='label2')
fig.legend(handles=[line1, line2])
```

**4. Labeling existing plot elements**

---

**Discouraged**

This call signature is discouraged, because the relation between plot elements and labels is only implicit by their order and can easily be mixed up.

---

To make a legend for all artists on all Axes, call this function with an iterable of strings, one for each legend item. For example:

```
fig, (ax1, ax2) = plt.subplots(1, 2)
ax1.plot([1, 3, 5], color='blue')
ax2.plot([2, 4, 6], color='red')
fig.legend(['the blues', 'the reds'])
```

**Parameters**

**handles**

[list of *Artist*, optional] A list of Artists (lines, patches) to be added to the legend. Use this together with *labels*, if you need full control on what is shown in the legend and the automatic mechanism described above is not sufficient.

The length of handles and labels should be the same in this case. If they are not, they are truncated to the smaller length.

**labels**

[list of str, optional] A list of labels to show next to the artists. Use this together with *handles*, if you need full control on what is shown in the legend and the automatic mechanism described above is not sufficient.

**Returns**

***Legend***

**Other Parameters**

**loc**

[str or pair of floats, default: 'upper right'] The location of the legend.

The strings `'upper left'`, `'upper right'`, `'lower left'`, `'lower right'` place the legend at the corresponding corner of the figure.

The strings `'upper center'`, `'lower center'`, `'center left'`, `'center right'` place the legend at the center of the corresponding edge of the figure.

The string `'center'` places the legend at the center of the figure.

The location can also be a 2-tuple giving the coordinates of the lower-left corner of the legend in figure coordinates (in which case *bbox_to_anchor* will be ignored).

For back-compatibility, `'center right'` (but no other location) can also be spelled `'right'`, and each "string" location can also be given as a numeric value:

| Location String | Location Code |
|---|---|
| 'best' (Axes only) | 0 |
| 'upper right' | 1 |
| 'upper left' | 2 |
| 'lower left' | 3 |
| 'lower right' | 4 |
| 'right' | 5 |
| 'center left' | 6 |
| 'center right' | 7 |
| 'lower center' | 8 |
| 'upper center' | 9 |
| 'center' | 10 |

If a figure is using the constrained layout manager, the string codes of the *loc* keyword argument can get better layout behaviour using the prefix 'outside'. There is ambiguity at the corners, so 'outside upper right' will make space for the legend above the rest of the axes in the layout, and 'outside right upper' will make space on the right side of the layout. In addition to the values of *loc* listed above, we have 'outside right upper', 'outside right lower', 'outside left upper', and 'outside left lower'. See *Legend guide* for more details.

**bbox_to_anchor**

[*BboxBase*, 2-tuple, or 4-tuple of floats] Box that is used to position the legend in conjunction with *loc*. Defaults to `axes.bbox` (if called as a method to *Axes.legend*) or `figure.bbox` (if *Figure.legend*). This argument allows arbitrary placement of the legend.

Bbox coordinates are interpreted in the coordinate system given by *bbox_transform*, with the default transform Axes or Figure coordinates, depending on which `legend` is called.

If a 4-tuple or *BboxBase* is given, then it specifies the bbox `(x, y, width, height)` that the legend is placed in. To put the legend in the best location in the bottom right quadrant of the Axes (or figure):

```
loc='best', bbox_to_anchor=(0.5, 0., 0.5, 0.5)
```

A 2-tuple `(x, y)` places the corner of the legend specified by *loc* at x, y. For example, to put the legend's upper right-hand corner in the center of the Axes (or figure) the following keywords can be used:

```
loc='upper right', bbox_to_anchor=(0.5, 0.5)
```

**ncols**

[int, default: 1] The number of columns that the legend has.

For backward compatibility, the spelling *ncol* is also supported but it is discouraged. If both are given, *ncols* takes precedence.

**prop**

[None or *FontProperties* or dict] The font properties of the legend. If None (default), the current *matplotlib.rcParams* will be used.

**fontsize**

[int or {'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large'}] The font size of the legend. If the value is numeric the size will be the absolute font size in points. String values are relative to the current default font size. This argument is only used if *prop* is not specified.

**labelcolor**

[str or list, default: *rcParams["legend.labelcolor"]* (default: `'None'`)] The color of the text in the legend. Either a valid color string (for example, 'red'), or a list of color strings. The labelcolor can also be made to match the color of the line or marker using 'linecolor', 'markerfacecolor' (or 'mfc'), or 'markeredgecolor' (or 'mec').

Labelcolor can be set globally using *rcParams["legend.labelcolor"]* (default: `'None'`). If None, use *rcParams["text.color"]* (default: `'black'`).

**numpoints**

[int, default: *rcParams["legend.numpoints"]* (default: `1`)] The number of marker points in the legend when creating a legend entry for a *Line2D* (line).

**scatterpoints**

[int, default: *rcParams["legend.scatterpoints"]* (default: `1`)] The number of marker points in the legend when creating a legend entry for a *PathCollection* (scatter plot).

**scatteryoffsets**

[iterable of floats, default: `[0.375, 0.5, 0.3125]`] The vertical offset (relative to the font size) for the markers created for a scatter plot legend entry. 0.0 is at the base the legend text, and 1.0 is at the top. To draw all markers at the same height, set to `[0.5]`.

**markerscale**

[float, default: *rcParams["legend.markerscale"]* (default: `1.0`)] The relative size of legend markers compared to the originally drawn ones.

**markerfirst**

[bool, default: True] If *True*, legend marker is placed to the left of the legend label. If *False*, legend marker is placed to the right of the legend label.

**reverse**

[bool, default: False] If *True*, the legend labels are displayed in reverse order from the input. If *False*, the legend labels are displayed in the same order as the input.

New in version 3.7.

**frameon**

[bool, default: *rcParams["legend.frameon"]* (default: `True`)] Whether the legend should be drawn on a patch (frame).

**fancybox**

[bool, default: *rcParams["legend.fancybox"]* (default: `True`)] Whether round edges should be enabled around the *FancyBboxPatch* which makes up the legend's background.

**shadow**

[None, bool or dict, default: *rcParams["legend.shadow"]* (default: `False`)] Whether to draw a shadow behind the legend. The shadow can be configured using *Patch* keywords. Customization via *rcParams["legend.shadow"]* (default: `False`) is currently not supported.

**framealpha**

[float, default: *rcParams["legend.framealpha"]* (default: `0.8`)] The alpha transparency of the legend's background. If *shadow* is activated and *framealpha* is `None`, the default value is ignored.

**facecolor**

["inherit" or color, default: *rcParams["legend.facecolor"]* (default: `'inherit'`)] The legend's background color. If `"inherit"`, use *rcParams["axes.facecolor"]* (default: `'white'`).

**edgecolor**

["inherit" or color, default: *rcParams["legend.edgecolor"]* (default: `'0.8'`)] The legend's background patch edge color. If `"inherit"`, use *rcParams["axes.edgecolor"]* (default: `'black'`).

**mode**

[{"expand", None}] If *mode* is set to `"expand"` the legend will be horizontally expanded to fill the Axes area (or *bbox_to_anchor* if defines the legend's size).

**bbox_transform**

[None or *Transform*] The transform for the bounding box (*bbox_to_anchor*). For a value of None (default) the Axes' transAxes transform will be used.

**title**

[str or None] The legend's title. Default is no title (None).

**title_fontproperties**

[None or *FontProperties* or dict] The font properties of the legend's title. If None (default), the *title_fontsize* argument will be used if present; if *title_fontsize* is also None, the current *rcParams["legend.title_fontsize"]* (default: None) will be used.

**title_fontsize**

[int or {'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large'}, default: *rcParams["legend.title_fontsize"]* (default: None)] The font size of the legend's title. Note: This cannot be combined with *title_fontproperties*. If you want to set the fontsize alongside other font properties, use the *size* parameter in *title_fontproperties*.

**alignment**

[{'center', 'left', 'right'}, default: 'center'] The alignment of the legend title and the box of entries. The entries are aligned as a single block, so that markers always lined up.

**borderpad**

[float, default: *rcParams["legend.borderpad"]* (default: 0.4)] The fractional whitespace inside the legend border, in font-size units.

**labelspacing**

[float, default: *rcParams["legend.labelspacing"]* (default: 0.5)] The vertical space between the legend entries, in font-size units.

**handlelength**

[float, default: *rcParams["legend.handlelength"]* (default: 2.0)] The length of the legend handles, in font-size units.

**handleheight**

[float, default: *rcParams["legend.handleheight"]* (default: 0.7)] The height of the legend handles, in font-size units.

**handletextpad**

[float, default: *rcParams["legend.handletextpad"]* (default: 0.8)] The pad between the legend handle and text, in font-size units.

**borderaxespad**

[float, default: *rcParams["legend.borderaxespad"]* (default: 0.5)]
The pad between the Axes and legend border, in font-size units.

**columnspacing**

[float, default: *rcParams["legend.columnspacing"]* (default: 2.0)]
The spacing between columns, in font-size units.

**handler_map**

[dict or None] The custom dictionary mapping instances or types to a legend handler. This *handler_map* updates the default handler map found at *matplotlib.legend.Legend.get_legend_handler_map*.

**draggable**

[bool, default: False] Whether the legend can be dragged with the mouse.

**See also:**

***Axes.legend***

### Notes

Some artists are not supported by this function. See *Legend guide* for details.

## Examples using `matplotlib.figure.Figure.legend`

- *Figure legend demo*
- *Legend guide*
- *Constrained layout guide*

## matplotlib.figure.Figure.text

`Figure.`**`text`**(*x*, *y*, *s*, *fontdict=None*, ***kwargs*)

Add text to figure.

**Parameters**

**x, y**

[float] The position to place the text. By default, this is in figure coordinates, floats in [0, 1]. The coordinate system can be changed using the *transform* keyword.

**s**

[str] The text string.

**fontdict**

[dict, optional] A dictionary to override the default text properties. If not given, the defaults are determined by *rcParams["font.*"]*. Properties passed as *kwargs* override the corresponding ones given in *fontdict*.

**Returns**

***Text***

**Other Parameters**

****kwargs**

[*Text* properties] Other miscellaneous text parameters.

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array a |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* | bool |
| *backgroundcolor* | *color* |
| *bbox* | dict with properties for *patches.FancyBboxPa* |
| *clip_box* | unknown |
| *clip_on* | unknown |
| *clip_path* | unknown |
| *color* or c | *color* |
| *figure* | *Figure* |
| *fontfamily* or family or fontname | {FONTNAME, 'serif', 'sans-serif', 'cursive', 'fantasy' |
| *fontproperties* or font or font_properties | *font_manager.FontProperties* or *str* or |
| *fontsize* or size | float or {'xx-small', 'x-small', 'small', 'medium', 'large |
| *fontstretch* or stretch | {a numeric value in range 0-1000, 'ultra-condensed', |
| *fontstyle* or style | {'normal', 'italic', 'oblique'} |
| *fontvariant* or variant | {'normal', 'small-caps'} |
| *fontweight* or weight | {a numeric value in range 0-1000, 'ultralight', 'light', |
| *gid* | str |
| *horizontalalignment* or ha | {'left', 'center', 'right'} |
| *in_layout* | bool |
| *label* | object |
| *linespacing* | float (multiple of font size) |
| *math_fontfamily* | str |
| *mouseover* | bool |
| *multialignment* or ma | {'left', 'right', 'center'} |
| *parse_math* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *position* | (float, float) |

Table 52 –

| Property | Description |
|---|---|
| *rasterized* | bool |
| *rotation* | float or {'vertical', 'horizontal'} |
| *rotation_mode* | {None, 'default', 'anchor'} |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *text* | object |
| *transform* | *Transform* |
| *transform_rotates_text* | bool |
| *url* | str |
| *usetex* | bool, default: *rcParams["text.usetex"]* (de |
| *verticalalignment* or va | {'baseline', 'bottom', 'center', 'center_baseline', 'top'} |
| *visible* | bool |
| *wrap* | bool |
| *x* | float |
| *y* | float |
| *zorder* | float |

**See also:**

***Axes.text***
***pyplot.text***

### Examples using `matplotlib.figure.Figure.text`

- *Custom Figure subclasses*
- *Boxplots*
- *The difference between \dfrac and \frac*
- *Fonts demo (object-oriented style)*
- *Fonts demo (keyword arguments)*
- *Convert texts to images*
- *STIX Fonts*
- *Unicode minus*
- *Usetex Fonteffects*
- *Text watermark*
- *Colormap reference*
- *Drawing fancy boxes*
- *Integral as the area under a curve*

- *XKCD*
- *Radar chart (aka spider or star chart)*
- *Menu*
- *Path effects guide*
- *Choosing Colormaps in Matplotlib*
- *Writing mathematical expressions*

## matplotlib.figure.Figure.suptitle

Figure.**suptitle**(*t*, ***kwargs*)

Add a centered suptitle to the figure.

**Parameters**

**t**

[str] The suptitle text.

**x**

[float, default: 0.5] The x location of the text in figure coordinates.

**y**

[float, default: 0.98] The y location of the text in figure coordinates.

**horizontalalignment, ha**

[{'center', 'left', 'right'}, default: center] The horizontal alignment of the text relative to (*x*, *y*).

**verticalalignment, va**

[{'top', 'center', 'bottom', 'baseline'}, default: top] The vertical alignment of the text relative to (*x*, *y*).

**fontsize, size**

[default: *rcParams["figure.titlesize"]* (default: 'large')] The font size of the text. See *Text.set_size* for possible values.

**fontweight, weight**

[default: *rcParams["figure.titleweight"]* (default: 'normal')] The font weight of the text. See *Text.set_weight* for possible values.

**Returns**

**text**

The *Text* instance of the suptitle.

**Other Parameters**

**fontproperties**

[None or dict, optional] A dict of font properties. If *fontproperties* is given the default values for font size and weight are taken from the *FontProperties* defaults. *rcParams["figure.titlesize"]* (default: `'large'`) and *rcParams["figure.titleweight"]* (default: `'normal'`) are ignored in this case.

****kwargs**

Additional kwargs are *matplotlib.text.Text* properties.

## Examples using `matplotlib.figure.Figure.suptitle`

- *Plotting categorical variables*
- *Errorbar subsampling*
- *Fill Between and Alpha*
- *Marker reference*
- *Mapping marker properties to multivariate data*
- *Image Masked*
- *Image nonuniform*
- *Multiple images*
- *Figure labels: suptitle, supxlabel, supylabel*
- *Using Gridspec to make multi-column/row subplot layouts*
- *Inverted axis*
- *Figure subfigures*
- *Multiple subplots*
- *Creating multiple subplots using plt.subplots*
- *Artist customization in box plots*
- *Plotting cumulative distributions*
- *Violin plot basics*
- *Scale invariant angle label*
- *Text Commands*
- *Usetex Fonteffects*
- *Colors in the default property cycle*
- *Creating a colormap from a list of colors*

- *Grayscale style sheet*
- *Simple Axes Divider 1*
- *Pan/zoom events of overlapping axes*
- *Stock prices over 32 years*
- *Figure/Axes enter and leave events*
- *2D and 3D Axes in same figure*
- *Spine placement*
- *Date tick locators and formatters*
- *Evans test*
- *Mouse Cursor*
- *GridSpec demo*
- *Constrained layout guide*
- *Arranging multiple Axes in a Figure*
- *Plotting dates and strings*
- *Placing colorbars*
- *Choosing Colormaps in Matplotlib*
- *Text in Matplotlib*
- *Writing mathematical expressions*

### matplotlib.figure.Figure.get_suptitle

Figure.**get_suptitle**()

Return the suptitle as string or an empty string if not set.

### matplotlib.figure.Figure.supxlabel

Figure.**supxlabel**(*t*, ***kwargs*)

Add a centered supxlabel to the figure.

**Parameters**

**t**

[str] The supxlabel text.

**x**

[float, default: 0.5] The x location of the text in figure coordinates.

**y**

[float, default: 0.01] The y location of the text in figure coordinates.

**horizontalalignment, ha**

[{'center', 'left', 'right'}, default: center] The horizontal alignment of the text relative to (*x*, *y*).

**verticalalignment, va**

[{'top', 'center', 'bottom', 'baseline'}, default: bottom] The vertical alignment of the text relative to (*x*, *y*).

**fontsize, size**

[default: *rcParams["figure.labelsize"]* (default: `'large'`)] The font size of the text. See *Text.set_size* for possible values.

**fontweight, weight**

[default: *rcParams["figure.labelweight"]* (default: `'normal'`)] The font weight of the text. See *Text.set_weight* for possible values.

**Returns**

**text**

The *Text* instance of the supxlabel.

**Other Parameters**

**fontproperties**

[None or dict, optional] A dict of font properties. If *fontproperties* is given the default values for font size and weight are taken from the *FontProperties* defaults. *rcParams["figure.labelsize"]* (default: `'large'`) and *rcParams["figure.labelweight"]* (default: `'normal'`) are ignored in this case.

****kwargs**

Additional kwargs are *matplotlib.text.Text* properties.

## Examples using `matplotlib.figure.Figure.supxlabel`

- *Figure labels: suptitle, supxlabel, supylabel*

### matplotlib.figure.Figure.get_supxlabel

Figure.**get_supxlabel**()

Return the supxlabel as string or an empty string if not set.

### matplotlib.figure.Figure.supylabel

Figure.**supylabel**(*t*, ***kwargs*)

Add a centered supylabel to the figure.

**Parameters**

**t**

[str] The supylabel text.

**x**

[float, default: 0.02] The x location of the text in figure coordinates.

**y**

[float, default: 0.5] The y location of the text in figure coordinates.

**horizontalalignment, ha**

[{'center', 'left', 'right'}, default: left] The horizontal alignment of the text relative to (*x*, *y*).

**verticalalignment, va**

[{'top', 'center', 'bottom', 'baseline'}, default: center] The vertical alignment of the text relative to (*x*, *y*).

**fontsize, size**

[default: *rcParams["figure.labelsize"]* (default: 'large')] The font size of the text. See *Text.set_size* for possible values.

**fontweight, weight**

[default: *rcParams["figure.labelweight"]* (default: 'normal')] The font weight of the text. See *Text.set_weight* for possible values.

**Returns**

**text**

The *Text* instance of the supylabel.

**Other Parameters**

**fontproperties**

[None or dict, optional] A dict of font properties. If *fontproperties* is given the default values for font size and weight are taken from the *FontProperties* defaults. *rcParams["figure.labelsize"]* (default: 'large') and *rcParams["figure.labelweight"]* (default: 'normal') are ignored in this case.

****kwargs**

Additional kwargs are *matplotlib.text.Text* properties.

### Examples using `matplotlib.figure.Figure.supylabel`

- *Figure labels: suptitle, supxlabel, supylabel*

## matplotlib.figure.Figure.get_supylabel

Figure.**get_supylabel**()

Return the supylabel as string or an empty string if not set.

## matplotlib.figure.Figure.align_labels

Figure.**align_labels**(*axs=None*)

Align the xlabels and ylabels of subplots with the same subplots row or column (respectively) if label alignment is being done automatically (i.e. the label position is not manually set).

Alignment persists for draw events after this is called.

**Parameters**

**axs**

[list of *Axes*] Optional list (or ndarray) of *Axes* to align the labels. Default is to align all Axes on the figure.

**See also:**

***matplotlib.figure.Figure.align_xlabels***
***matplotlib.figure.Figure.align_ylabels***
***matplotlib.figure.Figure.align_titles***

### Examples using `matplotlib.figure.Figure.align_labels`

- *Aligning Labels and Titles*

## matplotlib.figure.Figure.align_xlabels

Figure.**align_xlabels**(*axs=None*)

Align the xlabels of subplots in the same subplot row if label alignment is being done automatically (i.e. the label position is not manually set).

Alignment persists for draw events after this is called.

If a label is on the bottom, it is aligned with labels on Axes that also have their label on the bottom and that have the same bottom-most subplot row. If the label is on the top, it is aligned with labels on Axes with the same top-most row.

**Parameters**

**axs**

[list of *Axes*] Optional list of (or `ndarray`) *Axes* to align the xlabels. Default is to align all Axes on the figure.

**See also:**

***matplotlib.figure.Figure.align_ylabels***
***matplotlib.figure.Figure.align_titles***
***matplotlib.figure.Figure.align_labels***

### Notes

This assumes that `axs` are from the same *GridSpec*, so that their *SubplotSpec* positions correspond to figure positions.

### Examples

Example with rotated xtick labels:

```
fig, axs = plt.subplots(1, 2)
for tick in axs[0].get_xticklabels():
    tick.set_rotation(55)
axs[0].set_xlabel('XLabel 0')
axs[1].set_xlabel('XLabel 1')
fig.align_xlabels()
```

## matplotlib.figure.Figure.align_ylabels

Figure.**align_ylabels**(*axs=None*)

Align the ylabels of subplots in the same subplot column if label alignment is being done automatically (i.e. the label position is not manually set).

Alignment persists for draw events after this is called.

If a label is on the left, it is aligned with labels on Axes that also have their label on the left and that have the same left-most subplot column. If the label is on the right, it is aligned with labels on Axes with the same right-most column.

**Parameters**

**axs**

[list of *Axes*] Optional list (or ndarray) of *Axes* to align the ylabels. Default is to align all Axes on the figure.

**See also:**

***matplotlib.figure.Figure.align_xlabels***
***matplotlib.figure.Figure.align_titles***
***matplotlib.figure.Figure.align_labels***

### Notes

This assumes that axs are from the same *GridSpec*, so that their *SubplotSpec* positions correspond to figure positions.

### Examples

Example with large yticks labels:

```
fig, axs = plt.subplots(2, 1)
axs[0].plot(np.arange(0, 1000, 50))
axs[0].set_ylabel('YLabel 0')
axs[1].set_ylabel('YLabel 1')
fig.align_ylabels()
```

### Examples using `matplotlib.figure.Figure.align_ylabels`

- *Align y-labels*

## matplotlib.figure.Figure.align_titles

`Figure.`**`align_titles`**(*axs=None*)

Align the titles of subplots in the same subplot row if title alignment is being done automatically (i.e. the title position is not manually set).

Alignment persists for draw events after this is called.

**Parameters**

**axs**

[list of *Axes*] Optional list of (or ndarray) *Axes* to align the titles. Default is to align all Axes on the figure.

**See also:**

***matplotlib.figure.Figure.align_xlabels***
***matplotlib.figure.Figure.align_ylabels***
***matplotlib.figure.Figure.align_labels***

### Notes

This assumes that `axs` are from the same *GridSpec*, so that their *SubplotSpec* positions correspond to figure positions.

### Examples

Example with titles:

```
fig, axs = plt.subplots(1, 2)
axs[0].set_aspect('equal')
axs[0].set_title('Title 0')
axs[1].set_title('Title 1')
fig.align_titles()
```

### Examples using `matplotlib.figure.Figure.align_titles`

- *Aligning Labels and Titles*

## matplotlib.figure.Figure.autofmt_xdate

Figure.**autofmt_xdate**(*bottom=0.2*, *rotation=30*, *ha='right'*, *which='major'*)

Date ticklabels often overlap, so it is useful to rotate them and right align them. Also, a common use case is a number of subplots with shared x-axis where the x-axis is date data. The ticklabels are often long, and it helps to rotate them on the bottom subplot and turn them off on other subplots, as well as turn off xlabels.

**Parameters**

**bottom**

[float, default: 0.2] The bottom of the subplots for *subplots_adjust*.

**rotation**

[float, default: 30 degrees] The rotation angle of the xtick labels in degrees.

**ha**

[{'left', 'center', 'right'}, default: 'right'] The horizontal alignment of the xticklabels.

**which**

[{'major', 'minor', 'both'}, default: 'major'] Selects which ticklabels to rotate.

### Examples using `matplotlib.figure.Figure.autofmt_xdate`

- *Fill Between and Alpha*
- *Date Demo Convert*

### Figure geometry

| | |
|---|---|
| *Figure.set_size_inches* | Set the figure size in inches. |
| *Figure.get_size_inches* | Return the current size of the figure in inches. |
| *Figure.set_figheight* | Set the height of the figure in inches. |
| *Figure.get_figheight* | Return the figure height in inches. |
| *Figure.set_figwidth* | Set the width of the figure in inches. |
| *Figure.get_figwidth* | Return the figure width in inches. |
| *Figure.dpi* | The resolution in dots per inch. |
| *Figure.set_dpi* | Set the resolution of the figure in dots-per-inch. |
| *Figure.set_dpi* | Set the resolution of the figure in dots-per-inch. |

### matplotlib.figure.Figure.set_size_inches

Figure.**set_size_inches**(*w*, *h=None*, *forward=True*)

Set the figure size in inches.

Call signatures:

```
fig.set_size_inches(w, h)  # OR
fig.set_size_inches((w, h))
```

**Parameters**

**w**

[(float, float) or float] Width and height in inches (if height not specified as a separate argument) or width.

**h**

[float] Height in inches.

**forward**

[bool, default: True] If `True`, the canvas size is automatically updated, e.g., you can resize the figure window from the shell.

**See also:**

*matplotlib.figure.Figure.get_size_inches*
*matplotlib.figure.Figure.set_figwidth*
*matplotlib.figure.Figure.set_figheight*

#### Notes

To transform from pixels to inches divide by *Figure.dpi*.

### matplotlib.figure.Figure.get_size_inches

Figure.**get_size_inches**()

Return the current size of the figure in inches.

**Returns**

**ndarray**

The size (width, height) of the figure in inches.

**See also:**

*matplotlib.figure.Figure.set_size_inches*
*matplotlib.figure.Figure.get_figwidth*
*matplotlib.figure.Figure.get_figheight*

### Notes

The size in pixels can be obtained by multiplying with *Figure.dpi*.

## Examples using `matplotlib.figure.Figure.get_size_inches`

- *List of named colors*

## matplotlib.figure.Figure.set_figheight

`Figure.`**`set_figheight`**(*val*, *forward=True*)

Set the height of the figure in inches.

**Parameters**

**val**

[float]

**forward**

[bool] See *set_size_inches*.

**See also:**

*matplotlib.figure.Figure.set_figwidth*
*matplotlib.figure.Figure.set_size_inches*

## matplotlib.figure.Figure.get_figheight

`Figure.`**`get_figheight`**()

Return the figure height in inches.

### matplotlib.figure.Figure.set_figwidth

Figure.**set_figwidth**(*val*, *forward=True*)

Set the width of the figure in inches.

**Parameters**

**val**

[float]

**forward**

[bool] See *set_size_inches*.

**See also:**

***matplotlib.figure.Figure.set_figheight***
***matplotlib.figure.Figure.set_size_inches***

### matplotlib.figure.Figure.get_figwidth

Figure.**get_figwidth**()

Return the figure width in inches.

### matplotlib.figure.Figure.dpi

**property** Figure.**dpi**

The resolution in dots per inch.

### matplotlib.figure.Figure.set_dpi

Figure.**set_dpi**(*val*)

Set the resolution of the figure in dots-per-inch.

**Parameters**

**val**

[float]

### Subplot layout

| | |
|---|---|
| *Figure.subplots_adjust* | Adjust the subplot layout parameters. |
| *Figure.set_layout_engine* | Set the layout engine for this figure. |
| *Figure.get_layout_engine* | |

### matplotlib.figure.Figure.subplots_adjust

Figure.**subplots_adjust**(*left=None*, *bottom=None*, *right=None*, *top=None*, *wspace=None*, *hspace=None*)

Adjust the subplot layout parameters.

Unset parameters are left unmodified; initial values are given by *rcParams["figure.subplot.[name]"]*.

**Parameters**

**left**

[float, optional] The position of the left edge of the subplots, as a fraction of the figure width.

**right**

[float, optional] The position of the right edge of the subplots, as a fraction of the figure width.

**bottom**

[float, optional] The position of the bottom edge of the subplots, as a fraction of the figure height.

**top**

[float, optional] The position of the top edge of the subplots, as a fraction of the figure height.

**wspace**

[float, optional] The width of the padding between subplots, as a fraction of the average Axes width.

**hspace**

[float, optional] The height of the padding between subplots, as a fraction of the average Axes height.

### Examples using `matplotlib.figure.Figure.subplots_adjust`

- *Marker reference*
- *Contour Image*
- *Programmatically controlling subplot adjustment*
- *Broken Axis*
- *Creating adjacent subplots*
- *Artist customization in box plots*
- *Boxplots*
- *Plot a confidence ellipse of a two-dimensional dataset*
- *Violin plot basics*
- *Bar of pie*
- *Align y-labels*
- *Multiline*
- *Text Commands*
- *Creating a colormap from a list of colors*
- *List of named colors*
- *Line, Poly and RegularPoly Collection with autoscaling*
- *Colorbar with AxesDivider*
- *Axis Direction*
- *floating_axes features*
- *Ticklabel alignment*
- *Ticklabel direction*
- *Simple axis direction*
- *Simple axis tick label and tick directions*
- *Simple Axis Pad*
- *Simple Axisline*
- *Radar chart (aka spider or star chart)*
- *Topographic hillshading*
- *Multiple y-axis with Spines*
- *Evans test*
- *Buttons*

- *Menu*
- *Thresholding an Image with RangeSlider*
- *Slider*
- *Snapping Sliders to Discrete Values*
- *Textbox*
- *The Lifecycle of a Plot*
- *Artist tutorial*
- *Styling with cycler*
- *Choosing Colormaps in Matplotlib*
- *Text in Matplotlib*
- *Annotations*

### matplotlib.figure.Figure.set_layout_engine

`Figure.`**`set_layout_engine`**`(`*layout=None*, ***kwargs*`)`

Set the layout engine for this figure.

**Parameters**

**layout**

[{'constrained', 'compressed', 'tight', 'none', *LayoutEngine*, None}]

- 'constrained' will use *ConstrainedLayoutEngine*
- 'compressed' will also use *ConstrainedLayoutEngine*, but with a correction that attempts to make a good layout for fixed-aspect ratio Axes.
- 'tight' uses *TightLayoutEngine*
- 'none' removes layout engine.

If a *LayoutEngine* instance, that instance will be used.

If None, the behavior is controlled by *rcParams["figure.autolayout"]* (default: False) (which if True behaves as if 'tight' was passed) and *rcParams["figure.constrained_layout.use"]* (default: False) (which if True behaves as if 'constrained' was passed). If both are True, *rcParams["figure.autolayout"]* (default: False) takes priority.

Users and libraries can define their own layout engines and pass the instance directly as well.

****kwargs**

The keyword arguments are passed to the layout engine to set things like padding and margin sizes. Only used if *layout* is a string.

### Examples using `matplotlib.figure.Figure.set_layout_engine`

- *Constrained layout guide*

## matplotlib.figure.Figure.get_layout_engine

`Figure.get_layout_engine()`

### Examples using `matplotlib.figure.Figure.get_layout_engine`

- *Left ventricle bullseye*
- *Custom tick formatter for time series*
- *Connection styles for annotations*
- *Constrained layout guide*

### Discouraged or deprecated

| | |
|---|---|
| *Figure.tight_layout* | Adjust the padding between and around subplots. |
| *Figure.set_tight_layout* | [*Deprecated*] Set whether and how *Figure.tight_layout* is called when drawing. |
| *Figure.get_tight_layout* | Return whether *Figure.tight_layout* is called when drawing. |
| *Figure.set_constrained_layout* | [*Deprecated*] Set whether constrained_layout is used upon drawing. |
| *Figure.get_constrained_layout* | Return whether constrained layout is being used. |
| *Figure.set_constrained_layout_pads* | [*Deprecated*] Set padding for constrained_layout. |
| *Figure.get_constrained_layout_pads* | [*Deprecated*] Get padding for constrained_layout. |

## matplotlib.figure.Figure.tight_layout

`Figure.tight_layout(*, pad=1.08, h_pad=None, w_pad=None, rect=None)`

Adjust the padding between and around subplots.

To exclude an artist on the Axes from the bounding box calculation that determines the subplot parameters (i.e. legend, or annotation), set `a.set_in_layout(False)` for that artist.

**Parameters**

**pad**

[float, default: 1.08] Padding between the figure edge and the edges of subplots, as a fraction of the font size.

**h_pad, w_pad**

[float, default: *pad*] Padding (height/width) between edges of adjacent subplots, as a fraction of the font size.

**rect**

[tuple (left, bottom, right, top), default: (0, 0, 1, 1)] A rectangle in normalized figure coordinates into which the whole subplots area (including labels) will fit.

**See also:**

***Figure.set_layout_engine***
***pyplot.tight_layout***

### Examples using `matplotlib.figure.Figure.tight_layout`

- *Filling the area between lines*
- *Hat graph*
- *Marker reference*
- *Scatter Demo2*
- *Contour Image*
- *Creating annotated heatmaps*
- *pcolor images*
- *pcolormesh*
- *QuadMesh Demo*
- *Triinterp Demo*
- *Equal axis aspect ratio*
- *Resizing Axes with tight layout*
- *Combining two subplots using subplots and GridSpec*
- *Creating multiple subplots using plt.subplots*
- *Plots with different scales*
- *Demo of the histogram function's different histtype settings*
- *The histogram (hist) function with multiple data sets*
- *Accented text*

- *Mathtext*
- *Drawing fancy boxes*
- *Log Demo*
- *Logit Demo*
- *Exploring normalizations*
- *Symlog Demo*
- *Bar demo with units*
- *Radian ticks*
- *Unit handling*
- *PGF fonts*
- *PGF texsystem*
- *Tight layout guide*
- *Choosing Colormaps in Matplotlib*

### matplotlib.figure.Figure.set_tight_layout

`Figure.`**`set_tight_layout`**(*tight*)

[*Deprecated*] Set whether and how *`Figure.tight_layout`* is called when drawing.

**Parameters**

**tight**

[bool or dict with keys "pad", "w_pad", "h_pad", "rect" or None] If a bool, sets whether to call *`Figure.tight_layout`* upon drawing. If `None`, use *`rcParams["figure.autolayout"]`* (default: `False`) instead. If a dict, pass it as kwargs to *`Figure.tight_layout`*, overriding the default paddings.

#### Notes

Deprecated since version 3.6: Use set_layout_engine instead.

### matplotlib.figure.Figure.get_tight_layout

Figure.**get_tight_layout**()

Return whether *Figure.tight_layout* is called when drawing.

### matplotlib.figure.Figure.set_constrained_layout

Figure.**set_constrained_layout**(*constrained*)

[*Deprecated*] Set whether constrained_layout is used upon drawing.

If None, *rcParams["figure.constrained_layout.use"]* (default: False) value will be used.

When providing a dict containing the keys w_pad, h_pad the default constrained_layout paddings will be overridden. These pads are in inches and default to 3.0/72.0. w_pad is the width padding and h_pad is the height padding.

**Parameters**

**constrained**

[bool or dict or None]

#### Notes

Deprecated since version 3.6: Use set_layout_engine('constrained') instead.

### matplotlib.figure.Figure.get_constrained_layout

Figure.**get_constrained_layout**()

Return whether constrained layout is being used.

See *Constrained layout guide*.

### matplotlib.figure.Figure.set_constrained_layout_pads

Figure.**set_constrained_layout_pads**(***kwargs*)

[*Deprecated*] Set padding for constrained_layout.

Tip: The parameters can be passed from a dictionary by using fig.set_constrained_layout(**pad_dict).

See *Constrained layout guide*.

**Parameters**

**w_pad**

[float, default: *rcParams["figure.constrained_layout.w_pad"]* (default: 0.04167)] Width padding in inches. This is the pad around Axes and is meant to make sure there is enough room for fonts to look good. Defaults to 3 pts = 0.04167 inches

**h_pad**

[float, default: *rcParams["figure.constrained_layout.h_pad"]* (default: 0.04167)] Height padding in inches. Defaults to 3 pts.

**wspace**

[float, default: *rcParams["figure.constrained_layout.wspace"]* (default: 0.02)] Width padding between subplots, expressed as a fraction of the subplot width. The total padding ends up being w_pad + wspace.

**hspace**

[float, default: *rcParams["figure.constrained_layout.hspace"]* (default: 0.02)] Height padding between subplots, expressed as a fraction of the subplot width. The total padding ends up being h_pad + hspace.

### Notes

Deprecated since version 3.6: Use figure.get_layout_engine().set() instead.

## matplotlib.figure.Figure.get_constrained_layout_pads

Figure.**get_constrained_layout_pads**(*relative=False*)

[*Deprecated*] Get padding for `constrained_layout`.

Returns a list of `w_pad, h_pad` in inches and `wspace` and `hspace` as fractions of the subplot. All values are None if `constrained_layout` is not used.

See *Constrained layout guide*.

**Parameters**

**relative**

[bool] If `True`, then convert from inches to figure relative.

### Notes

Deprecated since version 3.6: Use fig.get_layout_engine().get() instead.

## Interactive

**See also:**

- *Event handling and picking*

| | |
|---|---|
| *Figure.ginput* | Blocking call to interact with a figure. |
| *Figure.add_axobserver* | Whenever the Axes state change, `func(self)` will be called. |
| *Figure.waitforbuttonpress* | Blocking call to interact with the figure. |
| *Figure.pick* | Process a pick event. |

## matplotlib.figure.Figure.ginput

Figure.**ginput**(*n=1*, *timeout=30*, *show_clicks=True*, *mouse_add=MouseButton.LEFT*, *mouse_pop=MouseButton.RIGHT*, *mouse_stop=MouseButton.MIDDLE*)

Blocking call to interact with a figure.

Wait until the user clicks *n* times on the figure, and return the coordinates of each click in a list.

There are three possible interactions:

- Add a point.
- Remove the most recently added point.
- Stop the interaction and return the points added so far.

The actions are assigned to mouse buttons via the arguments *mouse_add*, *mouse_pop* and *mouse_stop*.

**Parameters**

**n**

[int, default: 1] Number of mouse clicks to accumulate. If negative, accumulate clicks until the input is terminated manually.

**timeout**

[float, default: 30 seconds] Number of seconds to wait before timing out. If zero or negative will never time out.

**show_clicks**

[bool, default: True] If True, show a red cross at the location of each click.

**mouse_add**

[*MouseButton* or None, default: *MouseButton.LEFT*] Mouse button used to add points.

**mouse_pop**

[*MouseButton* or None, default: *MouseButton.RIGHT*] Mouse button used to remove the most recently added point.

**mouse_stop**

[*MouseButton* or None, default: *MouseButton.MIDDLE*] Mouse button used to stop input.

**Returns**

**list of tuples**

A list of the clicked (x, y) coordinates.

### Notes

The keyboard can also be used to select points in case your mouse does not have one or more of the buttons. The delete and backspace keys act like right-clicking (i.e., remove last point), the enter key terminates input and any other key (not already used by the window manager) selects a point.

## matplotlib.figure.Figure.add_axobserver

`Figure.`**`add_axobserver`**(*func*)

Whenever the Axes state change, `func(self)` will be called.

## matplotlib.figure.Figure.waitforbuttonpress

`Figure.`**`waitforbuttonpress`**(*timeout=-1*)

Blocking call to interact with the figure.

Wait for user input and return True if a key was pressed, False if a mouse button was pressed and None if no input was given within *timeout* seconds. Negative values deactivate *timeout*.

### matplotlib.figure.Figure.pick

Figure.**pick**(*mouseevent*)

Process a pick event.

Each child artist will fire a pick event if *mouseevent* is over the artist and the artist has picker set.

**See also:**

***Artist.set_picker***, ***Artist.get_picker***, ***Artist.pickable***

## Modifying appearance

| | |
|---|---|
| *Figure.set_frameon* | Set the figure's background patch visibility, i.e. whether the figure background will be drawn. |
| *Figure.get_frameon* | Return the figure's background patch visibility, i.e. whether the figure background will be drawn. |
| *Figure.set_linewidth* | Set the line width of the Figure rectangle. |
| *Figure.get_linewidth* | Get the line width of the Figure rectangle. |
| *Figure.set_facecolor* | Set the face color of the Figure rectangle. |
| *Figure.get_facecolor* | Get the face color of the Figure rectangle. |
| *Figure.set_edgecolor* | Set the edge color of the Figure rectangle. |
| *Figure.get_edgecolor* | Get the edge color of the Figure rectangle. |

### matplotlib.figure.Figure.set_frameon

Figure.**set_frameon**(*b*)

Set the figure's background patch visibility, i.e. whether the figure background will be drawn. Equivalent to `Figure.patch.set_visible()`.

**Parameters**

**b**

[bool]

### matplotlib.figure.Figure.get_frameon

Figure.**get_frameon**()

Return the figure's background patch visibility, i.e. whether the figure background will be drawn. Equivalent to `Figure.patch.get_visible()`.

### matplotlib.figure.Figure.set_linewidth

Figure.**set_linewidth**(*linewidth*)

Set the line width of the Figure rectangle.

**Parameters**

**linewidth**

[number]

### matplotlib.figure.Figure.get_linewidth

Figure.**get_linewidth**()

Get the line width of the Figure rectangle.

### matplotlib.figure.Figure.set_facecolor

Figure.**set_facecolor**(*color*)

Set the face color of the Figure rectangle.

**Parameters**

**color**

[*color*]

### matplotlib.figure.Figure.get_facecolor

Figure.**get_facecolor**()

Get the face color of the Figure rectangle.

### matplotlib.figure.Figure.set_edgecolor

Figure.**set_edgecolor**(*color*)

Set the edge color of the Figure rectangle.

**Parameters**

**color**

[*color*]

### matplotlib.figure.Figure.get_edgecolor

Figure.**get_edgecolor**()

Get the edge color of the Figure rectangle.

### Adding and getting Artists

| | |
|---|---|
| *Figure.add_artist* | Add an *Artist* to the figure. |
| *Figure.get_children* | Get a list of artists contained in the figure. |
| *Figure.figimage* | Add a non-resampled image to the figure. |

### matplotlib.figure.Figure.add_artist

Figure.**add_artist**(*artist*, *clip=False*)

Add an *Artist* to the figure.

Usually artists are added to *Axes* objects using *Axes.add_artist*; this method can be used in the rare cases where one needs to add artists directly to the figure instead.

**Parameters**

**artist**

[*Artist*] The artist to add to the figure. If the added artist has no transform previously set, its transform will be set to `figure.transSubfigure`.

**clip**

[bool, default: False] Whether the added artist should be clipped by the figure patch.

**Returns**

***Artist***

The added artist.

### Examples using `matplotlib.figure.Figure.add_artist`

- *Multiple Axes animation*
- *Adding lines to figures*
- *Constrained layout guide*
- *Annotations*

### matplotlib.figure.Figure.get_children

Figure.**get_children**()

Get a list of artists contained in the figure.

### matplotlib.figure.Figure.figimage

Figure.**figimage**(*X*, *xo=0*, *yo=0*, *alpha=None*, *norm=None*, *cmap=None*, *vmin=None*, *vmax=None*, *origin=None*, *resize=False*, ***kwargs*)

Add a non-resampled image to the figure.

The image is attached to the lower or upper left corner depending on *origin*.

**Parameters**

**X**

The image data. This is an array of one of the following shapes:

- (M, N): an image with scalar data. Color-mapping is controlled by *cmap*, *norm*, *vmin*, and *vmax*.
- (M, N, 3): an image with RGB values (0-1 float or 0-255 int).
- (M, N, 4): an image with RGBA values (0-1 float or 0-255 int), i.e. including transparency.

**xo, yo**

[int] The *x*/*y* image offset in pixels.

**alpha**

[None or float] The alpha blending value.

**cmap**

[str or *Colormap*, default: *rcParams["image.cmap"]* (default: `'viridis'`)] The Colormap instance or registered colormap name used to map scalar data to colors.

This parameter is ignored if *X* is RGB(A).

**norm**

[str or *Normalize*, optional] The normalization method used to scale scalar data to the [0, 1] range before mapping to colors using *cmap*. By default, a linear scaling is used, mapping the lowest value to 0 and the highest to 1.

If given, this can be one of the following:

- An instance of *Normalize* or one of its subclasses (see *Colormap normalization*).

- A scale name, i.e. one of "linear", "log", "symlog", "logit", etc. For a list of available scales, call *matplotlib.scale.get_scale_names()*. In that case, a suitable *Normalize* subclass is dynamically generated and instantiated.

This parameter is ignored if *X* is RGB(A).

**vmin, vmax**

[float, optional] When using scalar data and no explicit *norm*, *vmin* and *vmax* define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data. It is an error to use *vmin*/*vmax* when a *norm* instance is given (but using a `str` *norm* name together with *vmin*/*vmax* is acceptable).

This parameter is ignored if *X* is RGB(A).

**origin**

[{'upper', 'lower'}, default: *rcParams["image.origin"]* (default: `'upper'`)] Indicates where the [0, 0] index of the array is in the upper left or lower left corner of the Axes.

**resize**

[bool] If *True*, resize the figure to match the given image size.

**Returns**

***matplotlib.image.FigureImage***

**Other Parameters**

****kwargs**

Additional kwargs are *Artist* kwargs passed on to *FigureImage*.

## Notes

figimage complements the Axes image (*imshow*) which will be resampled to fit the current Axes. If you want a resampled image to fill the entire figure, you can define an *Axes* with extent [0, 0, 1, 1].

## Examples

```
f = plt.figure()
nx = int(f.get_figwidth() * f.dpi)
ny = int(f.get_figheight() * f.dpi)
data = np.random.random((ny, nx))
f.figimage(data)
plt.show()
```

### Examples using `matplotlib.figure.Figure.figimage`

- *Figimage Demo*
- *Watermark image*
- *Convert texts to images*

### Getting and modifying state

**See also:**

- *Interactive figures*

| | |
|---|---|
| `Figure.clear` | Clear the figure. |
| `Figure.gca` | Get the current Axes. |
| `Figure.sca` | Set the current Axes to be *a* and return *a*. |
| `Figure.get_tightbbox` | Return a (tight) bounding box of the figure *in inches*. |
| `Figure.get_window_extent` | Get the artist's bounding box in display space. |
| `Figure.show` | If using a GUI backend with pyplot, display the figure window. |
| `Figure.set_canvas` | Set the canvas that contains the figure |
| `Figure.draw` | Draw the Artist (and its children) using the given renderer. |
| `Figure.draw_without_rendering` | Draw the figure with no output. |
| `Figure.draw_artist` | Draw `Artist` *a* only. |

### matplotlib.figure.Figure.clear

Figure.**clear**(*keep_observers=False*)

Clear the figure.

**Parameters**

**keep_observers**

[bool, default: False] Set *keep_observers* to True if, for example, a gui widget is tracking the Axes in the figure.

### matplotlib.figure.Figure.gca

Figure.**gca**()

Get the current Axes.

If there is currently no Axes on this Figure, a new one is created using *Figure.add_subplot*. (To test whether there is currently an Axes on a Figure, check whether `figure.axes` is empty. To test whether there is currently a Figure on the pyplot figure stack, check whether *pyplot.get_fignums()* is empty.)

### matplotlib.figure.Figure.sca

Figure.**sca**(*a*)

Set the current Axes to be *a* and return *a*.

### matplotlib.figure.Figure.get_tightbbox

Figure.**get_tightbbox**(*renderer=None*, *, *bbox_extra_artists=None*)

Return a (tight) bounding box of the figure *in inches*.

Note that *FigureBase* differs from all other artists, which return their *Bbox* in pixels.

Artists that have `artist.set_in_layout(False)` are not included in the bbox.

**Parameters**

**renderer**

[*RendererBase* subclass] Renderer that will be used to draw the figures (i.e. `fig.canvas.get_renderer()`)

**bbox_extra_artists**

[list of *Artist* or `None`] List of artists to include in the tight bounding box. If `None` (default), then all artist children of each Axes are included in the tight bounding box.

**Returns**

***BboxBase***

containing the bounding box (in figure inches).

### matplotlib.figure.Figure.get_window_extent

Figure.**get_window_extent**(*renderer=None*)

Get the artist's bounding box in display space.

The bounding box' width and height are nonnegative.

Subclasses should override for inclusion in the bounding box "tight" calculation. Default is to return an empty bounding box at 0, 0.

Be careful when using this function, the results will not update if the artist window extent of the artist changes. The extent can change due to any changes in the transform stack, such as changing the Axes limits, the figure size, or the canvas used (as is done when saving a figure). This can lead to unexpected behavior where interactive figures will look fine on the screen, but will save incorrectly.

### matplotlib.figure.Figure.show

Figure.**show**(*warn=True*)

If using a GUI backend with pyplot, display the figure window.

If the figure was not created using *figure*, it will lack a *FigureManagerBase*, and this method will raise an AttributeError.

> **Warning:** This does not manage an GUI event loop. Consequently, the figure may only be shown briefly or not shown at all if you or your environment are not managing an event loop.
>
> Use cases for *Figure.show* include running this from a GUI application (where there is persistently an event loop running) or from a shell, like IPython, that install an input hook to allow the interactive shell to accept input while the figure is also being shown and interactive. Some, but not all, GUI toolkits will register an input hook on import. See *Command prompt integration* for more details.
>
> If you're in a shell without input hook integration or executing a python script, you should use *matplotlib.pyplot.show* with `block=True` instead, which takes care of starting and running the event loop for you.

**Parameters**

**warn**

[bool, default: True] If `True` and we are not running headless (i.e. on Linux with an unset DISPLAY), issue warning when called on a non-GUI backend.

### Examples using `matplotlib.figure.Figure.show`

- *Polygon Selector*

## matplotlib.figure.Figure.set_canvas

`Figure.`**`set_canvas`**(*canvas*)

Set the canvas that contains the figure

**Parameters**

**canvas**

[FigureCanvas]

## matplotlib.figure.Figure.draw

`Figure.`**`draw`**(*renderer*)

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*`Artist.get_visible`* returns False).

**Parameters**

**renderer**

[*`RendererBase`* subclass.]

### Notes

This method is overridden in the Artist subclasses.

## matplotlib.figure.Figure.draw_without_rendering

`Figure.`**`draw_without_rendering`**()

Draw the figure with no output. Useful to get the final size of artists that require a draw before their size is known (e.g. text).

### matplotlib.figure.Figure.draw_artist

Figure.**draw_artist**(*a*)

Draw *Artist* *a* only.

### Examples using `matplotlib.figure.Figure.draw_artist`

- *Faster rendering by using blitting*

## SubFigure

Matplotlib has the concept of a *SubFigure*, which is a logical figure inside a parent *Figure*. It has many of the same methods as the parent. See *Nested Axes layouts*.

## SubFigure class

| | |
|---|---|
| *SubFigure* | Logical figure that can be placed inside a figure. |

### matplotlib.figure.SubFigure

**class** matplotlib.figure.**SubFigure**(*parent*, *subplotspec*, *, *facecolor=None*, *edgecolor=None*, *linewidth=0.0*, *frameon=None*, ***kwargs*)

Logical figure that can be placed inside a figure.

See *SubFigure* for an index of methods on this class. Typically instantiated using *Figure.add_subfigure* or *SubFigure.add_subfigure*, or *SubFigure.subfigures*. A subfigure has the same methods as a figure except for those particularly tied to the size or dpi of the figure, and is confined to a prescribed region of the figure. For example the following puts two subfigures side-by-side:

```
fig = plt.figure()
sfigs = fig.subfigures(1, 2)
axsL = sfigs[0].subplots(1, 2)
axsR = sfigs[1].subplots(2, 1)
```

See *Figure subfigures*

**Note:** The *subfigure* concept is new in v3.4, and the API is still provisional.

**Parameters**

**parent**

[*Figure* or *SubFigure*] Figure or subfigure that contains the SubFigure. SubFigures can be nested.

**subplotspec**

[*gridspec.SubplotSpec*] Defines the region in a parent gridspec where the subfigure will be placed.

**facecolor**

[default: `"none"`] The figure patch face color; transparent by default.

**edgecolor**

[default: *rcParams["figure.edgecolor"]* (default: `'white'`)] The figure patch edge color.

**linewidth**

[float] The linewidth of the frame (i.e. the edge linewidth of the figure patch).

**frameon**

[bool, default: *rcParams["figure.frameon"]* (default: `True`)] If `False`, suppress drawing the figure background patch.

**Other Parameters**

****kwargs**

[*SubFigure* properties, optional]

| Property | Description |
|---|---|
| agg_fil | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image |
| alpha | scalar or None |
| animated | bool |
| clip_bo | BboxBase or None |
| clip_on | bool |
| clip_pa | Patch or (Path, Transform) or None |
| dpi | float |
| edgecolor | color |
| facecolor | color |
| figure | Figure |
| frameon | bool |
| gid | str |
| in_layo | bool |
| label | object |
| linewid | number |
| mouseov | bool |
| path_ef | list of AbstractPathEffect |
| picker | None or bool or float or callable |
| rasterized | bool |
| sketch_ | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| transform | Transform |
| url | str |
| visible | bool |
| zorder | float |

### Adding Axes and SubFigures

| | |
|---|---|
| *SubFigure.add_axes* | Add an *Axes* to the figure. |
| *SubFigure.add_subplot* | Add an *Axes* to the figure as part of a subplot arrangement. |
| *SubFigure.subplots* | Add a set of subplots to this figure. |
| *SubFigure.subplot_mosaic* | Build a layout of Axes based on ASCII art or nested lists. |
| *SubFigure.add_gridspec* | Low-level API for creating a *GridSpec* that has this figure as a parent. |
| *SubFigure.delaxes* | Remove the *Axes* *ax* from the figure; update the current Axes. |
| *SubFigure.add_subfigure* | Add a *SubFigure* to the figure as part of a subplot arrangement. |
| *SubFigure.subfigures* | Add a set of subfigures to this figure or subfigure. |

### matplotlib.figure.SubFigure.add_axes

SubFigure.**add_axes**(**args*, ***kwargs*)

Add an *Axes* to the figure.

Call signatures:

```
add_axes(rect, projection=None, polar=False, **kwargs)
add_axes(ax)
```

**Parameters**

**rect**

[tuple (left, bottom, width, height)] The dimensions (left, bottom, width, height) of the new *Axes*. All quantities are in fractions of figure width and height.

**projection**

[{None, 'aitoff', 'hammer', 'lambert', 'mollweide', 'polar', 'rectilinear', str}, optional] The projection type of the *Axes*. *str* is the name of a custom projection, see *projections*. The default None results in a 'rectilinear' projection.

**polar**

[bool, default: False] If True, equivalent to projection='polar'.

**axes_class**

[subclass type of *Axes*, optional] The *axes.Axes* subclass that is instantiated. This parameter is incompatible with *projection* and *polar*. See *axisartist* for examples.

**sharex, sharey**

[*Axes*, optional] Share the x or y *axis* with sharex and/or sharey. The axis will have the same limits, ticks, and scale as the axis of the shared Axes.

**label**

[str] A label for the returned Axes.

### Returns

***Axes*, or a subclass of *Axes***

The returned Axes class depends on the projection used. It is *Axes* if rectilinear projection is used and *projections.polar.PolarAxes* if polar projection is used.

### Other Parameters

****kwargs**

This method also takes the keyword arguments for the returned Axes class. The keyword arguments for the rectilinear Axes class *Axes* can be found in the following table but there might also be other keyword arguments if another projection is used, see the actual Axes class.

| Property | Description |
|---|---|
| *adjustable* | {'box', 'datalim'} |
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi v |
| *alpha* | scalar or None |
| *anchor* | (float, float) or {'C', 'SW', 'S', 'SE', 'E', 'NE', ...} |
| *animated* | bool |
| *aspect* | {'auto', 'equal'} or float |
| *autoscale_on* | bool |
| *autoscalex_on* | unknown |
| *autoscaley_on* | unknown |
| *axes_locator* | Callable[[Axes, Renderer], Bbox] |
| *axisbelow* | bool or 'line' |
| *box_aspect* | float or None |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *facecolor* or fc | *color* |
| *figure* | *Figure* |
| *forward_navigation_events* | bool or "auto" |
| *frame_on* | bool |
| *gid* | str |
| *in_layout* | bool |

Table 53 – continued from

| Property | Description |
|---|---|
| *label* | object |
| *mouseover* | bool |
| *navigate* | bool |
| *navigate_mode* | unknown |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *position* | [left, bottom, width, height] or *Bbox* |
| *prop_cycle* | *Cycler* |
| *rasterization_zorder* | float or None |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *subplotspec* | unknown |
| *title* | str |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *xbound* | (lower: float, upper: float) |
| *xlabel* | str |
| *xlim* | (left: float, right: float) |
| *xmargin* | float greater than -0.5 |
| *xscale* | unknown |
| *xticklabels* | unknown |
| *xticks* | unknown |
| *ybound* | (lower: float, upper: float) |
| *ylabel* | str |
| *ylim* | (bottom: float, top: float) |
| *ymargin* | float greater than -0.5 |
| *yscale* | unknown |
| *yticklabels* | unknown |
| *yticks* | unknown |
| *zorder* | float |

**See also:**

***Figure.add_subplot***
***pyplot.subplot***
***pyplot.axes***
***Figure.subplots***
***pyplot.subplots***

### Notes

In rare circumstances, *add_axes* may be called with a single argument, an Axes instance already created in the present figure but not in the figure's list of Axes.

### Examples

Some simple examples:

```
rect = l, b, w, h
fig = plt.figure()
fig.add_axes(rect)
fig.add_axes(rect, frameon=False, facecolor='g')
fig.add_axes(rect, polar=True)
ax = fig.add_axes(rect, projection='polar')
fig.delaxes(ax)
fig.add_axes(ax)
```

## matplotlib.figure.SubFigure.add_subplot

SubFigure.**add_subplot**(**args*, ***kwargs*)

Add an *Axes* to the figure as part of a subplot arrangement.

Call signatures:

```
add_subplot(nrows, ncols, index, **kwargs)
add_subplot(pos, **kwargs)
add_subplot(ax)
add_subplot()
```

**Parameters**

***args**

[int, (int, int, *index*), or *SubplotSpec*, default: (1, 1, 1)] The position of the subplot described by one of

- Three integers (*nrows*, *ncols*, *index*). The subplot will take the *index* position on a grid with *nrows* rows and *ncols* columns. *index* starts at 1 in the upper left corner and increases to the right. *index* can also be a two-tuple specifying the (*first*, *last*) indices (1-based, and including *last*) of the subplot, e.g., `fig.add_subplot(3, 1, (1, 2))` makes a subplot that spans the upper 2/3 of the figure.
- A 3-digit integer. The digits are interpreted as if given separately as three single-digit integers, i.e. `fig.add_subplot(235)` is the same as `fig.add_subplot(2, 3, 5)`. Note that this can only be used if there are no more than 9 subplots.

- A *SubplotSpec*.

In rare circumstances, *add_subplot* may be called with a single argument, a subplot Axes instance already created in the present figure but not in the figure's list of Axes.

**projection**

[{None, 'aitoff', 'hammer', 'lambert', 'mollweide', 'polar', 'rectilinear', str}, optional] The projection type of the subplot (*Axes*). *str* is the name of a custom projection, see *projections*. The default None results in a 'rectilinear' projection.

**polar**

[bool, default: False] If True, equivalent to projection='polar'.

**axes_class**

[subclass type of *Axes*, optional] The *axes.Axes* subclass that is instantiated. This parameter is incompatible with *projection* and *polar*. See *axisartist* for examples.

**sharex, sharey**

[*Axes*, optional] Share the x or y *axis* with sharex and/or sharey. The axis will have the same limits, ticks, and scale as the axis of the shared Axes.

**label**

[str] A label for the returned Axes.

### Returns

***Axes***

The Axes of the subplot. The returned Axes can actually be an instance of a subclass, such as *projections.polar.PolarAxes* for polar projections.

### Other Parameters

****kwargs**

This method also takes the keyword arguments for the returned Axes base class; except for the *figure* argument. The keyword arguments for the rectilinear base class *Axes* can be found in the following table but there might also be other keyword arguments if another projection is used.

| Property | Description |
|---|---|
| *adjustable* | {'box', 'datalim'} |
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi v |
| *alpha* | scalar or None |

Table 54 – continued from

| Property | Description |
|---|---|
| *anchor* | (float, float) or {'C', 'SW', 'S', 'SE', 'E', 'NE', ...} |
| *animated* | bool |
| *aspect* | {'auto', 'equal'} or float |
| *autoscale_on* | bool |
| *autoscalex_on* | unknown |
| *autoscaley_on* | unknown |
| *axes_locator* | Callable[[Axes, Renderer], Bbox] |
| *axisbelow* | bool or 'line' |
| *box_aspect* | float or None |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *facecolor* or fc | *color* |
| *figure* | *Figure* |
| *forward_navigation_events* | bool or "auto" |
| *frame_on* | bool |
| *gid* | str |
| *in_layout* | bool |
| *label* | object |
| *mouseover* | bool |
| *navigate* | bool |
| *navigate_mode* | unknown |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *position* | [left, bottom, width, height] or *Bbox* |
| *prop_cycle* | *Cycler* |
| *rasterization_zorder* | float or None |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *subplotspec* | unknown |
| *title* | str |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *xbound* | (lower: float, upper: float) |
| *xlabel* | str |
| *xlim* | (left: float, right: float) |
| *xmargin* | float greater than -0.5 |
| *xscale* | unknown |
| *xticklabels* | unknown |
| *xticks* | unknown |
| *ybound* | (lower: float, upper: float) |
| *ylabel* | str |

Table 54 – continued from

| Property | Description |
|---|---|
| *ylim* | (bottom: float, top: float) |
| *ymargin* | float greater than -0.5 |
| *yscale* | unknown |
| *yticklabels* | unknown |
| *yticks* | unknown |
| *zorder* | float |

**See also:**

***Figure.add_axes***
***pyplot.subplot***
***pyplot.axes***
***Figure.subplots***
***pyplot.subplots***

### Examples

```
fig = plt.figure()

fig.add_subplot(231)
ax1 = fig.add_subplot(2, 3, 1)  # equivalent but more general

fig.add_subplot(232, frameon=False)  # subplot with no frame
fig.add_subplot(233, projection='polar')  # polar subplot
fig.add_subplot(234, sharex=ax1)  # subplot sharing x-axis with ax1
fig.add_subplot(235, facecolor="red")  # red subplot

ax1.remove()  # delete ax1 from the figure
fig.add_subplot(ax1)  # add ax1 back to the figure
```

## matplotlib.figure.SubFigure.subplots

SubFigure.**subplots**(*nrows=1*, *ncols=1*, ***, *sharex=False*, *sharey=False*, *squeeze=True*, *width_ratios=None*, *height_ratios=None*, *subplot_kw=None*, *gridspec_kw=None*)

Add a set of subplots to this figure.

This utility wrapper makes it convenient to create common layouts of subplots in a single call.

**Parameters**

**nrows, ncols**

[int, default: 1] Number of rows/columns of the subplot grid.

**sharex, sharey**

[bool or {'none', 'all', 'row', 'col'}, default: False] Controls sharing of x-axis (*sharex*) or y-axis (*sharey*):

- True or 'all': x- or y-axis will be shared among all subplots.
- False or 'none': each subplot x- or y-axis will be independent.
- 'row': each subplot row will share an x- or y-axis.
- 'col': each subplot column will share an x- or y-axis.

When subplots have a shared x-axis along a column, only the x tick labels of the bottom subplot are created. Similarly, when subplots have a shared y-axis along a row, only the y tick labels of the first column subplot are created. To later turn other subplots' ticklabels on, use *tick_params*.

When subplots have a shared axis that has units, calling *Axis.set_units* will update each axis with the new units.

**squeeze**

[bool, default: True]

- If True, extra dimensions are squeezed out from the returned array of Axes:
  - if only one subplot is constructed (nrows=ncols=1), the resulting single Axes object is returned as a scalar.
  - for Nx1 or 1xM subplots, the returned object is a 1D numpy object array of Axes objects.
  - for NxM, subplots with N>1 and M>1 are returned as a 2D array.
- If False, no squeezing at all is done: the returned Axes object is always a 2D array containing Axes instances, even if it ends up being 1x1.

**width_ratios**

[array-like of length *ncols*, optional] Defines the relative widths of the columns. Each column gets a relative width of `width_ratios[i] / sum(width_ratios)`. If not given, all columns will have the same width. Equivalent to `gridspec_kw={'width_ratios': [...]}`.

**height_ratios**

[array-like of length *nrows*, optional] Defines the relative heights of the rows. Each row gets a relative height of `height_ratios[i] / sum(height_ratios)`. If not given, all rows will have the same height. Equivalent to `gridspec_kw={'height_ratios': [...]}`.

**subplot_kw**

[dict, optional] Dict with keywords passed to the *Figure.add_subplot* call used to create each subplot.

**gridspec_kw**

[dict, optional] Dict with keywords passed to the *GridSpec* constructor used to create the grid the subplots are placed on.

**Returns**

***Axes* or array of Axes**

Either a single *Axes* object or an array of Axes objects if more than one subplot was created. The dimensions of the resulting array can be controlled with the *squeeze* keyword, see above.

**See also:**

***pyplot.subplots***
***Figure.add_subplot***
***pyplot.subplot***

## Examples

```
# First create some toy data:
x = np.linspace(0, 2*np.pi, 400)
y = np.sin(x**2)

# Create a figure
fig = plt.figure()

# Create a subplot
ax = fig.subplots()
ax.plot(x, y)
ax.set_title('Simple plot')

# Create two subplots and unpack the output array immediately
ax1, ax2 = fig.subplots(1, 2, sharey=True)
ax1.plot(x, y)
ax1.set_title('Sharing Y axis')
ax2.scatter(x, y)

# Create four polar Axes and access them through the returned array
axes = fig.subplots(2, 2, subplot_kw=dict(projection='polar'))
axes[0, 0].plot(x, y)
axes[1, 1].scatter(x, y)

# Share an X-axis with each column of subplots
fig.subplots(2, 2, sharex='col')

# Share a Y-axis with each row of subplots
fig.subplots(2, 2, sharey='row')
```

(continues on next page)

(continued from previous page)

```
# Share both X- and Y-axes with all subplots
fig.subplots(2, 2, sharex='all', sharey='all')

# Note that this is the same as
fig.subplots(2, 2, sharex=True, sharey=True)
```

### Examples using `matplotlib.figure.SubFigure.subplots`

- *Figure subfigures*
- *Tick formatters*
- *Axis ticks*

## matplotlib.figure.SubFigure.subplot_mosaic

SubFigure.**subplot_mosaic**(*mosaic*, *, *sharex=False*, *sharey=False*, *width_ratios=None*, *height_ratios=None*, *empty_sentinel='.'*, *subplot_kw=None*, *per_subplot_kw=None*, *gridspec_kw=None*)

Build a layout of Axes based on ASCII art or nested lists.

This is a helper function to build complex GridSpec layouts visually.

See *Complex and semantic figure composition (subplot_mosaic)* for an example and full API documentation

**Parameters**

**mosaic**

[list of list of {hashable or nested} or str] A visual layout of how you want your Axes to be arranged labeled as strings. For example

```
x = [['A panel', 'A panel', 'edge'],
     ['C panel', '.',       'edge']]
```

produces 4 Axes:

- 'A panel' which is 1 row high and spans the first two columns
- 'edge' which is 2 rows high and is on the right edge
- 'C panel' which in 1 row and 1 column wide in the bottom left
- a blank space 1 row and 1 column wide in the bottom center

Any of the entries in the layout can be a list of lists of the same form to create nested layouts.

If input is a str, then it can either be a multi-line string of the form

```
'''
AAE
C.E
'''
```

where each character is a column and each line is a row. Or it can be a single-line string where rows are separated by `;`:

```
'AB;CC'
```

The string notation allows only single character Axes labels and does not support nesting but is very terse.

The Axes identifiers may be `str` or a non-iterable hashable object (e.g. `tuple` s may not be used).

**sharex, sharey**

[bool, default: False] If True, the x-axis (*sharex*) or y-axis (*sharey*) will be shared among all subplots. In that case, tick label visibility and axis units behave as for *subplots*. If False, each subplot's x- or y-axis will be independent.

**width_ratios**

[array-like of length *ncols*, optional] Defines the relative widths of the columns. Each column gets a relative width of `width_ratios[i] / sum(width_ratios)`. If not given, all columns will have the same width. Equivalent to `gridspec_kw={'width_ratios': [...]}`. In the case of nested layouts, this argument applies only to the outer layout.

**height_ratios**

[array-like of length *nrows*, optional] Defines the relative heights of the rows. Each row gets a relative height of `height_ratios[i] / sum(height_ratios)`. If not given, all rows will have the same height. Equivalent to `gridspec_kw={'height_ratios': [...]}`. In the case of nested layouts, this argument applies only to the outer layout.

**subplot_kw**

[dict, optional] Dictionary with keywords passed to the *Figure.add_subplot* call used to create each subplot. These values may be overridden by values in *per_subplot_kw*.

**per_subplot_kw**

[dict, optional] A dictionary mapping the Axes identifiers or tuples of identifiers to a dictionary of keyword arguments to be passed to the *Figure.add_subplot* call used to create each subplot. The values in these dictionaries have precedence over the values in *subplot_kw*.

If *mosaic* is a string, and thus all keys are single characters, it is possible to use a single string instead of a tuple as keys; i.e. `"AB"` is equivalent to `("A", "B")`.

New in version 3.7.

**gridspec_kw**

[dict, optional] Dictionary with keywords passed to the *GridSpec* constructor used to create the grid the subplots are placed on. In the case of nested layouts, this argument applies only to the outer layout. For more complex layouts, users should use *Figure.subfigures* to create the nesting.

**empty_sentinel**

[object, optional] Entry in the layout to mean "leave this space empty". Defaults to `'.'`. Note, if *layout* is a string, it is processed via `inspect.cleandoc` to remove leading white space, which may interfere with using white-space as the empty sentinel.

**Returns**

**dict[label, Axes]**

A dictionary mapping the labels to the Axes objects. The order of the Axes is left-to-right and top-to-bottom of their position in the total layout.

## Examples using `matplotlib.figure.SubFigure.subplot_mosaic`

- *Complex and semantic figure composition (subplot_mosaic)*

## matplotlib.figure.SubFigure.add_gridspec

SubFigure.**add_gridspec**(*nrows=1*, *ncols=1*, ***kwargs*)

Low-level API for creating a *GridSpec* that has this figure as a parent.

This is a low-level API, allowing you to create a gridspec and subsequently add subplots based on the gridspec. Most users do not need that freedom and should use the higher-level methods *subplots* or *subplot_mosaic*.

**Parameters**

**nrows**

[int, default: 1] Number of rows in grid.

**ncols**

[int, default: 1] Number of columns in grid.

**Returns**

*GridSpec*

**Other Parameters**

****kwargs**

Keyword arguments are passed to *GridSpec*.

**See also:**

***matplotlib.pyplot.subplots***

### Examples

Adding a subplot that spans two rows:

```
fig = plt.figure()
gs = fig.add_gridspec(2, 2)
ax1 = fig.add_subplot(gs[0, 0])
ax2 = fig.add_subplot(gs[1, 0])
# spans two rows:
ax3 = fig.add_subplot(gs[:, 1])
```

## matplotlib.figure.SubFigure.delaxes

SubFigure.**delaxes**(*ax*)

Remove the *Axes* *ax* from the figure; update the current Axes.

## matplotlib.figure.SubFigure.add_subfigure

SubFigure.**add_subfigure**(*subplotspec*, ***kwargs*)

Add a *SubFigure* to the figure as part of a subplot arrangement.

**Parameters**

**subplotspec**

[*gridspec.SubplotSpec*] Defines the region in a parent gridspec where the subfigure will be placed.

**Returns**

***SubFigure***

**Other Parameters**

****kwargs**

Are passed to the *SubFigure* object.

**See also:**

*Figure.subfigures*

## matplotlib.figure.SubFigure.subfigures

SubFigure.**subfigures**(*nrows=1*, *ncols=1*, *squeeze=True*, *wspace=None*, *hspace=None*, *width_ratios=None*, *height_ratios=None*, ***kwargs*)

Add a set of subfigures to this figure or subfigure.

A subfigure has the same artist methods as a figure, and is logically the same as a figure, but cannot print itself. See *Figure subfigures*.

**Note:** The *subfigure* concept is new in v3.4, and the API is still provisional.

### Parameters

**nrows, ncols**

[int, default: 1] Number of rows/columns of the subfigure grid.

**squeeze**

[bool, default: True] If True, extra dimensions are squeezed out from the returned array of subfigures.

**wspace, hspace**

[float, default: None] The amount of width/height reserved for space between subfigures, expressed as a fraction of the average subfigure width/height. If not given, the values will be inferred from rcParams if using constrained layout (see *ConstrainedLayoutEngine*), or zero if not using a layout engine.

**width_ratios**

[array-like of length *ncols*, optional] Defines the relative widths of the columns. Each column gets a relative width of `width_ratios[i] / sum(width_ratios)`. If not given, all columns will have the same width.

**height_ratios**

[array-like of length *nrows*, optional] Defines the relative heights of the rows. Each row gets a relative height of `height_ratios[i] / sum(height_ratios)`. If not given, all rows will have the same height.

### Annotating

| | |
|---|---|
| *SubFigure.colorbar* | Add a colorbar to a plot. |
| *SubFigure.legend* | Place a legend on the figure. |
| *SubFigure.text* | Add text to figure. |
| *SubFigure.suptitle* | Add a centered suptitle to the figure. |
| *SubFigure.get_suptitle* | Return the suptitle as string or an empty string if not set. |
| *SubFigure.supxlabel* | Add a centered supxlabel to the figure. |
| *SubFigure.get_supxlabel* | Return the supxlabel as string or an empty string if not set. |
| *SubFigure.supylabel* | Add a centered supylabel to the figure. |
| *SubFigure.get_supylabel* | Return the supylabel as string or an empty string if not set. |
| *SubFigure.align_labels* | Align the xlabels and ylabels of subplots with the same subplots row or column (respectively) if label alignment is being done automatically (i.e. the label position is not manually set). |
| *SubFigure.align_xlabels* | Align the xlabels of subplots in the same subplot row if label alignment is being done automatically (i.e. the label position is not manually set). |
| *SubFigure.align_ylabels* | Align the ylabels of subplots in the same subplot column if label alignment is being done automatically (i.e. the label position is not manually set). |
| *SubFigure.align_titles* | Align the titles of subplots in the same subplot row if title alignment is being done automatically (i.e. the title position is not manually set). |

### matplotlib.figure.SubFigure.colorbar

SubFigure.**colorbar**(*mappable*, *cax=None*, *ax=None*, *use_gridspec=True*, ***kwargs*)

Add a colorbar to a plot.

**Parameters**

**mappable**

The *matplotlib.cm.ScalarMappable* (i.e., *AxesImage*, *ContourSet*, etc.) described by this colorbar. This argument is mandatory for the *Figure.colorbar* method but optional for the *pyplot.colorbar* function, which sets the default to the current image.

Note that one can create a *ScalarMappable* "on-the-fly" to generate colorbars not attached to a previously drawn artist, e.g.

```
fig.colorbar(cm.ScalarMappable(norm=norm, cmap=cmap),␣
↪ax=ax)
```

**cax**

[*Axes*, optional] Axes into which the colorbar will be drawn. If `None`, then a new Axes is created and the space for it will be stolen from the Axes(s) specified in *ax*.

**ax**

[*Axes* or iterable or `numpy.ndarray` of Axes, optional] The one or more parent Axes from which space for a new colorbar Axes will be stolen. This parameter is only used if *cax* is not set.

Defaults to the Axes that contains the mappable used to create the colorbar.

**use_gridspec**

[bool, optional] If *cax* is `None`, a new *cax* is created as an instance of Axes. If *ax* is positioned with a subplotspec and *use_gridspec* is `True`, then *cax* is also positioned with a subplotspec.

**Returns**

**colorbar**

[*Colorbar*]

**Other Parameters**

**location**

[None or {'left', 'right', 'top', 'bottom'}] The location, relative to the parent Axes, where the colorbar Axes is created. It also determines the *orientation* of the colorbar (colorbars on the left and right are vertical, colorbars at the top and bottom are horizontal). If None, the location will come from the *orientation* if it is set (vertical colorbars on the right, horizontal ones at the bottom), or default to 'right' if *orientation* is unset.

**orientation**

[None or {'vertical', 'horizontal'}] The orientation of the colorbar. It is preferable to set the *location* of the colorbar, as that also determines the *orientation*; passing incompatible values for *location* and *orientation* raises an exception.

**fraction**

[float, default: 0.15] Fraction of original Axes to use for colorbar.

**shrink**

[float, default: 1.0] Fraction by which to multiply the size of the colorbar.

**aspect**

[float, default: 20] Ratio of long to short dimensions.

**pad**

[float, default: 0.05 if vertical, 0.15 if horizontal] Fraction of original Axes between colorbar and new image Axes.

**anchor**

[(float, float), optional] The anchor point of the colorbar Axes. Defaults to (0.0, 0.5) if vertical; (0.5, 1.0) if horizontal.

**panchor**

[(float, float), or *False*, optional] The anchor point of the colorbar parent Axes. If *False*, the parent axes' anchor will be unchanged. Defaults to (1.0, 0.5) if vertical; (0.5, 0.0) if horizontal.

**extend**

[{'neither', 'both', 'min', 'max'}] Make pointed end(s) for out-of-range values (unless 'neither'). These are set for a given colormap using the colormap set_under and set_over methods.

**extendfrac**

[{*None*, 'auto', length, lengths}] If set to *None*, both the minimum and maximum triangular colorbar extensions will have a length of 5% of the interior colorbar length (this is the default setting).

If set to 'auto', makes the triangular colorbar extensions the same lengths as the interior boxes (when *spacing* is set to 'uniform') or the same lengths as the respective adjacent interior boxes (when *spacing* is set to 'proportional').

If a scalar, indicates the length of both the minimum and maximum triangular colorbar extensions as a fraction of the interior colorbar length. A two-element sequence of fractions may also be given, indicating the lengths of the minimum and maximum colorbar extensions respectively as a fraction of the interior colorbar length.

**extendrect**

[bool] If *False* the minimum and maximum colorbar extensions will be triangular (the default). If *True* the extensions will be rectangular.

**spacing**

[{'uniform', 'proportional'}] For discrete colorbars (`BoundaryNorm` or contours), 'uniform' gives each color the same space; 'proportional' makes the space proportional to the data interval.

**ticks**

[None or list of ticks or Locator] If None, ticks are determined automatically from the input.

**format**

[None or str or Formatter] If None, *ScalarFormatter* is used. Format strings, e.g., `"%4.2e"` or `"{x:.2e}"`, are supported. An alternative *Formatter* may be given instead.

**drawedges**

[bool] Whether to draw lines at color boundaries.

**label**

[str] The label on the colorbar's long axis.

**boundaries, values**

[None or a sequence] If unset, the colormap will be displayed on a 0-1 scale. If sequences, *values* must have a length 1 less than *boundaries*. For each region delimited by adjacent entries in *boundaries*, the color mapped to the corresponding value in values will be used. Normally only useful for indexed colors (i.e. `norm=NoNorm()`) or other unusual circumstances.

### Notes

If *mappable* is a *ContourSet*, its *extend* kwarg is included automatically.

The *shrink* kwarg provides a simple way to scale the colorbar with respect to the Axes. Note that if *cax* is specified, it determines the size of the colorbar, and *shrink* and *aspect* are ignored.

For more precise control, you can manually specify the positions of the axes objects in which the mappable and the colorbar are drawn. In this case, do not use any of the Axes properties kwargs.

It is known that some vector graphics viewers (svg and pdf) render white gaps between segments of the colorbar. This is due to bugs in the viewers, not Matplotlib. As a workaround, the colorbar can be rendered with overlapping segments:

```
cbar = colorbar()
cbar.solids.set_edgecolor("face")
draw()
```

However, this has negative consequences in other circumstances, e.g. with semi-transparent images (alpha < 1) and colorbar extensions; therefore, this workaround is not used by default (see issue #1188).

### Examples using `matplotlib.figure.SubFigure.colorbar`

- *Figure subfigures*

### matplotlib.figure.SubFigure.legend

SubFigure.**legend**(**args*, ***kwargs*)

Place a legend on the figure.

Call signatures:

```
legend()
legend(handles, labels)
legend(handles=handles)
legend(labels)
```

The call signatures correspond to the following different ways to use this method:

**1. Automatic detection of elements to be shown in the legend**

The elements to be added to the legend are automatically determined, when you do not pass in any extra arguments.

In this case, the labels are taken from the artist. You can specify them either at artist creation or by calling the *set_label()* method on the artist:

```
ax.plot([1, 2, 3], label='Inline label')
fig.legend()
```

or:

```
line, = ax.plot([1, 2, 3])
line.set_label('Label via method')
fig.legend()
```

Specific lines can be excluded from the automatic legend element selection by defining a label starting with an underscore. This is default for all artists, so calling *Figure.legend* without any arguments and without setting the labels manually will result in no legend being drawn.

**2. Explicitly listing the artists and labels in the legend**

For full control of which artists have a legend entry, it is possible to pass an iterable of legend artists followed by an iterable of legend labels respectively:

```
fig.legend([line1, line2, line3], ['label1', 'label2', 'label3'])
```

**3. Explicitly listing the artists in the legend**

This is similar to 2, but the labels are taken from the artists' label properties. Example:

```
line1, = ax1.plot([1, 2, 3], label='label1')
line2, = ax2.plot([1, 2, 3], label='label2')
fig.legend(handles=[line1, line2])
```

**4. Labeling existing plot elements**

---

**Discouraged**

This call signature is discouraged, because the relation between plot elements and labels is only implicit by their order and can easily be mixed up.

---

To make a legend for all artists on all Axes, call this function with an iterable of strings, one for each legend item. For example:

```
fig, (ax1, ax2) = plt.subplots(1, 2)
ax1.plot([1, 3, 5], color='blue')
ax2.plot([2, 4, 6], color='red')
fig.legend(['the blues', 'the reds'])
```

**Parameters**

**handles**

[list of *Artist*, optional] A list of Artists (lines, patches) to be added to the legend. Use this together with *labels*, if you need full control on what is shown in the legend and the automatic mechanism described above is not sufficient.

The length of handles and labels should be the same in this case. If they are not, they are truncated to the smaller length.

**labels**

[list of str, optional] A list of labels to show next to the artists. Use this together with *handles*, if you need full control on what is shown in the legend and the automatic mechanism described above is not sufficient.

**Returns**

***Legend***

**Other Parameters**

**loc**

[str or pair of floats, default: 'upper right'] The location of the legend.

The strings `'upper left'`, `'upper right'`, `'lower left'`, `'lower right'` place the legend at the corresponding corner of the figure.

The strings `'upper center'`, `'lower center'`, `'center left'`, `'center right'` place the legend at the center of the corresponding edge of the figure.

The string `'center'` places the legend at the center of the figure.

The location can also be a 2-tuple giving the coordinates of the lower-left corner of the legend in figure coordinates (in which case *bbox_to_anchor* will be ignored).

For back-compatibility, `'center right'` (but no other location) can also be spelled `'right'`, and each "string" location can also be given as a numeric value:

| Location String | Location Code |
|---|---|
| 'best' (Axes only) | 0 |
| 'upper right' | 1 |
| 'upper left' | 2 |
| 'lower left' | 3 |
| 'lower right' | 4 |
| 'right' | 5 |
| 'center left' | 6 |
| 'center right' | 7 |
| 'lower center' | 8 |
| 'upper center' | 9 |
| 'center' | 10 |

If a figure is using the constrained layout manager, the string codes of the *loc* keyword argument can get better layout behaviour using the prefix 'outside'. There is ambiguity at the corners, so 'outside upper right' will make space for the legend above the rest of the axes in the layout, and 'outside right upper' will make space on the right side of the layout. In addition to the values of *loc* listed above, we have 'outside right upper', 'outside right lower', 'outside left upper', and 'outside left lower'. See *Legend guide* for more details.

**bbox_to_anchor**

[*BboxBase*, 2-tuple, or 4-tuple of floats] Box that is used to position the legend in conjunction with *loc*. Defaults to `axes.bbox` (if called as a method to *Axes.legend*) or `figure.bbox` (if *Figure.legend*). This argument allows arbitrary placement of the legend.

Bbox coordinates are interpreted in the coordinate system given by *bbox_transform*, with the default transform Axes or Figure coordinates, depending on which `legend` is called.

If a 4-tuple or *BboxBase* is given, then it specifies the bbox `(x, y, width, height)` that the legend is placed in. To put the legend in the best location in the bottom right quadrant of the Axes (or figure):

```
loc='best', bbox_to_anchor=(0.5, 0., 0.5, 0.5)
```

A 2-tuple `(x, y)` places the corner of the legend specified by *loc* at x, y. For example, to put the legend's upper right-hand corner in the center of the Axes (or figure) the following keywords can be used:

```
loc='upper right', bbox_to_anchor=(0.5, 0.5)
```

**ncols**

[int, default: 1] The number of columns that the legend has.

For backward compatibility, the spelling *ncol* is also supported but it is discouraged. If both are given, *ncols* takes precedence.

**prop**

[None or *FontProperties* or dict] The font properties of the legend. If None (default), the current *matplotlib.rcParams* will be used.

**fontsize**

[int or {'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large'}] The font size of the legend. If the value is numeric the size will be the absolute font size in points. String values are relative to the current default font size. This argument is only used if *prop* is not specified.

**labelcolor**

[str or list, default: *rcParams["legend.labelcolor"]* (default: `'None'`)] The color of the text in the legend. Either a valid color string (for example, 'red'), or a list of color strings. The labelcolor can also be made to match the color of the line or marker using 'linecolor', 'markerfacecolor' (or 'mfc'), or 'markeredgecolor' (or 'mec').

Labelcolor can be set globally using *rcParams["legend.labelcolor"]* (default: `'None'`). If None, use *rcParams["text.color"]* (default: `'black'`).

**numpoints**

[int, default: *rcParams["legend.numpoints"]* (default: `1`)] The number of marker points in the legend when creating a legend entry for a *Line2D* (line).

**scatterpoints**

[int, default: *rcParams["legend.scatterpoints"]* (default: `1`)] The number of marker points in the legend when creating a legend entry for a *PathCollection* (scatter plot).

**scatteryoffsets**

[iterable of floats, default: `[0.375, 0.5, 0.3125]`] The vertical offset (relative to the font size) for the markers created for a scatter plot legend entry. 0.0 is at the base the legend text, and 1.0 is at the top. To draw all markers at the same height, set to `[0.5]`.

**markerscale**

[float, default: *rcParams["legend.markerscale"]* (default: `1.0`)] The relative size of legend markers compared to the originally drawn ones.

**markerfirst**

[bool, default: True] If *True*, legend marker is placed to the left of the legend label. If *False*, legend marker is placed to the right of the legend label.

**reverse**

[bool, default: False] If *True*, the legend labels are displayed in reverse order from the input. If *False*, the legend labels are displayed in the same order as the input.

New in version 3.7.

**frameon**

[bool, default: *rcParams["legend.frameon"]* (default: `True`)] Whether the legend should be drawn on a patch (frame).

**fancybox**

[bool, default: *rcParams["legend.fancybox"]* (default: `True`)] Whether round edges should be enabled around the *FancyBboxPatch* which makes up the legend's background.

**shadow**

[None, bool or dict, default: *rcParams["legend.shadow"]* (default: `False`)] Whether to draw a shadow behind the legend. The shadow can be configured using *Patch* keywords. Customization via *rcParams["legend.shadow"]* (default: `False`) is currently not supported.

**framealpha**

[float, default: *rcParams["legend.framealpha"]* (default: `0.8`)] The alpha transparency of the legend's background. If *shadow* is activated and *framealpha* is `None`, the default value is ignored.

**facecolor**

["inherit" or color, default: *rcParams["legend.facecolor"]* (default: `'inherit'`)] The legend's background color. If `"inherit"`, use *rcParams["axes.facecolor"]* (default: `'white'`).

**edgecolor**

["inherit" or color, default: *rcParams["legend.edgecolor"]* (default: `'0.8'`)] The legend's background patch edge color. If `"inherit"`, use *rcParams["axes.edgecolor"]* (default: `'black'`).

**mode**

[{"expand", None}] If *mode* is set to `"expand"` the legend will be horizontally expanded to fill the Axes area (or *bbox_to_anchor* if defines the legend's size).

**bbox_transform**

[None or *Transform*] The transform for the bounding box (*bbox_to_anchor*). For a value of `None` (default) the Axes' `transAxes` transform will be used.

**title**

[str or None] The legend's title. Default is no title (`None`).

**title_fontproperties**

[None or *FontProperties* or dict] The font properties of the legend's title. If None (default), the *title_fontsize* argument will be used if present; if *title_fontsize* is also None, the current *rcParams["legend.title_fontsize"]* (default: `None`) will be used.

**title_fontsize**

[int or {'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large'}, default: *rcParams["legend.title_fontsize"]* (default: None)] The font size of the legend's title. Note: This cannot be combined with *title_fontproperties*. If you want to set the fontsize alongside other font properties, use the *size* parameter in *title_fontproperties*.

**alignment**

[{'center', 'left', 'right'}, default: 'center'] The alignment of the legend title and the box of entries. The entries are aligned as a single block, so that markers always lined up.

**borderpad**

[float, default: *rcParams["legend.borderpad"]* (default: 0.4)] The fractional whitespace inside the legend border, in font-size units.

**labelspacing**

[float, default: *rcParams["legend.labelspacing"]* (default: 0.5)] The vertical space between the legend entries, in font-size units.

**handlelength**

[float, default: *rcParams["legend.handlelength"]* (default: 2.0)] The length of the legend handles, in font-size units.

**handleheight**

[float, default: *rcParams["legend.handleheight"]* (default: 0.7)] The height of the legend handles, in font-size units.

**handletextpad**

[float, default: *rcParams["legend.handletextpad"]* (default: 0.8)] The pad between the legend handle and text, in font-size units.

**borderaxespad**

[float, default: *rcParams["legend.borderaxespad"]* (default: 0.5)] The pad between the Axes and legend border, in font-size units.

**columnspacing**

[float, default: *rcParams["legend.columnspacing"]* (default: 2.0)] The spacing between columns, in font-size units.

**handler_map**

[dict or None] The custom dictionary mapping instances or types to a legend handler. This *handler_map* updates the default handler map found at *matplotlib.legend.Legend.get_legend_handler_map*.

**draggable**

[bool, default: False] Whether the legend can be dragged with the mouse.

**See also:**

*Axes.legend*

### Notes

Some artists are not supported by this function. See *Legend guide* for details.

## matplotlib.figure.SubFigure.text

SubFigure.**text**(*x*, *y*, *s*, *fontdict=None*, ***kwargs*)

Add text to figure.

**Parameters**

**x, y**

[float] The position to place the text. By default, this is in figure coordinates, floats in [0, 1]. The coordinate system can be changed using the *transform* keyword.

**s**

[str] The text string.

**fontdict**

[dict, optional] A dictionary to override the default text properties. If not given, the defaults are determined by `rcParams["font.*"]`. Properties passed as *kwargs* override the corresponding ones given in *fontdict*.

**Returns**

*Text*

**Other Parameters**

****kwargs**

[*Text* properties] Other miscellaneous text parameters.

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array a |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* | bool |
| *backgroundcolor* | *color* |
| *bbox* | dict with properties for *patches.FancyBboxPa* |

Table 55 –

| Property | Description |
|---|---|
| *clip_box* | unknown |
| *clip_on* | unknown |
| *clip_path* | unknown |
| *color* or c | *color* |
| *figure* | *Figure* |
| *fontfamily* or family or fontname | {FONTNAME, 'serif', 'sans-serif', 'cursive', 'fantasy' |
| *fontproperties* or font or font_properties | *font_manager.FontProperties* or *str* or |
| *fontsize* or size | float or {'xx-small', 'x-small', 'small', 'medium', 'large |
| *fontstretch* or stretch | {a numeric value in range 0-1000, 'ultra-condensed', |
| *fontstyle* or style | {'normal', 'italic', 'oblique'} |
| *fontvariant* or variant | {'normal', 'small-caps'} |
| *fontweight* or weight | {a numeric value in range 0-1000, 'ultralight', 'light', |
| *gid* | str |
| *horizontalalignment* or ha | {'left', 'center', 'right'} |
| *in_layout* | bool |
| *label* | object |
| *linespacing* | float (multiple of font size) |
| *math_fontfamily* | str |
| *mouseover* | bool |
| *multialignment* or ma | {'left', 'right', 'center'} |
| *parse_math* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *position* | (float, float) |
| *rasterized* | bool |
| *rotation* | float or {'vertical', 'horizontal'} |
| *rotation_mode* | {None, 'default', 'anchor'} |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *text* | object |
| *transform* | *Transform* |
| *transform_rotates_text* | bool |
| *url* | str |
| *usetex* | bool, default: *rcParams["text.usetex"]* (de |
| *verticalalignment* or va | {'baseline', 'bottom', 'center', 'center_baseline', 'top'} |
| *visible* | bool |
| *wrap* | bool |
| *x* | float |
| *y* | float |
| *zorder* | float |

**See also:**

***Axes.text***
***pyplot.text***

### matplotlib.figure.SubFigure.suptitle

SubFigure.**suptitle**(*t*, ***kwargs*)

Add a centered suptitle to the figure.

**Parameters**

**t**

[str] The suptitle text.

**x**

[float, default: 0.5] The x location of the text in figure coordinates.

**y**

[float, default: 0.98] The y location of the text in figure coordinates.

**horizontalalignment, ha**

[{'center', 'left', 'right'}, default: center] The horizontal alignment of the text relative to (*x*, *y*).

**verticalalignment, va**

[{'top', 'center', 'bottom', 'baseline'}, default: top] The vertical alignment of the text relative to (*x*, *y*).

**fontsize, size**

[default: *rcParams["figure.titlesize"]* (default: 'large')] The font size of the text. See *Text.set_size* for possible values.

**fontweight, weight**

[default: *rcParams["figure.titleweight"]* (default: 'normal')] The font weight of the text. See *Text.set_weight* for possible values.

**Returns**

**text**

The *Text* instance of the suptitle.

**Other Parameters**

**fontproperties**

[None or dict, optional] A dict of font properties. If *fontproperties* is given the default values for font size and weight are taken from the *FontProperties* defaults. *rcParams["figure.titlesize"]* (default: 'large') and *rcParams["figure.titleweight"]* (default: 'normal') are ignored in this case.

****kwargs**

Additional kwargs are *matplotlib.text.Text* properties.

### Examples using `matplotlib.figure.SubFigure.suptitle`

- *Figure subfigures*
- *Tick formatters*
- *Axis ticks*

## matplotlib.figure.SubFigure.get_suptitle

SubFigure.**get_suptitle**()

Return the suptitle as string or an empty string if not set.

## matplotlib.figure.SubFigure.supxlabel

SubFigure.**supxlabel**(*t*, ***kwargs*)

Add a centered supxlabel to the figure.

**Parameters**

**t**

[str] The supxlabel text.

**x**

[float, default: 0.5] The x location of the text in figure coordinates.

**y**

[float, default: 0.01] The y location of the text in figure coordinates.

**horizontalalignment, ha**

[{'center', 'left', 'right'}, default: center] The horizontal alignment of the text relative to (*x*, *y*).

**verticalalignment, va**

[{'top', 'center', 'bottom', 'baseline'}, default: bottom] The vertical alignment of the text relative to (*x*, *y*).

**fontsize, size**

[default: *rcParams["figure.labelsize"]* (default: `'large'`)] The font size of the text. See *Text.set_size* for possible values.

**fontweight, weight**

[default: *rcParams["figure.labelweight"]* (default: 'normal')] The font weight of the text. See *Text.set_weight* for possible values.

**Returns**

**text**

The *Text* instance of the supxlabel.

**Other Parameters**

**fontproperties**

[None or dict, optional] A dict of font properties. If *fontproperties* is given the default values for font size and weight are taken from the *FontProperties* defaults. *rcParams["figure.labelsize"]* (default: 'large') and *rcParams["figure.labelweight"]* (default: 'normal') are ignored in this case.

****kwargs**

Additional kwargs are *matplotlib.text.Text* properties.

## matplotlib.figure.SubFigure.get_supxlabel

SubFigure.**get_supxlabel**()

Return the supxlabel as string or an empty string if not set.

## matplotlib.figure.SubFigure.supylabel

SubFigure.**supylabel**(*t*, ***kwargs*)

Add a centered supylabel to the figure.

**Parameters**

**t**

[str] The supylabel text.

**x**

[float, default: 0.02] The x location of the text in figure coordinates.

**y**

[float, default: 0.5] The y location of the text in figure coordinates.

**horizontalalignment, ha**

[{'center', 'left', 'right'}, default: left] The horizontal alignment of the text relative to (*x*, *y*).

**verticalalignment, va**

[{'top', 'center', 'bottom', 'baseline'}, default: center] The vertical alignment of the text relative to (*x*, *y*).

**fontsize, size**

[default: *rcParams["figure.labelsize"]* (default: `'large'`)] The font size of the text. See *Text.set_size* for possible values.

**fontweight, weight**

[default: *rcParams["figure.labelweight"]* (default: `'normal'`)] The font weight of the text. See *Text.set_weight* for possible values.

**Returns**

**text**

The *Text* instance of the supylabel.

**Other Parameters**

**fontproperties**

[None or dict, optional] A dict of font properties. If *fontproperties* is given the default values for font size and weight are taken from the *FontProperties* defaults. *rcParams["figure.labelsize"]* (default: `'large'`) and *rcParams["figure.labelweight"]* (default: `'normal'`) are ignored in this case.

****kwargs**

Additional kwargs are *matplotlib.text.Text* properties.

## matplotlib.figure.SubFigure.get_supylabel

`SubFigure.`**`get_supylabel`**`()`

Return the supylabel as string or an empty string if not set.

### matplotlib.figure.SubFigure.align_labels

SubFigure.**align_labels**(*axs=None*)

Align the xlabels and ylabels of subplots with the same subplots row or column (respectively) if label alignment is being done automatically (i.e. the label position is not manually set).

Alignment persists for draw events after this is called.

**Parameters**

**axs**

[list of *Axes*] Optional list (or ndarray) of *Axes* to align the labels. Default is to align all Axes on the figure.

**See also:**

***matplotlib.figure.Figure.align_xlabels***
***matplotlib.figure.Figure.align_ylabels***
***matplotlib.figure.Figure.align_titles***

### matplotlib.figure.SubFigure.align_xlabels

SubFigure.**align_xlabels**(*axs=None*)

Align the xlabels of subplots in the same subplot row if label alignment is being done automatically (i.e. the label position is not manually set).

Alignment persists for draw events after this is called.

If a label is on the bottom, it is aligned with labels on Axes that also have their label on the bottom and that have the same bottom-most subplot row. If the label is on the top, it is aligned with labels on Axes with the same top-most row.

**Parameters**

**axs**

[list of *Axes*] Optional list of (or ndarray) *Axes* to align the xlabels. Default is to align all Axes on the figure.

**See also:**

***matplotlib.figure.Figure.align_ylabels***
***matplotlib.figure.Figure.align_titles***
***matplotlib.figure.Figure.align_labels***

### Notes

This assumes that `axs` are from the same *GridSpec*, so that their *SubplotSpec* positions correspond to figure positions.

### Examples

Example with rotated xtick labels:

```
fig, axs = plt.subplots(1, 2)
for tick in axs[0].get_xticklabels():
    tick.set_rotation(55)
axs[0].set_xlabel('XLabel 0')
axs[1].set_xlabel('XLabel 1')
fig.align_xlabels()
```

## matplotlib.figure.SubFigure.align_ylabels

SubFigure.**align_ylabels**(*axs=None*)

Align the ylabels of subplots in the same subplot column if label alignment is being done automatically (i.e. the label position is not manually set).

Alignment persists for draw events after this is called.

If a label is on the left, it is aligned with labels on Axes that also have their label on the left and that have the same left-most subplot column. If the label is on the right, it is aligned with labels on Axes with the same right-most column.

**Parameters**

**axs**

[list of *Axes*] Optional list (or `ndarray`) of *Axes* to align the ylabels. Default is to align all Axes on the figure.

**See also:**

***matplotlib.figure.Figure.align_xlabels***
***matplotlib.figure.Figure.align_titles***
***matplotlib.figure.Figure.align_labels***

### Notes

This assumes that `axs` are from the same *GridSpec*, so that their *SubplotSpec* positions correspond to figure positions.

### Examples

Example with large yticks labels:

```
fig, axs = plt.subplots(2, 1)
axs[0].plot(np.arange(0, 1000, 50))
axs[0].set_ylabel('YLabel 0')
axs[1].set_ylabel('YLabel 1')
fig.align_ylabels()
```

## matplotlib.figure.SubFigure.align_titles

SubFigure.**align_titles**(*axs=None*)

Align the titles of subplots in the same subplot row if title alignment is being done automatically (i.e. the title position is not manually set).

Alignment persists for draw events after this is called.

**Parameters**

**axs**

[list of *Axes*] Optional list of (or ndarray) *Axes* to align the titles. Default is to align all Axes on the figure.

**See also:**

***matplotlib.figure.Figure.align_xlabels***
***matplotlib.figure.Figure.align_ylabels***
***matplotlib.figure.Figure.align_labels***

### Notes

This assumes that `axs` are from the same *GridSpec*, so that their *SubplotSpec* positions correspond to figure positions.

### Examples

Example with titles:

```
fig, axs = plt.subplots(1, 2)
axs[0].set_aspect('equal')
axs[0].set_title('Title 0')
axs[1].set_title('Title 1')
fig.align_titles()
```

## Adding and getting Artists

| | |
|---|---|
| *SubFigure.add_artist* | Add an *Artist* to the figure. |
| *SubFigure.get_children* | Get a list of artists contained in the figure. |

## matplotlib.figure.SubFigure.add_artist

SubFigure.**add_artist**(*artist*, *clip=False*)

Add an *Artist* to the figure.

Usually artists are added to *Axes* objects using *Axes.add_artist*; this method can be used in the rare cases where one needs to add artists directly to the figure instead.

**Parameters**

**artist**

[*Artist*] The artist to add to the figure. If the added artist has no transform previously set, its transform will be set to `figure.transSubfigure`.

**clip**

[bool, default: False] Whether the added artist should be clipped by the figure patch.

**Returns**

***Artist***

The added artist.

### matplotlib.figure.SubFigure.get_children

SubFigure.**get_children**()

Get a list of artists contained in the figure.

### Modifying appearance

| | |
|---|---|
| *SubFigure.set_frameon* | Set the figure's background patch visibility, i.e. whether the figure background will be drawn. |
| *SubFigure.get_frameon* | Return the figure's background patch visibility, i.e. whether the figure background will be drawn. |
| *SubFigure.set_linewidth* | Set the line width of the Figure rectangle. |
| *SubFigure.get_linewidth* | Get the line width of the Figure rectangle. |
| *SubFigure.set_facecolor* | Set the face color of the Figure rectangle. |
| *SubFigure.get_facecolor* | Get the face color of the Figure rectangle. |
| *SubFigure.set_edgecolor* | Set the edge color of the Figure rectangle. |
| *SubFigure.get_edgecolor* | Get the edge color of the Figure rectangle. |

### matplotlib.figure.SubFigure.set_frameon

SubFigure.**set_frameon**(*b*)

Set the figure's background patch visibility, i.e. whether the figure background will be drawn. Equivalent to `Figure.patch.set_visible()`.

**Parameters**

**b**

[bool]

### matplotlib.figure.SubFigure.get_frameon

SubFigure.**get_frameon**()

Return the figure's background patch visibility, i.e. whether the figure background will be drawn. Equivalent to `Figure.patch.get_visible()`.

### matplotlib.figure.SubFigure.set_linewidth

SubFigure.**set_linewidth**(*linewidth*)

Set the line width of the Figure rectangle.

**Parameters**

**linewidth**

[number]

### matplotlib.figure.SubFigure.get_linewidth

SubFigure.**get_linewidth**()

Get the line width of the Figure rectangle.

### matplotlib.figure.SubFigure.set_facecolor

SubFigure.**set_facecolor**(*color*)

Set the face color of the Figure rectangle.

**Parameters**

**color**

[*color*]

### Examples using `matplotlib.figure.SubFigure.set_facecolor`

- *Figure subfigures*

### matplotlib.figure.SubFigure.get_facecolor

SubFigure.**get_facecolor**()

Get the face color of the Figure rectangle.

### matplotlib.figure.SubFigure.set_edgecolor

SubFigure.**set_edgecolor**(*color*)

Set the edge color of the Figure rectangle.

**Parameters**

**color**

[*color*]

### matplotlib.figure.SubFigure.get_edgecolor

SubFigure.**get_edgecolor**()

Get the edge color of the Figure rectangle.

### Passthroughs

| | |
|---|---|
| *SubFigure.set_dpi* | Set the resolution of parent figure in dots-per-inch. |
| *SubFigure.get_dpi* | Return the resolution of the parent figure in dots-per-inch as a float. |

### matplotlib.figure.SubFigure.set_dpi

SubFigure.**set_dpi**(*val*)

Set the resolution of parent figure in dots-per-inch.

**Parameters**

**val**

[float]

### matplotlib.figure.SubFigure.get_dpi

SubFigure.**get_dpi**()

Return the resolution of the parent figure in dots-per-inch as a float.

### FigureBase parent class

**class** matplotlib.figure.**FigureBase**(***kwargs*)

Base class for *Figure* and *SubFigure* containing the methods that add artists to the figure or subfigure, create Axes, etc.

**add_artist**(*artist*, *clip=False*)

Add an *Artist* to the figure.

Usually artists are added to *Axes* objects using *Axes.add_artist*; this method can be used in the rare cases where one needs to add artists directly to the figure instead.

**Parameters**

**artist**

[*Artist*] The artist to add to the figure. If the added artist has no transform previously set, its transform will be set to `figure.transSubfigure`.

**clip**

[bool, default: False] Whether the added artist should be clipped by the figure patch.

**Returns**

***Artist***

The added artist.

**add_axes**(**args*, ***kwargs*)

Add an *Axes* to the figure.

Call signatures:

```
add_axes(rect, projection=None, polar=False, **kwargs)
add_axes(ax)
```

**Parameters**

**rect**

[tuple (left, bottom, width, height)] The dimensions (left, bottom, width, height) of the new *Axes*. All quantities are in fractions of figure width and height.

**projection**

[{None, 'aitoff', 'hammer', 'lambert', 'mollweide', 'polar', 'rectilinear', str}, optional] The projection type of the *Axes*. *str* is the name of a custom projection, see *projections*. The default None results in a 'rectilinear' projection.

**polar**

[bool, default: False] If True, equivalent to projection='polar'.

**axes_class**

[subclass type of *Axes*, optional] The *axes.Axes* subclass that is instantiated. This parameter is incompatible with *projection* and *polar*. See *axisartist* for examples.

**sharex, sharey**

[*Axes*, optional] Share the x or y *axis* with sharex and/or sharey. The axis will have the same limits, ticks, and scale as the axis of the shared Axes.

**label**

[str] A label for the returned Axes.

### Returns

***Axes*, or a subclass of *Axes***

The returned Axes class depends on the projection used. It is *Axes* if rectilinear projection is used and *projections.polar.PolarAxes* if polar projection is used.

### Other Parameters

****kwargs**

This method also takes the keyword arguments for the returned Axes class. The keyword arguments for the rectilinear Axes class *Axes* can be found in the following table but there might also be other keyword arguments if another projection is used, see the actual Axes class.

| Property | Description |
|---|---|
| *adjustable* | {'box', 'datalim'} |
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dp |
| *alpha* | scalar or None |
| *anchor* | (float, float) or {'C', 'SW', 'S', 'SE', 'E', 'NE', ...} |
| *animated* | bool |
| *aspect* | {'auto', 'equal'} or float |
| *autoscale_on* | bool |
| *autoscalex_on* | unknown |
| *autoscaley_on* | unknown |
| *axes_locator* | Callable[[Axes, Renderer], Bbox] |
| *axisbelow* | bool or 'line' |
| *box_aspect* | float or None |
| *clip_box* | *BboxBase* or None |

Table 56 – continued fr

| Property | Description |
|---|---|
| clip_on | bool |
| clip_path | Patch or (Path, Transform) or None |
| facecolor or fc | color |
| figure | Figure |
| forward_navigation_events | bool or "auto" |
| frame_on | bool |
| gid | str |
| in_layout | bool |
| label | object |
| mouseover | bool |
| navigate | bool |
| navigate_mode | unknown |
| path_effects | list of AbstractPathEffect |
| picker | None or bool or float or callable |
| position | [left, bottom, width, height] or Bbox |
| prop_cycle | Cycler |
| rasterization_zorder | float or None |
| rasterized | bool |
| sketch_params | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| subplotspec | unknown |
| title | str |
| transform | Transform |
| url | str |
| visible | bool |
| xbound | (lower: float, upper: float) |
| xlabel | str |
| xlim | (left: float, right: float) |
| xmargin | float greater than -0.5 |
| xscale | unknown |
| xticklabels | unknown |
| xticks | unknown |
| ybound | (lower: float, upper: float) |
| ylabel | str |
| ylim | (bottom: float, top: float) |
| ymargin | float greater than -0.5 |
| yscale | unknown |
| yticklabels | unknown |
| yticks | unknown |
| zorder | float |

**See also:**

*Figure.add_subplot*
*pyplot.subplot*
*pyplot.axes*
*Figure.subplots*
*pyplot.subplots*

### Notes

In rare circumstances, *add_axes* may be called with a single argument, an Axes instance already created in the present figure but not in the figure's list of Axes.

### Examples

Some simple examples:

```
rect = l, b, w, h
fig = plt.figure()
fig.add_axes(rect)
fig.add_axes(rect, frameon=False, facecolor='g')
fig.add_axes(rect, polar=True)
ax = fig.add_axes(rect, projection='polar')
fig.delaxes(ax)
fig.add_axes(ax)
```

**add_gridspec**(*nrows=1*, *ncols=1*, ***kwargs*)

Low-level API for creating a *GridSpec* that has this figure as a parent.

This is a low-level API, allowing you to create a gridspec and subsequently add subplots based on the gridspec. Most users do not need that freedom and should use the higher-level methods *subplots* or *subplot_mosaic*.

**Parameters**

**nrows**

[int, default: 1] Number of rows in grid.

**ncols**

[int, default: 1] Number of columns in grid.

**Returns**

*GridSpec*

**Other Parameters**

****kwargs**

Keyword arguments are passed to *GridSpec*.

**See also:**

*matplotlib.pyplot.subplots*

### Examples

Adding a subplot that spans two rows:

```
fig = plt.figure()
gs = fig.add_gridspec(2, 2)
ax1 = fig.add_subplot(gs[0, 0])
ax2 = fig.add_subplot(gs[1, 0])
# spans two rows:
ax3 = fig.add_subplot(gs[:, 1])
```

**add_subfigure**(*subplotspec*, ***kwargs*)

Add a *SubFigure* to the figure as part of a subplot arrangement.

**Parameters**

**subplotspec**

[*gridspec.SubplotSpec*] Defines the region in a parent gridspec where the subfigure will be placed.

**Returns**

*SubFigure*

**Other Parameters**

****kwargs**

Are passed to the *SubFigure* object.

**See also:**

*Figure.subfigures*

**add_subplot**(**args*, ***kwargs*)

Add an *Axes* to the figure as part of a subplot arrangement.

Call signatures:

```
add_subplot(nrows, ncols, index, **kwargs)
add_subplot(pos, **kwargs)
add_subplot(ax)
add_subplot()
```

**Parameters**

***args**

[int, (int, int, *index*), or *SubplotSpec*, default: (1, 1, 1)] The position of the subplot described by one of

- Three integers (*nrows*, *ncols*, *index*). The subplot will take the *index* position on a grid with *nrows* rows and *ncols* columns. *index* starts at 1 in the upper left corner and increases to the right. *index* can also be a two-tuple specifying the (*first*, *last*) indices (1-based, and including *last*) of the subplot, e.g., `fig.add_subplot(3, 1, (1, 2))` makes a subplot that spans the upper 2/3 of the figure.
- A 3-digit integer. The digits are interpreted as if given separately as three single-digit integers, i.e. `fig.add_subplot(235)` is the same as `fig.add_subplot(2, 3, 5)`. Note that this can only be used if there are no more than 9 subplots.
- A *SubplotSpec*.

In rare circumstances, *add_subplot* may be called with a single argument, a subplot Axes instance already created in the present figure but not in the figure's list of Axes.

**projection**

[{None, 'aitoff', 'hammer', 'lambert', 'mollweide', 'polar', 'rectilinear', str}, optional] The projection type of the subplot (*Axes*). *str* is the name of a custom projection, see *projections*. The default None results in a 'rectilinear' projection.

**polar**

[bool, default: False] If True, equivalent to projection='polar'.

**axes_class**

[subclass type of *Axes*, optional] The *axes.Axes* subclass that is instantiated. This parameter is incompatible with *projection* and *polar*. See *axisartist* for examples.

**sharex, sharey**

[*Axes*, optional] Share the x or y *axis* with sharex and/or sharey. The axis will have the same limits, ticks, and scale as the axis of the shared Axes.

**label**

[str] A label for the returned Axes.

**Returns**

***Axes***

The Axes of the subplot. The returned Axes can actually be an instance of a subclass, such as *projections.polar.PolarAxes* for polar projections.

### Other Parameters

#### **kwargs

This method also takes the keyword arguments for the returned Axes base class; except for the *figure* argument. The keyword arguments for the rectilinear base class *Axes* can be found in the following table but there might also be other keyword arguments if another projection is used.

| Property | Description |
|---|---|
| *adjustable* | {'box', 'datalim'} |
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dp |
| *alpha* | scalar or None |
| *anchor* | (float, float) or {'C', 'SW', 'S', 'SE', 'E', 'NE', ...} |
| *animated* | bool |
| *aspect* | {'auto', 'equal'} or float |
| *autoscale_on* | bool |
| *autoscalex_on* | unknown |
| *autoscaley_on* | unknown |
| *axes_locator* | Callable[[Axes, Renderer], Bbox] |
| *axisbelow* | bool or 'line' |
| *box_aspect* | float or None |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *facecolor* or fc | *color* |
| *figure* | *Figure* |
| *forward_navigation_events* | bool or "auto" |
| *frame_on* | bool |
| *gid* | str |
| *in_layout* | bool |
| *label* | object |
| *mouseover* | bool |
| *navigate* | bool |
| *navigate_mode* | unknown |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *position* | [left, bottom, width, height] or *Bbox* |
| *prop_cycle* | *Cycler* |
| *rasterization_zorder* | float or None |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *subplotspec* | unknown |
| *title* | str |
| *transform* | *Transform* |

Table 57 – continued fr

| Property | Description |
|---|---|
| *url* | str |
| *visible* | bool |
| *xbound* | (lower: float, upper: float) |
| *xlabel* | str |
| *xlim* | (left: float, right: float) |
| *xmargin* | float greater than -0.5 |
| *xscale* | unknown |
| *xticklabels* | unknown |
| *xticks* | unknown |
| *ybound* | (lower: float, upper: float) |
| *ylabel* | str |
| *ylim* | (bottom: float, top: float) |
| *ymargin* | float greater than -0.5 |
| *yscale* | unknown |
| *yticklabels* | unknown |
| *yticks* | unknown |
| *zorder* | float |

**See also:**

***Figure.add_axes***
***pyplot.subplot***
***pyplot.axes***
***Figure.subplots***
***pyplot.subplots***

### Examples

```
fig = plt.figure()

fig.add_subplot(231)
ax1 = fig.add_subplot(2, 3, 1)  # equivalent but more general

fig.add_subplot(232, frameon=False)  # subplot with no frame
fig.add_subplot(233, projection='polar')  # polar subplot
fig.add_subplot(234, sharex=ax1)  # subplot sharing x-axis with ax1
fig.add_subplot(235, facecolor="red")  # red subplot

ax1.remove()  # delete ax1 from the figure
fig.add_subplot(ax1)  # add ax1 back to the figure
```

**align_labels**(*axs=None*)

Align the xlabels and ylabels of subplots with the same subplots row or column (respectively) if label alignment is being done automatically (i.e. the label position is not manually set).

Alignment persists for draw events after this is called.

**Parameters**

**axs**

[list of *Axes*] Optional list (or `ndarray`) of *Axes* to align the labels. Default is to align all Axes on the figure.

**See also:**

***matplotlib.figure.Figure.align_xlabels***
***matplotlib.figure.Figure.align_ylabels***
***matplotlib.figure.Figure.align_titles***

**align_titles**(*axs=None*)

Align the titles of subplots in the same subplot row if title alignment is being done automatically (i.e. the title position is not manually set).

Alignment persists for draw events after this is called.

**Parameters**

**axs**

[list of *Axes*] Optional list of (or ndarray) *Axes* to align the titles. Default is to align all Axes on the figure.

**See also:**

***matplotlib.figure.Figure.align_xlabels***
***matplotlib.figure.Figure.align_ylabels***
***matplotlib.figure.Figure.align_labels***

## Notes

This assumes that `axs` are from the same *GridSpec*, so that their *SubplotSpec* positions correspond to figure positions.

## Examples

Example with titles:

```
fig, axs = plt.subplots(1, 2)
axs[0].set_aspect('equal')
axs[0].set_title('Title 0')
axs[1].set_title('Title 1')
fig.align_titles()
```

**align_xlabels**(*axs=None*)

Align the xlabels of subplots in the same subplot row if label alignment is being done automatically (i.e. the label position is not manually set).

Alignment persists for draw events after this is called.

If a label is on the bottom, it is aligned with labels on Axes that also have their label on the bottom and that have the same bottom-most subplot row. If the label is on the top, it is aligned with labels on Axes with the same top-most row.

**Parameters**

**axs**

[list of *Axes*] Optional list of (or `ndarray`) *Axes* to align the xlabels. Default is to align all Axes on the figure.

**See also:**

***matplotlib.figure.Figure.align_ylabels***
***matplotlib.figure.Figure.align_titles***
***matplotlib.figure.Figure.align_labels***

### Notes

This assumes that `axs` are from the same *GridSpec*, so that their *SubplotSpec* positions correspond to figure positions.

### Examples

Example with rotated xtick labels:

```
fig, axs = plt.subplots(1, 2)
for tick in axs[0].get_xticklabels():
    tick.set_rotation(55)
axs[0].set_xlabel('XLabel 0')
axs[1].set_xlabel('XLabel 1')
fig.align_xlabels()
```

**align_ylabels**(*axs=None*)

Align the ylabels of subplots in the same subplot column if label alignment is being done automatically (i.e. the label position is not manually set).

Alignment persists for draw events after this is called.

If a label is on the left, it is aligned with labels on Axes that also have their label on the left and that have the same left-most subplot column. If the label is on the right, it is aligned with labels on Axes with the same right-most column.

**Parameters**

**axs**

[list of *Axes*] Optional list (or `ndarray`) of *Axes* to align the ylabels. Default is to align all Axes on the figure.

**See also:**

***matplotlib.figure.Figure.align_xlabels***
***matplotlib.figure.Figure.align_titles***
***matplotlib.figure.Figure.align_labels***

### Notes

This assumes that `axs` are from the same *GridSpec*, so that their *SubplotSpec* positions correspond to figure positions.

### Examples

Example with large yticks labels:

```
fig, axs = plt.subplots(2, 1)
axs[0].plot(np.arange(0, 1000, 50))
axs[0].set_ylabel('YLabel 0')
axs[1].set_ylabel('YLabel 1')
fig.align_ylabels()
```

**autofmt_xdate**(*bottom=0.2*, *rotation=30*, *ha='right'*, *which='major'*)

Date ticklabels often overlap, so it is useful to rotate them and right align them. Also, a common use case is a number of subplots with shared x-axis where the x-axis is date data. The ticklabels are often long, and it helps to rotate them on the bottom subplot and turn them off on other subplots, as well as turn off xlabels.

**Parameters**

**bottom**

[float, default: 0.2] The bottom of the subplots for *subplots_adjust*.

**rotation**

[float, default: 30 degrees] The rotation angle of the xtick labels in degrees.

**ha**

[{'left', 'center', 'right'}, default: 'right'] The horizontal alignment of the xticklabels.

**which**

[{'major', 'minor', 'both'}, default: 'major'] Selects which ticklabels to rotate.

**clear**(*keep_observers=False*)

Clear the figure.

**Parameters**

**keep_observers**

[bool, default: False] Set *keep_observers* to True if, for example, a gui widget is tracking the Axes in the figure.

**clf**(*keep_observers=False*)

[*Discouraged*] Alias for the `clear()` method.

---

**Discouraged**

The use of `clf()` is discouraged. Use `clear()` instead.

---

**Parameters**

**keep_observers**

[bool, default: False] Set *keep_observers* to True if, for example, a gui widget is tracking the Axes in the figure.

**colorbar**(*mappable*, *cax=None*, *ax=None*, *use_gridspec=True*, ***kwargs*)

Add a colorbar to a plot.

**Parameters**

**mappable**

The `matplotlib.cm.ScalarMappable` (i.e., `AxesImage`, `ContourSet`, etc.) described by this colorbar. This argument is mandatory for the `Figure.colorbar` method but optional for the `pyplot.colorbar` function, which sets the default to the current image.

Note that one can create a `ScalarMappable` "on-the-fly" to generate colorbars not attached to a previously drawn artist, e.g.

```
fig.colorbar(cm.ScalarMappable(norm=norm, cmap=cmap),␣
↪ax=ax)
```

**cax**

[`Axes`, optional] Axes into which the colorbar will be drawn. If `None`, then a new Axes is created and the space for it will be stolen from the Axes(s) specified in *ax*.

**ax**

[*Axes* or iterable or `numpy.ndarray` of Axes, optional] The one or more parent Axes from which space for a new colorbar Axes will be stolen. This parameter is only used if *cax* is not set.

Defaults to the Axes that contains the mappable used to create the colorbar.

**use_gridspec**

[bool, optional] If *cax* is `None`, a new *cax* is created as an instance of Axes. If *ax* is positioned with a subplotspec and *use_gridspec* is `True`, then *cax* is also positioned with a subplotspec.

**Returns**

**colorbar**

[*Colorbar*]

**Other Parameters**

**location**

[None or {'left', 'right', 'top', 'bottom'}] The location, relative to the parent Axes, where the colorbar Axes is created. It also determines the *orientation* of the colorbar (colorbars on the left and right are vertical, colorbars at the top and bottom are horizontal). If None, the location will come from the *orientation* if it is set (vertical colorbars on the right, horizontal ones at the bottom), or default to 'right' if *orientation* is unset.

**orientation**

[None or {'vertical', 'horizontal'}] The orientation of the colorbar. It is preferable to set the *location* of the colorbar, as that also determines the *orientation*; passing incompatible values for *location* and *orientation* raises an exception.

**fraction**

[float, default: 0.15] Fraction of original Axes to use for colorbar.

**shrink**

[float, default: 1.0] Fraction by which to multiply the size of the colorbar.

**aspect**

[float, default: 20] Ratio of long to short dimensions.

**pad**

[float, default: 0.05 if vertical, 0.15 if horizontal] Fraction of original Axes between colorbar and new image Axes.

**anchor**

[(float, float), optional] The anchor point of the colorbar Axes. Defaults to (0.0, 0.5) if vertical; (0.5, 1.0) if horizontal.

**panchor**

[(float, float), or *False*, optional] The anchor point of the colorbar parent Axes. If *False*, the parent axes' anchor will be unchanged. Defaults to (1.0, 0.5) if vertical; (0.5, 0.0) if horizontal.

**extend**

[{'neither', 'both', 'min', 'max'}] Make pointed end(s) for out-of-range values (unless 'neither'). These are set for a given colormap using the colormap set_under and set_over methods.

**extendfrac**

[{*None*, 'auto', length, lengths}] If set to *None*, both the minimum and maximum triangular colorbar extensions will have a length of 5% of the interior colorbar length (this is the default setting).

If set to 'auto', makes the triangular colorbar extensions the same lengths as the interior boxes (when *spacing* is set to 'uniform') or the same lengths as the respective adjacent interior boxes (when *spacing* is set to 'proportional').

If a scalar, indicates the length of both the minimum and maximum triangular colorbar extensions as a fraction of the interior colorbar length. A two-element sequence of fractions may also be given, indicating the lengths of the minimum and maximum colorbar extensions respectively as a fraction of the interior colorbar length.

**extendrect**

[bool] If *False* the minimum and maximum colorbar extensions will be triangular (the default). If *True* the extensions will be rectangular.

**spacing**

[{'uniform', 'proportional'}] For discrete colorbars (`BoundaryNorm` or contours), 'uniform' gives each color the same space; 'proportional' makes the space proportional to the data interval.

**ticks**

[None or list of ticks or Locator] If None, ticks are determined automatically from the input.

**format**

[None or str or Formatter] If None, `ScalarFormatter` is used. Format strings, e.g., `"%4.2e"` or `"{x:.2e}"`, are supported. An alternative `Formatter` may be given instead.

**drawedges**

[bool] Whether to draw lines at color boundaries.

**label**

[str] The label on the colorbar's long axis.

**boundaries, values**

[None or a sequence] If unset, the colormap will be displayed on a 0-1 scale. If sequences, *values* must have a length 1 less than *boundaries*. For each region delimited by adjacent entries in *boundaries*, the color mapped to the corresponding value in values will be used. Normally only useful for indexed colors (i.e. `norm=NoNorm()`) or other unusual circumstances.

### Notes

If *mappable* is a *ContourSet*, its *extend* kwarg is included automatically.

The *shrink* kwarg provides a simple way to scale the colorbar with respect to the Axes. Note that if *cax* is specified, it determines the size of the colorbar, and *shrink* and *aspect* are ignored.

For more precise control, you can manually specify the positions of the axes objects in which the mappable and the colorbar are drawn. In this case, do not use any of the Axes properties kwargs.

It is known that some vector graphics viewers (svg and pdf) render white gaps between segments of the colorbar. This is due to bugs in the viewers, not Matplotlib. As a workaround, the colorbar can be rendered with overlapping segments:

```
cbar = colorbar()
cbar.solids.set_edgecolor("face")
draw()
```

However, this has negative consequences in other circumstances, e.g. with semi-transparent images (alpha < 1) and colorbar extensions; therefore, this workaround is not used by default (see issue #1188).

**contains**(*mouseevent*)

Test whether the mouse event occurred on the figure.

**Returns**

**bool, {}**

**delaxes**(*ax*)

Remove the *Axes* *ax* from the figure; update the current Axes.

**draw**(*renderer*)

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

**Parameters**

**renderer**

[*RendererBase* subclass.]

### Notes

This method is overridden in the Artist subclasses.

**property frameon**

Return the figure's background patch visibility, i.e. whether the figure background will be drawn. Equivalent to `Figure.patch.get_visible()`.

**gca**()

Get the current Axes.

If there is currently no Axes on this Figure, a new one is created using *Figure.add_subplot*. (To test whether there is currently an Axes on a Figure, check whether `figure.axes` is empty. To test whether there is currently a Figure on the pyplot figure stack, check whether *pyplot.get_fignums()* is empty.)

**get_children**()

Get a list of artists contained in the figure.

**get_default_bbox_extra_artists**()

Return a list of Artists typically used in *Figure.get_tightbbox*.

**get_edgecolor**()

Get the edge color of the Figure rectangle.

**get_facecolor**()

Get the face color of the Figure rectangle.

**get_frameon**()

Return the figure's background patch visibility, i.e. whether the figure background will be drawn. Equivalent to `Figure.patch.get_visible()`.

**get_linewidth**()

Get the line width of the Figure rectangle.

**get_suptitle**()

Return the suptitle as string or an empty string if not set.

**get_supxlabel**()

Return the supxlabel as string or an empty string if not set.

**get_supylabel**()

Return the supylabel as string or an empty string if not set.

**get_tightbbox**(*renderer=None*, *, *bbox_extra_artists=None*)

Return a (tight) bounding box of the figure *in inches*.

Note that *FigureBase* differs from all other artists, which return their *Bbox* in pixels.

Artists that have `artist.set_in_layout(False)` are not included in the bbox.

**Parameters**

**renderer**

[*RendererBase* subclass] Renderer that will be used to draw the figures (i.e. `fig.canvas.get_renderer()`)

**bbox_extra_artists**

[list of *Artist* or `None`] List of artists to include in the tight bounding box. If `None` (default), then all artist children of each Axes are included in the tight bounding box.

**Returns**

***BboxBase***

containing the bounding box (in figure inches).

**get_window_extent**(*renderer=None*)

Get the artist's bounding box in display space.

The bounding box' width and height are nonnegative.

Subclasses should override for inclusion in the bounding box "tight" calculation. Default is to return an empty bounding box at 0, 0.

Be careful when using this function, the results will not update if the artist window extent of the artist changes. The extent can change due to any changes in the transform stack, such as changing the Axes limits, the figure size, or the canvas used (as is done when saving a figure). This can lead to unexpected behavior where interactive figures will look fine on the screen, but will save incorrectly.

**legend**(**args*, ***kwargs*)

Place a legend on the figure.

Call signatures:

```
legend()
legend(handles, labels)
legend(handles=handles)
legend(labels)
```

The call signatures correspond to the following different ways to use this method:

**1. Automatic detection of elements to be shown in the legend**

The elements to be added to the legend are automatically determined, when you do not pass in any extra arguments.

In this case, the labels are taken from the artist. You can specify them either at artist creation or by calling the *set_label()* method on the artist:

```
ax.plot([1, 2, 3], label='Inline label')
fig.legend()
```

or:

```
line, = ax.plot([1, 2, 3])
line.set_label('Label via method')
fig.legend()
```

Specific lines can be excluded from the automatic legend element selection by defining a label starting with an underscore. This is default for all artists, so calling *Figure.legend* without any arguments and without setting the labels manually will result in no legend being drawn.

**2. Explicitly listing the artists and labels in the legend**

For full control of which artists have a legend entry, it is possible to pass an iterable of legend artists followed by an iterable of legend labels respectively:

```
fig.legend([line1, line2, line3], ['label1', 'label2', 'label3'])
```

**3. Explicitly listing the artists in the legend**

This is similar to 2, but the labels are taken from the artists' label properties. Example:

```
line1, = ax1.plot([1, 2, 3], label='label1')
line2, = ax2.plot([1, 2, 3], label='label2')
fig.legend(handles=[line1, line2])
```

**4. Labeling existing plot elements**

---

**Discouraged**

This call signature is discouraged, because the relation between plot elements and labels is only implicit by their order and can easily be mixed up.

---

To make a legend for all artists on all Axes, call this function with an iterable of strings, one for each legend item. For example:

```
fig, (ax1, ax2) = plt.subplots(1, 2)
ax1.plot([1, 3, 5], color='blue')
ax2.plot([2, 4, 6], color='red')
fig.legend(['the blues', 'the reds'])
```

**Parameters**

**handles**

[list of *Artist*, optional] A list of Artists (lines, patches) to be added to the legend. Use this together with *labels*, if you need full control on what is shown in the legend and the automatic mechanism described above is not sufficient.

The length of handles and labels should be the same in this case. If they are not, they are truncated to the smaller length.

**labels**

[list of str, optional] A list of labels to show next to the artists. Use this together with *handles*, if you need full control on what is shown in the legend and the automatic mechanism described above is not sufficient.

**Returns**

*Legend*

**Other Parameters**

**loc**

[str or pair of floats, default: 'upper right'] The location of the legend.

The strings `'upper left'`, `'upper right'`, `'lower left'`, `'lower right'` place the legend at the corresponding corner of the figure.

The strings `'upper center'`, `'lower center'`, `'center left'`, `'center right'` place the legend at the center of the corresponding edge of the figure.

The string `'center'` places the legend at the center of the figure.

The location can also be a 2-tuple giving the coordinates of the lower-left corner of the legend in figure coordinates (in which case *bbox_to_anchor* will be ignored).

For back-compatibility, `'center right'` (but no other location) can also be spelled `'right'`, and each "string" location can also be given as a numeric value:

| Location String | Location Code |
|---|---|
| 'best' (Axes only) | 0 |
| 'upper right' | 1 |
| 'upper left' | 2 |
| 'lower left' | 3 |
| 'lower right' | 4 |
| 'right' | 5 |
| 'center left' | 6 |
| 'center right' | 7 |
| 'lower center' | 8 |
| 'upper center' | 9 |
| 'center' | 10 |

If a figure is using the constrained layout manager, the string codes of the *loc* keyword argument can get better layout behaviour using the prefix 'outside'. There is ambiguity at the corners, so 'outside upper right' will make space for the legend above the rest of the axes in the layout, and 'outside right upper' will make space

on the right side of the layout. In addition to the values of *loc* listed above, we have 'outside right upper', 'outside right lower', 'outside left upper', and 'outside left lower'. See *Legend guide* for more details.

**bbox_to_anchor**

[*BboxBase*, 2-tuple, or 4-tuple of floats] Box that is used to position the legend in conjunction with *loc*. Defaults to `axes.bbox` (if called as a method to *Axes.legend*) or `figure.bbox` (if *Figure.legend*). This argument allows arbitrary placement of the legend.

Bbox coordinates are interpreted in the coordinate system given by *bbox_transform*, with the default transform Axes or Figure coordinates, depending on which `legend` is called.

If a 4-tuple or *BboxBase* is given, then it specifies the bbox `(x, y, width, height)` that the legend is placed in. To put the legend in the best location in the bottom right quadrant of the Axes (or figure):

```
loc='best', bbox_to_anchor=(0.5, 0., 0.5, 0.5)
```

A 2-tuple `(x, y)` places the corner of the legend specified by *loc* at x, y. For example, to put the legend's upper right-hand corner in the center of the Axes (or figure) the following keywords can be used:

```
loc='upper right', bbox_to_anchor=(0.5, 0.5)
```

**ncols**

[int, default: 1] The number of columns that the legend has.

For backward compatibility, the spelling *ncol* is also supported but it is discouraged. If both are given, *ncols* takes precedence.

**prop**

[None or *FontProperties* or dict] The font properties of the legend. If None (default), the current *matplotlib.rcParams* will be used.

**fontsize**

[int or {'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large'}] The font size of the legend. If the value is numeric the size will be the absolute font size in points. String values are relative to the current default font size. This argument is only used if *prop* is not specified.

**labelcolor**

[str or list, default: *rcParams["legend.labelcolor"]* (default: `'None'`)] The color of the text in the legend. Either a valid color string (for example, 'red'), or a list of color strings. The labelcolor can also be made to match the color of the line or marker using 'linecolor', 'markerfacecolor' (or 'mfc'), or 'markeredgecolor' (or 'mec').

Labelcolor can be set globally using *rcParams["legend.labelcolor"]* (default: 'None'). If None, use *rcParams["text.color"]* (default: 'black').

**numpoints**

[int, default: *rcParams["legend.numpoints"]* (default: 1)] The number of marker points in the legend when creating a legend entry for a *Line2D* (line).

**scatterpoints**

[int, default: *rcParams["legend.scatterpoints"]* (default: 1)] The number of marker points in the legend when creating a legend entry for a *PathCollection* (scatter plot).

**scatteryoffsets**

[iterable of floats, default: [0.375, 0.5, 0.3125]] The vertical offset (relative to the font size) for the markers created for a scatter plot legend entry. 0.0 is at the base the legend text, and 1.0 is at the top. To draw all markers at the same height, set to [0.5].

**markerscale**

[float, default: *rcParams["legend.markerscale"]* (default: 1.0)] The relative size of legend markers compared to the originally drawn ones.

**markerfirst**

[bool, default: True] If *True*, legend marker is placed to the left of the legend label. If *False*, legend marker is placed to the right of the legend label.

**reverse**

[bool, default: False] If *True*, the legend labels are displayed in reverse order from the input. If *False*, the legend labels are displayed in the same order as the input.

New in version 3.7.

**frameon**

[bool, default: *rcParams["legend.frameon"]* (default: True)] Whether the legend should be drawn on a patch (frame).

**fancybox**

[bool, default: *rcParams["legend.fancybox"]* (default: True)] Whether round edges should be enabled around the *FancyBboxPatch* which makes up the legend's background.

**shadow**

[None, bool or dict, default: *rcParams["legend.shadow"]* (default: False)] Whether to draw a shadow behind the legend. The shadow can be con-

figured using *Patch* keywords. Customization via *rcParams["legend.shadow"]* (default: False) is currently not supported.

**framealpha**

[float, default: *rcParams["legend.framealpha"]* (default: 0.8)] The alpha transparency of the legend's background. If *shadow* is activated and *framealpha* is None, the default value is ignored.

**facecolor**

["inherit" or color, default: *rcParams["legend.facecolor"]* (default: 'inherit')] The legend's background color. If "inherit", use *rcParams["axes.facecolor"]* (default: 'white').

**edgecolor**

["inherit" or color, default: *rcParams["legend.edgecolor"]* (default: '0.8')] The legend's background patch edge color. If "inherit", use *rcParams["axes.edgecolor"]* (default: 'black').

**mode**

[{"expand", None}] If *mode* is set to "expand" the legend will be horizontally expanded to fill the Axes area (or *bbox_to_anchor* if defines the legend's size).

**bbox_transform**

[None or *Transform*] The transform for the bounding box (*bbox_to_anchor*). For a value of None (default) the Axes' transAxes transform will be used.

**title**

[str or None] The legend's title. Default is no title (None).

**title_fontproperties**

[None or *FontProperties* or dict] The font properties of the legend's title. If None (default), the *title_fontsize* argument will be used if present; if *title_fontsize* is also None, the current *rcParams["legend.title_fontsize"]* (default: None) will be used.

**title_fontsize**

[int or {'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large'}, default: *rcParams["legend.title_fontsize"]* (default: None)] The font size of the legend's title. Note: This cannot be combined with *title_fontproperties*. If you want to set the fontsize alongside other font properties, use the *size* parameter in *title_fontproperties*.

**alignment**

[{'center', 'left', 'right'}, default: 'center'] The alignment of the legend title and the box of entries. The entries are aligned as a single block, so that markers always lined up.

**borderpad**

[float, default: *rcParams["legend.borderpad"]* (default: 0.4)] The fractional whitespace inside the legend border, in font-size units.

**labelspacing**

[float, default: *rcParams["legend.labelspacing"]* (default: 0.5)] The vertical space between the legend entries, in font-size units.

**handlelength**

[float, default: *rcParams["legend.handlelength"]* (default: 2.0)] The length of the legend handles, in font-size units.

**handleheight**

[float, default: *rcParams["legend.handleheight"]* (default: 0.7)] The height of the legend handles, in font-size units.

**handletextpad**

[float, default: *rcParams["legend.handletextpad"]* (default: 0.8)] The pad between the legend handle and text, in font-size units.

**borderaxespad**

[float, default: *rcParams["legend.borderaxespad"]* (default: 0.5)] The pad between the Axes and legend border, in font-size units.

**columnspacing**

[float, default: *rcParams["legend.columnspacing"]* (default: 2.0)] The spacing between columns, in font-size units.

**handler_map**

[dict or None] The custom dictionary mapping instances or types to a legend handler. This *handler_map* updates the default handler map found at *matplotlib.legend.Legend.get_legend_handler_map*.

**draggable**

[bool, default: False] Whether the legend can be dragged with the mouse.

**See also:**

***Axes.legend***

### Notes

Some artists are not supported by this function. See *Legend guide* for details.

**sca**(*a*)

Set the current Axes to be *a* and return *a*.

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *edgecolor=<UNSET>*, *facecolor=<UNSET>*, *frameon=<UNSET>*, *gid=<UNSET>*, *in_layout=<UNSET>*, *label=<UNSET>*, *linewidth=<UNSET>*, *mouseover=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| agg_fil | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image |
| alpha | scalar or None |
| animated | bool |
| clip_bo | BboxBase or None |
| clip_on | bool |
| clip_pa | Patch or (Path, Transform) or None |
| edgecolor | color |
| facecolor | color |
| figure | Figure |
| frameon | bool |
| gid | str |
| in_layo | bool |
| label | object |
| linewid | number |
| mouseov | bool |
| path_ef | list of AbstractPathEffect |
| picker | None or bool or float or callable |
| rasterized | bool |
| sketch_ | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| transform | Transform |
| url | str |
| visible | bool |
| zorder | float |

**set_edgecolor**(*color*)

Set the edge color of the Figure rectangle.

**Parameters**

**color**

[*color*]

**set_facecolor**(*color*)

Set the face color of the Figure rectangle.

**Parameters**

**color**

[*color*]

**set_frameon**(*b*)

Set the figure's background patch visibility, i.e. whether the figure background will be drawn. Equivalent to `Figure.patch.set_visible()`.

**Parameters**

**b**

[bool]

**set_linewidth**(*linewidth*)

Set the line width of the Figure rectangle.

**Parameters**

**linewidth**

[number]

**subfigures**(*nrows=1*, *ncols=1*, *squeeze=True*, *wspace=None*, *hspace=None*, *width_ratios=None*, *height_ratios=None*, ***kwargs*)

Add a set of subfigures to this figure or subfigure.

A subfigure has the same artist methods as a figure, and is logically the same as a figure, but cannot print itself. See *Figure subfigures*.

---

**Note:** The *subfigure* concept is new in v3.4, and the API is still provisional.

---

**Parameters**

**nrows, ncols**

[int, default: 1] Number of rows/columns of the subfigure grid.

**squeeze**

[bool, default: True] If True, extra dimensions are squeezed out from the returned array of subfigures.

**wspace, hspace**

[float, default: None] The amount of width/height reserved for space between subfigures, expressed as a fraction of the average subfigure width/height. If not given, the values will be inferred from rcParams if using constrained layout (see `ConstrainedLayoutEngine`), or zero if not using a layout engine.

**width_ratios**

[array-like of length *ncols*, optional] Defines the relative widths of the columns. Each column gets a relative width of `width_ratios[i] / sum(width_ratios)`. If not given, all columns will have the same width.

**height_ratios**

[array-like of length *nrows*, optional] Defines the relative heights of the rows. Each row gets a relative height of `height_ratios[i] / sum(height_ratios)`. If not given, all rows will have the same height.

**subplot_mosaic**(*mosaic*, *, *sharex=False*, *sharey=False*, *width_ratios=None*, *height_ratios=None*, *empty_sentinel='.'*, *subplot_kw=None*, *per_subplot_kw=None*, *gridspec_kw=None*)

Build a layout of Axes based on ASCII art or nested lists.

This is a helper function to build complex GridSpec layouts visually.

See *Complex and semantic figure composition (subplot_mosaic)* for an example and full API documentation

**Parameters**

**mosaic**

[list of list of {hashable or nested} or str] A visual layout of how you want your Axes to be arranged labeled as strings. For example

```
x = [['A panel', 'A panel', 'edge'],
     ['C panel', '.',       'edge']]
```

produces 4 Axes:

- 'A panel' which is 1 row high and spans the first two columns
- 'edge' which is 2 rows high and is on the right edge
- 'C panel' which in 1 row and 1 column wide in the bottom left
- a blank space 1 row and 1 column wide in the bottom center

Any of the entries in the layout can be a list of lists of the same form to create nested layouts.

If input is a str, then it can either be a multi-line string of the form

```
'''
AAE
```

(continues on next page)

(continued from previous page)

```
C.E
'''
```

where each character is a column and each line is a row. Or it can be a single-line string where rows are separated by `;`:

```
'AB;CC'
```

The string notation allows only single character Axes labels and does not support nesting but is very terse.

The Axes identifiers may be `str` or a non-iterable hashable object (e.g. `tuple`s may not be used).

**sharex, sharey**

[bool, default: False] If True, the x-axis (*sharex*) or y-axis (*sharey*) will be shared among all subplots. In that case, tick label visibility and axis units behave as for *subplots*. If False, each subplot's x- or y-axis will be independent.

**width_ratios**

[array-like of length *ncols*, optional] Defines the relative widths of the columns. Each column gets a relative width of `width_ratios[i] / sum(width_ratios)`. If not given, all columns will have the same width. Equivalent to `gridspec_kw={'width_ratios': [...]}`. In the case of nested layouts, this argument applies only to the outer layout.

**height_ratios**

[array-like of length *nrows*, optional] Defines the relative heights of the rows. Each row gets a relative height of `height_ratios[i] / sum(height_ratios)`. If not given, all rows will have the same height. Equivalent to `gridspec_kw={'height_ratios': [...]}`. In the case of nested layouts, this argument applies only to the outer layout.

**subplot_kw**

[dict, optional] Dictionary with keywords passed to the *Figure.add_subplot* call used to create each subplot. These values may be overridden by values in *per_subplot_kw*.

**per_subplot_kw**

[dict, optional] A dictionary mapping the Axes identifiers or tuples of identifiers to a dictionary of keyword arguments to be passed to the *Figure.add_subplot* call used to create each subplot. The values in these dictionaries have precedence over the values in *subplot_kw*.

If *mosaic* is a string, and thus all keys are single characters, it is possible to use a single string instead of a tuple as keys; i.e. `"AB"` is equivalent to `("A", "B")`.

New in version 3.7.

**gridspec_kw**

[dict, optional] Dictionary with keywords passed to the *GridSpec* constructor used to create the grid the subplots are placed on. In the case of nested layouts, this argument applies only to the outer layout. For more complex layouts, users should use *Figure.subfigures* to create the nesting.

**empty_sentinel**

[object, optional] Entry in the layout to mean "leave this space empty". Defaults to `'.'`. Note, if *layout* is a string, it is processed via `inspect.cleandoc` to remove leading white space, which may interfere with using white-space as the empty sentinel.

**Returns**

**dict[label, Axes]**

A dictionary mapping the labels to the Axes objects. The order of the Axes is left-to-right and top-to-bottom of their position in the total layout.

**subplots**(*nrows=1*, *ncols=1*, ***, *sharex=False*, *sharey=False*, *squeeze=True*, *width_ratios=None*, *height_ratios=None*, *subplot_kw=None*, *gridspec_kw=None*)

Add a set of subplots to this figure.

This utility wrapper makes it convenient to create common layouts of subplots in a single call.

**Parameters**

**nrows, ncols**

[int, default: 1] Number of rows/columns of the subplot grid.

**sharex, sharey**

[bool or {'none', 'all', 'row', 'col'}, default: False] Controls sharing of x-axis (*sharex*) or y-axis (*sharey*):

- True or 'all': x- or y-axis will be shared among all subplots.
- False or 'none': each subplot x- or y-axis will be independent.
- 'row': each subplot row will share an x- or y-axis.
- 'col': each subplot column will share an x- or y-axis.

When subplots have a shared x-axis along a column, only the x tick labels of the bottom subplot are created. Similarly, when subplots have a shared y-axis along a row, only the y tick labels of the first column subplot are created. To later turn other subplots' ticklabels on, use *tick_params*.

When subplots have a shared axis that has units, calling *Axis.set_units* will update each axis with the new units.

**squeeze**

[bool, default: True]

- If True, extra dimensions are squeezed out from the returned array of Axes:
  - if only one subplot is constructed (nrows=ncols=1), the resulting single Axes object is returned as a scalar.
  - for Nx1 or 1xM subplots, the returned object is a 1D numpy object array of Axes objects.
  - for NxM, subplots with N>1 and M>1 are returned as a 2D array.
- If False, no squeezing at all is done: the returned Axes object is always a 2D array containing Axes instances, even if it ends up being 1x1.

**width_ratios**

[array-like of length *ncols*, optional] Defines the relative widths of the columns. Each column gets a relative width of `width_ratios[i] / sum(width_ratios)`. If not given, all columns will have the same width. Equivalent to `gridspec_kw={'width_ratios': [...]}`.

**height_ratios**

[array-like of length *nrows*, optional] Defines the relative heights of the rows. Each row gets a relative height of `height_ratios[i] / sum(height_ratios)`. If not given, all rows will have the same height. Equivalent to `gridspec_kw={'height_ratios': [...]}`.

**subplot_kw**

[dict, optional] Dict with keywords passed to the *Figure.add_subplot* call used to create each subplot.

**gridspec_kw**

[dict, optional] Dict with keywords passed to the *GridSpec* constructor used to create the grid the subplots are placed on.

**Returns**

***Axes* or array of Axes**

Either a single *Axes* object or an array of Axes objects if more than one subplot was created. The dimensions of the resulting array can be controlled with the *squeeze* keyword, see above.

**See also:**

***pyplot.subplots***
***Figure.add_subplot***
***pyplot.subplot***

### Examples

```
# First create some toy data:
x = np.linspace(0, 2*np.pi, 400)
y = np.sin(x**2)

# Create a figure
fig = plt.figure()

# Create a subplot
ax = fig.subplots()
ax.plot(x, y)
ax.set_title('Simple plot')

# Create two subplots and unpack the output array immediately
ax1, ax2 = fig.subplots(1, 2, sharey=True)
ax1.plot(x, y)
ax1.set_title('Sharing Y axis')
ax2.scatter(x, y)

# Create four polar Axes and access them through the returned array
axes = fig.subplots(2, 2, subplot_kw=dict(projection='polar'))
axes[0, 0].plot(x, y)
axes[1, 1].scatter(x, y)

# Share an X-axis with each column of subplots
fig.subplots(2, 2, sharex='col')

# Share a Y-axis with each row of subplots
fig.subplots(2, 2, sharey='row')

# Share both X- and Y-axes with all subplots
fig.subplots(2, 2, sharex='all', sharey='all')

# Note that this is the same as
fig.subplots(2, 2, sharex=True, sharey=True)
```

**subplots_adjust**(*left=None*, *bottom=None*, *right=None*, *top=None*, *wspace=None*, *hspace=None*)

Adjust the subplot layout parameters.

Unset parameters are left unmodified; initial values are given by *rcParams["figure.subplot.[name]"]*.

#### Parameters

**left**

[float, optional] The position of the left edge of the subplots, as a fraction of the figure width.

**right**

[float, optional] The position of the right edge of the subplots, as a fraction of the figure width.

**bottom**

[float, optional] The position of the bottom edge of the subplots, as a fraction of the figure height.

**top**

[float, optional] The position of the top edge of the subplots, as a fraction of the figure height.

**wspace**

[float, optional] The width of the padding between subplots, as a fraction of the average Axes width.

**hspace**

[float, optional] The height of the padding between subplots, as a fraction of the average Axes height.

**suptitle**(*t*, ***kwargs*)

Add a centered suptitle to the figure.

**Parameters**

**t**

[str] The suptitle text.

**x**

[float, default: 0.5] The x location of the text in figure coordinates.

**y**

[float, default: 0.98] The y location of the text in figure coordinates.

**horizontalalignment, ha**

[{'center', 'left', 'right'}, default: center] The horizontal alignment of the text relative to (*x*, *y*).

**verticalalignment, va**

[{'top', 'center', 'bottom', 'baseline'}, default: top] The vertical alignment of the text relative to (*x*, *y*).

**fontsize, size**

[default: *rcParams["figure.titlesize"]* (default: `'large'`)] The font size of the text. See *Text.set_size* for possible values.

**fontweight, weight**

[default: *rcParams["figure.titleweight"]* (default: 'normal')] The font weight of the text. See *Text.set_weight* for possible values.

**Returns**

**text**

The *Text* instance of the suptitle.

**Other Parameters**

**fontproperties**

[None or dict, optional] A dict of font properties. If *fontproperties* is given the default values for font size and weight are taken from the *FontProperties* defaults. *rcParams["figure.titlesize"]* (default: 'large') and *rcParams["figure.titleweight"]* (default: 'normal') are ignored in this case.

****kwargs**

Additional kwargs are *matplotlib.text.Text* properties.

**supxlabel**(*t*, ***kwargs*)

Add a centered supxlabel to the figure.

**Parameters**

**t**

[str] The supxlabel text.

**x**

[float, default: 0.5] The x location of the text in figure coordinates.

**y**

[float, default: 0.01] The y location of the text in figure coordinates.

**horizontalalignment, ha**

[{'center', 'left', 'right'}, default: center] The horizontal alignment of the text relative to (*x*, *y*).

**verticalalignment, va**

[{'top', 'center', 'bottom', 'baseline'}, default: bottom] The vertical alignment of the text relative to (*x*, *y*).

**fontsize, size**

[default: *rcParams["figure.labelsize"]* (default: 'large')] The font size of the text. See *Text.set_size* for possible values.

**fontweight, weight**

[default: *rcParams["figure.labelweight"]* (default: 'normal')] The font weight of the text. See *Text.set_weight* for possible values.

**Returns**

**text**

The *Text* instance of the supxlabel.

**Other Parameters**

**fontproperties**

[None or dict, optional] A dict of font properties. If *fontproperties* is given the default values for font size and weight are taken from the *FontProperties* defaults. *rcParams["figure.labelsize"]* (default: 'large') and *rcParams["figure.labelweight"]* (default: 'normal') are ignored in this case.

****kwargs**

Additional kwargs are *matplotlib.text.Text* properties.

**supylabel**(*t*, ***kwargs*)

Add a centered supylabel to the figure.

**Parameters**

**t**

[str] The supylabel text.

**x**

[float, default: 0.02] The x location of the text in figure coordinates.

**y**

[float, default: 0.5] The y location of the text in figure coordinates.

**horizontalalignment, ha**

[{'center', 'left', 'right'}, default: left] The horizontal alignment of the text relative to (*x*, *y*).

**verticalalignment, va**

[{'top', 'center', 'bottom', 'baseline'}, default: center] The vertical alignment of the text relative to (*x*, *y*).

**fontsize, size**

[default: *rcParams["figure.labelsize"]* (default: 'large')] The font size of the text. See *Text.set_size* for possible values.

**fontweight, weight**

[default: *rcParams["figure.labelweight"]* (default: 'normal')] The font weight of the text. See *Text.set_weight* for possible values.

**Returns**

**text**

The *Text* instance of the supylabel.

**Other Parameters**

**fontproperties**

[None or dict, optional] A dict of font properties. If *fontproperties* is given the default values for font size and weight are taken from the *FontProperties* defaults. *rcParams["figure.labelsize"]* (default: 'large') and *rcParams["figure.labelweight"]* (default: 'normal') are ignored in this case.

****kwargs**

Additional kwargs are *matplotlib.text.Text* properties.

**text**(*x*, *y*, *s*, *fontdict=None*, ***kwargs*)

Add text to figure.

**Parameters**

**x, y**

[float] The position to place the text. By default, this is in figure coordinates, floats in [0, 1]. The coordinate system can be changed using the *transform* keyword.

**s**

[str] The text string.

**fontdict**

[dict, optional] A dictionary to override the default text properties. If not given, the defaults are determined by *rcParams["font.*"]*. Properties passed as *kwargs* override the corresponding ones given in *fontdict*.

**Returns**

***Text***

**Other Parameters**

****kwargs**

[*Text* properties] Other miscellaneous text parameters.

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* | bool |
| *backgroundcolor* | *color* |
| *bbox* | dict with properties for *patches.FancyBboxP* |
| *clip_box* | unknown |
| *clip_on* | unknown |
| *clip_path* | unknown |
| *color* or c | *color* |
| *figure* | *Figure* |
| *fontfamily* or family or fontname | {FONTNAME, 'serif', 'sans-serif', 'cursive', 'fantas |
| *fontproperties* or font or font_properties | *font_manager.FontProperties* or *str* c |
| *fontsize* or size | float or {'xx-small', 'x-small', 'small', 'medium', 'la |
| *fontstretch* or stretch | {a numeric value in range 0-1000, 'ultra-condense |
| *fontstyle* or style | {'normal', 'italic', 'oblique'} |
| *fontvariant* or variant | {'normal', 'small-caps'} |
| *fontweight* or weight | {a numeric value in range 0-1000, 'ultralight', 'ligh |
| *gid* | str |
| *horizontalalignment* or ha | {'left', 'center', 'right'} |
| *in_layout* | bool |
| *label* | object |
| *linespacing* | float (multiple of font size) |
| *math_fontfamily* | str |
| *mouseover* | bool |
| *multialignment* or ma | {'left', 'right', 'center'} |
| *parse_math* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *position* | (float, float) |
| *rasterized* | bool |
| *rotation* | float or {'vertical', 'horizontal'} |
| *rotation_mode* | {None, 'default', 'anchor'} |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *text* | object |
| *transform* | *Transform* |
| *transform_rotates_text* | bool |
| *url* | str |
| *usetex* | bool, default: *rcParams["text.usetex"]* ( |
| *verticalalignment* or va | {'baseline', 'bottom', 'center', 'center_baseline', 'top |

Table 58

| Property | Description |
|---|---|
| *visible* | bool |
| *wrap* | bool |
| *x* | float |
| *y* | float |
| *zorder* | float |

**See also:**

***Axes.text***
***pyplot.text***

## Helper functions

matplotlib.figure.**figaspect**(*arg*)

Calculate the width and height for a figure with a specified aspect ratio.

While the height is taken from *rcParams["figure.figsize"]* (default: `[6.4, 4.8]`), the width is adjusted to match the desired aspect ratio. Additionally, it is ensured that the width is in the range [4., 16.] and the height is in the range [2., 16.]. If necessary, the default height is adjusted to ensure this.

**Parameters**

**arg**

[float or 2D array] If a float, this defines the aspect ratio (i.e. the ratio height / width). In case of an array the aspect ratio is number of rows / number of columns, so that the array could be fitted in the figure undistorted.

**Returns**

**width, height**

[float] The figure size in inches.

### Notes

If you want to create an Axes within the figure, that still preserves the aspect ratio, be sure to create it with equal width and height. See examples below.

Thanks to Fernando Perez for this function.

### Examples

Make a figure twice as tall as it is wide:

```
w, h = figaspect(2.)
fig = Figure(figsize=(w, h))
ax = fig.add_axes([0.1, 0.1, 0.8, 0.8])
ax.imshow(A, **kwargs)
```

Make a figure with the proper aspect for an array:

```
A = rand(5, 3)
w, h = figaspect(A)
fig = Figure(figsize=(w, h))
ax = fig.add_axes([0.1, 0.1, 0.8, 0.8])
ax.imshow(A, **kwargs)
```

## 7.2.24 matplotlib.font_manager

A module for finding, managing, and using fonts across platforms.

This module provides a single *FontManager* instance, `fontManager`, that can be shared across backends and platforms. The *findfont* function returns the best TrueType (TTF) font file in the local or system font path that matches the specified *FontProperties* instance. The *FontManager* also handles Adobe Font Metrics (AFM) font files for use by the PostScript backend. The *FontManager.addfont* function adds a custom font from a file without installing it into your operating system.

The design is based on the W3C Cascading Style Sheet, Level 1 (CSS1) font specification. Future versions may implement the Level 2 or 2.1 specifications.

**class** matplotlib.font_manager.**FontManager**(*size=None*, *weight='normal'*)

Bases: `object`

On import, the *FontManager* singleton instance creates a list of ttf and afm fonts and caches their *FontProperties*. The *FontManager.findfont* method does a nearest neighbor search to find the font that most closely matches the specification. If no good enough match is found, the default font is returned.

Fonts added with the *FontManager.addfont* method will not persist in the cache; therefore, *addfont* will need to be called every time Matplotlib is imported. This method should only be used if and when a font cannot be installed on your operating system by other means.

### Notes

The *FontManager.addfont* method must be called on the global *FontManager* instance.

Example usage:

```
import matplotlib.pyplot as plt
from matplotlib import font_manager

font_dirs = ["/resources/fonts"]  # The path to the custom font file.
font_files = font_manager.findSystemFonts(fontpaths=font_dirs)

for font_file in font_files:
    font_manager.fontManager.addfont(font_file)
```

**addfont**(*path*)

Cache the properties of the font at *path* to make it available to the *FontManager*. The type of font is inferred from the path suffix.

**Parameters**

**path**

[str or path-like]

### Notes

This method is useful for adding a custom font without installing it in your operating system. See the *FontManager* singleton instance for usage and caveats about this function.

**property defaultFont**

**findfont**(*prop*, *fontext='ttf'*, *directory=None*, *fallback_to_default=True*, *rebuild_if_missing=True*)

Find the path to the font file most closely matching the given font properties.

**Parameters**

**prop**

[str or *FontProperties*] The font properties to search for. This can be either a *FontProperties* object or a string defining a fontconfig patterns.

**fontext**

[{'ttf', 'afm'}, default: 'ttf'] The extension of the font file:

- 'ttf': TrueType and OpenType fonts (.ttf, .ttc, .otf)
- 'afm': Adobe Font Metrics (.afm)

**directory**

[str, optional] If given, only search this directory and its subdirectories.

**fallback_to_default**

[bool] If True, will fall back to the default font family (usually "DejaVu Sans" or "Helvetica") if the first lookup hard-fails.

**rebuild_if_missing**

[bool] Whether to rebuild the font cache and search again if the first match appears to point to a nonexisting font (i.e., the font cache contains outdated entries).

**Returns**

**str**

The filename of the best matching font.

### Notes

This performs a nearest neighbor search. Each font is given a similarity score to the target font properties. The first font with the highest score is returned. If no matches below a certain threshold are found, the default font (usually DejaVu Sans) is returned.

The result is cached, so subsequent lookups don't have to perform the O(n) nearest neighbor search.

See the W3C Cascading Style Sheet, Level 1 documentation for a description of the font finding algorithm.

**static get_default_size**()

Return the default font size.

**get_default_weight**()

Return the default font weight.

**get_font_names**()

Return the list of available fonts.

**score_family**(*families*, *family2*)

Return a match score between the list of font families in *families* and the font family name *family2*.

An exact match at the head of the list returns 0.0.

A match further down the list will return between 0 and 1.

No match will return 1.0.

**score_size**(*size1*, *size2*)

Return a match score between *size1* and *size2*.

If *size2* (the size specified in the font file) is 'scalable', this function always returns 0.0, since any font size can be generated.

Otherwise, the result is the absolute distance between *size1* and *size2*, normalized so that the usual range of font sizes (6pt - 72pt) will lie between 0.0 and 1.0.

**score_stretch**(*stretch1*, *stretch2*)

Return a match score between *stretch1* and *stretch2*.

The result is the absolute value of the difference between the CSS numeric values of *stretch1* and *stretch2*, normalized between 0.0 and 1.0.

**score_style**(*style1*, *style2*)

Return a match score between *style1* and *style2*.

An exact match returns 0.0.

A match between 'italic' and 'oblique' returns 0.1.

No match returns 1.0.

**score_variant**(*variant1*, *variant2*)

Return a match score between *variant1* and *variant2*.

An exact match returns 0.0, otherwise 1.0.

**score_weight**(*weight1*, *weight2*)

Return a match score between *weight1* and *weight2*.

The result is 0.0 if both weight1 and weight 2 are given as strings and have the same value.

Otherwise, the result is the absolute value of the difference between the CSS numeric values of *weight1* and *weight2*, normalized between 0.05 and 1.0.

**set_default_weight**(*weight*)

Set the default font weight. The initial value is 'normal'.

**class** matplotlib.font_manager.**FontProperties**(*family=None*, *style=None*, *variant=None*, *weight=None*, *stretch=None*, *size=None*, *fname=None*, *math_fontfamily=None*)

Bases: object

A class for storing and manipulating font properties.

The font properties are the six properties described in the W3C Cascading Style Sheet, Level 1 font specification and *math_fontfamily* for math fonts:

- family: A list of font names in decreasing order of priority. The items may include a generic font family name, either 'sans-serif', 'serif', 'cursive', 'fantasy', or 'monospace'. In that case, the actual font to be used will be looked up from the associated rcParam during the search process in *findfont*. Default: *rcParams["font.family"]* (default: ['sans-serif'])
- style: Either 'normal', 'italic' or 'oblique'. Default: *rcParams["font.style"]* (default: 'normal')
- variant: Either 'normal' or 'small-caps'. Default: *rcParams["font.variant"]* (default: 'normal')
- stretch: A numeric value in the range 0-1000 or one of 'ultra-condensed', 'extra-condensed', 'condensed', 'semi-condensed', 'normal', 'semi-expanded', 'expanded', 'extra-expanded' or 'ultra-expanded'. Default: *rcParams["font.stretch"]* (default: 'normal')

- weight: A numeric value in the range 0-1000 or one of 'ultralight', 'light', 'normal', 'regular', 'book', 'medium', 'roman', 'semibold', 'demibold', 'demi', 'bold', 'heavy', 'extra bold', 'black'. Default: *rcParams["font.weight"]* (default: 'normal')
- size: Either a relative value of 'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large' or an absolute font size, e.g., 10. Default: *rcParams["font.size"]* (default: 10.0)
- math_fontfamily: The family of fonts used to render math text. Supported values are: 'dejavusans', 'dejavuserif', 'cm', 'stix', 'stixsans' and 'custom'. Default: *rcParams["mathtext.fontset"]* (default: 'dejavusans')

Alternatively, a font may be specified using the absolute path to a font file, by using the *fname* kwarg. However, in this case, it is typically simpler to just pass the path (as a pathlib.Path, not a str) to the *font* kwarg of the *Text* object.

The preferred usage of font sizes is to use the relative values, e.g., 'large', instead of absolute font sizes, e.g., 12. This approach allows all text sizes to be made larger or smaller based on the font manager's default font size.

This class will also accept a fontconfig pattern, if it is the only argument provided. This support does not depend on fontconfig; we are merely borrowing its pattern syntax for use here.

Note that Matplotlib's internal font manager and fontconfig use a different algorithm to lookup fonts, so the results of the same pattern may be different in Matplotlib than in other applications that use fontconfig.

**copy**()

Return a copy of self.

**get_family**()

Return a list of individual font family names or generic family names.

The font families or generic font families (which will be resolved from their respective rcParams when searching for a matching font) in the order of preference.

**get_file**()

Return the filename of the associated font.

**get_fontconfig_pattern**()

Get a fontconfig pattern suitable for looking up the font as specified with fontconfig's fc-match utility.

This support does not depend on fontconfig; we are merely borrowing its pattern syntax for use here.

**get_math_fontfamily**()

Return the name of the font family used for math text.

The default font is *rcParams["mathtext.fontset"]* (default: 'dejavusans').

**get_name**()

Return the name of the font that best matches the font properties.

**get_size**()

Return the font size.

**get_size_in_points**()

Return the font size.

**get_slant**()

Return the font style. Values are: 'normal', 'italic' or 'oblique'.

**get_stretch**()

Return the font stretch or width. Options are: 'ultra-condensed', 'extra-condensed', 'condensed', 'semi-condensed', 'normal', 'semi-expanded', 'expanded', 'extra-expanded', 'ultra-expanded'.

**get_style**()

Return the font style. Values are: 'normal', 'italic' or 'oblique'.

**get_variant**()

Return the font variant. Values are: 'normal' or 'small-caps'.

**get_weight**()

Set the font weight. Options are: A numeric value in the range 0-1000 or one of 'light', 'normal', 'regular', 'book', 'medium', 'roman', 'semibold', 'demibold', 'demi', 'bold', 'heavy', 'extra bold', 'black'

**set_family**(*family*)

Change the font family. Can be either an alias (generic name is CSS parlance), such as: 'serif', 'sans-serif', 'cursive', 'fantasy', or 'monospace', a real font name or a list of real font names. Real font names are not supported when *rcParams["text.usetex"]* (default: `False`) is `True`. Default: *rcParams["font.family"]* (default: `['sans-serif']`)

**set_file**(*file*)

Set the filename of the fontfile to use. In this case, all other properties will be ignored.

**set_fontconfig_pattern**(*pattern*)

Set the properties by parsing a fontconfig *pattern*.

This support does not depend on fontconfig; we are merely borrowing its pattern syntax for use here.

**set_math_fontfamily**(*fontfamily*)

Set the font family for text in math mode.

If not set explicitly, *rcParams["mathtext.fontset"]* (default: `'dejavusans'`) will be used.

**Parameters**

**fontfamily**

[str] The name of the font family.

Available font families are defined in the *default matplotlibrc file*.

**See also:**

*text.Text.get_math_fontfamily*

**set_name**(*family*)

Change the font family. Can be either an alias (generic name is CSS parlance), such as: 'serif', 'sans-serif', 'cursive', 'fantasy', or 'monospace', a real font name or a list of real font names. Real font names are not supported when *rcParams["text.usetex"]* (default: `False`) is `True`. Default: *rcParams["font.family"]* (default: `['sans-serif']`)

**set_size**(*size*)

Set the font size.

**Parameters**

**size**

[float or {'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large'}, default: *rcParams["font.size"]* (default: `10.0`)] If a float, the font size in points. The string values denote sizes relative to the default font size.

**set_slant**(*style*)

Set the font style.

**Parameters**

**style**

[{'normal', 'italic', 'oblique'}, default: *rcParams["font.style"]* (default: `'normal'`)]

**set_stretch**(*stretch*)

Set the font stretch or width.

**Parameters**

**stretch**

[int or {'ultra-condensed', 'extra-condensed', 'condensed', 'semi-condensed', 'normal', 'semi-expanded', 'expanded', 'extra-expanded', 'ultra-expanded'}, default: *rcParams["font.stretch"]* (default: `'normal'`)] If int, must be in the range 0-1000.

**set_style**(*style*)

Set the font style.

**Parameters**

**style**

[{'normal', 'italic', 'oblique'}, default: *rcParams["font.style"]* (default: 'normal')]

**set_variant**(*variant*)

Set the font variant.

**Parameters**

**variant**

[{'normal', 'small-caps'}, default: *rcParams["font.variant"]* (default: 'normal')]

**set_weight**(*weight*)

Set the font weight.

**Parameters**

**weight**

[int or {'ultralight', 'light', 'normal', 'regular', 'book', 'medium', 'roman', 'semibold', 'demibold', 'demi', 'bold', 'heavy', 'extra bold', 'black'}, default: *rcParams["font.weight"]* (default: 'normal')] If int, must be in the range 0-1000.

matplotlib.font_manager.**afmFontProperty**(*fontpath*, *font*)

Extract information from an AFM font file.

**Parameters**

**fontpath**

[str] The filename corresponding to *font*.

**font**

[AFM] The AFM font file from which information will be extracted.

**Returns**

***FontEntry***

The extracted font properties.

matplotlib.font_manager.**findSystemFonts**(*fontpaths=None*, *fontext='ttf'*)

Search for fonts in the specified font paths. If no paths are given, will use a standard set of system paths, as well as the list of fonts tracked by fontconfig if fontconfig is installed and available. A list of TrueType fonts are returned by default with AFM fonts as an option.

matplotlib.font_manager.**findfont**(*prop*, *fontext='ttf'*, *directory=None*, *fallback_to_default=True*, *rebuild_if_missing=True*)

Find the path to the font file most closely matching the given font properties.

**Parameters**

**prop**

[str or *FontProperties*] The font properties to search for. This can be either a *FontProperties* object or a string defining a fontconfig patterns.

**fontext**

[{'ttf', 'afm'}, default: 'ttf'] The extension of the font file:

- 'ttf': TrueType and OpenType fonts (.ttf, .ttc, .otf)
- 'afm': Adobe Font Metrics (.afm)

**directory**

[str, optional] If given, only search this directory and its subdirectories.

**fallback_to_default**

[bool] If True, will fall back to the default font family (usually "DejaVu Sans" or "Helvetica") if the first lookup hard-fails.

**rebuild_if_missing**

[bool] Whether to rebuild the font cache and search again if the first match appears to point to a nonexisting font (i.e., the font cache contains outdated entries).

**Returns**

**str**

The filename of the best matching font.

### Notes

This performs a nearest neighbor search. Each font is given a similarity score to the target font properties. The first font with the highest score is returned. If no matches below a certain threshold are found, the default font (usually DejaVu Sans) is returned.

The result is cached, so subsequent lookups don't have to perform the O(n) nearest neighbor search.

See the W3C Cascading Style Sheet, Level 1 documentation for a description of the font finding algorithm.

matplotlib.font_manager.**get_font**(*font_filepaths*, *hinting_factor=None*)

Get an *ft2font.FT2Font* object given a list of file paths.

**Parameters**

**font_filepaths**

[Iterable[str, Path, bytes], str, Path, bytes] Relative or absolute paths to the font files to be used.

If a single string, bytes, or `pathlib.Path`, then it will be treated as a list with that entry only.

If more than one filepath is passed, then the returned FT2Font object will fall back through the fonts, in the order given, to find a needed glyph.

**Returns**

*ft2font.FT2Font*

matplotlib.font_manager.**get_font_names**()

Return the list of available fonts.

matplotlib.font_manager.**get_fontext_synonyms**(*fontext*)

Return a list of file extensions that are synonyms for the given file extension *fileext*.

matplotlib.font_manager.**is_opentype_cff_font**(*filename*)

Return whether the given font is a Postscript Compact Font Format Font embedded in an OpenType wrapper. Used by the PostScript and PDF backends that cannot subset these fonts.

matplotlib.font_manager.**json_dump**(*data*, *filename*)

Dump *FontManager* *data* as JSON to the file named *filename*.

**See also:**

*json_load*

### Notes

File paths that are children of the Matplotlib data path (typically, fonts shipped with Matplotlib) are stored relative to that data path (to remain valid across virtualenvs).

This function temporarily locks the output file to prevent multiple processes from overwriting one another's output.

matplotlib.font_manager.**json_load**(*filename*)

Load a *FontManager* from the JSON file named *filename*.

**See also:**

*json_dump*

matplotlib.font_manager.**list_fonts**(*directory*, *extensions*)

Return a list of all fonts matching any of the extensions, found recursively under the directory.

matplotlib.font_manager.**ttfFontProperty**(*font*)

Extract information from a TrueType font file.

**Parameters**

**font**

[*FT2Font*] The TrueType font file from which information will be extracted.

**Returns**

***FontEntry***

The extracted font properties.

matplotlib.font_manager.**win32FontDirectory**()

Return the user-specified font directory for Win32. This is looked up from the registry key

```
\\HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Explorer\
↪Shell Folders\Fonts
```

If the key is not found, `%WINDIR%\Fonts` will be returned.

matplotlib.font_manager.**fontManager**

The global instance of *FontManager*.

**class** matplotlib.font_manager.**FontEntry**(*fname=''*, *name=''*, *style='normal'*, *variant='normal'*, *weight='normal'*, *stretch='normal'*, *size='medium'*)

A class for storing Font properties.

It is used when populating the font lookup dictionary.

### 7.2.25 matplotlib.ft2font

**class** matplotlib.ft2font.**FT2Font**(*filename*, *hinting_factor=8*, ***, *_fallback_list=None*, *_kerning_factor=0*)

Bases: object

Create a new FT2Font object.

**Parameters**

**filename**

[str or file-like] The source of the font data in a format (ttf or ttc) that FreeType can read

**hinting_factor**

[int, optional] Must be positive. Used to scale the hinting in the x-direction

**_fallback_list**

[list of FT2Font, optional] A list of FT2Font objects used to find missing glyphs.

> **Warning:** This API is both private and provisional: do not use it directly

**_kerning_factor**

[int, optional] Used to adjust the degree of kerning.

> **Warning:** This API is private: do not use it directly

**Attributes**

**num_faces**

[int] Number of faces in file.

**face_flags, style_flags**

[int] Face and style flags; see the ft2font constants.

**num_glyphs**

[int] Number of glyphs in the face.

**family_name, style_name**

[str] Face family and style name.

**num_fixed_sizes**

[int] Number of bitmap in the face.

**scalable**

[bool] Whether face is scalable; attributes after this one are only defined for scalable faces.

**bbox**

[tuple[int, int, int, int]] Face global bounding box (xmin, ymin, xmax, ymax).

**units_per_EM**

[int] Number of font units covered by the EM.

**ascender, descender**

[int] Ascender and descender in 26.6 units.

**height**

[int] Height in 26.6 units; used to compute a default line spacing (baseline-to-baseline distance).

**max_advance_width, max_advance_height**

[int] Maximum horizontal and vertical cursor advance for all glyphs.

**underline_position, underline_thickness**

[int] Vertical position and thickness of the underline bar.

**postscript_name**

[str] PostScript name of the font.

**ascender**

**bbox**

**clear**(*self*)

Clear all the glyphs, reset for a new call to *set_text*.

**descender**

**draw_glyph_to_bitmap**(*self*, *image*, *x*, *y*, *glyph*, *antialiased=True*)

Draw a single glyph to the bitmap at pixel locations x, y Note it is your responsibility to set up the bitmap manually with `set_bitmap_size(w, h)` before this call is made.

If you want automatic layout, use *set_text* in combinations with *draw_glyphs_to_bitmap*. This function is instead intended for people who want to render individual glyphs (e.g., returned by *load_char*) at precise locations.

**draw_glyphs_to_bitmap**(*self*, *antialiased=True*)

Draw the glyphs that were loaded by *set_text* to the bitmap. The bitmap size will be automatically set to include the glyphs.

**face_flags**

**family_name**

**fname**

**get_bitmap_offset**(*self*)

Get the (x, y) offset in 26.6 subpixels for the bitmap if ink hangs left or below (0, 0). Since Matplotlib only supports left-to-right text, y is always 0.

**get_char_index**(*self*, *codepoint*)

Return the glyph index corresponding to a character *codepoint*.

**get_charmap**(*self*)

Return a dict that maps the character codes of the selected charmap (Unicode by default) to their corresponding glyph indices.

**get_descent**(*self*)

Get the descent in 26.6 subpixels of the current string set by *set_text*. The rotation of the string is accounted for. To get the descent in pixels, divide this value by 64.

**get_glyph_name**(*self*, *index*)

Retrieve the ASCII name of a given glyph *index* in a face.

Due to Matplotlib's internal design, for fonts that do not contain glyph names (per FT_FACE_FLAG_GLYPH_NAMES), this returns a made-up name which does *not* roundtrip through *get_name_index*.

**get_image**(*self*)

Return the underlying image buffer for this font object.

**get_kerning**(*self*, *left*, *right*, *mode*)

Get the kerning between *left* and *right* glyph indices. *mode* is a kerning mode constant:

- KERNING_DEFAULT - Return scaled and grid-fitted kerning distances
- KERNING_UNFITTED - Return scaled but un-grid-fitted kerning distances
- KERNING_UNSCALED - Return the kerning vector in original font units

**get_name_index**(*self*, *name*)

Return the glyph index of a given glyph *name*. The glyph index 0 means 'undefined character code'.

**get_num_glyphs**(*self*)

Return the number of loaded glyphs.

**get_path**(*self*)

Get the path data from the currently loaded glyph as a tuple of vertices, codes.

**get_ps_font_info**(*self*)

Return the information in the PS Font Info structure.

**get_sfnt**(*self*)

Load the entire SFNT names table, as a dict whose keys are (platform-ID, ISO-encoding-scheme, language-code, and description) tuples.

**get_sfnt_table**(*self*, *name*)

Return one of the following SFNT tables: head, maxp, OS/2, hhea, vhea, post, or pclt.

**get_width_height**(*self*)

Get the width and height in 26.6 subpixels of the current string set by *set_text*. The rotation of the string is accounted for. To get width and height in pixels, divide these values by 64.

**get_xys**(*self*, *antialiased=True*)

Get the xy locations of the current glyphs.

Deprecated since version 3.8.

**height**

**load_char**(*self*, *charcode*, *flags=32*)

Load character with *charcode* in current fontfile and set glyph. *flags* can be a bitwise-or of the LOAD_XXX constants; the default value is LOAD_FORCE_AUTOHINT. Return value is a Glyph object, with attributes

- width: glyph width
- height: glyph height
- bbox: the glyph bbox (xmin, ymin, xmax, ymax)
- horiBearingX: left side bearing in horizontal layouts
- horiBearingY: top side bearing in horizontal layouts
- horiAdvance: advance width for horizontal layout
- vertBearingX: left side bearing in vertical layouts
- vertBearingY: top side bearing in vertical layouts
- vertAdvance: advance height for vertical layout

**load_glyph**(*self*, *glyphindex*, *flags=32*)

Load character with *glyphindex* in current fontfile and set glyph. *flags* can be a bitwise-or of the LOAD_XXX constants; the default value is LOAD_FORCE_AUTOHINT. Return value is a Glyph object, with attributes

- width: glyph width
- height: glyph height
- bbox: the glyph bbox (xmin, ymin, xmax, ymax)
- horiBearingX: left side bearing in horizontal layouts
- horiBearingY: top side bearing in horizontal layouts
- horiAdvance: advance width for horizontal layout
- vertBearingX: left side bearing in vertical layouts
- vertBearingY: top side bearing in vertical layouts
- vertAdvance: advance height for vertical layout

**max_advance_height**

**max_advance_width**

**num_charmaps**

**num_faces**

**num_fixed_sizes**

**num_glyphs**

**postscript_name**

**scalable**

**select_charmap**(*self*, *i*)

Select a charmap by its FT_Encoding number.

**set_charmap**(*self*, *i*)

Make the i-th charmap current.

**set_size**(*self*, *ptsize*, *dpi*)

Set the point size and dpi of the text.

**set_text**(*self*, *string*, *angle=0.0*, *flags=32*)

Set the text *string* and *angle*. *flags* can be a bitwise-or of the LOAD_XXX constants; the default value is LOAD_FORCE_AUTOHINT. You must call this before *draw_glyphs_to_bitmap*. A sequence of x,y positions is returned.

**style_flags**

**style_name**

**underline_position**

**underline_thickness**

**units_per_EM**

**class** matplotlib.ft2font.**FT2Image**

Bases: object

**draw_rect**(*self*, *x0*, *y0*, *x1*, *y1*)

Draw an empty rectangle to the image.

Deprecated since version 3.8.

**draw_rect_filled**(*self*, *x0*, *y0*, *x1*, *y1*)

Draw a filled rectangle to the image.

### 7.2.26 matplotlib.gridspec

*gridspec* contains classes that help to layout multiple *Axes* in a grid-like pattern within a figure.

The *GridSpec* specifies the overall grid structure. Individual cells within the grid are referenced by *SubplotSpec*s.

Often, users need not access this module directly, and can use higher-level methods like *subplots*, *subplot_mosaic* and *subfigures*. See the tutorial *Arranging multiple Axes in a Figure* for a guide.

## Classes

| | |
|---|---|
| GridSpec(nrows, ncols[, figure, left, ...]) | A grid layout to place subplots within a figure. |
| SubplotSpec(gridspec, num1[, num2]) | The location of a subplot in a GridSpec. |
| GridSpecBase(nrows, ncols[, height_ratios, ...]) | A base class of GridSpec that specifies the geometry of the grid that a subplot will be placed. |
| GridSpecFromSubplotSpec(nrows, ncols, ...[, ...]) | GridSpec whose subplot layout parameters are inherited from the location specified by a given SubplotSpec. |
| SubplotParams([left, bottom, right, top, ...]) | Parameters defining the positioning of a subplots grid in a figure. |

## matplotlib.gridspec.GridSpec

**class** matplotlib.gridspec.**GridSpec**(*nrows*, *ncols*, *figure=None*, *left=None*, *bottom=None*, *right=None*, *top=None*, *wspace=None*, *hspace=None*, *width_ratios=None*, *height_ratios=None*)

Bases: GridSpecBase

A grid layout to place subplots within a figure.

The location of the grid cells is determined in a similar way to SubplotParams using *left*, *right*, *top*, *bottom*, *wspace* and *hspace*.

Indexing a GridSpec instance returns a SubplotSpec.

**Parameters**

**nrows, ncols**

[int] The number of rows and columns of the grid.

**figure**

[Figure, optional] Only used for constrained layout to create a proper layoutgrid.

**left, right, top, bottom**

[float, optional] Extent of the subplots as a fraction of figure width or height. Left cannot be larger than right, and bottom cannot be larger than top. If not given, the values will be inferred from a figure or rcParams at draw time. See also GridSpec.get_subplot_params.

**wspace**

[float, optional] The amount of width reserved for space between subplots, expressed as a fraction of the average axis width. If not given, the values will be inferred from a figure or rcParams when necessary. See also GridSpec.get_subplot_params.

**hspace**

[float, optional] The amount of height reserved for space between subplots, expressed as a fraction of the average axis height. If not given, the values will be inferred from a figure or rcParams when necessary. See also *GridSpec.get_subplot_params*.

**width_ratios**

[array-like of length *ncols*, optional] Defines the relative widths of the columns. Each column gets a relative width of `width_ratios[i] / sum(width_ratios)`. If not given, all columns will have the same width.

**height_ratios**

[array-like of length *nrows*, optional] Defines the relative heights of the rows. Each row gets a relative height of `height_ratios[i] / sum(height_ratios)`. If not given, all rows will have the same height.

**get_subplot_params**(*figure=None*)

Return the *SubplotParams* for the GridSpec.

In order of precedence the values are taken from

- non-*None* attributes of the GridSpec
- the provided *figure*
- *rcParams["figure.subplot.*"]*

Note that the `figure` attribute of the GridSpec is always ignored.

**locally_modified_subplot_params**()

Return a list of the names of the subplot parameters explicitly set in the GridSpec.

This is a subset of the attributes of *SubplotParams*.

**tight_layout**(*figure*, *renderer=None*, *pad=1.08*, *h_pad=None*, *w_pad=None*, *rect=None*)

Adjust subplot parameters to give specified padding.

**Parameters**

**figure**

[*Figure*] The figure.

**renderer**

[*RendererBase* subclass, optional] The renderer to be used.

**pad**

[float] Padding between the figure edge and the edges of subplots, as a fraction of the font-size.

**h_pad, w_pad**

[float, optional] Padding (height/width) between edges of adjacent subplots. Defaults to *pad*.

**rect**

[tuple (left, bottom, right, top), default: None] (left, bottom, right, top) rectangle in normalized figure coordinates that the whole subplots area (including labels) will fit into. Default (None) is the whole figure.

**update**(***kwargs*)

Update the subplot parameters of the grid.

Parameters that are not explicitly given are not changed. Setting a parameter to *None* resets it to `rcParams["figure.subplot.*"]`.

**Parameters**

**left, right, top, bottom**

[float or None, optional] Extent of the subplots as a fraction of figure width or height.

**wspace, hspace**

[float, optional] Spacing between the subplots as a fraction of the average subplot width / height.

### Examples using `matplotlib.gridspec.GridSpec`

- *Scatter plot with histograms*
- *Streamplot*
- *Resizing Axes with constrained layout*
- *Resizing Axes with tight layout*
- *Combining two subplots using subplots and GridSpec*
- *Using Gridspec to make multi-column/row subplot layouts*
- *Nested Gridspecs*
- *Figure subfigures*
- *Creating multiple subplots using plt.subplots*
- *Custom spines with axisartist*
- *GridSpec demo*
- *Nested GridSpecs*
- *origin and extent in imshow*

- *Constrained layout guide*
- *Tight layout guide*
- *Arranging multiple Axes in a Figure*

### matplotlib.gridspec.SubplotSpec

**class** matplotlib.gridspec.**SubplotSpec**(*gridspec*, *num1*, *num2=None*)

Bases: object

The location of a subplot in a *GridSpec*.

---

**Note:** Likely, you will never instantiate a *SubplotSpec* yourself. Instead, you will typically obtain one from a *GridSpec* using item-access.

---

**Parameters**

**gridspec**

[*GridSpec*] The GridSpec, which the subplot is referencing.

**num1, num2**

[int] The subplot will occupy the *num1*-th cell of the given *gridspec*. If *num2* is provided, the subplot will span between *num1*-th cell and *num2*-th cell **inclusive**.

The index starts from 0.

**property colspan**

The columns spanned by this subplot, as a range object.

**get_geometry**()

Return the subplot geometry as tuple `(n_rows, n_cols, start, stop)`.

The indices *start* and *stop* define the range of the subplot within the *GridSpec*. *stop* is inclusive (i.e. for a single cell `start == stop`).

**get_gridspec**()

**get_position**(*figure*)

Update the subplot position from `figure.subplotpars`.

**get_topmost_subplotspec**()

Return the topmost *SubplotSpec* instance associated with the subplot.

**is_first_col**()

**is_first_row**()

**is_last_col**()

**is_last_row**()

**property num2**

**property rowspan**

The rows spanned by this subplot, as a `range` object.

**subgridspec**(*nrows*, *ncols*, ***kwargs*)

Create a GridSpec within this subplot.

The created *GridSpecFromSubplotSpec* will have this *SubplotSpec* as a parent.

**Parameters**

**nrows**

[int] Number of rows in grid.

**ncols**

[int] Number of columns in grid.

**Returns**

***GridSpecFromSubplotSpec***

**Other Parameters**

****kwargs**

All other parameters are passed to *GridSpecFromSubplotSpec*.

**See also:**

***matplotlib.pyplot.subplots***

### Examples

Adding three subplots in the space occupied by a single subplot:

```
fig = plt.figure()
gs0 = fig.add_gridspec(3, 1)
ax1 = fig.add_subplot(gs0[0])
ax2 = fig.add_subplot(gs0[1])
gssub = gs0[2].subgridspec(1, 3)
for i in range(3):
    fig.add_subplot(gssub[0, i])
```

### Examples using `matplotlib.gridspec.SubplotSpec`

- *Nested GridSpecs*
- *Constrained layout guide*
- *Arranging multiple Axes in a Figure*

## matplotlib.gridspec.GridSpecBase

**class** matplotlib.gridspec.**GridSpecBase**(*nrows*, *ncols*, *height_ratios=None*, *width_ratios=None*)

Bases: `object`

A base class of GridSpec that specifies the geometry of the grid that a subplot will be placed.

**Parameters**

**nrows, ncols**

[int] The number of rows and columns of the grid.

**width_ratios**

[array-like of length *ncols*, optional] Defines the relative widths of the columns. Each column gets a relative width of `width_ratios[i] / sum(width_ratios)`. If not given, all columns will have the same width.

**height_ratios**

[array-like of length *nrows*, optional] Defines the relative heights of the rows. Each row gets a relative height of `height_ratios[i] / sum(height_ratios)`. If not given, all rows will have the same height.

**get_geometry**()

Return a tuple containing the number of rows and columns in the grid.

**get_grid_positions**(*fig*)

Return the positions of the grid cells in figure coordinates.

**Parameters**

**fig**

[*Figure*] The figure the grid should be applied to. The subplot parameters (margins and spacing between subplots) are taken from *fig*.

**Returns**

**bottoms, tops, lefts, rights**

[array] The bottom, top, left, right positions of the grid cells in figure coordinates.

**get_height_ratios**()

Return the height ratios.

This is *None* if no height ratios have been set explicitly.

**get_subplot_params**(*figure=None*)

**get_width_ratios**()

Return the width ratios.

This is *None* if no width ratios have been set explicitly.

**property ncols**

The number of columns in the grid.

**new_subplotspec**(*loc*, *rowspan=1*, *colspan=1*)

Create and return a *SubplotSpec* instance.

**Parameters**

**loc**

[(int, int)] The position of the subplot in the grid as `(row_index, column_index)`.

**rowspan, colspan**

[int, default: 1] The number of rows and columns the subplot should span in the grid.

**property nrows**

The number of rows in the grid.

**set_height_ratios**(*height_ratios*)

Set the relative heights of the rows.

*height_ratios* must be of length *nrows*. Each row gets a relative height of `height_ratios[i] / sum(height_ratios)`.

**set_width_ratios**(*width_ratios*)

Set the relative widths of the columns.

*width_ratios* must be of length *ncols*. Each column gets a relative width of `width_ratios[i] / sum(width_ratios)`.

**subplots**(*, *sharex=False*, *sharey=False*, *squeeze=True*, *subplot_kw=None*)

Add all subplots specified by this *GridSpec* to its parent figure.

See *Figure.subplots* for detailed documentation.

### Examples using `matplotlib.gridspec.GridSpecBase`

- *Resizing Axes with constrained layout*
- *Using Gridspec to make multi-column/row subplot layouts*
- *Nested Gridspecs*
- *GridSpec demo*
- *origin and extent in imshow*
- *Constrained layout guide*
- *Tight layout guide*

## matplotlib.gridspec.GridSpecFromSubplotSpec

**class** matplotlib.gridspec.**GridSpecFromSubplotSpec**(*nrows*, *ncols*, *subplot_spec*, *wspace=None*, *hspace=None*, *height_ratios=None*, *width_ratios=None*)

Bases: *GridSpecBase*

GridSpec whose subplot layout parameters are inherited from the location specified by a given SubplotSpec.

**Parameters**

**nrows, ncols**

[int] Number of rows and number of columns of the grid.

**subplot_spec**

[SubplotSpec] Spec from which the layout parameters are inherited.

**wspace, hspace**

[float, optional] See *GridSpec* for more details. If not specified default values (from the figure or rcParams) are used.

**height_ratios**

[array-like of length *nrows*, optional] See *GridSpecBase* for details.

**width_ratios**

[array-like of length *ncols*, optional] See *GridSpecBase* for details.

**get_subplot_params**(*figure=None*)

Return a dictionary of subplot layout parameters.

**get_topmost_subplotspec**()

Return the topmost *SubplotSpec* instance associated with the subplot.

### Examples using `matplotlib.gridspec.GridSpecFromSubplotSpec`

- *Resizing Axes with constrained layout*
- *Nested Gridspecs*
- *Nested GridSpecs*
- *Constrained layout guide*
- *Arranging multiple Axes in a Figure*

## matplotlib.gridspec.SubplotParams

**class** matplotlib.gridspec.**SubplotParams**(*left=None*, *bottom=None*, *right=None*, *top=None*, *wspace=None*, *hspace=None*)

Bases: `object`

Parameters defining the positioning of a subplots grid in a figure.

Defaults are given by *`rcParams["figure.subplot.[name]"]`*.

**Parameters**

**left**

[float] The position of the left edge of the subplots, as a fraction of the figure width.

**right**

[float] The position of the right edge of the subplots, as a fraction of the figure width.

**bottom**

[float] The position of the bottom edge of the subplots, as a fraction of the figure height.

**top**

[float] The position of the top edge of the subplots, as a fraction of the figure height.

**wspace**

[float] The width of the padding between subplots, as a fraction of the average Axes width.

**hspace**

[float] The height of the padding between subplots, as a fraction of the average Axes height.

**update**(*left=None*, *bottom=None*, *right=None*, *top=None*, *wspace=None*, *hspace=None*)

Update the dimensions of the passed parameters. *None* means unchanged.

### Examples using `matplotlib.gridspec.SubplotParams`

- *Programmatically controlling subplot adjustment*

### 7.2.27 `matplotlib.hatch`

Contains classes for generating hatch patterns.

**class** matplotlib.hatch.**Circles**(*hatch*, *density*)

Bases: *Shapes*

**class** matplotlib.hatch.**HatchPatternBase**

Bases: object

The base class for a hatch pattern.

**class** matplotlib.hatch.**HorizontalHatch**(*hatch*, *density*)

Bases: *HatchPatternBase*

**set_vertices_and_codes**(*vertices*, *codes*)

**class** matplotlib.hatch.**LargeCircles**(*hatch*, *density*)

Bases: *Circles*

**size = 0.35**

**class** matplotlib.hatch.**NorthEastHatch**(*hatch*, *density*)

Bases: *HatchPatternBase*

**set_vertices_and_codes**(*vertices*, *codes*)

**class** matplotlib.hatch.**Shapes**(*hatch*, *density*)

Bases: *HatchPatternBase*

**filled = False**

**set_vertices_and_codes**(*vertices*, *codes*)

**class** matplotlib.hatch.**SmallCircles**(*hatch*, *density*)

Bases: *Circles*

**size = 0.2**

**class** matplotlib.hatch.**SmallFilledCircles**(*hatch*, *density*)

Bases: *Circles*

**filled = True**

**size = 0.1**

**class** matplotlib.hatch.**SouthEastHatch**(*hatch*, *density*)

Bases: *HatchPatternBase*

**set_vertices_and_codes**(*vertices*, *codes*)

**class** matplotlib.hatch.**Stars**(*hatch*, *density*)

Bases: *Shapes*

**filled = True**

**size = 0.3333333333333333**

**class** matplotlib.hatch.**VerticalHatch**(*hatch*, *density*)

Bases: *HatchPatternBase*

**set_vertices_and_codes**(*vertices*, *codes*)

matplotlib.hatch.**get_path**(*hatchpattern*, *density=6*)

Given a hatch specifier, *hatchpattern*, generates Path to render the hatch in a unit square. *density* is the number of lines per unit square.

### 7.2.28 matplotlib.image

The image module supports basic image loading, rescaling and display operations.

**class** matplotlib.image.**AxesImage**(*ax*, *, *cmap=None*, *norm=None*, *interpolation=None*, *origin=None*, *extent=None*, *filternorm=True*, *filterrad=4.0*, *resample=False*, *interpolation_stage=None*, ***kwargs*)

Bases: _ImageBase

An image attached to an Axes.

**Parameters**

**ax**

[*Axes*] The Axes the image will belong to.

**cmap**

[str or *Colormap*, default: *rcParams["image.cmap"]* (default: 'viridis')] The Colormap instance or registered colormap name used to map scalar data to colors.

**norm**

[str or *Normalize*] Maps luminance to 0-1.

**interpolation**

[str, default: *rcParams["image.interpolation"]* (default: 'antialiased')] Supported values are 'none', 'antialiased', 'nearest', 'bilinear', 'bicubic', 'spline16', 'spline36', 'hanning', 'hamming', 'hermite', 'kaiser', 'quadric', 'catrom', 'gaussian', 'bessel', 'mitchell', 'sinc', 'lanczos', 'blackman'.

**interpolation_stage**

[{'data', 'rgba'}, default: 'data'] If 'data', interpolation is carried out on the data provided by the user. If 'rgba', the interpolation is carried out after the colormapping has been applied (visual interpolation).

**origin**

[{'upper', 'lower'}, default: *rcParams["image.origin"]* (default: `'upper'`)] Place the [0, 0] index of the array in the upper left or lower left corner of the Axes. The convention 'upper' is typically used for matrices and images.

**extent**

[tuple, optional] The data axes (left, right, bottom, top) for making image plots registered with data plots. Default is to label the pixel centers with the zero-based row and column indices.

**filternorm**

[bool, default: True] A parameter for the antigrain image resize filter (see the antigrain documentation). If filternorm is set, the filter normalizes integer values and corrects the rounding errors. It doesn't do anything with the source floating point values, it corrects only integers according to the rule of 1.0 which means that any sum of pixel weights must be equal to 1.0. So, the filter function must produce a graph of the proper shape.

**filterrad**

[float > 0, default: 4] The filter radius for filters that have a radius parameter, i.e. when interpolation is one of: 'sinc', 'lanczos' or 'blackman'.

**resample**

[bool, default: False] When True, use a full resampling method. When False, only resample when the output image is larger than the input image.

****kwargs**

[*Artist* properties]

**Parameters**

**norm**

[*Normalize* (or subclass thereof) or str or None] The normalizing object which scales data, typically into the interval `[0, 1]`. If a `str`, a *Normalize* subclass is dynamically generated based on the scale with the corresponding name. If *None*, *norm* defaults to a *colors.Normalize* object which initializes its scaling based on the first data processed.

**cmap**

[str or *Colormap*] The colormap used to map normalized data values to RGBA colors.

**get_cursor_data**(*event*)

Return the image value at the event position or *None* if the event is outside the image.

**See also:**

*matplotlib.artist.Artist.get_cursor_data*

**get_extent**()

Return the image extent as tuple (left, right, bottom, top).

**get_window_extent**(*renderer=None*)

Get the artist's bounding box in display space.

The bounding box' width and height are nonnegative.

Subclasses should override for inclusion in the bounding box "tight" calculation. Default is to return an empty bounding box at 0, 0.

Be careful when using this function, the results will not update if the artist window extent of the artist changes. The extent can change due to any changes in the transform stack, such as changing the Axes limits, the figure size, or the canvas used (as is done when saving a figure). This can lead to unexpected behavior where interactive figures will look fine on the screen, but will save incorrectly.

**make_image**(*renderer*, *magnification=1.0*, *unsampled=False*)

Normalize, rescale, and colormap this image's data for rendering using *renderer*, with the given *magnification*.

If *unsampled* is True, the image will not be scaled, but an appropriate affine transformation will be returned instead.

**Returns**

**image**

[(M, N, 4) numpy.uint8 array] The RGBA image, resampled unless *unsampled* is True.

**x, y**

[float] The upper left corner where the image should be drawn, in pixel space.

**trans**

[*Affine2D*] The affine transformation from image to pixel space.

**set**(**, agg_filter=<UNSET>, alpha=<UNSET>, animated=<UNSET>, array=<UNSET>, clim=<UNSET>, clip_box=<UNSET>, clip_on=<UNSET>, clip_path=<UNSET>, cmap=<UNSET>, data=<UNSET>, extent=<UNSET>, filternorm=<UNSET>, filterrad=<UNSET>, gid=<UNSET>, in_layout=<UNSET>, interpolation=<UNSET>, interpolation_stage=<UNSET>, label=<UNSET>, mouseover=<UNSET>, norm=<UNSET>, path_effects=<UNSET>, picker=<UNSET>, rasterized=<UNSET>, resample=<UNSET>, sketch_params=<UNSET>, snap=<UNSET>, transform=<UNSET>, url=<UNSET>, visible=<UNSET>, zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and retu |
| alpha | float or 2D array-like or None |
| *animated* | bool |
| array | array-like |
| *clim* | (vmin: float, vmax: float) |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *cmap* | *Colormap* or str or None |
| data | array-like or PIL.Image.Image |
| *extent* | 4-tuple of float |
| *figure* | *Figure* |
| filternorm | bool |
| filterrad | positive float |
| *gid* | str |
| *in_layout* | bool |
| interpolation | {'antialiased', 'nearest', 'bilinear', 'bicubic', 'spline16', 'spline36', 'hanning', ' |
| interpolation_stage | {'data', 'rgba'} or None |
| *label* | object |
| *mouseover* | bool |
| *norm* | *Normalize* or str or None |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *rasterized* | bool |
| resample | bool or None |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *zorder* | float |

**set_extent**(*extent*, ***kwargs*)

Set the image extent.

**Parameters**

**extent**

[4-tuple of float] The position and size of the image as tuple `(left, right, bottom, top)` in data coordinates.

****kwargs**

Other parameters from which unit info (i.e., the *xunits*, *yunits*, *zunits* (for 3D Axes), *runits* and *thetaunits* (for polar Axes) entries are applied, if present.

### Notes

This updates `ax.dataLim`, and, if autoscaling, sets `ax.viewLim` to tightly fit the image, regardless of `dataLim`. Autoscaling state is not changed, so following this with `ax.autoscale_view()` will redo the autoscaling in accord with `dataLim`.

**class** matplotlib.image.**BboxImage**(*bbox*, ***, *cmap=None*, *norm=None*, *interpolation=None*, *origin=None*, *filternorm=True*, *filterrad=4.0*, *resample=False*, ***kwargs*)

Bases: `_ImageBase`

The Image class whose size is determined by the given bbox.

cmap is a colors.Colormap instance norm is a colors.Normalize instance to map luminance to 0-1

kwargs are an optional list of Artist keyword args

**contains**(*mouseevent*)

Test whether the mouse event occurred within the image.

**get_window_extent**(*renderer=None*)

Get the artist's bounding box in display space.

The bounding box' width and height are nonnegative.

Subclasses should override for inclusion in the bounding box "tight" calculation. Default is to return an empty bounding box at 0, 0.

Be careful when using this function, the results will not update if the artist window extent of the artist changes. The extent can change due to any changes in the transform stack, such as changing the Axes limits, the figure size, or the canvas used (as is done when saving a figure). This can lead to unexpected behavior where interactive figures will look fine on the screen, but will save incorrectly.

**make_image**(*renderer*, *magnification=1.0*, *unsampled=False*)

Normalize, rescale, and colormap this image's data for rendering using *renderer*, with the given *magnification*.

If *unsampled* is True, the image will not be scaled, but an appropriate affine transformation will be returned instead.

**Returns**

**image**

[(M, N, 4) numpy.uint8 array] The RGBA image, resampled unless *unsampled* is True.

**x, y**

[float] The upper left corner where the image should be drawn, in pixel space.

**trans**

[*Affine2D*] The affine transformation from image to pixel space.

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *array=<UNSET>*, *clim=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *cmap=<UNSET>*, *data=<UNSET>*, *filternorm=<UNSET>*, *filterrad=<UNSET>*, *gid=<UNSET>*, *in_layout=<UNSET>*, *interpolation=<UNSET>*, *interpolation_stage=<UNSET>*, *label=<UNSET>*, *mouseover=<UNSET>*, *norm=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *rasterized=<UNSET>*, *resample=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and retu |
| alpha | float or 2D array-like or None |
| *animated* | bool |
| array | array-like |
| *clim* | (vmin: float, vmax: float) |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *cmap* | *Colormap* or str or None |
| data | array-like or PIL.Image.Image |
| *figure* | *Figure* |
| filternorm | bool |
| filterrad | positive float |
| *gid* | str |
| *in_layout* | bool |
| interpolation | {'antialiased', 'nearest', 'bilinear', 'bicubic', 'spline16', 'spline36', 'hanning', ' |
| interpolation_stage | {'data', 'rgba'} or None |
| *label* | object |
| *mouseover* | bool |
| *norm* | *Normalize* or str or None |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *rasterized* | bool |
| resample | bool or None |

Table 60 – continue

| Property | Description |
|---|---|
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *zorder* | float |

**class** matplotlib.image.**FigureImage**(*fig*, *, *cmap=None*, *norm=None*, *offsetx=0*, *offsety=0*, *origin=None*, ***kwargs*)

Bases: _ImageBase

An image attached to a figure.

cmap is a colors.Colormap instance norm is a colors.Normalize instance to map luminance to 0-1

kwargs are an optional list of Artist keyword args

**get_extent**()

Return the image extent as tuple (left, right, bottom, top).

**make_image**(*renderer*, *magnification=1.0*, *unsampled=False*)

Normalize, rescale, and colormap this image's data for rendering using *renderer*, with the given *magnification*.

If *unsampled* is True, the image will not be scaled, but an appropriate affine transformation will be returned instead.

**Returns**

**image**

[(M, N, 4) numpy.uint8 array] The RGBA image, resampled unless *unsampled* is True.

**x, y**

[float] The upper left corner where the image should be drawn, in pixel space.

**trans**

[*Affine2D*] The affine transformation from image to pixel space.

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *array=<UNSET>*, *clim=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *cmap=<UNSET>*, *data=<UNSET>*, *filternorm=<UNSET>*, *filterrad=<UNSET>*, *gid=<UNSET>*, *in_layout=<UNSET>*, *interpolation=<UNSET>*, *interpolation_stage=<UNSET>*, *label=<UNSET>*, *mouseover=<UNSET>*, *norm=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *rasterized=<UNSET>*, *resample=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and retu |
| alpha | float or 2D array-like or None |
| *animated* | bool |
| array | array-like |
| *clim* | (vmin: float, vmax: float) |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *cmap* | *Colormap* or str or None |
| *data* | unknown |
| *figure* | *Figure* |
| filternorm | bool |
| filterrad | positive float |
| *gid* | str |
| *in_layout* | bool |
| interpolation | {'antialiased', 'nearest', 'bilinear', 'bicubic', 'spline16', 'spline36', 'hanning', ' |
| interpolation_stage | {'data', 'rgba'} or None |
| *label* | object |
| *mouseover* | bool |
| *norm* | *Normalize* or str or None |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *rasterized* | bool |
| resample | bool or None |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *zorder* | float |

**set_data**(*A*)

Set the image array.

**zorder = 0**

**class** matplotlib.image.**NonUniformImage**(*ax*, *, *interpolation='nearest'*, ***kwargs*)

Bases: *AxesImage*

**Parameters**

**ax**

[*Axes*] The Axes the image will belong to.

**interpolation**

[{'nearest', 'bilinear'}, default: 'nearest'] The interpolation scheme used in the resampling.

****kwargs**

All other keyword arguments are identical to those of *AxesImage*.

**get_cursor_data**(*event*)

Return the image value at the event position or *None* if the event is outside the image.

**See also:**

***matplotlib.artist.Artist.get_cursor_data***

**get_extent**()

Return the image extent as tuple (left, right, bottom, top).

**make_image**(*renderer*, *magnification=1.0*, *unsampled=False*)

Normalize, rescale, and colormap this image's data for rendering using *renderer*, with the given *magnification*.

If *unsampled* is True, the image will not be scaled, but an appropriate affine transformation will be returned instead.

**Returns**

**image**

[(M, N, 4) numpy.uint8 array] The RGBA image, resampled unless *unsampled* is True.

**x, y**

[float] The upper left corner where the image should be drawn, in pixel space.

**trans**

[*Affine2D*] The affine transformation from image to pixel space.

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *array=<UNSET>*, *clim=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *cmap=<UNSET>*, *data=<UNSET>*, *extent=<UNSET>*, *filternorm=<UNSET>*, *filterrad=<UNSET>*, *gid=<UNSET>*, *in_layout=<UNSET>*, *interpolation=<UNSET>*, *interpolation_stage=<UNSET>*, *label=<UNSET>*, *mouseover=<UNSET>*, *norm=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *rasterized=<UNSET>*, *resample=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and retu |
| alpha | float or 2D array-like or None |
| *animated* | bool |
| array | unknown |
| *clim* | (vmin: float, vmax: float) |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *cmap* | unknown |
| *data* | unknown |
| *extent* | 4-tuple of float |
| *figure* | *Figure* |
| filternorm | unknown |
| filterrad | unknown |
| *gid* | str |
| *in_layout* | bool |
| *interpolation* | {'nearest', 'bilinear'} or None |
| interpolation_stage | {'data', 'rgba'} or None |
| *label* | object |
| *mouseover* | bool |
| *norm* | unknown |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *rasterized* | bool |
| resample | bool or None |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *zorder* | float |

**set_array**(**args*)

Retained for backwards compatibility - use set_data instead.

**Parameters**

**A**

[array-like]

**set_cmap**(*cmap*)

Set the colormap for luminance data.

**Parameters**

**cmap**

[*Colormap* or str or None]

**set_data**(*x*, *y*, *A*)

Set the grid for the pixel centers, and the pixel values.

**Parameters**

**x, y**

[1D array-like] Monotonic arrays of shapes (N,) and (M,), respectively, specifying pixel centers.

**A**

[array-like] (M, N) `ndarray` or masked array of values to be colormapped, or (M, N, 3) RGB array, or (M, N, 4) RGBA array.

**set_filternorm**(*filternorm*)

Set whether the resize filter normalizes the weights.

See help for *imshow*.

**Parameters**

**filternorm**

[bool]

**set_filterrad**(*filterrad*)

Set the resize filter radius only applicable to some interpolation schemes -- see help for imshow

**Parameters**

**filterrad**

[positive float]

**set_interpolation**(*s*)

**Parameters**

**s**

[{'nearest', 'bilinear'} or None] If None, use *rcParams["image.interpolation"]* (default: 'antialiased').

**set_norm**(*norm*)

Set the normalization instance.

**Parameters**

**norm**

[*Normalize* or str or None]

### Notes

If there are any colorbars using the mappable for this norm, setting the norm of the mappable will reset the norm, locator, and formatters on the colorbar to default.

**class** matplotlib.image.**PcolorImage**(*ax*, *x=None*, *y=None*, *A=None*, ***, *cmap=None*, *norm=None*, ***kwargs*)

Bases: *AxesImage*

Make a pcolor-style plot with an irregular rectangular grid.

This uses a variation of the original irregular image code, and it is used by pcolorfast for the corresponding grid type.

**Parameters**

**ax**

[*Axes*] The Axes the image will belong to.

**x, y**

[1D array-like, optional] Monotonic arrays of length N+1 and M+1, respectively, specifying rectangle boundaries. If not given, will default to `range(N + 1)` and `range(M + 1)`, respectively.

**A**

[array-like] The data to be color-coded. The interpretation depends on the shape:

- (M, N) `ndarray` or masked array: values to be colormapped
- (M, N, 3): RGB array
- (M, N, 4): RGBA array

**cmap**

[str or *Colormap*, default: *rcParams["image.cmap"]* (default: `'viridis'`)] The Colormap instance or registered colormap name used to map scalar data to colors.

**norm**

[str or *Normalize*] Maps luminance to 0-1.

****kwargs**

[*Artist* properties]

**get_cursor_data**(*event*)

Return the image value at the event position or *None* if the event is outside the image.

**See also:**

***matplotlib.artist.Artist.get_cursor_data***

**make_image**(*renderer*, *magnification=1.0*, *unsampled=False*)

Normalize, rescale, and colormap this image's data for rendering using *renderer*, with the given *magnification*.

If *unsampled* is True, the image will not be scaled, but an appropriate affine transformation will be returned instead.

**Returns**

**image**

[(M, N, 4) `numpy.uint8` array] The RGBA image, resampled unless *unsampled* is True.

**x, y**

[float] The upper left corner where the image should be drawn, in pixel space.

**trans**

[*Affine2D*] The affine transformation from image to pixel space.

**set**(**, agg_filter=<UNSET>, alpha=<UNSET>, animated=<UNSET>, array=<UNSET>, clim=<UNSET>, clip_box=<UNSET>, clip_on=<UNSET>, clip_path=<UNSET>, cmap=<UNSET>, data=<UNSET>, extent=<UNSET>, filternorm=<UNSET>, filterrad=<UNSET>, gid=<UNSET>, in_layout=<UNSET>, interpolation=<UNSET>, interpolation_stage=<UNSET>, label=<UNSET>, mouseover=<UNSET>, norm=<UNSET>, path_effects=<UNSET>, picker=<UNSET>, rasterized=<UNSET>, resample=<UNSET>, sketch_params=<UNSET>, snap=<UNSET>, transform=<UNSET>, url=<UNSET>, visible=<UNSET>, zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| agg_filter | a filter function, which takes a (m, n, 3) float array and a dpi value, and retu |
| alpha | float or 2D array-like or None |
| animated | bool |
| array | unknown |
| clim | (vmin: float, vmax: float) |
| clip_box | BboxBase or None |
| clip_on | bool |
| clip_path | Patch or (Path, Transform) or None |
| cmap | Colormap or str or None |
| data | unknown |
| extent | 4-tuple of float |
| figure | Figure |
| filternorm | bool |
| filterrad | positive float |
| gid | str |
| in_layout | bool |
| interpolation | {'antialiased', 'nearest', 'bilinear', 'bicubic', 'spline16', 'spline36', 'hanning', ' |
| interpolation_stage | {'data', 'rgba'} or None |
| label | object |
| mouseover | bool |
| norm | Normalize or str or None |
| path_effects | list of AbstractPathEffect |
| picker | None or bool or float or callable |
| rasterized | bool |
| resample | bool or None |
| sketch_params | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| transform | Transform |
| url | str |
| visible | bool |
| zorder | float |

**set_array**(**args*)

Retained for backwards compatibility - use set_data instead.

**Parameters**

**A**

[array-like]

**set_data**(*x*, *y*, *A*)

Set the grid for the rectangle boundaries, and the data values.

**Parameters**

**x, y**

[1D array-like, optional] Monotonic arrays of length N+1 and M+1, respectively, specifying rectangle boundaries. If not given, will default to `range(N + 1)` and `range(M + 1)`, respectively.

**A**

[array-like] The data to be color-coded. The interpretation depends on the shape:

- (M, N) `ndarray` or masked array: values to be colormapped
- (M, N, 3): RGB array
- (M, N, 4): RGBA array

`matplotlib.image.`**`composite_images`**(*images*, *renderer*, *magnification=1.0*)

Composite a number of RGBA images into one. The images are composited in the order in which they appear in the *images* list.

**Parameters**

**images**

[list of Images] Each must have a `make_image` method. For each image, `can_composite` should return `True`, though this is not enforced by this function. Each image must have a purely affine transformation with no shear.

**renderer**

[*RendererBase*]

**magnification**

[float, default: 1] The additional magnification to apply for the renderer in use.

**Returns**

**image**

[(M, N, 4) `numpy.uint8` array] The composited RGBA image.

**offset_x, offset_y**

[float] The (left, bottom) offset where the composited image should be placed in the output figure.

`matplotlib.image.`**`imread`**(*fname*, *format=None*)

Read an image from a file into an array.

**Note:** This function exists for historical reasons. It is recommended to use `PIL.Image.open` instead for loading images.

**Parameters**

**fname**

[str or file-like] The image file to read: a filename, a URL or a file-like object opened in read-binary mode.

Passing a URL is deprecated. Please open the URL for reading and pass the result to Pillow, e.g. with `np.array(PIL.Image.open(urllib.request.urlopen(url)))`.

**format**

[str, optional] The image file format assumed for reading the data. The image is loaded as a PNG file if *format* is set to "png", if *fname* is a path or opened file with a ".png" extension, or if it is a URL. In all other cases, *format* is ignored and the format is auto-detected by `PIL.Image.open`.

**Returns**

**numpy.array**

The image data. The returned array has shape

- (M, N) for grayscale images.
- (M, N, 3) for RGB images.
- (M, N, 4) for RGBA images.

PNG images are returned as float arrays (0-1). All other formats are returned as int arrays, with a bit depth determined by the file's contents.

matplotlib.image.**imsave**(*fname*, *arr*, *vmin=None*, *vmax=None*, *cmap=None*, *format=None*, *origin=None*, *dpi=100*, *, *metadata=None*, *pil_kwargs=None*)

Colormap and save an array as an image file.

RGB(A) images are passed through. Single channel images will be colormapped according to *cmap* and *norm*.

**Note:** If you want to save a single channel image as gray scale please use an image I/O library (such as pillow, tifffile, or imageio) directly.

**Parameters**

**fname**

[str or path-like or file-like] A path or a file-like object to store the image in. If *format* is not set, then the output format is inferred from the extension of *fname*, if any, and from *rcParams["savefig.format"]* (default: 'png') otherwise. If *format* is set, it determines the output format.

**arr**

[array-like] The image data. The shape can be one of MxN (luminance), MxNx3 (RGB) or MxNx4 (RGBA).

**vmin, vmax**

[float, optional] *vmin* and *vmax* set the color scaling for the image by fixing the values that map to the colormap color limits. If either *vmin* or *vmax* is None, that limit is determined from the *arr* min/max value.

**cmap**

[str or *Colormap*, default: *rcParams["image.cmap"]* (default: 'viridis')] A Colormap instance or registered colormap name. The colormap maps scalar data to colors. It is ignored for RGB(A) data.

**format**

[str, optional] The file format, e.g. 'png', 'pdf', 'svg', ... The behavior when this is unset is documented under *fname*.

**origin**

[{'upper', 'lower'}, default: *rcParams["image.origin"]* (default: 'upper')] Indicates whether the `(0, 0)` index of the array is in the upper left or lower left corner of the Axes.

**dpi**

[float] The DPI to store in the metadata of the file. This does not affect the resolution of the output image. Depending on file format, this may be rounded to the nearest integer.

**metadata**

[dict, optional] Metadata in the image file. The supported keys depend on the output format, see the documentation of the respective backends for more information. Currently only supported for "png", "pdf", "ps", "eps", and "svg".

**pil_kwargs**

[dict, optional] Keyword arguments passed to `PIL.Image.Image.save`. If the 'pnginfo' key is present, it completely overrides *metadata*, including the default 'Software' key.

`matplotlib.image.`**`pil_to_array`**(*pilImage*)

Load a PIL image and return it as a numpy int array.

**Returns**

**numpy.array**

The array shape depends on the image type:

- (M, N) for grayscale images.
- (M, N, 3) for RGB images.
- (M, N, 4) for RGBA images.

matplotlib.image.**thumbnail**(*infile*, *thumbfile*, *scale=0.1*, *interpolation='bilinear'*, *preview=False*)

Make a thumbnail of image in *infile* with output filename *thumbfile*.

See *Image Thumbnail*.

**Parameters**

**infile**

[str or file-like] The image file. Matplotlib relies on Pillow for image reading, and thus supports a wide range of file formats, including PNG, JPG, TIFF and others.

**thumbfile**

[str or file-like] The thumbnail filename.

**scale**

[float, default: 0.1] The scale factor for the thumbnail.

**interpolation**

[str, default: 'bilinear'] The interpolation scheme used in the resampling. See the *interpolation* parameter of `imshow` for possible values.

**preview**

[bool, default: False] If True, the default backend (presumably a user interface backend) will be used which will cause a figure to be raised if `show` is called. If it is False, the figure is created using `FigureCanvasBase` and the drawing backend is selected as `Figure.savefig` would normally do.

**Returns**

***Figure***

The figure instance containing the thumbnail.

### 7.2.29 matplotlib.layout_engine

Classes to layout elements in a *Figure*.

Figures have a `layout_engine` property that holds a subclass of *LayoutEngine* defined here (or *None* for no layout). At draw time `figure.get_layout_engine().execute()` is called, the goal of which is usually to rearrange Axes on the figure to produce a pleasing layout. This is like a `draw` callback but with two differences. First, when printing we disable the layout engine for the final draw. Second, it is useful to know the layout engine while the figure is being created. In particular, colorbars are made differently with different layout engines (for historical reasons).

Matplotlib supplies two layout engines, *TightLayoutEngine* and *ConstrainedLayoutEngine*. Third parties can create their own layout engine by subclassing *LayoutEngine*.

**class** matplotlib.layout_engine.**ConstrainedLayoutEngine**(*, *h_pad=None*, *w_pad=None*, *hspace=None*, *wspace=None*, *rect=(0, 0, 1, 1)*, *compress=False*, ***kwargs*)

Implements the `constrained_layout` geometry management. See *Constrained layout guide* for details.

Initialize `constrained_layout` settings.

**Parameters**

**h_pad, w_pad**

[float] Padding around the Axes elements in inches. Default to *rcParams["figure.constrained_layout.h_pad"]* (default: `0.04167`) and *rcParams["figure.constrained_layout.w_pad"]* (default: `0.04167`).

**hspace, wspace**

[float] Fraction of the figure to dedicate to space between the axes. These are evenly spread between the gaps between the Axes. A value of 0.2 for a three-column layout would have a space of 0.1 of the figure width between each column. If h/wspace < h/w_pad, then the pads are used instead. Default to *rcParams["figure.constrained_layout.hspace"]* (default: `0.02`) and *rcParams["figure.constrained_layout.wspace"]* (default: `0.02`).

**rect**

[tuple of 4 floats] Rectangle in figure coordinates to perform constrained layout in (left, bottom, width, height), each from 0-1.

**compress**

[bool] Whether to shift Axes so that white space in between them is removed. This is useful for simple grids of fixed-aspect Axes (e.g. a grid of images). See *Grids of fixed aspect-ratio Axes: "compressed" layout*.

**property adjust_compatible**

Return a boolean if the layout engine is compatible with `subplots_adjust`.

**property colorbar_gridspec**

Return a boolean if the layout engine creates colorbars using a gridspec.

**execute**(*fig*)

Perform constrained_layout and move and resize Axes accordingly.

**Parameters**

**fig**

[`Figure` to perform layout on.]

**get**()

Return copy of the parameters for the layout engine.

**set**(*, *h_pad=None*, *w_pad=None*, *hspace=None*, *wspace=None*, *rect=None*)

Set the pads for constrained_layout.

**Parameters**

**h_pad, w_pad**

[float] Padding around the Axes elements in inches. Default to `rcParams["figure.constrained_layout.h_pad"]` (default: `0.04167`) and `rcParams["figure.constrained_layout.w_pad"]` (default: `0.04167`).

**hspace, wspace**

[float] Fraction of the figure to dedicate to space between the axes. These are evenly spread between the gaps between the Axes. A value of 0.2 for a three-column layout would have a space of 0.1 of the figure width between each column. If h/wspace < h/w_pad, then the pads are used instead. Default to `rcParams["figure.constrained_layout.hspace"]` (default: `0.02`) and `rcParams["figure.constrained_layout.wspace"]` (default: `0.02`).

**rect**

[tuple of 4 floats] Rectangle in figure coordinates to perform constrained layout in (left, bottom, width, height), each from 0-1.

**class** matplotlib.layout_engine.**LayoutEngine**(***kwargs*)

Base class for Matplotlib layout engines.

A layout engine can be passed to a figure at instantiation or at any time with *set_layout_engine*. Once attached to a figure, the layout engine `execute` function is called at draw time by *draw*, providing a special draw-time hook.

---

**Note:** However, note that layout engines affect the creation of colorbars, so *set_layout_engine* should be called before any colorbars are created.

---

Currently, there are two properties of *LayoutEngine* classes that are consulted while manipulating the figure:

- **engine.colorbar_gridspec tells *Figure.colorbar* whether to make the**

  axes using the gridspec method (see *colorbar.make_axes_gridspec*) or not (see *colorbar.make_axes*);

- **engine.adjust_compatible stops *Figure.subplots_adjust* from being**

  run if it is not compatible with the layout engine.

To implement a custom *LayoutEngine*:

1. override `_adjust_compatible` and `_colorbar_gridspec`
2. override *LayoutEngine.set* to update *self._params*
3. override *LayoutEngine.execute* with your implementation

**property adjust_compatible**

Return a boolean if the layout engine is compatible with *subplots_adjust*.

**property colorbar_gridspec**

Return a boolean if the layout engine creates colorbars using a gridspec.

**execute**(*fig*)

Execute the layout on the figure given by *fig*.

**get**()

Return copy of the parameters for the layout engine.

**set**(***kwargs*)

Set the parameters for the layout engine.

**class** matplotlib.layout_engine.**PlaceHolderLayoutEngine**(*adjust_compatible*, *colorbar_gridspec*, ***kwargs*)

This layout engine does not adjust the figure layout at all.

The purpose of this *LayoutEngine* is to act as a placeholder when the user removes a layout engine to ensure an incompatible *LayoutEngine* cannot be set later.

**Parameters**

**adjust_compatible, colorbar_gridspec**

[bool] Allow the PlaceHolderLayoutEngine to mirror the behavior of whatever layout engine it is replacing.

**property adjust_compatible**

Return a boolean if the layout engine is compatible with *subplots_adjust*.

**property colorbar_gridspec**

Return a boolean if the layout engine creates colorbars using a gridspec.

**execute**(*fig*)

Do nothing.

**get**()

Return copy of the parameters for the layout engine.

**set**(***kwargs*)

Set the parameters for the layout engine.

**class** matplotlib.layout_engine.**TightLayoutEngine**(*, *pad=1.08*, *h_pad=None*, *w_pad=None*, *rect=(0, 0, 1, 1)*, ***kwargs*)

Implements the `tight_layout` geometry management. See *Tight layout guide* for details.

Initialize tight_layout engine.

**Parameters**

**pad**

[float, default: 1.08] Padding between the figure edge and the edges of subplots, as a fraction of the font size.

**h_pad, w_pad**

[float] Padding (height/width) between edges of adjacent subplots. Defaults to *pad*.

**rect**

[tuple (left, bottom, right, top), default: (0, 0, 1, 1).] rectangle in normalized figure coordinates that the subplots (including labels) will fit into.

**property adjust_compatible**

Return a boolean if the layout engine is compatible with *subplots_adjust*.

**property colorbar_gridspec**

Return a boolean if the layout engine creates colorbars using a gridspec.

**execute**(*fig*)

Execute tight_layout.

This decides the subplot parameters given the padding that will allow the Axes labels to not be covered by other labels and Axes.

**Parameters**

**fig**

[*Figure* to perform layout on.]

**See also:**

***figure.Figure.tight_layout***
***pyplot.tight_layout***

**get**()

Return copy of the parameters for the layout engine.

**set**(*, *pad=None*, *w_pad=None*, *h_pad=None*, *rect=None*)

Set the pads for tight_layout.

**Parameters**

**pad**

[float] Padding between the figure edge and the edges of subplots, as a fraction of the font size.

**w_pad, h_pad**

[float] Padding (width/height) between edges of adjacent subplots. Defaults to *pad*.

**rect**

[tuple (left, bottom, right, top)] rectangle in normalized figure coordinates that the subplots (including labels) will fit into.

### 7.2.30 matplotlib.legend

The legend module defines the Legend class, which is responsible for drawing legends associated with Axes and/or figures.

---

**Important:** It is unlikely that you would ever create a Legend instance manually. Most users would normally create a legend via the *legend* function. For more details on legends there is also a *legend guide*.

---

The *Legend* class is a container of legend handles and legend texts.

The legend handler map specifies how to create legend handles from artists (lines, patches, etc.) in the Axes or figures. Default legend handlers are defined in the *legend_handler* module. While not all artist types are covered by the default legend handlers, custom legend handlers can be defined to support arbitrary objects.

See the :ref`<legend_guide>` for more information.

**class** matplotlib.legend.**DraggableLegend**(*legend*, *use_blit=False*, *update='loc'*)

Bases: *DraggableOffsetBox*

Wrapper around a *Legend* to support mouse dragging.

**Parameters**

**legend**

[*Legend*] The *Legend* instance to wrap.

**use_blit**

[bool, optional] Use blitting for faster image composition. For details see *FuncAnimation*.

**update**

[{'loc', 'bbox'}, optional] If "loc", update the *loc* parameter of the legend upon finalizing. If "bbox", update the *bbox_to_anchor* parameter.

**finalize_offset**()

**class** matplotlib.legend.**Legend**(*parent*, *handles*, *labels*, ***, *loc=None*, *numpoints=None*, *markerscale=None*, *markerfirst=True*, *reverse=False*, *scatterpoints=None*, *scatteryoffsets=None*, *prop=None*, *fontsize=None*, *labelcolor=None*, *borderpad=None*, *labelspacing=None*, *handlelength=None*, *handleheight=None*, *handletextpad=None*, *borderaxespad=None*, *columnspacing=None*, *ncols=1*, *mode=None*, *fancybox=None*, *shadow=None*, *title=None*, *title_fontsize=None*, *framealpha=None*, *edgecolor=None*, *facecolor=None*, *bbox_to_anchor=None*, *bbox_transform=None*, *frameon=None*, *handler_map=None*, *title_fontproperties=None*, *alignment='center'*, *ncol=1*, *draggable=False*)

Bases: *Artist*

Place a legend on the figure/axes.

**Parameters**

**parent**

[*Axes* or *Figure*] The artist that contains the legend.

**handles**

[list of (*Artist* or tuple of *Artist*)] A list of Artists (lines, patches) to be added to the legend.

**labels**

[list of str] A list of labels to show next to the artists. The length of handles and labels should be the same. If they are not, they are truncated to the length of the shorter list.

### Other Parameters

**loc**

[str or pair of floats, default: *rcParams["legend.loc"]* (default: 'best') for Axes, 'upper right' for Figure] The location of the legend.

The strings 'upper left', 'upper right', 'lower left', 'lower right' place the legend at the corresponding corner of the axes/figure.

The strings 'upper center', 'lower center', 'center left', 'center right' place the legend at the center of the corresponding edge of the axes/figure.

The string 'center' places the legend at the center of the axes/figure.

The string 'best' places the legend at the location, among the nine locations defined so far, with the minimum overlap with other drawn artists. This option can be quite slow for plots with large amounts of data; your plotting speed may benefit from providing a specific location.

The location can also be a 2-tuple giving the coordinates of the lower-left corner of the legend in axes/figure coordinates (in which case *bbox_to_anchor* will be ignored).

For back-compatibility, 'center right' (but no other location) can also be spelled 'right', and each "string" location can also be given as a numeric value:

| Location String | Location Code |
|---|---|
| 'best' (Axes only) | 0 |
| 'upper right' | 1 |
| 'upper left' | 2 |
| 'lower left' | 3 |
| 'lower right' | 4 |
| 'right' | 5 |
| 'center left' | 6 |
| 'center right' | 7 |
| 'lower center' | 8 |
| 'upper center' | 9 |
| 'center' | 10 |

If a figure is using the constrained layout manager, the string codes of the *loc* keyword argument can get better layout behaviour using the prefix 'outside'. There is ambiguity at the corners, so 'outside upper right' will make space for the legend above the rest of the axes in the layout, and 'outside right upper' will make space on the right side of the layout. In addition to the values of *loc* listed above, we have 'outside right upper', 'outside right lower', 'outside left upper', and 'outside left lower'. See *Legend guide* for more details.

**bbox_to_anchor**

[*BboxBase*, 2-tuple, or 4-tuple of floats] Box that is used to position the legend in conjunction with *loc*. Defaults to `axes.bbox` (if called as a method to *Axes.legend*) or `figure.bbox` (if *Figure.legend*). This argument allows arbitrary placement of the legend.

Bbox coordinates are interpreted in the coordinate system given by *bbox_transform*, with the default transform Axes or Figure coordinates, depending on which `legend` is called.

If a 4-tuple or *BboxBase* is given, then it specifies the bbox `(x, y, width, height)` that the legend is placed in. To put the legend in the best location in the bottom right quadrant of the Axes (or figure):

```
loc='best', bbox_to_anchor=(0.5, 0., 0.5, 0.5)
```

A 2-tuple `(x, y)` places the corner of the legend specified by *loc* at x, y. For example, to put the legend's upper right-hand corner in the center of the Axes (or figure) the following keywords can be used:

```
loc='upper right', bbox_to_anchor=(0.5, 0.5)
```

**ncols**

[int, default: 1] The number of columns that the legend has.

For backward compatibility, the spelling *ncol* is also supported but it is discouraged. If both are given, *ncols* takes precedence.

**prop**

[None or *FontProperties* or dict] The font properties of the legend. If None (default), the current *matplotlib.rcParams* will be used.

**fontsize**

[int or {'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large'}] The font size of the legend. If the value is numeric the size will be the absolute font size in points. String values are relative to the current default font size. This argument is only used if *prop* is not specified.

**labelcolor**

[str or list, default: *rcParams["legend.labelcolor"]* (default: `'None'`)] The color of the text in the legend. Either a valid color string (for

example, 'red'), or a list of color strings. The labelcolor can also be made to match the color of the line or marker using 'linecolor', 'markerfacecolor' (or 'mfc'), or 'markeredgecolor' (or 'mec').

Labelcolor can be set globally using *rcParams["legend.labelcolor"]* (default: 'None'). If None, use *rcParams["text.color"]* (default: 'black').

**numpoints**

[int, default: *rcParams["legend.numpoints"]* (default: 1)] The number of marker points in the legend when creating a legend entry for a *Line2D* (line).

**scatterpoints**

[int, default: *rcParams["legend.scatterpoints"]* (default: 1)] The number of marker points in the legend when creating a legend entry for a *PathCollection* (scatter plot).

**scatteryoffsets**

[iterable of floats, default: [0.375, 0.5, 0.3125]] The vertical offset (relative to the font size) for the markers created for a scatter plot legend entry. 0.0 is at the base the legend text, and 1.0 is at the top. To draw all markers at the same height, set to [0.5].

**markerscale**

[float, default: *rcParams["legend.markerscale"]* (default: 1.0)] The relative size of legend markers compared to the originally drawn ones.

**markerfirst**

[bool, default: True] If *True*, legend marker is placed to the left of the legend label. If *False*, legend marker is placed to the right of the legend label.

**reverse**

[bool, default: False] If *True*, the legend labels are displayed in reverse order from the input. If *False*, the legend labels are displayed in the same order as the input.

New in version 3.7.

**frameon**

[bool, default: *rcParams["legend.frameon"]* (default: True)] Whether the legend should be drawn on a patch (frame).

**fancybox**

[bool, default: *rcParams["legend.fancybox"]* (default: True)] Whether round edges should be enabled around the *FancyBboxPatch* which makes up the legend's background.

**shadow**

[None, bool or dict, default: *rcParams["legend.shadow"]* (default: False)] Whether to draw a shadow behind the legend. The shadow can be

configured using *Patch* keywords. Customization via *rcParams["legend.shadow"]* (default: False) is currently not supported.

**framealpha**

[float, default: *rcParams["legend.framealpha"]* (default: 0.8)] The alpha transparency of the legend's background. If *shadow* is activated and *framealpha* is None, the default value is ignored.

**facecolor**

["inherit" or color, default: *rcParams["legend.facecolor"]* (default: 'inherit')] The legend's background color. If "inherit", use *rcParams["axes.facecolor"]* (default: 'white').

**edgecolor**

["inherit" or color, default: *rcParams["legend.edgecolor"]* (default: '0.8')] The legend's background patch edge color. If "inherit", use *rcParams["axes.edgecolor"]* (default: 'black').

**mode**

[{"expand", None}] If *mode* is set to "expand" the legend will be horizontally expanded to fill the Axes area (or *bbox_to_anchor* if defines the legend's size).

**bbox_transform**

[None or *Transform*] The transform for the bounding box (*bbox_to_anchor*). For a value of None (default) the Axes' transAxes transform will be used.

**title**

[str or None] The legend's title. Default is no title (None).

**title_fontproperties**

[None or *FontProperties* or dict] The font properties of the legend's title. If None (default), the *title_fontsize* argument will be used if present; if *title_fontsize* is also None, the current *rcParams["legend.title_fontsize"]* (default: None) will be used.

**title_fontsize**

[int or {'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large'}, default: *rcParams["legend.title_fontsize"]* (default: None)] The font size of the legend's title. Note: This cannot be combined with *title_fontproperties*. If you want to set the fontsize alongside other font properties, use the *size* parameter in *title_fontproperties*.

**alignment**

[{'center', 'left', 'right'}, default: 'center'] The alignment of the legend title and the box of entries. The entries are aligned as a single block, so that markers always lined up.

**borderpad**

[float, default: *rcParams["legend.borderpad"]* (default: 0.4)] The fractional whitespace inside the legend border, in font-size units.

**labelspacing**

[float, default: *rcParams["legend.labelspacing"]* (default: 0.5)] The vertical space between the legend entries, in font-size units.

**handlelength**

[float, default: *rcParams["legend.handlelength"]* (default: 2.0)] The length of the legend handles, in font-size units.

**handleheight**

[float, default: *rcParams["legend.handleheight"]* (default: 0.7)] The height of the legend handles, in font-size units.

**handletextpad**

[float, default: *rcParams["legend.handletextpad"]* (default: 0.8)] The pad between the legend handle and text, in font-size units.

**borderaxespad**

[float, default: *rcParams["legend.borderaxespad"]* (default: 0.5)] The pad between the Axes and legend border, in font-size units.

**columnspacing**

[float, default: *rcParams["legend.columnspacing"]* (default: 2.0)] The spacing between columns, in font-size units.

**handler_map**

[dict or None] The custom dictionary mapping instances or types to a legend handler. This *handler_map* updates the default handler map found at *matplotlib.legend.Legend.get_legend_handler_map*.

**draggable**

[bool, default: False] Whether the legend can be dragged with the mouse.

**Attributes**

**legend_handles**

List of *Artist* objects added as legend entries.

New in version 3.7.

```
codes = {'best': 0, 'center': 10, 'center left': 6, 'center
right': 7, 'lower center': 8, 'lower left': 3, 'lower right': 4,
'right': 5, 'upper center': 9, 'upper left': 2, 'upper right': 1}
```

**contains**(*mouseevent*)

Test whether the artist contains the mouse event.

**Parameters**

**mouseevent**

[*MouseEvent*]

**Returns**

**contains**

[bool] Whether any values are within the radius.

**details**

[dict] An artist-specific dictionary of details of the event context, such as which points are contained in the pick radius. See the individual Artist subclasses for details.

**draw**(*renderer*)

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

**Parameters**

**renderer**

[*RendererBase* subclass.]

### Notes

This method is overridden in the Artist subclasses.

**draw_frame**(*b*)

Set whether the legend box patch is drawn.

**Parameters**

**b**

[bool]

**get_alignment**()

Get the alignment value of the legend box

**get_bbox_to_anchor**()

Return the bbox that the legend will be anchored to.

**get_children**()

Return a list of the child *Artist*s of this *Artist*.

**classmethod get_default_handler_map**()

Return the global default handler map, shared by all legends.

**get_draggable**()

Return `True` if the legend is draggable, `False` otherwise.

**get_frame**()

Return the *Rectangle* used to frame the legend.

**get_frame_on**()

Get whether the legend box patch is drawn.

**static get_legend_handler**(*legend_handler_map*, *orig_handle*)

Return a legend handler from *legend_handler_map* that corresponds to *orig_handler*.

*legend_handler_map* should be a dictionary object (that is returned by the get_legend_handler_map method).

It first checks if the *orig_handle* itself is a key in the *legend_handler_map* and return the associated value. Otherwise, it checks for each of the classes in its method-resolution-order. If no matching key is found, it returns `None`.

**get_legend_handler_map**()

Return this legend instance's handler map.

**get_lines**()

Return the list of *Line2D*s in the legend.

**get_patches**()

Return the list of *Patch*s in the legend.

**get_texts**()

Return the list of *Text*s in the legend.

**get_tightbbox**(*renderer=None*)

Like *Artist.get_window_extent*, but includes any clipping.

**Parameters**

**renderer**

[*RendererBase* subclass, optional] renderer that will be used to draw the figures (i.e. `fig.canvas.get_renderer()`)

**Returns**

***Bbox* or None**

The enclosing bounding box (in figure pixel coordinates). Returns None if clipping results in no intersection.

**get_title**()

Return the *Text* instance for the legend title.

**get_window_extent**(*renderer=None*)

Get the artist's bounding box in display space.

The bounding box' width and height are nonnegative.

Subclasses should override for inclusion in the bounding box "tight" calculation. Default is to return an empty bounding box at 0, 0.

Be careful when using this function, the results will not update if the artist window extent of the artist changes. The extent can change due to any changes in the transform stack, such as changing the Axes limits, the figure size, or the canvas used (as is done when saving a figure). This can lead to unexpected behavior where interactive figures will look fine on the screen, but will save incorrectly.

**set**(*, *agg_filter=<UNSET>*, *alignment=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *bbox_to_anchor=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *draggable=<UNSET>*, *frame_on=<UNSET>*, *gid=<UNSET>*, *in_layout=<UNSET>*, *label=<UNSET>*, *loc=<UNSET>*, *mouseover=<UNSET>*, *ncols=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *title=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| agg_filt | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image |
| alignment | {'center', 'left', 'right'}. |
| alpha | scalar or None |
| animated | bool |
| bbox_to_ | BboxBase or tuple |
| clip_box | BboxBase or None |
| clip_on | bool |
| clip_path | Patch or (Path, Transform) or None |
| draggable | bool |
| figure | Figure |
| frame_on | bool |
| gid | str |
| in_layout | bool |
| label | object |
| loc | str or pair of floats, default: rcParams["legend.loc"] (default: 'best') for Axes, 'upper right' for Figure |
| mouseover | bool |
| ncols | unknown |
| path_effects | list of AbstractPathEffect |
| picker | None or bool or float or callable |
| rasterized | bool |
| sketch_par | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| title | str |
| transform | Transform |
| url | str |
| visible | bool |
| zorder | float |

**set_alignment**(*alignment*)

Set the alignment of the legend title and the box of entries.

The entries are aligned as a single block, so that markers always lined up.

**Parameters**

**alignment**

[{'center', 'left', 'right'}.]

**set_bbox_to_anchor**(*bbox*, *transform=None*)

Set the bbox that the legend will be anchored to.

**Parameters**

**bbox**

[*BboxBase* or tuple] The bounding box can be specified in the following ways:

- A *BboxBase* instance
- A tuple of `(left, bottom, width, height)` in the given transform (normalized axes coordinate if None)
- A tuple of `(left, bottom)` where the width and height will be assumed to be zero.
- *None*, to remove the bbox anchoring, and use the parent bbox.

**transform**

[*Transform*, optional] A transform to apply to the bounding box. If not specified, this will use a transform to the bounding box of the parent.

**classmethod set_default_handler_map**(*handler_map*)

Set the global default handler map, shared by all legends.

**set_draggable**(*state*, *use_blit=False*, *update='loc'*)

Enable or disable mouse dragging support of the legend.

**Parameters**

**state**

[bool] Whether mouse dragging is enabled.

**use_blit**

[bool, optional] Use blitting for faster image composition. For details see *FuncAnimation*.

**update**

[{'loc', 'bbox'}, optional] The legend parameter to be changed when dragged:

- 'loc': update the *loc* parameter of the legend
- 'bbox': update the *bbox_to_anchor* parameter of the legend

**Returns**

***DraggableLegend* or *None***

If *state* is `True` this returns the *DraggableLegend* helper instance. Otherwise this returns *None*.

**set_frame_on**(*b*)

Set whether the legend box patch is drawn.

**Parameters**

**b**

[bool]

**set_loc**(*loc=None*)

Set the location of the legend.

New in version 3.8.

**Parameters**

**loc**

[str or pair of floats, default: *rcParams["legend.loc"]* (default: `'best'`) for Axes, 'upper right' for Figure] The location of the legend.

The strings `'upper left'`, `'upper right'`, `'lower left'`, `'lower right'` place the legend at the corresponding corner of the axes/figure.

The strings `'upper center'`, `'lower center'`, `'center left'`, `'center right'` place the legend at the center of the corresponding edge of the axes/figure.

The string `'center'` places the legend at the center of the axes/figure.

The string `'best'` places the legend at the location, among the nine locations defined so far, with the minimum overlap with other drawn artists. This option can be quite slow for plots with large amounts of data; your plotting speed may benefit from providing a specific location.

The location can also be a 2-tuple giving the coordinates of the lower-left corner of the legend in axes/figure coordinates (in which case *bbox_to_anchor* will be ignored).

For back-compatibility, `'center right'` (but no other location) can also be spelled `'right'`, and each "string" location can also be given as a numeric value:

| Location String | Location Code |
|---|---|
| 'best' (Axes only) | 0 |
| 'upper right' | 1 |
| 'upper left' | 2 |
| 'lower left' | 3 |
| 'lower right' | 4 |
| 'right' | 5 |
| 'center left' | 6 |
| 'center right' | 7 |
| 'lower center' | 8 |
| 'upper center' | 9 |
| 'center' | 10 |

If a figure is using the constrained layout manager, the string codes of the *loc* keyword argument can get better layout behaviour using the prefix 'outside'. There is ambiguity at the corners, so 'outside upper right' will make space for the legend above the rest of the axes in the layout, and 'outside right upper' will make space on the right side of the layout. In addition to the values of *loc* listed above, we have 'outside right upper', 'outside right lower', 'outside left upper', and 'outside left lower'. See *Legend guide* for more details.

**set_ncols**(*ncols*)

Set the number of columns.

**set_title**(*title*, *prop=None*)

Set legend title and title style.

**Parameters**

**title**

[str] The legend title.

**prop**

[*font_manager.FontProperties* or `str` or `pathlib.Path`] The font properties of the legend title. If a `str`, it is interpreted as a fontconfig pattern parsed by *FontProperties*. If a `pathlib.Path`, it is interpreted as the absolute path to a font file.

**classmethod update_default_handler_map**(*handler_map*)

Update the global default handler map, shared by all legends.

**zorder = 5**

### 7.2.31 matplotlib.legend_handler

Default legend handlers.

---

**Important:** This is a low-level legend API, which most end users do not need.

We recommend that you are familiar with the *legend guide* before reading this documentation.

---

Legend handlers are expected to be a callable object with a following signature:

```
legend_handler(legend, orig_handle, fontsize, handlebox)
```

Where *legend* is the legend itself, *orig_handle* is the original plot, *fontsize* is the fontsize in pixels, and *handlebox* is an `OffsetBox` instance. Within the call, you should create relevant artists (using relevant properties from the *legend* and/or *orig_handle*) and add them into the *handlebox*. The artists need to be scaled according to the *fontsize* (note that the size is in pixels, i.e., this is dpi-scaled value).

This module includes definition of several legend handler classes derived from the base class (HandlerBase) with the following method:

```
def legend_artist(self, legend, orig_handle, fontsize, handlebox)
```

**class** matplotlib.legend_handler.**HandlerBase**(*xpad=0.0*, *ypad=0.0*, *update_func=None*)

A base class for default legend handlers.

The derived classes are meant to override *create_artists* method, which has the following signature:

```
def create_artists(self, legend, orig_handle,
                   xdescent, ydescent, width, height, fontsize,
                   trans):
```

The overridden method needs to create artists of the given transform that fits in the given dimension (xdescent, ydescent, width, height) that are scaled by fontsize if necessary.

**Parameters**

**xpad**

[float, optional] Padding in x-direction.

**ypad**

[float, optional] Padding in y-direction.

**update_func**

[callable, optional] Function for updating the legend handler properties from another legend handler, used by `update_prop`.

**adjust_drawing_area**(*legend*, *orig_handle*, *xdescent*, *ydescent*, *width*, *height*, *fontsize*)

**create_artists**(*legend*, *orig_handle*, *xdescent*, *ydescent*, *width*, *height*, *fontsize*, *trans*)

Return the legend artists generated.

**Parameters**

**legend**

[*Legend*] The legend for which these legend artists are being created.

**orig_handle**

[*Artist* or similar] The object for which these legend artists are being created.

**xdescent, ydescent, width, height**

[int] The rectangle (*xdescent*, *ydescent*, *width*, *height*) that the legend artists being created should fit within.

**fontsize**

[int] The fontsize in pixels. The legend artists being created should be scaled according to the given fontsize.

**trans**

[*Transform*] The transform that is applied to the legend artists being created. Typically from unit coordinates in the handler box to screen coordinates.

**legend_artist**(*legend*, *orig_handle*, *fontsize*, *handlebox*)

Return the artist that this HandlerBase generates for the given original artist/handle.

**Parameters**

**legend**

[*Legend*] The legend for which these legend artists are being created.

**orig_handle**

[*matplotlib.artist.Artist* or similar] The object for which these legend artists are being created.

**fontsize**

[int] The fontsize in pixels. The artists being created should be scaled according to the given fontsize.

**handlebox**

[*OffsetBox*] The box which has been created to hold this legend entry's artists. Artists created in the *legend_artist* method must be added to this handlebox inside this method.

**update_prop**(*legend_handle*, *orig_handle*, *legend*)

**class** matplotlib.legend_handler.**HandlerCircleCollection**(*yoffsets=None*, *sizes=None*, ***kwargs*)

Handler for *CircleCollection*s.

**Parameters**

**numpoints**

[int] Number of points to show in legend entry.

**yoffsets**

[array of floats] Length *numpoints* list of y offsets for each point in legend entry.

****kwargs**

Keyword arguments forwarded to *HandlerNpoints*.

**create_collection**(*orig_handle*, *sizes*, *offsets*, *offset_transform*)

**class** matplotlib.legend_handler.**HandlerErrorbar**(*xerr_size=0.5*, *yerr_size=None*, *marker_pad=0.3*, *numpoints=None*, ***kwargs*)

Handler for Errorbars.

**Parameters**

**marker_pad**

[float] Padding between points in legend entry.

**numpoints**

[int] Number of points to show in legend entry.

****kwargs**

Keyword arguments forwarded to *HandlerBase*.

**create_artists**(*legend*, *orig_handle*, *xdescent*, *ydescent*, *width*, *height*, *fontsize*, *trans*)

Return the legend artists generated.

**Parameters**

**legend**

[*Legend*] The legend for which these legend artists are being created.

**orig_handle**

[*Artist* or similar] The object for which these legend artists are being created.

**xdescent, ydescent, width, height**

[int] The rectangle (*xdescent*, *ydescent*, *width*, *height*) that the legend artists being created should fit within.

**fontsize**

[int] The fontsize in pixels. The legend artists being created should be scaled according to the given fontsize.

**trans**

[*Transform*] The transform that is applied to the legend artists being created. Typically from unit coordinates in the handler box to screen coordinates.

**get_err_size**(*legend*, *xdescent*, *ydescent*, *width*, *height*, *fontsize*)

**class** matplotlib.legend_handler.**HandlerLine2D**(*marker_pad=0.3*, *numpoints=None*, ***kwargs*)

Handler for *Line2D* instances.

**See also:**

***HandlerLine2DCompound***

An earlier handler implementation, which used one artist for the line and another for the marker(s).

**Parameters**

**marker_pad**

[float] Padding between points in legend entry.

**numpoints**

[int] Number of points to show in legend entry.

****kwargs**

Keyword arguments forwarded to *HandlerBase*.

**create_artists**(*legend*, *orig_handle*, *xdescent*, *ydescent*, *width*, *height*, *fontsize*, *trans*)

Return the legend artists generated.

**Parameters**

**legend**

[*Legend*] The legend for which these legend artists are being created.

**orig_handle**

[*Artist* or similar] The object for which these legend artists are being created.

**xdescent, ydescent, width, height**

[int] The rectangle (*xdescent*, *ydescent*, *width*, *height*) that the legend artists being created should fit within.

**fontsize**

[int] The fontsize in pixels. The legend artists being created should be scaled according to the given fontsize.

**trans**

[*Transform*] The transform that is applied to the legend artists being created. Typically from unit coordinates in the handler box to screen coordinates.

**class** matplotlib.legend_handler.**HandlerLine2DCompound**(*marker_pad=0.3*, *numpoints=None*, ***kwargs*)

Original handler for *Line2D* instances, that relies on combining a line-only with a marker-only artist. May be deprecated in the future.

**Parameters**

**marker_pad**

[float] Padding between points in legend entry.

**numpoints**

[int] Number of points to show in legend entry.

****kwargs**

Keyword arguments forwarded to *HandlerBase*.

**create_artists**(*legend*, *orig_handle*, *xdescent*, *ydescent*, *width*, *height*, *fontsize*, *trans*)

Return the legend artists generated.

**Parameters**

**legend**

[*Legend*] The legend for which these legend artists are being created.

**orig_handle**

[*Artist* or similar] The object for which these legend artists are being created.

**xdescent, ydescent, width, height**

[int] The rectangle (*xdescent*, *ydescent*, *width*, *height*) that the legend artists being created should fit within.

**fontsize**

[int] The fontsize in pixels. The legend artists being created should be scaled according to the given fontsize.

**trans**

[*Transform*] The transform that is applied to the legend artists being created. Typically from unit coordinates in the handler box to screen coordinates.

**class** matplotlib.legend_handler.**HandlerLineCollection**(*marker_pad=0.3*, *numpoints=None*, ***kwargs*)

Handler for *LineCollection* instances.

**Parameters**

**marker_pad**

[float] Padding between points in legend entry.

**numpoints**

[int] Number of points to show in legend entry.

****kwargs**

Keyword arguments forwarded to *HandlerBase*.

**create_artists**(*legend*, *orig_handle*, *xdescent*, *ydescent*, *width*, *height*, *fontsize*, *trans*)

Return the legend artists generated.

**Parameters**

**legend**

[*Legend*] The legend for which these legend artists are being created.

**orig_handle**

[*Artist* or similar] The object for which these legend artists are being created.

**xdescent, ydescent, width, height**

[int] The rectangle (*xdescent*, *ydescent*, *width*, *height*) that the legend artists being created should fit within.

**fontsize**

[int] The fontsize in pixels. The legend artists being created should be scaled according to the given fontsize.

**trans**

[*Transform*] The transform that is applied to the legend artists being created. Typically from unit coordinates in the handler box to screen coordinates.

**get_numpoints**(*legend*)

**class** matplotlib.legend_handler.**HandlerNpoints**(*marker_pad=0.3*, *numpoints=None*, ***kwargs*)

A legend handler that shows *numpoints* points in the legend entry.

**Parameters**

**marker_pad**

[float] Padding between points in legend entry.

**numpoints**

[int] Number of points to show in legend entry.

****kwargs**

Keyword arguments forwarded to *HandlerBase*.

**get_numpoints**(*legend*)

**get_xdata**(*legend*, *xdescent*, *ydescent*, *width*, *height*, *fontsize*)

**class** matplotlib.legend_handler.**HandlerNpointsYoffsets**(*numpoints=None*, *yoffsets=None*, ***kwargs*)

A legend handler that shows *numpoints* in the legend, and allows them to be individually offset in the y-direction.

**Parameters**

**numpoints**

[int] Number of points to show in legend entry.

**yoffsets**

[array of floats] Length *numpoints* list of y offsets for each point in legend entry.

****kwargs**

Keyword arguments forwarded to *HandlerNpoints*.

**get_ydata**(*legend*, *xdescent*, *ydescent*, *width*, *height*, *fontsize*)

**class** matplotlib.legend_handler.**HandlerPatch**(*patch_func=None*, ***kwargs*)

Handler for *Patch* instances.

**Parameters**

**patch_func**

[callable, optional] The function that creates the legend key artist. *patch_func* should have the signature:

```
def patch_func(legend=legend, orig_handle=orig_handle,
               xdescent=xdescent, ydescent=ydescent,
               width=width, height=height,␣
↪fontsize=fontsize)
```

Subsequently, the created artist will have its `update_prop` method called and the appropriate transform will be applied.

****kwargs**

Keyword arguments forwarded to *HandlerBase*.

**create_artists**(*legend*, *orig_handle*, *xdescent*, *ydescent*, *width*, *height*, *fontsize*, *trans*)

Return the legend artists generated.

**Parameters**

**legend**

[*Legend*] The legend for which these legend artists are being created.

**orig_handle**

[*Artist* or similar] The object for which these legend artists are being created.

**xdescent, ydescent, width, height**

[int] The rectangle (*xdescent*, *ydescent*, *width*, *height*) that the legend artists being created should fit within.

**fontsize**

[int] The fontsize in pixels. The legend artists being created should be scaled according to the given fontsize.

**trans**

[*Transform*] The transform that is applied to the legend artists being created. Typically from unit coordinates in the handler box to screen coordinates.

**class** matplotlib.legend_handler.**HandlerPathCollection**(*yoffsets=None*, *sizes=None*, ***kwargs*)

Handler for *PathCollection*s, which are used by *scatter*.

**Parameters**

**numpoints**

[int] Number of points to show in legend entry.

**yoffsets**

[array of floats] Length *numpoints* list of y offsets for each point in legend entry.

****kwargs**

Keyword arguments forwarded to *HandlerNpoints*.

**create_collection**(*orig_handle*, *sizes*, *offsets*, *offset_transform*)

**class** matplotlib.legend_handler.**HandlerPolyCollection**(*xpad=0.0*, *ypad=0.0*, *update_func=None*)

Handler for *PolyCollection* used in *fill_between* and *stackplot*.

**Parameters**

**xpad**

[float, optional] Padding in x-direction.

**ypad**

[float, optional] Padding in y-direction.

**update_func**

[callable, optional] Function for updating the legend handler properties from another legend handler, used by *update_prop*.

**create_artists**(*legend*, *orig_handle*, *xdescent*, *ydescent*, *width*, *height*, *fontsize*, *trans*)

Return the legend artists generated.

**Parameters**

**legend**

[*Legend*] The legend for which these legend artists are being created.

**orig_handle**

[*Artist* or similar] The object for which these legend artists are being created.

**xdescent, ydescent, width, height**

[int] The rectangle (*xdescent*, *ydescent*, *width*, *height*) that the legend artists being created should fit within.

**fontsize**

[int] The fontsize in pixels. The legend artists being created should be scaled according to the given fontsize.

**trans**

[*Transform*] The transform that is applied to the legend artists being created. Typically from unit coordinates in the handler box to screen coordinates.

**class** matplotlib.legend_handler.**HandlerRegularPolyCollection**(*yoffsets=None*, *sizes=None*, ***kwargs*)

Handler for *RegularPolyCollection*s.

**Parameters**

**numpoints**

[int] Number of points to show in legend entry.

**yoffsets**

[array of floats] Length *numpoints* list of y offsets for each point in legend entry.

****kwargs**

Keyword arguments forwarded to *HandlerNpoints*.

**create_artists**(*legend*, *orig_handle*, *xdescent*, *ydescent*, *width*, *height*, *fontsize*, *trans*)

Return the legend artists generated.

**Parameters**

**legend**

[*Legend*] The legend for which these legend artists are being created.

**orig_handle**

[*Artist* or similar] The object for which these legend artists are being created.

**xdescent, ydescent, width, height**

[int] The rectangle (*xdescent*, *ydescent*, *width*, *height*) that the legend artists being created should fit within.

**fontsize**

[int] The fontsize in pixels. The legend artists being created should be scaled according to the given fontsize.

**trans**

[*Transform*] The transform that is applied to the legend artists being created. Typically from unit coordinates in the handler box to screen coordinates.

**create_collection**(*orig_handle*, *sizes*, *offsets*, *offset_transform*)

**get_numpoints**(*legend*)

**get_sizes**(*legend*, *orig_handle*, *xdescent*, *ydescent*, *width*, *height*, *fontsize*)

**update_prop**(*legend_handle*, *orig_handle*, *legend*)

**class** matplotlib.legend_handler.**HandlerStem**(*marker_pad=0.3*, *numpoints=None*, *bottom=None*, *yoffsets=None*, ***kwargs*)

Handler for plots produced by *stem*.

**Parameters**

**marker_pad**

[float, default: 0.3] Padding between points in legend entry.

**numpoints**

[int, optional] Number of points to show in legend entry.

**bottom**

[float, optional]

**yoffsets**

[array of floats, optional] Length *numpoints* list of y offsets for each point in legend entry.

****kwargs**

Keyword arguments forwarded to *HandlerNpointsYoffsets*.

**create_artists**(*legend*, *orig_handle*, *xdescent*, *ydescent*, *width*, *height*, *fontsize*, *trans*)

Return the legend artists generated.

**Parameters**

**legend**

[*Legend*] The legend for which these legend artists are being created.

**orig_handle**

[*Artist* or similar] The object for which these legend artists are being created.

**xdescent, ydescent, width, height**

[int] The rectangle (*xdescent*, *ydescent*, *width*, *height*) that the legend artists being created should fit within.

**fontsize**

[int] The fontsize in pixels. The legend artists being created should be scaled according to the given fontsize.

**trans**

[*Transform*] The transform that is applied to the legend artists being created. Typically from unit coordinates in the handler box to screen coordinates.

**get_ydata**(*legend*, *xdescent*, *ydescent*, *width*, *height*, *fontsize*)

**class** matplotlib.legend_handler.**HandlerStepPatch**(*xpad=0.0*, *ypad=0.0*, *update_func=None*)

Handler for *StepPatch* instances.

**Parameters**

**xpad**

[float, optional] Padding in x-direction.

**ypad**

[float, optional] Padding in y-direction.

**update_func**

[callable, optional] Function for updating the legend handler properties from another legend handler, used by *update_prop*.

**create_artists**(*legend*, *orig_handle*, *xdescent*, *ydescent*, *width*, *height*, *fontsize*, *trans*)

Return the legend artists generated.

**Parameters**

**legend**

[*Legend*] The legend for which these legend artists are being created.

**orig_handle**

[*Artist* or similar] The object for which these legend artists are being created.

**xdescent, ydescent, width, height**

[int] The rectangle (*xdescent*, *ydescent*, *width*, *height*) that the legend artists being created should fit within.

**fontsize**

[int] The fontsize in pixels. The legend artists being created should be scaled according to the given fontsize.

**trans**

[*Transform*] The transform that is applied to the legend artists being created. Typically from unit coordinates in the handler box to screen coordinates.

**class** matplotlib.legend_handler.**HandlerTuple**(*ndivide=1*, *pad=None*, ***kwargs*)

Handler for Tuple.

**Parameters**

**ndivide**

[int or None, default: 1] The number of sections to divide the legend area into. If None, use the length of the input tuple.

**pad**

[float, default: `rcParams["legend.borderpad"]` (default: `0.4`)] Padding in units of fraction of font size.

****kwargs**

Keyword arguments forwarded to `HandlerBase`.

**create_artists**(*legend*, *orig_handle*, *xdescent*, *ydescent*, *width*, *height*, *fontsize*, *trans*)

Return the legend artists generated.

**Parameters**

**legend**

[`Legend`] The legend for which these legend artists are being created.

**orig_handle**

[`Artist` or similar] The object for which these legend artists are being created.

**xdescent, ydescent, width, height**

[int] The rectangle (*xdescent*, *ydescent*, *width*, *height*) that the legend artists being created should fit within.

**fontsize**

[int] The fontsize in pixels. The legend artists being created should be scaled according to the given fontsize.

**trans**

[`Transform`] The transform that is applied to the legend artists being created. Typically from unit coordinates in the handler box to screen coordinates.

matplotlib.legend_handler.**update_from_first_child**(*tgt*, *src*)

### 7.2.32 matplotlib.lines

2D lines with support for a variety of line styles, markers, colors, etc.

### Classes

| | |
|---|---|
| *Line2D*(xdata, ydata, *[, linewidth, ...]) | A line - the line can have both a solid linestyle connecting all the vertices, and a marker at each vertex. |
| *VertexSelector*(line) | Manage the callbacks to maintain a list of selected vertices for *Line2D*. |
| *AxLine*(xy1, xy2, slope, **kwargs) | A helper class that implements *axline*, by recomputing the artist transform at draw time. |

### matplotlib.lines.Line2D

**class** matplotlib.lines.**Line2D**(*xdata*, *ydata*, *, *linewidth=None*, *linestyle=None*, *color=None*, *gapcolor=None*, *marker=None*, *markersize=None*, *markeredgewidth=None*, *markeredgecolor=None*, *markerfacecolor=None*, *markerfacecoloralt='none'*, *fillstyle=None*, *antialiased=None*, *dash_capstyle=None*, *solid_capstyle=None*, *dash_joinstyle=None*, *solid_joinstyle=None*, *pickradius=5*, *drawstyle=None*, *markevery=None*, **kwargs*)

Bases: *Artist*

A line - the line can have both a solid linestyle connecting all the vertices, and a marker at each vertex. Additionally, the drawing of the solid line is influenced by the drawstyle, e.g., one can create "stepped" lines in various styles.

Create a *Line2D* instance with *x* and *y* data in sequences of *xdata*, *ydata*.

Additional keyword arguments are *Line2D* properties:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* or aa | bool |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* or c | *color* |
| *dash_capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *dash_joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *dashes* | sequence of floats (on/off ink in points) or (None, None) |
| *data* | (2, N) array or two 1D arrays |
| *drawstyle* or ds | {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'default' |
| *figure* | *Figure* |

Table 64 – continued from previou

| Property | Description |
|---|---|
| *fillstyle* | {'full', 'left', 'right', 'bottom', 'top', 'none'} |
| *gapcolor* | *color* or None |
| *gid* | str |
| *in_layout* | bool |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float |
| *marker* | marker style string, *Path* or *MarkerStyle* |
| *markeredgecolor* or mec | *color* |
| *markeredgewidth* or mew | float |
| *markerfacecolor* or mfc | *color* |
| *markerfacecoloralt* or mfcalt | *color* |
| *markersize* or ms | float |
| *markevery* | None or int or (int, int) or slice or list[int] or float or (float, float) or list[ |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | float or callable[[Artist, Event], tuple[bool, dict]] |
| *pickradius* | float |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *solid_capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *solid_joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *transform* | unknown |
| *url* | str |
| *visible* | bool |
| *xdata* | 1D array |
| *ydata* | 1D array |
| *zorder* | float |

See *set_linestyle()* for a description of the line styles, *set_marker()* for a description of the markers, and *set_drawstyle()* for a description of the draw styles.

**contains**(*mouseevent*)

Test whether *mouseevent* occurred on the line.

An event is deemed to have occurred "on" the line if it is less than `self.pickradius` (default: 5 points) away from it. Use *get_pickradius* or *set_pickradius* to get or set the pick radius.

**Parameters**

**mouseevent**

[*MouseEvent*]

**Returns**

**contains**

[bool] Whether any values are within the radius.

**details**

[dict] A dictionary `{'ind': pointlist}`, where *pointlist* is a list of points of the line that are within the pickradius around the event position.

TODO: sort returned indices by distance

**draw**(*renderer*)

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

**Parameters**

**renderer**

[*RendererBase* subclass.]

### Notes

This method is overridden in the Artist subclasses.

**drawStyleKeys = ['default', 'steps-mid', 'steps-pre', 'steps-post', 'steps']**

**drawStyles = {'default': '_draw_lines', 'steps': '_draw_steps_pre', 'steps-mid': '_draw_steps_mid', 'steps-post': '_draw_steps_post', 'steps-pre': '_draw_steps_pre'}**

**fillStyles = ('full', 'left', 'right', 'bottom', 'top', 'none')**

**filled_markers = ('.', 'o', 'v', '^', '<', '>', '8', 's', 'p', '*', 'h', 'H', 'D', 'd', 'P', 'X')**

**get_aa**()

Alias for *get_antialiased*.

**get_antialiased**()

Return whether antialiased rendering is used.

**get_bbox**()

Get the bounding box of this line.

**get_c**()

Alias for *get_color*.

**get_color**()

Return the line color.

See also *set_color*.

**get_dash_capstyle**()

Return the *CapStyle* for dashed lines.

See also *set_dash_capstyle*.

**get_dash_joinstyle**()

Return the *JoinStyle* for dashed lines.

See also *set_dash_joinstyle*.

**get_data**(*orig=True*)

Return the line data as an `(xdata, ydata)` pair.

If *orig* is *True*, return the original data.

**get_drawstyle**()

Return the drawstyle.

See also *set_drawstyle*.

**get_ds**()

Alias for *get_drawstyle*.

**get_fillstyle**()

Return the marker fill style.

See also *set_fillstyle*.

**get_gapcolor**()

Return the line gapcolor.

See also *set_gapcolor*.

**get_linestyle**()

Return the linestyle.

See also *set_linestyle*.

**get_linewidth**()

Return the linewidth in points.

See also *set_linewidth*.

**get_ls**()

Alias for *get_linestyle*.

**get_lw**()

Alias for *get_linewidth*.

**get_marker**()

Return the line marker.

See also *set_marker*.

**get_markeredgecolor**()

Return the marker edge color.

See also *set_markeredgecolor*.

**get_markeredgewidth**()

Return the marker edge width in points.

See also *set_markeredgewidth*.

**get_markerfacecolor**()

Return the marker face color.

See also *set_markerfacecolor*.

**get_markerfacecoloralt**()

Return the alternate marker face color.

See also *set_markerfacecoloralt*.

**get_markersize**()

Return the marker size in points.

See also *set_markersize*.

**get_markevery**()

Return the markevery setting for marker subsampling.

See also *set_markevery*.

**get_mec**()

Alias for *get_markeredgecolor*.

**get_mew**()

Alias for *get_markeredgewidth*.

**get_mfc**()

Alias for *get_markerfacecolor*.

**get_mfcalt**()

Alias for *get_markerfacecoloralt*.

**get_ms**()

Alias for *get_markersize*.

**get_path**()

Return the *Path* associated with this line.

**get_pickradius**()

Return the pick radius used for containment tests.

See *contains* for more details.

**get_solid_capstyle**()

Return the *CapStyle* for solid lines.

See also *set_solid_capstyle*.

**get_solid_joinstyle**()

Return the *JoinStyle* for solid lines.

See also *set_solid_joinstyle*.

**get_window_extent**(*renderer=None*)

Get the artist's bounding box in display space.

The bounding box' width and height are nonnegative.

Subclasses should override for inclusion in the bounding box "tight" calculation. Default is to return an empty bounding box at 0, 0.

Be careful when using this function, the results will not update if the artist window extent of the artist changes. The extent can change due to any changes in the transform stack, such as changing the Axes limits, the figure size, or the canvas used (as is done when saving a figure). This can lead to unexpected behavior where interactive figures will look fine on the screen, but will save incorrectly.

**get_xdata**(*orig=True*)

Return the xdata.

If *orig* is *True*, return the original data, else the processed data.

**get_xydata**()

Return the *xy* data as a (N, 2) array.

**get_ydata**(*orig=True*)

Return the ydata.

If *orig* is *True*, return the original data, else the processed data.

**is_dashed**()

Return whether line has a dashed linestyle.

A custom linestyle is assumed to be dashed, we do not inspect the `onoffseq` directly.

See also *set_linestyle*.

**lineStyles = {'': '_draw_nothing', ' ': '_draw_nothing', '-': '_draw_solid', '--': '_draw_dashed', '-.': '_draw_dash_dot', ':': '_draw_dotted', 'None': '_draw_nothing'}**

```
markers = {' ': 'nothing', '': 'nothing', '*': 'star', '+':
'plus', ',': 'pixel', '.': 'point', '1': 'tri_down', '2':
'tri_up', '3': 'tri_left', '4': 'tri_right', '8': 'octagon', '<':
'triangle_left', '>': 'triangle_right', 'D': 'diamond', 'H':
'hexagon2', 'None': 'nothing', 'P': 'plus_filled', 'X': 'x_filled',
'^': 'triangle_up', '_': 'hline', 'd': 'thin_diamond', 'h':
'hexagon1', 'none': 'nothing', 'o': 'circle', 'p': 'pentagon',
's': 'square', 'v': 'triangle_down', 'x': 'x', '|': 'vline', 0:
'tickleft', 1: 'tickright', 10: 'caretupbase', 11:
'caretdownbase', 2: 'tickup', 3: 'tickdown', 4: 'caretleft', 5:
'caretright', 6: 'caretup', 7: 'caretdown', 8: 'caretleftbase', 9:
'caretrightbase'}
```

**property pickradius**

Return the pick radius used for containment tests.

See *contains* for more details.

**recache**(*always=False*)

**recache_always**()

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *antialiased=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *color=<UNSET>*, *dash_capstyle=<UNSET>*, *dash_joinstyle=<UNSET>*, *dashes=<UNSET>*, *data=<UNSET>*, *drawstyle=<UNSET>*, *fillstyle=<UNSET>*, *gapcolor=<UNSET>*, *gid=<UNSET>*, *in_layout=<UNSET>*, *label=<UNSET>*, *linestyle=<UNSET>*, *linewidth=<UNSET>*, *marker=<UNSET>*, *markeredgecolor=<UNSET>*, *markeredgewidth=<UNSET>*, *markerfacecolor=<UNSET>*, *markerfacecoloralt=<UNSET>*, *markersize=<UNSET>*, *markevery=<UNSET>*, *mouseover=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *pickradius=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *solid_capstyle=<UNSET>*, *solid_joinstyle=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *xdata=<UNSET>*, *ydata=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and retur |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* | bool |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* | *color* |

Table 65 – continued from previou

| Property | Description |
|---|---|
| *dash_capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *dash_joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *dashes* | sequence of floats (on/off ink in points) or (None, None) |
| *data* | (2, N) array or two 1D arrays |
| *drawstyle* | {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'default' |
| *figure* | *Figure* |
| *fillstyle* | {'full', 'left', 'right', 'bottom', 'top', 'none'} |
| *gapcolor* | *color* or None |
| *gid* | str |
| *in_layout* | bool |
| *label* | object |
| *linestyle* | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* | float |
| *marker* | marker style string, *Path* or *MarkerStyle* |
| *markeredgecolor* | *color* |
| *markeredgewidth* | float |
| *markerfacecolor* | *color* |
| *markerfacecoloralt* | *color* |
| *markersize* | float |
| *markevery* | None or int or (int, int) or slice or list[int] or float or (float, float) or list[bool |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | float or callable[[Artist, Event], tuple[bool, dict]] |
| *pickradius* | float |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *solid_capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *solid_joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *transform* | unknown |
| *url* | str |
| *visible* | bool |
| *xdata* | 1D array |
| *ydata* | 1D array |
| *zorder* | float |

**set_aa**(*b*)

Alias for *set_antialiased*.

**set_antialiased**(*b*)

Set whether to use antialiased rendering.

**Parameters**

**b**

[bool]

**set_c**(*color*)

Alias for *set_color*.

**set_color**(*color*)

Set the color of the line.

**Parameters**

**color**

[*color*]

**set_dash_capstyle**(*s*)

How to draw the end caps if the line is *is_dashed*.

The default capstyle is *rcParams["lines.dash_capstyle"]* (default: `<CapStyle.butt: 'butt'>`).

**Parameters**

**s**

[*CapStyle* or {'butt', 'projecting', 'round'}]

**set_dash_joinstyle**(*s*)

How to join segments of the line if it *is_dashed*.

The default joinstyle is *rcParams["lines.dash_joinstyle"]* (default: `<JoinStyle.round: 'round'>`).

**Parameters**

**s**

[*JoinStyle* or {'miter', 'round', 'bevel'}]

**set_dashes**(*seq*)

Set the dash sequence.

The dash sequence is a sequence of floats of even length describing the length of dashes and spaces in points.

For example, (5, 2, 1, 2) describes a sequence of 5 point and 1 point dashes separated by 2 point spaces.

See also *set_gapcolor*, which allows those spaces to be filled with a color.

**Parameters**

**seq**

[sequence of floats (on/off ink in points) or (None, None)] If *seq* is empty or `(None, None)`, the linestyle will be set to solid.

**set_data**(**args*)

Set the x and y data.

**Parameters**

***args**

[(2, N) array or two 1D arrays]

**See also:**

*set_xdata*
*set_ydata*

**set_drawstyle**(*drawstyle*)

Set the drawstyle of the plot.

The drawstyle determines how the points are connected.

**Parameters**

**drawstyle**

[{'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'default'] For 'default', the points are connected with straight lines.

The steps variants connect the points with step-like lines, i.e. horizontal lines with vertical steps. They differ in the location of the step:

- 'steps-pre': The step is at the beginning of the line segment, i.e. the line will be at the y-value of point to the right.
- 'steps-mid': The step is halfway between the points.
- 'steps-post: The step is at the end of the line segment, i.e. the line will be at the y-value of the point to the left.
- 'steps' is equal to 'steps-pre' and is maintained for backward-compatibility.

For examples see *Step Demo*.

**set_ds**(*drawstyle*)

Alias for *set_drawstyle*.

**set_fillstyle**(*fs*)

Set the marker fill style.

**Parameters**

**fs**

[{'full', 'left', 'right', 'bottom', 'top', 'none'}] Possible values:

- 'full': Fill the whole marker with the *markerfacecolor*.
- 'left', 'right', 'bottom', 'top': Fill the marker half at the given side with the *markerfacecolor*. The other half of the marker is filled with *markerfacecoloralt*.
- 'none': No filling.

For examples see *Marker fill styles*.

**set_gapcolor**(*gapcolor*)

Set a color to fill the gaps in the dashed line style.

---

**Note:** Striped lines are created by drawing two interleaved dashed lines. There can be overlaps between those two, which may result in artifacts when using transparency.

This functionality is experimental and may change.

---

**Parameters**

**gapcolor**

[*color* or None] The color with which to fill the gaps. If None, the gaps are unfilled.

**set_linestyle**(*ls*)

Set the linestyle of the line.

**Parameters**

**ls**

[{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}] Possible values:

- A string:

| linestyle | description |
|---|---|
| `'-'` or `'solid'` | solid line |
| `'--'` or `'dashed'` | dashed line |
| `'-.'` or `'dashdot'` | dash-dotted line |
| `':'` or `'dotted'` | dotted line |
| `'none'`, `'None'`, `' '`, or `''` | draw nothing |

- Alternatively a dash tuple of the following form can be provided:

```
(offset, onoffseq)
```

where `onoffseq` is an even length tuple of on and off ink in points. See also *set_dashes()*.

For examples see *Linestyles*.

**set_linewidth**(*w*)

Set the line width in points.

**Parameters**

**w**

[float] Line width, in points.

**set_ls**(*ls*)

Alias for *set_linestyle*.

**set_lw**(*w*)

Alias for *set_linewidth*.

**set_marker**(*marker*)

Set the line marker.

**Parameters**

**marker**

[marker style string, *Path* or *MarkerStyle*] See *markers* for full description of possible arguments.

**set_markeredgecolor**(*ec*)

Set the marker edge color.

**Parameters**

**ec**

[*color*]

**set_markeredgewidth**(*ew*)

Set the marker edge width in points.

**Parameters**

**ew**

[float] Marker edge width, in points.

**set_markerfacecolor**(*fc*)

Set the marker face color.

**Parameters**

**fc**

[*color*]

**set_markerfacecoloralt**(*fc*)

Set the alternate marker face color.

**Parameters**

**fc**

[*color*]

**set_markersize**(*sz*)

Set the marker size in points.

**Parameters**

**sz**

[float] Marker size, in points.

**set_markevery**(*every*)

Set the markevery property to subsample the plot when using markers.

e.g., if `every=5`, every 5-th marker will be plotted.

**Parameters**

**every**

[None or int or (int, int) or slice or list[int] or float or (float, float) or list[bool]] Which markers to plot.

- `every=None`: every point will be plotted.
- `every=N`: every N-th marker will be plotted starting with marker 0.
- `every=(start, N)`: every N-th marker, starting at index *start*, will be plotted.
- `every=slice(start, end, N)`: every N-th marker, starting at index *start*, up to but not including index *end*, will be plotted.
- `every=[i, j, m, ...]`: only markers at the given indices will be plotted.
- `every=[True, False, True, ...]`: only positions that are True will be plotted. The list must have the same length as the data points.

- `every=0.1`, (i.e. a float): markers will be spaced at approximately equal visual distances along the line; the distance along the line between markers is determined by multiplying the display-coordinate distance of the Axes bounding-box diagonal by the value of *every*.
- `every=(0.5, 0.1)` (i.e. a length-2 tuple of float): similar to `every=0.1` but the first marker will be offset along the line by 0.5 multiplied by the display-coordinate-diagonal-distance along the line.

For examples see *Markevery Demo*.

### Notes

Setting *markevery* will still only draw markers at actual data points. While the float argument form aims for uniform visual spacing, it has to coerce from the ideal spacing to the nearest available data point. Depending on the number and distribution of data points, the result may still not look evenly spaced.

When using a start offset to specify the first marker, the offset will be from the first data point which may be different from the first the visible data point if the plot is zoomed in.

If zooming in on a plot when using float arguments then the actual data points that have markers will change because the distance between markers is always determined from the display-coordinates axes-bounding-box-diagonal regardless of the actual axes data limits.

**set_mec**(*ec*)

Alias for *set_markeredgecolor*.

**set_mew**(*ew*)

Alias for *set_markeredgewidth*.

**set_mfc**(*fc*)

Alias for *set_markerfacecolor*.

**set_mfcalt**(*fc*)

Alias for *set_markerfacecoloralt*.

**set_ms**(*sz*)

Alias for *set_markersize*.

**set_picker**(*p*)

Set the event picker details for the line.

**Parameters**

**p**

[float or callable[[Artist, Event], tuple[bool, dict]]] If a float, it is used as the pick radius in points.

**set_pickradius**(*pickradius*)

Set the pick radius used for containment tests.

See *contains* for more details.

**Parameters**

**pickradius**

[float] Pick radius, in points.

**set_solid_capstyle**(*s*)

How to draw the end caps if the line is solid (not *is_dashed*)

The default capstyle is *rcParams["lines.solid_capstyle"]* (default: `<CapStyle.projecting: 'projecting'>`).

**Parameters**

**s**

[*CapStyle* or {'butt', 'projecting', 'round'}]

**set_solid_joinstyle**(*s*)

How to join segments if the line is solid (not *is_dashed*).

The default joinstyle is *rcParams["lines.solid_joinstyle"]* (default: `<JoinStyle.round: 'round'>`).

**Parameters**

**s**

[*JoinStyle* or {'miter', 'round', 'bevel'}]

**set_transform**(*t*)

Set the artist transform.

**Parameters**

**t**

[*Transform*]

**set_xdata**(*x*)

Set the data array for x.

**Parameters**

**x**

[1D array]

**See also:**

*set_data*
*set_ydata*

**set_ydata**(*y*)

Set the data array for y.

**Parameters**

**y**

[1D array]

**See also:**

*set_data*
*set_xdata*

**update_from**(*other*)

Copy properties from *other* to self.

**zorder = 2**

### Examples using `matplotlib.lines.Line2D`

- *Customizing dashed line styles*
- *Stem Plot*
- *Figure labels: suptitle, supxlabel, supylabel*
- *Boxplots*
- *Scale invariant angle label*
- *Annotating a plot*
- *Annotating Plots*
- *Annotation Polar*
- *Composing Custom Legends*
- *Annotation arrow style reference*
- *Figure legend demo*
- *Legend Demo*
- *Artist within an artist*
- *Reference for Matplotlib artists*

- *PathPatch object*
- *Parasite Simple*
- *Parasite Axes demo*
- *Parasite axis demo*
- *Stock prices over 32 years*
- *Decay*
- *The double pendulum problem*
- *Multiple Axes animation*
- *Animated line plot*
- *Oscilloscope*
- *MATPLOTLIB UNCHAINED*
- *Cross-hair cursor*
- *Data browser*
- *Legend picking*
- *Looking Glass*
- *Pick event demo*
- *Pick event demo 2*
- *Poly Editor*
- *Resampling Data*
- *Anchored Artists*
- *Adding lines to figures*
- *Patheffect Demo*
- *Set and get properties*
- *SVG Filter Line*
- *SkewT-logP diagram: using transforms and custom projections*
- *Multiple y-axis with Spines*
- *Custom tick formatter for time series*
- *Fig Axes Customize Simple*
- *Artist tests*
- *Annotated cursor*
- *Buttons*
- *Check buttons*

- *Radio Buttons*
- *Thresholding an Image with RangeSlider*
- *Slider*
- *Snapping Sliders to Discrete Values*
- *Span Selector*
- *Textbox*
- *Simple Legend02*
- *Pyplot tutorial*
- *Artist tutorial*
- *Quick start guide*
- *Animations using Matplotlib*
- *Faster rendering by using blitting*
- *Transformations Tutorial*
- *Legend guide*
- *Annotations*

### matplotlib.lines.VertexSelector

**class** matplotlib.lines.**VertexSelector**(*line*)

Bases: `object`

Manage the callbacks to maintain a list of selected vertices for *`Line2D`*. Derived classes should override the *`process_selected`* method to do something with the picks.

Here is an example which highlights the selected verts with red circles:

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.lines as lines

class HighlightSelected(lines.VertexSelector):
    def __init__(self, line, fmt='ro', **kwargs):
        super().__init__(line)
        self.markers, = self.axes.plot([], [], fmt, **kwargs)

    def process_selected(self, ind, xs, ys):
        self.markers.set_data(xs, ys)
        self.canvas.draw()

fig, ax = plt.subplots()
x, y = np.random.rand(2, 30)
line, = ax.plot(x, y, 'bs-', picker=5)
```

(continues on next page)

(continued from previous page)

```
selector = HighlightSelected(line)
plt.show()
```

**Parameters**

**line**

[*Line2D*] The line must already have been added to an *Axes* and must have its picker property set.

**property canvas**

**onpick**(*event*)

When the line is picked, update the set of selected indices.

**process_selected**(*ind*, *xs*, *ys*)

Default "do nothing" implementation of the *process_selected* method.

**Parameters**

**ind**

[list of int] The indices of the selected vertices.

**xs, ys**

[array-like] The coordinates of the selected vertices.

## matplotlib.lines.AxLine

**class** matplotlib.lines.**AxLine**(*xy1*, *xy2*, *slope*, ***kwargs*)

Bases: *Line2D*

A helper class that implements *axline*, by recomputing the artist transform at draw time.

**Parameters**

**xy1**

[(float, float)] The first set of (x, y) coordinates for the line to pass through.

**xy2**

[(float, float) or None] The second set of (x, y) coordinates for the line to pass through. Both *xy2* and *slope* must be passed, but one of them must be None.

**slope**

[float or None] The slope of the line. Both *xy2* and *slope* must be passed, but one of them must be None.

**draw**(*renderer*)

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

**Parameters**

**renderer**

[*RendererBase* subclass.]

### Notes

This method is overridden in the Artist subclasses.

**get_slope**()

Return the *slope* value of the line.

**get_transform**()

Return the *Transform* instance used by this artist.

**get_xy1**()

Return the *xy1* value of the line.

**get_xy2**()

Return the *xy2* value of the line.

**set**(**, agg_filter=<UNSET>, alpha=<UNSET>, animated=<UNSET>, antialiased=<UNSET>, clip_box=<UNSET>, clip_on=<UNSET>, clip_path=<UNSET>, color=<UNSET>, dash_capstyle=<UNSET>, dash_joinstyle=<UNSET>, dashes=<UNSET>, data=<UNSET>, drawstyle=<UNSET>, fillstyle=<UNSET>, gapcolor=<UNSET>, gid=<UNSET>, in_layout=<UNSET>, label=<UNSET>, linestyle=<UNSET>, linewidth=<UNSET>, marker=<UNSET>, markeredgecolor=<UNSET>, markeredgewidth=<UNSET>, markerfacecolor=<UNSET>, markerfacecoloralt=<UNSET>, markersize=<UNSET>, markevery=<UNSET>, mouseover=<UNSET>, path_effects=<UNSET>, picker=<UNSET>, pickradius=<UNSET>, rasterized=<UNSET>, sketch_params=<UNSET>, slope=<UNSET>, snap=<UNSET>, solid_capstyle=<UNSET>, solid_joinstyle=<UNSET>, transform=<UNSET>, url=<UNSET>, visible=<UNSET>, xdata=<UNSET>, xy1=<UNSET>, xy2=<UNSET>, ydata=<UNSET>, zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* or aa | bool |

Table 66 – continued from pre

| Property | Description |
|---|---|
| clip_box | BboxBase or None |
| clip_on | bool |
| clip_path | Patch or (Path, Transform) or None |
| color or c | color |
| dash_capstyle | CapStyle or {'butt', 'projecting', 'round'} |
| dash_joinstyle | JoinStyle or {'miter', 'round', 'bevel'} |
| dashes | sequence of floats (on/off ink in points) or (None, None) |
| data | (2, N) array or two 1D arrays |
| drawstyle or ds | {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'defa |
| figure | Figure |
| fillstyle | {'full', 'left', 'right', 'bottom', 'top', 'none'} |
| gapcolor | color or None |
| gid | str |
| in_layout | bool |
| label | object |
| linestyle or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| linewidth or lw | float |
| marker | marker style string, Path or MarkerStyle |
| markeredgecolor or mec | color |
| markeredgewidth or mew | float |
| markerfacecolor or mfc | color |
| markerfacecoloralt or mfcalt | color |
| markersize or ms | float |
| markevery | None or int or (int, int) or slice or list[int] or float or (float, float) o |
| mouseover | bool |
| path_effects | list of AbstractPathEffect |
| picker | float or callable[[Artist, Event], tuple[bool, dict]] |
| pickradius | float |
| rasterized | bool |
| sketch_params | (scale: float, length: float, randomness: float) |
| slope | float |
| snap | bool or None |
| solid_capstyle | CapStyle or {'butt', 'projecting', 'round'} |
| solid_joinstyle | JoinStyle or {'miter', 'round', 'bevel'} |
| transform | Transform |
| url | str |
| visible | bool |
| xdata | 1D array |
| xy1 | unknown |
| xy2 | unknown |
| ydata | 1D array |
| zorder | float |

**set_slope**(*slope*)

Set the *slope* value of the line.

**Parameters**

**slope**

[float] The slope of the line.

**set_xy1** (*x*, *y*)

Set the *xy1* value of the line.

**Parameters**

**x, y**

[float] Points for the line to pass through.

**set_xy2** (*x*, *y*)

Set the *xy2* value of the line.

**Parameters**

**x, y**

[float] Points for the line to pass through.

## Functions

| | |
|---|---|
| *segment_hits*(cx, cy, x, y, radius) | Return the indices of the segments in the polyline with coordinates (*cx*, *cy*) that are within a distance *radius* of the point (*x*, *y*). |

### matplotlib.lines.segment_hits

matplotlib.lines.**segment_hits** (*cx*, *cy*, *x*, *y*, *radius*)

Return the indices of the segments in the polyline with coordinates (*cx*, *cy*) that are within a distance *radius* of the point (*x*, *y*).

### 7.2.33 matplotlib.markers

Functions to handle markers; used by the marker functionality of *plot*, *scatter*, and *errorbar*.

All possible markers are defined here:

| marker | symbol | description |
|---|---|---|
| "." | | point |
| "," |  | pixel |
| "o" | | circle |
| "v" | | triangle_down |
| "^" | | triangle_up |
| "<" | | triangle_left |
| ">" | | triangle_right |
| "1" | | tri_down |
| "2" | | tri_up |
| "3" | | tri_left |
| "4" | | tri_right |
| "8" | | octagon |
| "s" | | square |
| "p" | | pentagon |
| "P" | | plus (filled) |
| "*" | | star |
| "h" | | hexagon1 |
| "H" | | hexagon2 |
| "+" | | plus |
| "x" | | x |
| "X" | | x (filled) |
| "D" | | diamond |
| "d" | | thin_diamond |
| "\|" | | vline |
| "_" |  | hline |
| 0 (TICKLEFT) |  | tickleft |
| 1 (TICKRIGHT) |  | tickright |
| 2 (TICKUP) |  | tickup |
| 3 (TICKDOWN) | | tickdown |
| 4 (CARETLEFT) |  | caretleft |

Table 67 – continued from previ

| marker | symbol | description |
|---|---|---|
| `5` (`CARETRIGHT`) | | caretright |
| `6` (`CARETUP`) | | caretup |
| `7` (`CARETDOWN`) | | caretdown |
| `8` (`CARETLEFTBASE`) | | caretleft (centered at base) |
| `9` (`CARETRIGHTBASE`) | | caretright (centered at base) |
| `10` (`CARETUPBASE`) | | caretup (centered at base) |
| `11` (`CARETDOWNBASE`) | | caretdown (centered at base) |
| `"none"` or `"None"` | | nothing |
| `" "` or `""` | | nothing |
| `"$...$"` | | Render the string using mathtext. E.g `"$f$"` for marker showing the letter `f`. |
| `verts` | | A list of (x, y) pairs used for Path vertices. The center of the marker is located a |
| `path` | | A *`Path`* instance. |
| `(numsides, 0, angle)` | | A regular polygon with `numsides` sides, rotated by `angle`. |
| `(numsides, 1, angle)` | | A star-like symbol with `numsides` sides, rotated by `angle`. |
| `(numsides, 2, angle)` | | An asterisk with `numsides` sides, rotated by `angle`. |

Note that special symbols can be defined via the *STIX math font*, e.g. `"$\u266B$"`. For an overview over the STIX font symbols refer to the STIX font table. Also see the *STIX Fonts*.

Integer numbers from `0` to `11` create lines and triangles. Those are equally accessible via capitalized variables, like `CARETDOWNBASE`. Hence the following are equivalent:

```
plt.plot([1, 2, 3], marker=11)
plt.plot([1, 2, 3], marker=matplotlib.markers.CARETDOWNBASE)
```

Markers join and cap styles can be customized by creating a new instance of MarkerStyle. A MarkerStyle can also have a custom *`Transform`* allowing it to be arbitrarily rotated or offset.

Examples showing the use of markers:

- *Marker reference*
- *Marker examples*
- *Mapping marker properties to multivariate data*

## Classes

| | |
|---|---|
| *MarkerStyle*(marker[, fillstyle, transform, ...]) | A class representing marker types. |

## matplotlib.markers.MarkerStyle

**class** matplotlib.markers.**MarkerStyle**(*marker*, *fillstyle=None*, *transform=None*, *capstyle=None*, *joinstyle=None*)

Bases: object

A class representing marker types.

Instances are immutable. If you need to change anything, create a new instance.

**Attributes**

**markers**

[dict] All known markers.

**filled_markers**

[tuple] All known filled markers. This is a subset of *markers*.

**fillstyles**

[tuple] The supported fillstyles.

**Parameters**

**marker**

[str, array-like, Path, MarkerStyle]

- Another instance of *MarkerStyle* copies the details of that *marker*.
- For other possible marker values, see the module docstring *matplotlib.markers*.

**fillstyle**

[str, default: *rcParams["markers.fillstyle"]* (default: 'full')] One of 'full', 'left', 'right', 'bottom', 'top', 'none'.

**transform**

[*Transform*, optional] Transform that will be combined with the native transform of the marker.

**capstyle**

[*CapStyle* or %(CapStyle)s, optional] Cap style that will override the default cap style of the marker.

**joinstyle**

[*JoinStyle* or %(JoinStyle)s, optional] Join style that will override the default join style of the marker.

**filled_markers = ('.', 'o', 'v', '^', '<', '>', '8', 's', 'p', '*', 'h', 'H', 'D', 'd', 'P', 'X')**

**fillstyles = ('full', 'left', 'right', 'bottom', 'top', 'none')**

**get_alt_path**()

Return a *Path* for the alternate part of the marker.

For unfilled markers, this is *None*; for filled markers, this is the area to be drawn with *markerfacecoloralt*.

**get_alt_transform**()

Return the transform to be applied to the *Path* from *MarkerStyle.get_alt_path()*.

**get_capstyle**()

**get_fillstyle**()

**get_joinstyle**()

**get_marker**()

**get_path**()

Return a *Path* for the primary part of the marker.

For unfilled markers this is the whole marker, for filled markers, this is the area to be drawn with *markerfacecolor*.

**get_snap_threshold**()

**get_transform**()

Return the transform to be applied to the *Path* from *MarkerStyle.get_path()*.

**get_user_transform**()

Return user supplied part of marker transform.

**is_filled**()

```
markers = {' ': 'nothing', '': 'nothing', '*': 'star', '+':
'plus', ',': 'pixel', '.': 'point', '1': 'tri_down', '2':
'tri_up', '3': 'tri_left', '4': 'tri_right', '8': 'octagon', '<':
'triangle_left', '>': 'triangle_right', 'D': 'diamond', 'H':
'hexagon2', 'None': 'nothing', 'P': 'plus_filled', 'X': 'x_filled',
'^': 'triangle_up', '_': 'hline', 'd': 'thin_diamond', 'h':
'hexagon1', 'none': 'nothing', 'o': 'circle', 'p': 'pentagon',
's': 'square', 'v': 'triangle_down', 'x': 'x', '|': 'vline', 0:
'tickleft', 1: 'tickright', 10: 'caretupbase', 11:
'caretdownbase', 2: 'tickup', 3: 'tickdown', 4: 'caretleft', 5:
'caretright', 6: 'caretup', 7: 'caretdown', 8: 'caretleftbase', 9:
'caretrightbase'}
```

**rotated**(*, *deg=None*, *rad=None*)

Return a new version of this marker rotated by specified angle.

**Parameters**

**deg**

[float, optional] Rotation angle in degrees.

**rad**

[float, optional] Rotation angle in radians.

**.. note:: You must specify exactly one of deg or rad.**

**scaled**(*sx*, *sy=None*)

Return new marker scaled by specified scale factors.

If *sy* is not given, the same scale is applied in both the *x*- and *y*-directions.

**Parameters**

**sx**

[float] *X*-direction scaling factor.

**sy**

[float, optional] *Y*-direction scaling factor.

**transformed**(*transform*)

Return a new version of this marker with the transform applied.

**Parameters**

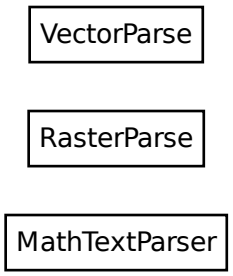
**transform**

[*Affine2D*] Transform will be combined with current user supplied transform.

### Examples using `matplotlib.markers.MarkerStyle`

- *Marker reference*
- *Mapping marker properties to multivariate data*
- *Ellipse with orientation arrow demo*

### 7.2.34 `matplotlib.mathtext`




A module for parsing a subset of the TeX math syntax and rendering it to a Matplotlib backend.

For a tutorial of its usage, see *Writing mathematical expressions*. This document is primarily concerned with implementation details.

The module uses pyparsing to parse the TeX expression.

The Bakoma distribution of the TeX Computer Modern fonts, and STIX fonts are supported. There is experimental support for using arbitrary fonts, but results may vary without proper tweaking and metrics for those fonts.

**class** matplotlib.mathtext.**MathTextParser**(*output*)

Bases: object

Create a MathTextParser for the given backend *output*.

**Parameters**

**output**

[{"path", "agg"}] Whether to return a *VectorParse* ("path") or a *RasterParse* ("agg", or its synonym "macosx").

**parse**(*s*, *dpi=72*, *prop=None*, ***, *antialiased=None*)

Parse the given math expression *s* at the given *dpi*. If *prop* is provided, it is a *FontProperties* object specifying the "default" font to use in the math expression, used for all non-math text.

The results are cached, so multiple calls to *parse* with the same expression should be fast.

Depending on the *output* type, this returns either a *VectorParse* or a *RasterParse*.

**class** matplotlib.mathtext.**RasterParse**(*ox*, *oy*, *width*, *height*, *depth*, *image*)

Bases: NamedTuple

The namedtuple type returned by `MathTextParser("agg").parse(...)`.

**Attributes**

**ox, oy**

[float] The offsets are always zero.

**width, height, depth**

[float] The global metrics.

**image**

[FT2Image] A raster image.

Create new instance of RasterParse(ox, oy, width, height, depth, image)

**depth**

Alias for field number 4

**height**

Alias for field number 3

**image**

Alias for field number 5

**ox**

Alias for field number 0

**oy**

Alias for field number 1

**width**

Alias for field number 2

**class** matplotlib.mathtext.**VectorParse**(*width*, *height*, *depth*, *glyphs*, *rects*)

Bases: NamedTuple

The namedtuple type returned by `MathTextParser("path").parse(...)`.

**Attributes**

**width, height, depth**

[float] The global metrics.

**glyphs**

[list] The glyphs including their positions.

**rect**

[list] The list of rectangles.

Create new instance of VectorParse(width, height, depth, glyphs, rects)

**depth**
Alias for field number 2

**glyphs**
Alias for field number 3

**height**
Alias for field number 1

**rects**
Alias for field number 4

**width**
Alias for field number 0

matplotlib.mathtext.**get_unicode_index**(*symbol*)

Return the integer index (from the Unicode table) of *symbol*.

**Parameters**

**symbol**

[str] A single (Unicode) character, a TeX command (e.g. r'pi') or a Type1 symbol name (e.g. 'phi').

matplotlib.mathtext.**math_to_image**(*s*, *filename_or_obj*, *prop=None*, *dpi=None*, *format=None*, *, *color=None*)

Given a math expression, renders it in a closely-clipped bounding box to an image file.

**Parameters**

**s**

[str] A math expression. The math portion must be enclosed in dollar signs.

**filename_or_obj**

[str or path-like or file-like] Where to write the image data.

**prop**

[*FontProperties*, optional] The size and style of the text.

**dpi**

[float, optional] The output dpi. If not set, the dpi is determined as for *Figure.savefig*.

**format**

[str, optional] The output format, e.g., 'svg', 'pdf', 'ps' or 'png'. If not set, the format is determined as for *Figure.savefig*.

**color**

[str, optional] Foreground color, defaults to *rcParams["text.color"]* (default: `'black'`).

### 7.2.35 matplotlib.mlab

Numerical Python functions written for compatibility with MATLAB commands with the same names. Most numerical Python functions can be found in the NumPy and SciPy libraries. What remains here is code for performing spectral computations and kernel density estimations.

#### Spectral functions

***cohere***

Coherence (normalized cross spectral density)

***csd***

Cross spectral density using Welch's average periodogram

***detrend***

Remove the mean or best fit line from an array

***psd***

Power spectral density using Welch's average periodogram

***specgram***

Spectrogram (spectrum over segments of time)

***complex_spectrum***

Return the complex-valued frequency spectrum of a signal

***magnitude_spectrum***

Return the magnitude of the frequency spectrum of a signal

***angle_spectrum***

Return the angle (wrapped phase) of the frequency spectrum of a signal

***phase_spectrum***

Return the phase (unwrapped angle) of the frequency spectrum of a signal

***detrend_mean***

Remove the mean from a line.

***detrend_linear***

Remove the best fit line from a line.

***detrend_none***

Return the original line.

**class** matplotlib.mlab.**GaussianKDE**(*dataset*, *bw_method=None*)

Bases: object

Representation of a kernel-density estimate using Gaussian kernels.

**Parameters**

**dataset**

[array-like] Datapoints to estimate from. In case of univariate data this is a 1-D array, otherwise a 2D array with shape (# of dims, # of data).

**bw_method**

[{'scott', 'silverman'} or float or callable, optional] The method used to calculate the estimator bandwidth. If a float, this will be used directly as `kde.factor`. If a callable, it should take a *GaussianKDE* instance as only parameter and return a float. If None (default), 'scott' is used.

**Attributes**

**dataset**

[ndarray] The dataset passed to the constructor.

**dim**

[int] Number of dimensions.

**num_dp**

[int] Number of datapoints.

**factor**

[float] The bandwidth factor, obtained from `kde.covariance_factor`, with which the covariance matrix is multiplied.

**covariance**

[ndarray] The covariance matrix of *dataset*, scaled by the calculated bandwidth (`kde.factor`).

**inv_cov**

[ndarray] The inverse of *covariance*.

### Methods

| | |
|---|---|
| **kde.evaluate(points)** | (ndarray) Evaluate the estimated pdf on a provided set of points. |
| **kde(points)** | (ndarray) Same as kde.evaluate(points) |

**covariance_factor**()

**evaluate**(*points*)

Evaluate the estimated pdf on a set of points.

**Parameters**

**points**

[(# of dimensions, # of points)-array] Alternatively, a (# of dimensions,) vector can be passed in and treated as a single point.

**Returns**

**(# of points,)-array**

The values at each point.

**Raises**

**ValueError**

[if the dimensionality of the input points is different] than the dimensionality of the KDE.

**scotts_factor**()

**silverman_factor**()

matplotlib.mlab.**angle_spectrum**(*x*, *Fs=None*, *window=None*, *pad_to=None*, *sides=None*)

Compute the angle of the frequency spectrum (wrapped phase spectrum) of *x*. Data is padded to a length of *pad_to* and the windowing function *window* is applied to the signal.

**Parameters**

**x**

[1-D array or sequence] Array or sequence containing the data

**Fs**

[float, default: 2] The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, *freqs*, in cycles per time unit.

**window**

[callable or ndarray, default: *window_hanning*] A function or a vector of length *NFFT*. To create window vectors see *window_hanning*, *window_none*, numpy.blackman, numpy.hamming, numpy.bartlett, scipy.signal, scipy.signal.get_window, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

**sides**

[{'default', 'onesided', 'twosided'}, optional] Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

**pad_to**

[int, optional] The number of points to which the data segment is padded when performing the FFT. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to fft. The default is None, which sets *pad_to* equal to the length of the input signal (i.e. no padding).

**Returns**

**spectrum**

[1-D array] The angle of the frequency spectrum (wrapped phase spectrum).

**freqs**

[1-D array] The frequencies corresponding to the elements in *spectrum*.

**See also:**

***psd***

Returns the power spectral density.

***complex_spectrum***

Returns the complex-valued frequency spectrum.

***magnitude_spectrum***

Returns the absolute value of the *complex_spectrum*.

***angle_spectrum***

Returns the angle of the *complex_spectrum*.

***phase_spectrum***

Returns the phase (unwrapped angle) of the *complex_spectrum*.

*specgram*

Can return the complex spectrum of segments within the signal.

matplotlib.mlab.**cohere**(*x*, *y*, *NFFT=256*, *Fs=2*, *detrend=<function detrend_none>*, *window=<function window_hanning>*, *noverlap=0*, *pad_to=None*, *sides='default'*, *scale_by_freq=None*)

The coherence between $x$ and $y$. Coherence is the normalized cross spectral density:

$$C_{xy} = \frac{|P_{xy}|^2}{P_{xx}P_{yy}}$$

**Parameters**

**x, y**

Array or sequence containing the data

**Fs**

[float, default: 2] The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, *freqs*, in cycles per time unit.

**window**

[callable or ndarray, default: *window_hanning*] A function or a vector of length *NFFT*. To create window vectors see *window_hanning*, *window_none*, numpy.blackman, numpy.hamming, numpy.bartlett, scipy.signal, scipy.signal.get_window, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

**sides**

[{'default', 'onesided', 'twosided'}, optional] Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

**pad_to**

[int, optional] The number of points to which the data segment is padded when performing the FFT. This can be different from *NFFT*, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to fft. The default is None, which sets *pad_to* equal to *NFFT*

**NFFT**

[int, default: 256] The number of data points used in each block for the FFT. A power 2 is most efficient. This should *NOT* be used to get zero padding, or the scaling of the result will be incorrect; use *pad_to* for this instead.

**detrend**

[{'none', 'mean', 'linear'} or callable, default: 'none'] The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the *detrend* parameter is a vector, in Matplotlib it is a function. The *mlab* module defines *detrend_none*, *detrend_mean*, and *detrend_linear*, but you can use a custom function as well. You can also use a string to choose one of the functions: 'none' calls *detrend_none*. 'mean' calls *detrend_mean*. 'linear' calls *detrend_linear*.

**scale_by_freq**

[bool, default: True] Whether the resulting density values should be scaled by the scaling frequency, which gives density in units of 1/Hz. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

**noverlap**

[int, default: 0 (no overlap)] The number of points of overlap between segments.

**Returns**

**Cxy**

[1-D array] The coherence vector.

**freqs**

[1-D array] The frequencies for the elements in *Cxy*.

**See also:**

***psd()*, *csd()***

For information about the methods used to compute $P_{xy}$, $P_{xx}$ and $P_{yy}$.

matplotlib.mlab.**complex_spectrum**(*x*, *Fs=None*, *window=None*, *pad_to=None*, *sides=None*)

Compute the complex-valued frequency spectrum of *x*. Data is padded to a length of *pad_to* and the windowing function *window* is applied to the signal.

**Parameters**

**x**

[1-D array or sequence] Array or sequence containing the data

**Fs**

[float, default: 2] The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, *freqs*, in cycles per time unit.

**window**

[callable or ndarray, default: *window_hanning*] A function or a vector of length *NFFT*. To create window vectors see *window_hanning*, *window_none*, numpy.blackman, numpy.hamming, numpy.bartlett, scipy.signal, scipy.signal.get_window, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

**sides**

[{'default', 'onesided', 'twosided'}, optional] Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

**pad_to**

[int, optional] The number of points to which the data segment is padded when performing the FFT. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to fft. The default is None, which sets *pad_to* equal to the length of the input signal (i.e. no padding).

**Returns**

**spectrum**

[1-D array] The complex-valued frequency spectrum.

**freqs**

[1-D array] The frequencies corresponding to the elements in *spectrum*.

**See also:**

***psd***

Returns the power spectral density.

***complex_spectrum***

Returns the complex-valued frequency spectrum.

***magnitude_spectrum***

Returns the absolute value of the *complex_spectrum*.

***angle_spectrum***

Returns the angle of the *complex_spectrum*.

***phase_spectrum***

Returns the phase (unwrapped angle) of the *complex_spectrum*.

***specgram***

Can return the complex spectrum of segments within the signal.

matplotlib.mlab.**csd**(*x*, *y*, *NFFT=None*, *Fs=None*, *detrend=None*, *window=None*, *noverlap=None*, *pad_to=None*, *sides=None*, *scale_by_freq=None*)

Compute the cross-spectral density.

The cross spectral density $P_{xy}$ by Welch's average periodogram method. The vectors $x$ and $y$ are divided into *NFFT* length segments. Each segment is detrended by function *detrend* and windowed by function *window*. *noverlap* gives the length of the overlap between segments. The product of the direct FFTs of $x$ and $y$ are averaged over each segment to compute $P_{xy}$, with a scaling to correct for power loss due to windowing.

If len($x$) < *NFFT* or len($y$) < *NFFT*, they will be zero padded to *NFFT*.

**Parameters**

**x, y**

[1-D arrays or sequences] Arrays or sequences containing the data

**Fs**

[float, default: 2] The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, *freqs*, in cycles per time unit.

**window**

[callable or ndarray, default: *window_hanning*] A function or a vector of length *NFFT*. To create window vectors see *window_hanning*, *window_none*, numpy.blackman, numpy.hamming, numpy.bartlett, scipy.signal, scipy.signal.get_window, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

**sides**

[{'default', 'onesided', 'twosided'}, optional] Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

**pad_to**

[int, optional] The number of points to which the data segment is padded when performing the FFT. This can be different from *NFFT*, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to fft. The default is None, which sets *pad_to* equal to *NFFT*

**NFFT**

[int, default: 256] The number of data points used in each block for the FFT. A power 2 is most efficient. This should *NOT* be used to get zero padding, or the scaling of the result will be incorrect; use *pad_to* for this instead.

**detrend**

[{'none', 'mean', 'linear'} or callable, default: 'none'] The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the *detrend* parameter is a vector, in Matplotlib it is a function. The *mlab* module defines *detrend_none*, *detrend_mean*, and *detrend_linear*, but you can use a custom function as well. You can also use a string to choose one of the functions: 'none' calls *detrend_none*. 'mean' calls *detrend_mean*. 'linear' calls *detrend_linear*.

**scale_by_freq**

[bool, default: True] Whether the resulting density values should be scaled by the scaling frequency, which gives density in units of 1/Hz. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

**noverlap**

[int, default: 0 (no overlap)] The number of points of overlap between segments.

**Returns**

**Pxy**

[1-D array] The values for the cross spectrum $P_{xy}$ before scaling (real valued)

**freqs**

[1-D array] The frequencies corresponding to the elements in *Pxy*

**See also:**

***psd***

equivalent to setting `y = x`.

## References

Bendat & Piersol -- Random Data: Analysis and Measurement Procedures, John Wiley & Sons (1986)

`matplotlib.mlab.`**`detrend`**(*x*, *key=None*, *axis=None*)

Return *x* with its trend removed.

**Parameters**

**x**

[array or sequence] Array or sequence containing the data.

**key**

[{'default', 'constant', 'mean', 'linear', 'none'} or function] The detrending algorithm to use. 'default', 'mean', and 'constant' are the same as *detrend_mean*. 'linear' is the same as *detrend_linear*. 'none' is the same as *detrend_none*. The

default is 'mean'. See the corresponding functions for more details regarding the algorithms. Can also be a function that carries out the detrend operation.

**axis**

[int] The axis along which to do the detrending.

**See also:**

***detrend_mean***

Implementation of the 'mean' algorithm.

***detrend_linear***

Implementation of the 'linear' algorithm.

***detrend_none***

Implementation of the 'none' algorithm.

matplotlib.mlab.**detrend_linear**(*y*)

Return *x* minus best fit line; 'linear' detrending.

**Parameters**

**y**

[0-D or 1-D array or sequence] Array or sequence containing the data

**See also:**

***detrend_mean***

Another detrend algorithm.

***detrend_none***

Another detrend algorithm.

***detrend***

A wrapper around all the detrend algorithms.

matplotlib.mlab.**detrend_mean**(*x*, *axis=None*)

Return *x* minus the mean(*x*).

**Parameters**

**x**

[array or sequence] Array or sequence containing the data Can have any dimensionality

**axis**

[int] The axis along which to take the mean. See `numpy.mean` for a description of this argument.

**See also:**

***detrend_linear***

Another detrend algorithm.

***detrend_none***

Another detrend algorithm.

***detrend***

A wrapper around all the detrend algorithms.

matplotlib.mlab.**detrend_none**(*x*, *axis=None*)

Return *x*: no detrending.

**Parameters**

**x**

[any object] An object containing the data

**axis**

[int] This parameter is ignored. It is included for compatibility with detrend_mean

**See also:**

***detrend_mean***

Another detrend algorithm.

***detrend_linear***

Another detrend algorithm.

***detrend***

A wrapper around all the detrend algorithms.

matplotlib.mlab.**magnitude_spectrum**(*x*, *Fs=None*, *window=None*, *pad_to=None*, *sides=None*)

Compute the magnitude (absolute value) of the frequency spectrum of *x*. Data is padded to a length of *pad_to* and the windowing function *window* is applied to the signal.

**Parameters**

**x**

[1-D array or sequence] Array or sequence containing the data

**Fs**

[float, default: 2] The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, *freqs*, in cycles per time unit.

**window**

[callable or ndarray, default: *window_hanning*] A function or a vector of length *NFFT*. To create window vectors see *window_hanning*, *window_none*, numpy.blackman, numpy.hamming, numpy.bartlett, scipy.signal, scipy.signal.get_window, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

**sides**

[{'default', 'onesided', 'twosided'}, optional] Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

**pad_to**

[int, optional] The number of points to which the data segment is padded when performing the FFT. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to fft. The default is None, which sets *pad_to* equal to the length of the input signal (i.e. no padding).

**Returns**

**spectrum**

[1-D array] The magnitude (absolute value) of the frequency spectrum.

**freqs**

[1-D array] The frequencies corresponding to the elements in *spectrum*.

**See also:**

***psd***

Returns the power spectral density.

***complex_spectrum***

Returns the complex-valued frequency spectrum.

***magnitude_spectrum***

Returns the absolute value of the *complex_spectrum*.

***angle_spectrum***

Returns the angle of the *complex_spectrum*.

***phase_spectrum***

Returns the phase (unwrapped angle) of the *complex_spectrum*.

***specgram***

Can return the complex spectrum of segments within the signal.

matplotlib.mlab.**phase_spectrum**(*x*, *Fs=None*, *window=None*, *pad_to=None*, *sides=None*)

Compute the phase of the frequency spectrum (unwrapped phase spectrum) of *x*. Data is padded to a length of *pad_to* and the windowing function *window* is applied to the signal.

**Parameters**

**x**

[1-D array or sequence] Array or sequence containing the data

**Fs**

[float, default: 2] The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, *freqs*, in cycles per time unit.

**window**

[callable or ndarray, default: *window_hanning*] A function or a vector of length *NFFT*. To create window vectors see *window_hanning*, *window_none*, numpy.blackman, numpy.hamming, numpy.bartlett, scipy.signal, scipy.signal.get_window, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

**sides**

[{'default', 'onesided', 'twosided'}, optional] Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

**pad_to**

[int, optional] The number of points to which the data segment is padded when performing the FFT. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to fft. The default is None, which sets *pad_to* equal to the length of the input signal (i.e. no padding).

**Returns**

**spectrum**

[1-D array] The phase of the frequency spectrum (unwrapped phase spectrum).

**freqs**

[1-D array] The frequencies corresponding to the elements in *spectrum*.

**See also:**

***psd***

Returns the power spectral density.

***complex_spectrum***

Returns the complex-valued frequency spectrum.

***magnitude_spectrum***

Returns the absolute value of the *complex_spectrum*.

***angle_spectrum***

Returns the angle of the *complex_spectrum*.

***phase_spectrum***

Returns the phase (unwrapped angle) of the *complex_spectrum*.

***specgram***

Can return the complex spectrum of segments within the signal.

matplotlib.mlab.**psd**(*x*, *NFFT=None*, *Fs=None*, *detrend=None*, *window=None*, *noverlap=None*, *pad_to=None*, *sides=None*, *scale_by_freq=None*)

Compute the power spectral density.

The power spectral density $P_{xx}$ by Welch's average periodogram method. The vector $x$ is divided into *NFFT* length segments. Each segment is detrended by function *detrend* and windowed by function *window*. *noverlap* gives the length of the overlap between segments. The $|\mathrm{fft}(i)|^2$ of each segment $i$ are averaged to compute $P_{xx}$.

If len($x$) < *NFFT*, it will be zero padded to *NFFT*.

**Parameters**

**x**

[1-D array or sequence] Array or sequence containing the data

**Fs**

[float, default: 2] The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, *freqs*, in cycles per time unit.

**window**

[callable or ndarray, default: *window_hanning*] A function or a vector of length *NFFT*. To create window vectors see *window_hanning*, *window_none*, numpy.blackman, numpy.hamming, numpy.bartlett, scipy.signal, scipy.signal.get_window, etc. If a function is passed

as the argument, it must take a data segment as an argument and return the windowed version of the segment.

**sides**

[{'default', 'onesided', 'twosided'}, optional] Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

**pad_to**

[int, optional] The number of points to which the data segment is padded when performing the FFT. This can be different from *NFFT*, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to `fft`. The default is None, which sets *pad_to* equal to *NFFT*

**NFFT**

[int, default: 256] The number of data points used in each block for the FFT. A power 2 is most efficient. This should *NOT* be used to get zero padding, or the scaling of the result will be incorrect; use *pad_to* for this instead.

**detrend**

[{'none', 'mean', 'linear'} or callable, default: 'none'] The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the *detrend* parameter is a vector, in Matplotlib it is a function. The `mlab` module defines `detrend_none`, `detrend_mean`, and `detrend_linear`, but you can use a custom function as well. You can also use a string to choose one of the functions: 'none' calls `detrend_none`. 'mean' calls `detrend_mean`. 'linear' calls `detrend_linear`.

**scale_by_freq**

[bool, default: True] Whether the resulting density values should be scaled by the scaling frequency, which gives density in units of 1/Hz. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

**noverlap**

[int, default: 0 (no overlap)] The number of points of overlap between segments.

**Returns**

**Pxx**

[1-D array] The values for the power spectrum $P_{xx}$ (real valued)

**freqs**

[1-D array] The frequencies corresponding to the elements in *Pxx*

**See also:**

***specgram***

*specgram* differs in the default overlap; in not returning the mean of the segment periodograms; and in returning the times of the segments.

***magnitude_spectrum***

returns the magnitude spectrum.

***csd***

returns the spectral density between two signals.

## References

Bendat & Piersol -- Random Data: Analysis and Measurement Procedures, John Wiley & Sons (1986)

matplotlib.mlab.**specgram**(*x*, *NFFT=None*, *Fs=None*, *detrend=None*, *window=None*, *noverlap=None*, *pad_to=None*, *sides=None*, *scale_by_freq=None*, *mode=None*)

Compute a spectrogram.

Compute and plot a spectrogram of data in *x*. Data are split into *NFFT* length segments and the spectrum of each section is computed. The windowing function *window* is applied to each segment, and the amount of overlap of each segment is specified with *noverlap*.

**Parameters**

**x**

[array-like] 1-D array or sequence.

**Fs**

[float, default: 2] The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, *freqs*, in cycles per time unit.

**window**

[callable or ndarray, default: *window_hanning*] A function or a vector of length *NFFT*. To create window vectors see *window_hanning*, *window_none*, numpy.blackman, numpy.hamming, numpy.bartlett, scipy.signal, scipy.signal.get_window, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

**sides**

[{'default', 'onesided', 'twosided'}, optional] Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

**pad_to**

[int, optional] The number of points to which the data segment is padded when performing the FFT. This can be different from *NFFT*, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to `fft`. The default is None, which sets *pad_to* equal to *NFFT*

**NFFT**

[int, default: 256] The number of data points used in each block for the FFT. A power 2 is most efficient. This should *NOT* be used to get zero padding, or the scaling of the result will be incorrect; use *pad_to* for this instead.

**detrend**

[{'none', 'mean', 'linear'} or callable, default: 'none'] The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the *detrend* parameter is a vector, in Matplotlib it is a function. The `mlab` module defines `detrend_none`, `detrend_mean`, and `detrend_linear`, but you can use a custom function as well. You can also use a string to choose one of the functions: 'none' calls `detrend_none`. 'mean' calls `detrend_mean`. 'linear' calls `detrend_linear`.

**scale_by_freq**

[bool, default: True] Whether the resulting density values should be scaled by the scaling frequency, which gives density in units of 1/Hz. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

**noverlap**

[int, default: 128] The number of points of overlap between blocks.

**mode**

[str, default: 'psd']

**What sort of spectrum to use:**

**'psd'**

Returns the power spectral density.

**'complex'**

Returns the complex-valued frequency spectrum.

**'magnitude'**

Returns the magnitude spectrum.

**'angle'**

Returns the phase spectrum without unwrapping.

**'phase'**

Returns the phase spectrum with unwrapping.

**Returns**

**spectrum**

[array-like] 2D array, columns are the periodograms of successive segments.

**freqs**

[array-like] 1-D array, frequencies corresponding to the rows in *spectrum*.

**t**

[array-like] 1-D array, the times corresponding to midpoints of segments (i.e the columns in *spectrum*).

**See also:**

***psd***

differs in the overlap and in the return values.

***complex_spectrum***

similar, but with complex valued frequencies.

***magnitude_spectrum***

similar single segment when *mode* is 'magnitude'.

***angle_spectrum***

similar to single segment when *mode* is 'angle'.

***phase_spectrum***

similar to single segment when *mode* is 'phase'.

### Notes

*detrend* and *scale_by_freq* only apply when *mode* is set to 'psd'.

`matplotlib.mlab.`**`window_hanning`**`(x)`

Return *x* times the Hanning (or Hann) window of len(*x*).

**See also:**

***window_none***

Another window algorithm.

matplotlib.mlab.**window_none**(*x*)

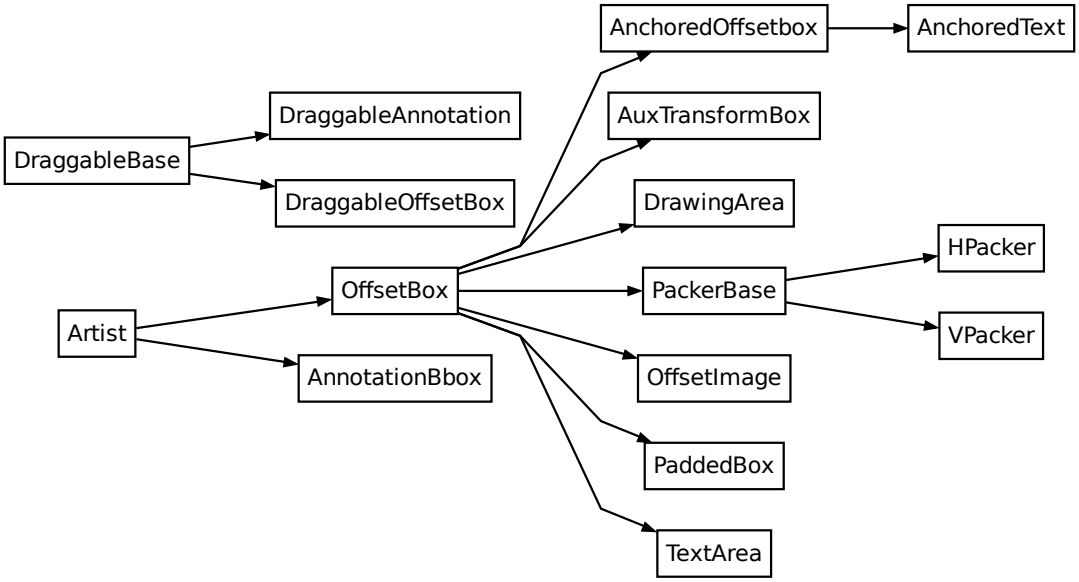
No window function; simply return *x*.

**See also:**

***window_hanning***

Another window algorithm.

### 7.2.36 matplotlib.offsetbox




Container classes for *Artist*s.

***OffsetBox***

The base of all container artists defined in this module.

***AnchoredOffsetbox***, ***AnchoredText***

Anchor and align an arbitrary *Artist* or a text relative to the parent axes or a specific anchor point.

***DrawingArea***

A container with fixed width and height. Children have a fixed position inside the container and may be clipped.

***HPacker***, ***VPacker***

Containers for layouting their children vertically or horizontally.

***PaddedBox***

A container to add a padding around an *Artist*.

***TextArea***

Contains a single *Text* instance.

**class** matplotlib.offsetbox.**AnchoredOffsetbox**(*loc*, *, *pad=0.4*, *borderpad=0.5*, *child=None*, *prop=None*, *frameon=True*, *bbox_to_anchor=None*, *bbox_transform=None*, ***kwargs*)

Bases: *OffsetBox*

An offset box placed according to location *loc*.

AnchoredOffsetbox has a single child. When multiple children are needed, use an extra OffsetBox to enclose them. By default, the offset box is anchored against its parent Axes. You may explicitly specify the *bbox_to_anchor*.

**Parameters**

**loc**

[str] The box location. Valid locations are 'upper left', 'upper center', 'upper right', 'center left', 'center', 'center right', 'lower left', 'lower center', 'lower right'. For backward compatibility, numeric values are accepted as well. See the parameter *loc* of *Legend* for details.

**pad**

[float, default: 0.4] Padding around the child as fraction of the fontsize.

**borderpad**

[float, default: 0.5] Padding between the offsetbox frame and the *bbox_to_anchor*.

**child**

[*OffsetBox*] The box that will be anchored.

**prop**

[*FontProperties*] This is only used as a reference for paddings. If not given, *rcParams["legend.fontsize"]* (default: 'medium') is used.

**frameon**

[bool] Whether to draw a frame around the box.

**bbox_to_anchor**

[*BboxBase*, 2-tuple, or 4-tuple of floats] Box that is used to position the legend in conjunction with *loc*.

**bbox_transform**

[None or *matplotlib.transforms.Transform*] The transform for the bounding box (*bbox_to_anchor*).

****kwargs**

All other parameters are passed on to *OffsetBox*.

## Notes

See *Legend* for a detailed description of the anchoring mechanism.

```
codes = {'center': 10, 'center left': 6, 'center right': 7, 'lower
center': 8, 'lower left': 3, 'lower right': 4, 'right': 5, 'upper
center': 9, 'upper left': 2, 'upper right': 1}
```

**draw**(*renderer*)

Update the location of children if necessary and draw them to the given *renderer*.

**get_bbox**(*renderer*)

Return the bbox of the offsetbox, ignoring parent offsets.

**get_bbox_to_anchor**()

Return the bbox that the box is anchored to.

**get_child**()

Return the child.

**get_children**()

Return the list of children.

**get_offset**(*bbox*, *renderer*)

Return the offset as a tuple (x, y).

The extent parameters have to be provided to handle the case where the offset is dynamically determined by a callable (see *set_offset*).

**Parameters**

**bbox**

[*Bbox*]

**renderer**

[*RendererBase* subclass]

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *bbox_to_anchor=<UNSET>*, *child=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *gid=<UNSET>*, *height=<UNSET>*, *in_layout=<UNSET>*, *label=<UNSET>*, *mouseover=<UNSET>*, *offset=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *width=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *agg_filt* | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image |
| *alpha* | scalar or None |
| *animated* | bool |
| *bbox_to_* | unknown |
| *child* | unknown |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_pat* | Patch or (Path, Transform) or None |
| *figure* | *Figure* |
| *gid* | str |
| *height* | float |
| *in_layou* | bool |
| *label* | object |
| *mouseove* | bool |
| *offset* | (float, float) or callable |
| *path_eff* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *rasterized* | bool |
| *sketch_p* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *width* | float |
| *zorder* | float |

**set_bbox_to_anchor**(*bbox*, *transform=None*)

Set the bbox that the box is anchored to.

*bbox* can be a Bbox instance, a list of [left, bottom, width, height], or a list of [left, bottom] where the width and height will be assumed to be zero. The bbox will be transformed to display coordinate by the given transform.

**set_child**(*child*)

Set the child to be anchored.

**update_frame**(*bbox*, *fontsize=None*)

**zorder = 5**

**class** matplotlib.offsetbox.**AnchoredText**(*s*, *loc*, *, *pad=0.4*, *borderpad=0.5*, *prop=None*, ***kwargs*)

Bases: *AnchoredOffsetbox*

AnchoredOffsetbox with Text.

**Parameters**

**s**

[str] Text.

**loc**

[str] Location code. See *AnchoredOffsetbox*.

**pad**

[float, default: 0.4] Padding around the text as fraction of the fontsize.

**borderpad**

[float, default: 0.5] Spacing between the offsetbox frame and the *bbox_to_anchor*.

**prop**

[dict, optional] Dictionary of keyword parameters to be passed to the *Text* instance contained inside AnchoredText.

****kwargs**

All other parameters are passed to *AnchoredOffsetbox*.

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *bbox_to_anchor=<UNSET>*, *child=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *gid=<UNSET>*, *height=<UNSET>*, *in_layout=<UNSET>*, *label=<UNSET>*, *mouseover=<UNSET>*, *offset=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *width=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| agg_filt | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image |
| alpha | scalar or None |
| animated | bool |
| bbox_to_ | unknown |
| child | unknown |
| clip_box | BboxBase or None |
| clip_on | bool |
| clip_pat | Patch or (Path, Transform) or None |
| figure | Figure |
| gid | str |
| height | float |
| in_layou | bool |
| label | object |
| mouseove | bool |
| offset | (float, float) or callable |
| path_eff | list of AbstractPathEffect |
| picker | None or bool or float or callable |
| rasterized | bool |
| sketch_p | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| transform | Transform |
| url | str |
| visible | bool |
| width | float |
| zorder | float |

**class** matplotlib.offsetbox.**AnnotationBbox**(*offsetbox*, *xy*, *xybox=None*, *xycoords='data'*, *boxcoords=None*, ***, *frameon=True*, *pad=0.4*, *annotation_clip=None*, *box_alignment=(0.5, 0.5)*, *bboxprops=None*, *arrowprops=None*, *fontsize=None*, ***kwargs*)

Bases: Artist, _AnnotationBase

Container for an OffsetBox referring to a specific position *xy*.

Optionally an arrow pointing from the offsetbox to *xy* can be drawn.

This is like *Annotation*, but with *OffsetBox* instead of *Text*.

**Parameters**

**offsetbox**

[*OffsetBox*]

**xy**

[(float, float)] The point *(x, y)* to annotate. The coordinate system is determined by *xycoords*.

**xybox**

[(float, float), default: *xy*] The position *(x, y)* to place the text at. The coordinate system is determined by *boxcoords*.

**xycoords**

[single or two-tuple of str or *Artist* or *Transform* or callable, default: 'data'] The coordinate system that *xy* is given in. See the parameter *xycoords* in *Annotation* for a detailed description.

**boxcoords**

[single or two-tuple of str or *Artist* or *Transform* or callable, default: value of *xycoords*] The coordinate system that *xybox* is given in. See the parameter *textcoords* in *Annotation* for a detailed description.

**frameon**

[bool, default: True] By default, the text is surrounded by a white *FancyBboxPatch* (accessible as the `patch` attribute of the *AnnotationBbox*). If *frameon* is set to False, this patch is made invisible.

**annotation_clip: bool or None, default: None**

Whether to clip (i.e. not draw) the annotation when the annotation point *xy* is outside the Axes area.

- If *True*, the annotation will be clipped when *xy* is outside the Axes.
- If *False*, the annotation will always be drawn.
- If *None*, the annotation will be clipped when *xy* is outside the Axes and *xycoords* is 'data'.

**pad**

[float, default: 0.4] Padding around the offsetbox.

**box_alignment**

[(float, float)] A tuple of two floats for a vertical and horizontal alignment of the offset box w.r.t. the *boxcoords*. The lower-left corner is (0, 0) and upper-right corner is (1, 1).

**bboxprops**

[dict, optional] A dictionary of properties to set for the annotation bounding box, for example *boxstyle* and *alpha*. See *FancyBboxPatch* for details.

**arrowprops: dict, optional**

Arrow properties, see *Annotation* for description.

**fontsize: float or str, optional**

Translated to points and passed as *mutation_scale* into *FancyBboxPatch* to scale attributes of the box style (e.g. pad or rounding_size). The name is chosen in analogy to *Text* where *fontsize* defines the mutation scale as well. If not given, *rcParams["legend.fontsize"]* (default: 'medium') is used. See *Text.set_fontsize* for valid values.

****kwargs**

Other *AnnotationBbox* properties. See *AnnotationBbox.set* for a list.

**property anncoords**

**contains**(*mouseevent*)

Test whether the artist contains the mouse event.

**Parameters**

**mouseevent**

[*MouseEvent*]

**Returns**

**contains**

[bool] Whether any values are within the radius.

**details**

[dict] An artist-specific dictionary of details of the event context, such as which points are contained in the pick radius. See the individual Artist subclasses for details.

**draw**(*renderer*)

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

**Parameters**

**renderer**

[*RendererBase* subclass.]

### Notes

This method is overridden in the Artist subclasses.

**get_children**()

Return a list of the child *Artist*s of this *Artist*.

**get_fontsize**()

Return the fontsize in points.

**get_tightbbox**(*renderer=None*)

Like *Artist.get_window_extent*, but includes any clipping.

**Parameters**

**renderer**

[*RendererBase* subclass, optional] renderer that will be used to draw the figures (i.e. `fig.canvas.get_renderer()`)

**Returns**

***Bbox* or None**

The enclosing bounding box (in figure pixel coordinates). Returns None if clipping results in no intersection.

**get_window_extent**(*renderer=None*)

Get the artist's bounding box in display space.

The bounding box' width and height are nonnegative.

Subclasses should override for inclusion in the bounding box "tight" calculation. Default is to return an empty bounding box at 0, 0.

Be careful when using this function, the results will not update if the artist window extent of the artist changes. The extent can change due to any changes in the transform stack, such as changing the Axes limits, the figure size, or the canvas used (as is done when saving a figure). This can lead to unexpected behavior where interactive figures will look fine on the screen, but will save incorrectly.

**set**(**, agg_filter=<UNSET>, alpha=<UNSET>, animated=<UNSET>, annotation_clip=<UNSET>, clip_box=<UNSET>, clip_on=<UNSET>, clip_path=<UNSET>, fontsize=<UNSET>, gid=<UNSET>, in_layout=<UNSET>, label=<UNSET>, mouseover=<UNSET>, path_effects=<UNSET>, picker=<UNSET>, rasterized=<UNSET>, sketch_params=<UNSET>, snap=<UNSET>, transform=<UNSET>, url=<UNSET>, visible=<UNSET>, zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| agg_filt | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image |
| alpha | scalar or None |
| animated | bool |
| annotation_cli | bool or None |
| clip_box | BboxBase or None |
| clip_on | bool |
| clip_path | Patch or (Path, Transform) or None |
| figure | unknown |
| fontsize | unknown |
| gid | str |
| in_layout | bool |
| label | object |
| mouseover | bool |
| path_effects | list of AbstractPathEffect |
| picker | None or bool or float or callable |
| rasterized | bool |
| sketch_params | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| transform | Transform |
| url | str |
| visible | bool |
| zorder | float |

**set_figure**(*fig*)

Set the *Figure* instance the artist belongs to.

**Parameters**

**fig**

[*Figure*]

**set_fontsize**(*s=None*)

Set the fontsize in points.

If *s* is not given, reset to *rcParams["legend.fontsize"]* (default: 'medium').

**update_positions**(*renderer*)

Update pixel positions for the annotated point, the text, and the arrow.

**property xyann**

**zorder = 3**

**class** matplotlib.offsetbox.**AuxTransformBox**(*aux_transform*)

Bases: *OffsetBox*

Offset Box with the aux_transform. Its children will be transformed with the aux_transform first then will be offsetted. The absolute coordinate of the aux_transform is meaning as it will be automatically adjust so that the left-lower corner of the bounding box of children will be set to (0, 0) before the offset transform.

It is similar to drawing area, except that the extent of the box is not predetermined but calculated from the window extent of its children. Furthermore, the extent of the children will be calculated in the transformed coordinate.

**add_artist**(*a*)

Add an *Artist* to the container box.

**draw**(*renderer*)

Update the location of children if necessary and draw them to the given *renderer*.

**get_bbox**(*renderer*)

Return the bbox of the offsetbox, ignoring parent offsets.

**get_offset**()

Return offset of the container.

**get_transform**()

Return the *Transform* applied to the children

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *gid=<UNSET>*, *height=<UNSET>*, *in_layout=<UNSET>*, *label=<UNSET>*, *mouseover=<UNSET>*, *offset=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *width=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| agg_fil | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image |
| alpha | scalar or None |
| animated | bool |
| clip_bo | BboxBase or None |
| clip_on | bool |
| clip_pa | Patch or (Path, Transform) or None |
| figure | Figure |
| gid | str |
| height | float |
| in_layo | bool |
| label | object |
| mouseov | bool |
| offset | (float, float) |
| path_ef | list of AbstractPathEffect |
| picker | None or bool or float or callable |
| rasterized | bool |
| sketch_ | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| transform | unknown |
| url | str |
| visible | bool |
| width | float |
| zorder | float |

**set_offset**(*xy*)

Set the offset of the container.

**Parameters**

**xy**

[(float, float)] The (x, y) coordinates of the offset in display units.

**set_transform**(*t*)

set_transform is ignored.

**class** matplotlib.offsetbox.**DraggableAnnotation**(*annotation*, *use_blit=False*)

Bases: DraggableBase

**save_offset**()

**update_offset**(*dx*, *dy*)

**class** matplotlib.offsetbox.**DraggableBase**(*ref_artist*, *use_blit=False*)

Bases: object

Helper base class for a draggable artist (legend, offsetbox).

Derived classes must override the following methods:

```
def save_offset(self):
    '''
    Called when the object is picked for dragging; should save the
    reference position of the artist.
    '''

def update_offset(self, dx, dy):
    '''
    Called during the dragging; (*dx*, *dy*) is the pixel offset from
    the point where the mouse drag started.
    '''
```

Optionally, you may override the following method:

```
def finalize_offset(self):
    '''Called when the mouse is released.'''
```

In the current implementation of *DraggableLegend* and *DraggableAnnotation*, *update_offset* places the artists in display coordinates, and *finalize_offset* recalculates their position in axes coordinate and set a relevant attribute.

**property canvas**

**property cids**

**disconnect**()

Disconnect the callbacks.

**finalize_offset**()

**on_motion**(*evt*)

**on_pick**(*evt*)

**on_release**(*event*)

**save_offset**()

**update_offset**(*dx*, *dy*)

**class** matplotlib.offsetbox.**DraggableOffsetBox**(*ref_artist*, *offsetbox*, *use_blit=False*)

Bases: *DraggableBase*

**get_loc_in_canvas**()

**save_offset**()

**update_offset**(*dx*, *dy*)

**class** matplotlib.offsetbox.**DrawingArea**(*width*, *height*, *xdescent=0.0*, *ydescent=0.0*, *clip=False*)

Bases: *OffsetBox*

The DrawingArea can contain any Artist as a child. The DrawingArea has a fixed width and height. The position of children relative to the parent is fixed. The children can be clipped at the boundaries of the parent.

**Parameters**

**width, height**

[float] Width and height of the container box.

**xdescent, ydescent**

[float] Descent of the box in x- and y-direction.

**clip**

[bool] Whether to clip the children to the box.

**add_artist**(*a*)

Add an *Artist* to the container box.

**property clip_children**

If the children of this DrawingArea should be clipped by DrawingArea bounding box.

**draw**(*renderer*)

Update the location of children if necessary and draw them to the given *renderer*.

**get_bbox**(*renderer*)

Return the bbox of the offsetbox, ignoring parent offsets.

**get_offset**()

Return offset of the container.

**get_transform**()

Return the *Transform* applied to the children.

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *gid=<UNSET>*, *height=<UNSET>*, *in_layout=<UNSET>*, *label=<UNSET>*, *mouseover=<UNSET>*, *offset=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *width=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| agg_fil | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image |
| alpha | scalar or None |
| animated | bool |
| clip_bo | BboxBase or None |
| clip_on | bool |
| clip_pa | Patch or (Path, Transform) or None |
| figure | Figure |
| gid | str |
| height | float |
| in_layo | bool |
| label | object |
| mouseov | bool |
| offset | (float, float) |
| path_ef | list of AbstractPathEffect |
| picker | None or bool or float or callable |
| rasterized | bool |
| sketch_ | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| transform | unknown |
| url | str |
| visible | bool |
| width | float |
| zorder | float |

**set_offset**(*xy*)

Set the offset of the container.

**Parameters**

**xy**

[(float, float)] The (x, y) coordinates of the offset in display units.

**set_transform**(*t*)

set_transform is ignored.

**class** matplotlib.offsetbox.**HPacker**(*pad=0.0*, *sep=0.0*, *width=None*, *height=None*, *align='baseline'*, *mode='fixed'*, *children=None*)

Bases: *PackerBase*

HPacker packs its children horizontally, automatically adjusting their relative positions at draw time.

**Parameters**

**pad**

[float, default: 0.0] The boundary padding in points.

**sep**

[float, default: 0.0] The spacing between items in points.

**width, height**

[float, optional] Width and height of the container box in pixels, calculated if *None*.

**align**

[{'top', 'bottom', 'left', 'right', 'center', 'baseline'}, default: 'baseline'] Alignment of boxes.

**mode**

[{'fixed', 'expand', 'equal'}, default: 'fixed'] The packing mode.

- 'fixed' packs the given *Artist*s tight with *sep* spacing.
- 'expand' uses the maximal available space to distribute the artists with equal spacing in between.
- 'equal': Each artist an equal fraction of the available space and is left-aligned (or top-aligned) therein.

**children**

[list of *Artist*] The artists to pack.

## Notes

*pad* and *sep* are in points and will be scaled with the renderer dpi, while *width* and *height* are in pixels.

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *gid=<UNSET>*, *height=<UNSET>*, *in_layout=<UNSET>*, *label=<UNSET>*, *mouseover=<UNSET>*, *offset=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *width=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| agg_fil | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image |
| alpha | scalar or None |
| animated | bool |
| clip_bo | BboxBase or None |
| clip_on | bool |
| clip_pa | Patch or (Path, Transform) or None |
| figure | Figure |
| gid | str |
| height | float |
| in_layo | bool |
| label | object |
| mouseov | bool |
| offset | (float, float) or callable |
| path_ef | list of AbstractPathEffect |
| picker | None or bool or float or callable |
| rasterized | bool |
| sketch_ | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| transform | Transform |
| url | str |
| visible | bool |
| width | float |
| zorder | float |

**class** matplotlib.offsetbox.**OffsetBox**(**args*, ***kwargs*)

Bases: Artist

The OffsetBox is a simple container artist.

The child artists are meant to be drawn at a relative position to its parent.

Being an artist itself, all parameters are passed on to Artist.

**property axes**

The *Axes* instance the artist resides in, or *None*.

**contains**(*mouseevent*)

Delegate the mouse event contains-check to the children.

As a container, the *OffsetBox* does not respond itself to mouseevents.

**Parameters**

**mouseevent**

[*MouseEvent*]

**Returns**

**contains**

[bool] Whether any values are within the radius.

**details**

[dict] An artist-specific dictionary of details of the event context, such as which points are contained in the pick radius. See the individual Artist subclasses for details.

**See also:**

***Artist.contains***

**draw**(*renderer*)

Update the location of children if necessary and draw them to the given *renderer*.

**get_bbox**(*renderer*)

Return the bbox of the offsetbox, ignoring parent offsets.

**get_children**()

Return a list of the child *Artist*s.

**get_offset**(*bbox*, *renderer*)

Return the offset as a tuple (x, y).

The extent parameters have to be provided to handle the case where the offset is dynamically determined by a callable (see *set_offset*).

**Parameters**

**bbox**

[*Bbox*]

**renderer**

[*RendererBase* subclass]

**get_visible_children**()

Return a list of the visible child *Artist*s.

**get_window_extent**(*renderer=None*)

Get the artist's bounding box in display space.

The bounding box' width and height are nonnegative.

Subclasses should override for inclusion in the bounding box "tight" calculation. Default is to return an empty bounding box at 0, 0.

Be careful when using this function, the results will not update if the artist window extent of the artist changes. The extent can change due to any changes in the transform stack, such as changing the Axes limits, the figure size, or the canvas used (as is done when saving a figure). This can lead to unexpected behavior where interactive figures will look fine on the screen, but will save incorrectly.

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *gid=<UNSET>*, *height=<UNSET>*, *in_layout=<UNSET>*, *label=<UNSET>*, *mouseover=<UNSET>*, *offset=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *width=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| agg_fil | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image |
| alpha | scalar or None |
| animated | bool |
| clip_bo | BboxBase or None |
| clip_on | bool |
| clip_pa | Patch or (Path, Transform) or None |
| figure | Figure |
| gid | str |
| height | float |
| in_layo | bool |
| label | object |
| mouseov | bool |
| offset | (float, float) or callable |
| path_ef | list of AbstractPathEffect |
| picker | None or bool or float or callable |
| rasterized | bool |
| sketch_ | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| transform | Transform |
| url | str |
| visible | bool |
| width | float |
| zorder | float |

**set_figure**(*fig*)

Set the Figure for the OffsetBox and all its children.

**Parameters**

**fig**

[Figure]

**set_height**(*height*)

Set the height of the box.

**Parameters**

**height**

[float]

**set_offset**(*xy*)

Set the offset.

**Parameters**

**xy**

[(float, float) or callable] The (x, y) coordinates of the offset in display units. These can either be given explicitly as a tuple (x, y), or by providing a function that converts the extent into the offset. This function must have the signature:

```
def offset(width, height, xdescent, ydescent, renderer) -
 ↪> (float, float)
```

**set_width**(*width*)

Set the width of the box.

**Parameters**

**width**

[float]

**class** matplotlib.offsetbox.**OffsetImage**(*arr*, *, *zoom=1*, *cmap=None*, *norm=None*, *interpolation=None*, *origin=None*, *filternorm=True*, *filterrad=4.0*, *resample=False*, *dpi_cor=True*, ***kwargs*)

Bases: *OffsetBox*

**draw**(*renderer*)

Update the location of children if necessary and draw them to the given *renderer*.

**get_bbox**(*renderer*)

Return the bbox of the offsetbox, ignoring parent offsets.

**get_children**()

Return a list of the child *Artist*s.

**get_data**()

**get_offset**()

Return offset of the container.

**get_zoom**()

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *data=<UNSET>*, *gid=<UNSET>*, *height=<UNSET>*, *in_layout=<UNSET>*, *label=<UNSET>*, *mouseover=<UNSET>*, *offset=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *width=<UNSET>*, *zoom=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| agg_fil | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image |
| alpha | scalar or None |
| animated | bool |
| clip_bo | BboxBase or None |
| clip_on | bool |
| clip_pa | Patch or (Path, Transform) or None |
| data | unknown |
| figure | Figure |
| gid | str |
| height | float |
| in_layo | bool |
| label | object |
| mouseov | bool |
| offset | (float, float) or callable |
| path_ef | list of AbstractPathEffect |
| picker | None or bool or float or callable |
| rasterized | bool |
| sketch_ | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| transform | Transform |
| url | str |
| visible | bool |
| width | float |
| zoom | unknown |
| zorder | float |

**set_data**(*arr*)

**set_zoom**(*zoom*)

**class** matplotlib.offsetbox.**PackerBase**(*pad=0.0*, *sep=0.0*, *width=None*, *height=None*, *align='baseline'*, *mode='fixed'*, *children=None*)

Bases: *OffsetBox*

**Parameters**

**pad**

[float, default: 0.0] The boundary padding in points.

**sep**

[float, default: 0.0] The spacing between items in points.

**width, height**

[float, optional] Width and height of the container box in pixels, calculated if *None*.

**align**

[{'top', 'bottom', 'left', 'right', 'center', 'baseline'}, default: 'baseline'] Alignment of boxes.

**mode**

[{'fixed', 'expand', 'equal'}, default: 'fixed'] The packing mode.

- 'fixed' packs the given *Artist*s tight with *sep* spacing.
- 'expand' uses the maximal available space to distribute the artists with equal spacing in between.
- 'equal': Each artist an equal fraction of the available space and is left-aligned (or top-aligned) therein.

**children**

[list of *Artist*] The artists to pack.

## Notes

*pad* and *sep* are in points and will be scaled with the renderer dpi, while *width* and *height* are in pixels.

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *gid=<UNSET>*, *height=<UNSET>*, *in_layout=<UNSET>*, *label=<UNSET>*, *mouseover=<UNSET>*, *offset=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *width=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| agg_fil | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image |
| alpha | scalar or None |
| animated | bool |
| clip_bo | BboxBase or None |
| clip_on | bool |
| clip_pa | Patch or (Path, Transform) or None |
| figure | Figure |
| gid | str |
| height | float |
| in_layo | bool |
| label | object |
| mouseov | bool |
| offset | (float, float) or callable |
| path_ef | list of AbstractPathEffect |
| picker | None or bool or float or callable |
| rasterized | bool |
| sketch_ | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| transform | Transform |
| url | str |
| visible | bool |
| width | float |
| zorder | float |

**class** matplotlib.offsetbox.**PaddedBox**(*child*, *pad=0.0*, *, *draw_frame=False*, *patch_attrs=None*)

Bases: OffsetBox

A container to add a padding around an Artist.

The PaddedBox contains a FancyBboxPatch that is used to visualize it when rendering.

**Parameters**

**child**

[*Artist*] The contained *Artist*.

**pad**

[float, default: 0.0] The padding in points. This will be scaled with the renderer dpi. In contrast, *width* and *height* are in *pixels* and thus not scaled.

**draw_frame**

[bool] Whether to draw the contained *FancyBboxPatch*.

**patch_attrs**

[dict or None] Additional parameters passed to the contained *FancyBboxPatch*.

**draw**(*renderer*)

Update the location of children if necessary and draw them to the given *renderer*.

**draw_frame**(*renderer*)

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *gid=<UNSET>*, *height=<UNSET>*, *in_layout=<UNSET>*, *label=<UNSET>*, *mouseover=<UNSET>*, *offset=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *width=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| agg_fil | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image |
| alpha | scalar or None |
| animated | bool |
| clip_bo | BboxBase or None |
| clip_on | bool |
| clip_pa | Patch or (Path, Transform) or None |
| figure | Figure |
| gid | str |
| height | float |
| in_layo | bool |
| label | object |
| mouseov | bool |
| offset | (float, float) or callable |
| path_ef | list of AbstractPathEffect |
| picker | None or bool or float or callable |
| rasterized | bool |
| sketch_ | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| transform | Transform |
| url | str |
| visible | bool |
| width | float |
| zorder | float |

**update_frame**(*bbox*, *fontsize=None*)

**class** matplotlib.offsetbox.**TextArea**(*s*, *, *textprops=None*, *multilinebaseline=False*)

Bases: OffsetBox

The TextArea is a container artist for a single Text instance.

The text is placed at (0, 0) with baseline+left alignment, by default. The width and height of the TextArea instance is the width and height of its child text.

**Parameters**

**s**

[str] The text to be displayed.

**textprops**

[dict, default: {}] Dictionary of keyword parameters to be passed to the *Text* instance in the TextArea.

**multilinebaseline**

[bool, default: False] Whether the baseline for multiline text is adjusted so that it is (approximately) center-aligned with single-line text.

**draw**(*renderer*)

Update the location of children if necessary and draw them to the given *renderer*.

**get_bbox**(*renderer*)

Return the bbox of the offsetbox, ignoring parent offsets.

**get_multilinebaseline**()

Get multilinebaseline.

**get_offset**()

Return offset of the container.

**get_text**()

Return the string representation of this area's text.

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *gid=<UNSET>*, *height=<UNSET>*, *in_layout=<UNSET>*, *label=<UNSET>*, *mouseover=<UNSET>*, *multilinebaseline=<UNSET>*, *offset=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *text=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *width=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| agg_filte | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image |
| alpha | scalar or None |
| animated | bool |
| clip_box | BboxBase or None |
| clip_on | bool |
| clip_path | Patch or (Path, Transform) or None |
| figure | Figure |
| gid | str |
| height | float |
| in_layout | bool |
| label | object |
| mouseover | bool |
| multilinebaseline | unknown |
| offset | (float, float) |
| path_effe | list of AbstractPathEffect |
| picker | None or bool or float or callable |
| rasterized | bool |
| sketch_pa | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| text | unknown |
| transform | unknown |
| url | str |
| visible | bool |
| width | float |
| zorder | float |

**set_multilinebaseline**(*t*)

Set multilinebaseline.

If True, the baseline for multiline text is adjusted so that it is (approximately) center-aligned with single-line text. This is used e.g. by the legend implementation so that single-line labels are baseline-aligned, but multiline labels are "center"-aligned with them.

**set_offset**(*xy*)

Set the offset of the container.

**Parameters**

**xy**

[(float, float)] The (x, y) coordinates of the offset in display units.

**set_text**(*s*)

Set the text of this area as a string.

**set_transform**(*t*)

set_transform is ignored.

**class** matplotlib.offsetbox.**VPacker**(*pad=0.0*, *sep=0.0*, *width=None*, *height=None*, *align='baseline'*, *mode='fixed'*, *children=None*)

Bases: *PackerBase*

VPacker packs its children vertically, automatically adjusting their relative positions at draw time.

**Parameters**

**pad**

[float, default: 0.0] The boundary padding in points.

**sep**

[float, default: 0.0] The spacing between items in points.

**width, height**

[float, optional] Width and height of the container box in pixels, calculated if *None*.

**align**

[{'top', 'bottom', 'left', 'right', 'center', 'baseline'}, default: 'baseline'] Alignment of boxes.

**mode**

[{'fixed', 'expand', 'equal'}, default: 'fixed'] The packing mode.

- 'fixed' packs the given *Artist*s tight with *sep* spacing.
- 'expand' uses the maximal available space to distribute the artists with equal spacing in between.
- 'equal': Each artist an equal fraction of the available space and is left-aligned (or top-aligned) therein.

**children**

[list of *Artist*] The artists to pack.

### Notes

*pad* and *sep* are in points and will be scaled with the renderer dpi, while *width* and *height* are in pixels.

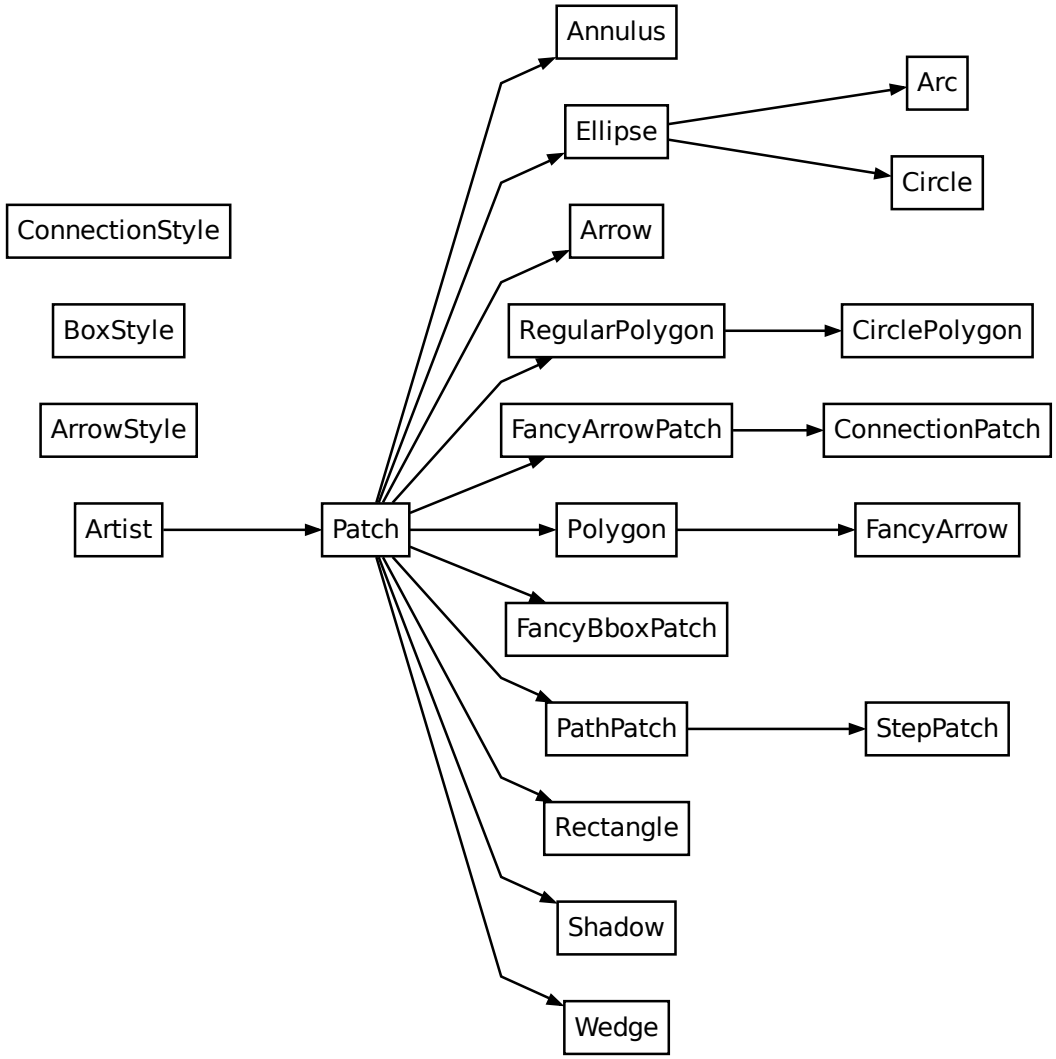
**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *gid=<UNSET>*, *height=<UNSET>*, *in_layout=<UNSET>*, *label=<UNSET>*, *mouseover=<UNSET>*, *offset=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *width=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| agg_fil | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image |
| alpha | scalar or None |
| animated | bool |
| clip_bo | BboxBase or None |
| clip_on | bool |
| clip_pa | Patch or (Path, Transform) or None |
| figure | Figure |
| gid | str |
| height | float |
| in_layo | bool |
| label | object |
| mouseov | bool |
| offset | (float, float) or callable |
| path_ef | list of AbstractPathEffect |
| picker | None or bool or float or callable |
| rasterized | bool |
| sketch_ | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| transform | Transform |
| url | str |
| visible | bool |
| width | float |
| zorder | float |

### 7.2.37 matplotlib.patches




Patches are *Artist*s with a face color and an edge color.

### Classes

| | |
|---|---|
| *Annulus*(xy, r, width[, angle]) | An elliptical annulus. |
| *Arc*(xy, width, height, *[, angle, theta1, ...]) | An elliptical arc, i.e. a segment of an ellipse. |
| *Arrow*(x, y, dx, dy, *[, width]) | An arrow patch. |
| *ArrowStyle*(stylename, **kwargs) | *ArrowStyle* is a container class which defines several arrowstyle classes, which is used to create an arrow path along a given path. |
| *BoxStyle*(stylename, **kwargs) | *BoxStyle* is a container class which defines several boxstyle classes, which are used for *FancyBboxPatch*. |
| *Circle*(xy[, radius]) | A circle patch. |
| *CirclePolygon*(xy[, radius, resolution]) | A polygon-approximation of a circle patch. |
| *ConnectionPatch*(xyA, xyB, coordsA[, ...]) | A patch that connects two points (possibly in different Axes). |
| *ConnectionStyle*(stylename, **kwargs) | *ConnectionStyle* is a container class which defines several connectionstyle classes, which is used to create a path between two points. |
| *Ellipse*(xy, width, height, *[, angle]) | A scale-free ellipse. |
| *FancyArrow*(x, y, dx, dy, *[, width, ...]) | Like Arrow, but lets you set head width and head height independently. |
| *FancyArrowPatch*([posA, posB, path, ...]) | A fancy arrow patch. |
| *FancyBboxPatch*(xy, width, height[, ...]) | A fancy box around a rectangle with lower left at *xy* = (*x*, *y*) with specified width and height. |
| *Patch*(*[, edgecolor, facecolor, color, ...]) | A patch is a 2D artist with a face color and an edge color. |
| *PathPatch*(path, **kwargs) | A general polycurve path patch. |
| *StepPatch*(values, edges, *[, orientation, ...]) | A path patch describing a stepwise constant function. |
| *Polygon*(xy, *[, closed]) | A general polygon patch. |
| *Rectangle*(xy, width, height, *[, angle, ...]) | A rectangle defined via an anchor point *xy* and its *width* and *height*. |
| *RegularPolygon*(xy, numVertices, *[, radius, ...]) | A regular polygon patch. |
| *Shadow*(patch, ox, oy, *[, shade]) | Create a shadow of the given *patch*. |
| *Wedge*(center, r, theta1, theta2, *[, width]) | Wedge shaped patch. |

### matplotlib.patches.Annulus

**class** matplotlib.patches.**Annulus**(*xy*, *r*, *width*, *angle=0.0*, ***kwargs*)

Bases: *Patch*

An elliptical annulus.

**Parameters**

**xy**

[(float, float)] xy coordinates of annulus centre.

**r**

[float or (float, float)] The radius, or semi-axes:

- If float: radius of the outer circle.
- If two floats: semi-major and -minor axes of outer ellipse.

**width**

[float] Width (thickness) of the annular ring. The width is measured inward from the outer ellipse so that for the inner ellipse the semi-axes are given by `r - width`. *width* must be less than or equal to the semi-minor axis.

**angle**

[float, default: 0] Rotation angle in degrees (anti-clockwise from the positive x-axis). Ignored for circular annuli (i.e., if *r* is a scalar).

****kwargs**

Keyword arguments control the *Patch* properties:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and retur |
| *alpha* | unknown |
| *animated* | bool |
| *antialiased* or aa | bool or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* | *color* |
| *edgecolor* or ec | *color* or None |
| *facecolor* or fc | *color* or None |
| *figure* | *Figure* |
| *fill* | bool |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |

Table 68 – continued from previc

| Property | Description |
|---|---|
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float or None |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *zorder* | float |

**property angle**

Return the angle of the annulus.

**property center**

Return the center of the annulus.

**get_angle**()

Return the angle of the annulus.

**get_center**()

Return the center of the annulus.

**get_path**()

Return the path of this patch.

**get_radii**()

Return the semi-major and semi-minor radii of the annulus.

**get_width**()

Return the width (thickness) of the annulus ring.

**property radii**

Return the semi-major and semi-minor radii of the annulus.

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *angle=<UNSET>*, *animated=<UNSET>*, *antialiased=<UNSET>*, *capstyle=<UNSET>*, *center=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *color=<UNSET>*, *edgecolor=<UNSET>*, *facecolor=<UNSET>*, *fill=<UNSET>*, *gid=<UNSET>*, *hatch=<UNSET>*, *in_layout=<UNSET>*, *joinstyle=<UNSET>*, *label=<UNSET>*, *linestyle=<UNSET>*, *linewidth=<UNSET>*, *mouseover=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *radii=<UNSET>*, *rasterized=<UNSET>*, *semimajor=<UNSET>*, *semiminor=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *width=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a |
| *alpha* | scalar or None |
| *angle* | float |
| *animated* | bool |
| *antialiased* or aa | bool or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *center* | (float, float) |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* | *color* |
| *edgecolor* or ec | *color* or None |
| *facecolor* or fc | *color* or None |
| *figure* | *Figure* |
| *fill* | bool |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float or None |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *radii* | float or (float, float) |
| *rasterized* | bool |
| *semimajor* | float |
| *semiminor* | float |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |

Table 69 – continued from previous p

| Property | Description |
|---|---|
| *url* | str |
| *visible* | bool |
| *width* | float |
| *zorder* | float |

**set_angle**(*angle*)

Set the tilt angle of the annulus.

**Parameters**

**angle**

[float]

**set_center**(*xy*)

Set the center of the annulus.

**Parameters**

**xy**

[(float, float)]

**set_radii**(*r*)

Set the semi-major ($a$) and semi-minor radii ($b$) of the annulus.

**Parameters**

**r**

[float or (float, float)] The radius, or semi-axes:

- If float: radius of the outer circle.
- If two floats: semi-major and -minor axes of outer ellipse.

**set_semimajor**(*a*)

Set the semi-major axis $a$ of the annulus.

**Parameters**

**a**

[float]

**set_semiminor**(*b*)

Set the semi-minor axis *b* of the annulus.

**Parameters**

**b**

[float]

**set_width**(*width*)

Set the width (thickness) of the annulus ring.

The width is measured inwards from the outer ellipse.

**Parameters**

**width**

[float]

**property width**

Return the width (thickness) of the annulus ring.

### matplotlib.patches.Arc

**class** matplotlib.patches.**Arc**(*xy*, *width*, *height*, *, *angle=0.0*, *theta1=0.0*, *theta2=360.0*, ***kwargs*)

Bases: *Ellipse*

An elliptical arc, i.e. a segment of an ellipse.

Due to internal optimizations, the arc cannot be filled.

**Parameters**

**xy**

[(float, float)] The center of the ellipse.

**width**

[float] The length of the horizontal axis.

**height**

[float] The length of the vertical axis.

**angle**

[float] Rotation of the ellipse in degrees (counterclockwise).

**theta1, theta2**

[float, default: 0, 360] Starting and ending angles of the arc in degrees. These values are relative to *angle*, e.g. if *angle* = 45 and *theta1* = 90 the absolute starting angle is 135. Default *theta1* = 0, *theta2* = 360, i.e. a complete ellipse. The arc is drawn in the counterclockwise direction. Angles greater than or equal to 360, or smaller than 0, are represented by an equivalent angle in the range [0, 360), by taking the input value mod 360.

**Other Parameters**

****kwargs**

[`Patch` properties] Most `Patch` properties are supported as keyword arguments, except *fill* and *facecolor* because filling is not supported.

| Property | Description |
|---|---|
| `agg_filter` | a filter function, which takes a (m, n, 3) float array and a dpi value, and returr |
| `alpha` | unknown |
| `animated` | bool |
| `antialiased` or aa | bool or None |
| `capstyle` | `CapStyle` or {'butt', 'projecting', 'round'} |
| `clip_box` | `BboxBase` or None |
| `clip_on` | bool |
| `clip_path` | Patch or (Path, Transform) or None |
| `color` | *color* |
| `edgecolor` or ec | *color* or None |
| `facecolor` or fc | *color* or None |
| `figure` | `Figure` |
| `fill` | bool |
| `gid` | str |
| `hatch` | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| `in_layout` | bool |
| `joinstyle` | `JoinStyle` or {'miter', 'round', 'bevel'} |
| `label` | object |
| `linestyle` or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| `linewidth` or lw | float or None |
| `mouseover` | bool |
| `path_effects` | list of `AbstractPathEffect` |
| `picker` | None or bool or float or callable |
| `rasterized` | bool |
| `sketch_params` | (scale: float, length: float, randomness: float) |
| `snap` | bool or None |
| `transform` | `Transform` |
| `url` | str |
| `visible` | bool |

Table 70 – continued from previc

| Property | Description |
|---|---|
| *zorder* | float |

**draw**(*renderer*)

Draw the arc to the given *renderer*.

### Notes

Ellipses are normally drawn using an approximation that uses eight cubic Bezier splines. The error of this approximation is 1.89818e-6, according to this unverified source:

> Lancaster, Don. *Approximating a Circle or an Ellipse Using Four Bezier Cubic Splines.*
>
> https://www.tinaja.com/glib/ellipse4.pdf

There is a use case where very large ellipses must be drawn with very high accuracy, and it is too expensive to render the entire ellipse with enough segments (either splines or line segments). Therefore, in the case where either radius of the ellipse is large enough that the error of the spline approximation will be visible (greater than one pixel offset from the ideal), a different technique is used.

In that case, only the visible parts of the ellipse are drawn, with each visible arc using a fixed number of spline segments (8). The algorithm proceeds as follows:

1. The points where the ellipse intersects the axes (or figure) bounding box are located. (This is done by performing an inverse transformation on the bbox such that it is relative to the unit circle -- this makes the intersection calculation much easier than doing rotated ellipse intersection directly.)

   This uses the "line intersecting a circle" algorithm from:

   > Vince, John. *Geometry for Computer Graphics: Formulae, Examples & Proofs.* London: Springer-Verlag, 2005.

2. The angles of each of the intersection points are calculated.
3. Proceeding counterclockwise starting in the positive x-direction, each of the visible arc-segments between the pairs of vertices are drawn using the Bezier arc approximation technique implemented in *Path.arc*.

**set**(**, agg_filter=<UNSET>, alpha=<UNSET>, angle=<UNSET>, animated=<UNSET>, antialiased=<UNSET>, capstyle=<UNSET>, center=<UNSET>, clip_box=<UNSET>, clip_on=<UNSET>, clip_path=<UNSET>, color=<UNSET>, edgecolor=<UNSET>, facecolor=<UNSET>, fill=<UNSET>, gid=<UNSET>, hatch=<UNSET>, height=<UNSET>, in_layout=<UNSET>, joinstyle=<UNSET>, label=<UNSET>, linestyle=<UNSET>, linewidth=<UNSET>, mouseover=<UNSET>, path_effects=<UNSET>, picker=<UNSET>, rasterized=<UNSET>, sketch_params=<UNSET>, snap=<UNSET>, transform=<UNSET>, url=<UNSET>, visible=<UNSET>, width=<UNSET>, zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a |
| *alpha* | scalar or None |
| *angle* | float |
| *animated* | bool |
| *antialiased* or aa | bool or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *center* | (float, float) |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* | *color* |
| *edgecolor* or ec | *color* or None |
| *facecolor* or fc | *color* or None |
| *figure* | *Figure* |
| *fill* | bool |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *height* | float |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float or None |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *width* | float |
| *zorder* | float |

### Examples using `matplotlib.patches.Arc`

- *Scale invariant angle label*
- *Ellipse with units*

## matplotlib.patches.Arrow

**class** matplotlib.patches.**Arrow**(*x*, *y*, *dx*, *dy*, *, *width=1.0*, ***kwargs*)

Bases: *Patch*

An arrow patch.

Draws an arrow from (*x*, *y*) to (*x* + *dx*, *y* + *dy*). The width of the arrow is scaled by *width*.

**Parameters**

**x**

[float] x coordinate of the arrow tail.

**y**

[float] y coordinate of the arrow tail.

**dx**

[float] Arrow length in the x direction.

**dy**

[float] Arrow length in the y direction.

**width**

[float, default: 1] Scale factor for the width of the arrow. With a default value of 1, the tail width is 0.2 and head width is 0.6.

****kwargs**

Keyword arguments control the *Patch* properties:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and retur |
| *alpha* | unknown |
| *animated* | bool |
| *antialiased* or aa | bool or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* | *color* |

Table 72 – continued from previc

| Property | Description |
|---|---|
| *edgecolor* or ec | *color* or None |
| *facecolor* or fc | *color* or None |
| *figure* | *Figure* |
| *fill* | bool |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float or None |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *zorder* | float |

**See also:**

***FancyArrow***

Patch that allows independent control of the head and tail properties.

**get_patch_transform**()

Return the *Transform* instance mapping patch coordinates to data coordinates.

For example, one may define a patch of a circle which represents a radius of 5 by providing coordinates for a unit circle, and a transform which scales the coordinates (the patch coordinate) by 5.

**get_path**()

Return the path of this patch.

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *antialiased=<UNSET>*, *capstyle=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *color=<UNSET>*, *data=<UNSET>*, *edgecolor=<UNSET>*, *facecolor=<UNSET>*, *fill=<UNSET>*, *gid=<UNSET>*, *hatch=<UNSET>*, *in_layout=<UNSET>*, *joinstyle=<UNSET>*, *label=<UNSET>*, *linestyle=<UNSET>*, *linewidth=<UNSET>*, *mouseover=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* or aa | bool or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* | *color* |
| *data* | unknown |
| *edgecolor* or ec | *color* or None |
| *facecolor* or fc | *color* or None |
| *figure* | *Figure* |
| *fill* | bool |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float or None |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *zorder* | float |

**set_data**(*x=None*, *y=None*, *dx=None*, *dy=None*, *width=None*)

Set *Arrow* x, y, dx, dy and width. Values left as None will not be updated.

**Parameters**

**x, y**

[float or None, default: None] The x and y coordinates of the arrow base.

**dx, dy**

[float or None, default: None] The length of the arrow along x and y direction.

**width**

[float or None, default: None] Width of full arrow tail.

### Examples using `matplotlib.patches.Arrow`

- *Arrow guide*
- *Reference for Matplotlib artists*

## matplotlib.patches.ArrowStyle

**class** matplotlib.patches.**ArrowStyle**(*stylename*, ***kwargs*)

Bases: `_Style`

*ArrowStyle* is a container class which defines several arrowstyle classes, which is used to create an arrow path along a given path. These are mainly used with *FancyArrowPatch*.

An arrowstyle object can be either created as:

```
ArrowStyle.Fancy(head_length=.4, head_width=.4, tail_width=.4)
```

or:

```
ArrowStyle("Fancy", head_length=.4, head_width=.4, tail_width=.4)
```

or:

```
ArrowStyle("Fancy, head_length=.4, head_width=.4, tail_width=.4")
```

The following classes are defined

| Class | Name | Attrs |
|---|---|---|
| Curve | `-` | None |
| CurveA | `<-` | head_length=0.4, head_width=0.2, widthA=1.0, widthB=1.0, lengthA=0.2, lengthB=0.2, angleA=0, angleB=0, scaleA=None, scaleB=None |
| CurveB | `->` | head_length=0.4, head_width=0.2, widthA=1.0, widthB=1.0, lengthA=0.2, lengthB=0.2, angleA=0, angleB=0, scaleA=None, scaleB=None |
| CurveAB | `<->` | head_length=0.4, head_width=0.2, widthA=1.0, widthB=1.0, lengthA=0.2, lengthB=0.2, angleA=0, angleB=0, scaleA=None, scaleB=None |
| CurveFilledA | `<\|-` | head_length=0.4, head_width=0.2, widthA=1.0, widthB=1.0, lengthA=0.2, lengthB=0.2, angleA=0, angleB=0, scaleA=None, scaleB=None |
| CurveFilledB | `-\|>` | head_length=0.4, head_width=0.2, widthA=1.0, widthB=1.0, lengthA=0.2, lengthB=0.2, angleA=0, angleB=0, scaleA=None, scaleB=None |
| CurveFilledAB | `<\|-\|` | head_length=0.4, head_width=0.2, widthA=1.0, widthB=1.0, lengthA=0.2, lengthB=0.2, angleA=0, angleB=0, scaleA=None, scaleB=None |
| BracketA | `]-` | widthA=1.0, lengthA=0.2, angleA=0 |
| BracketB | `-[` | widthB=1.0, lengthB=0.2, angleB=0 |
| BracketAB | `]-[` | widthA=1.0, lengthA=0.2, angleA=0, widthB=1.0, lengthB=0.2, angleB=0 |
| BarAB | `\|-\|` | widthA=1.0, angleA=0, widthB=1.0, angleB=0 |
| BracketCurve | `]->` | widthA=1.0, lengthA=0.2, angleA=None |
| CurveBracket | `<-[` | widthB=1.0, lengthB=0.2, angleB=None |
| Simple | `simple` | head_length=0.5, head_width=0.5, tail_width=0.2 |
| Fancy | `fancy` | head_length=0.4, head_width=0.4, tail_width=0.4 |
| Wedge | `wedge` | tail_width=0.3, shrink_factor=0.5 |

For an overview of the visual appearance, see *Annotation arrow style reference*.

An instance of any arrow style class is a callable object, whose call signature is:

```
__call__(self, path, mutation_size, linewidth, aspect_ratio=1.)
```

and it returns a tuple of a `Path` instance and a boolean value. *path* is a `Path` instance along which the arrow will be drawn. *mutation_size* and *aspect_ratio* have the same meaning as in `BoxStyle`. *linewidth* is a line width to be stroked. This is meant to be used to correct the location of the head so that it does not overshoot the destination point, but not all classes support it.

### Notes

*angleA* and *angleB* specify the orientation of the bracket, as either a clockwise or counterclockwise angle depending on the arrow type. 0 degrees means perpendicular to the line connecting the arrow's head and tail.

Return the instance of the subclass with the given style name.

**class BarAB**(*widthA=1.0*, *angleA=0*, *widthB=1.0*, *angleB=0*)

Bases: _Curve

An arrow with vertical bars | at both ends.

**Parameters**

**widthA, widthB**

[float, default: 1.0] Width of the bracket.

**angleA, angleB**

[float, default: 0 degrees] Orientation of the bracket, as a counterclockwise angle. 0 degrees means perpendicular to the line.

**arrow = '|-|'**

**class BracketA**(*widthA=1.0*, *lengthA=0.2*, *angleA=0*)

Bases: _Curve

An arrow with an outward square bracket at its start.

**Parameters**

**widthA**

[float, default: 1.0] Width of the bracket.

**lengthA**

[float, default: 0.2] Length of the bracket.

**angleA**

[float, default: 0 degrees] Orientation of the bracket, as a counterclockwise angle. 0 degrees means perpendicular to the line.

**arrow = ']-'**

**class BracketAB**(*widthA=1.0*, *lengthA=0.2*, *angleA=0*, *widthB=1.0*, *lengthB=0.2*, *angleB=0*)

Bases: _Curve

An arrow with outward square brackets at both ends.

**Parameters**

**widthA, widthB**

[float, default: 1.0] Width of the bracket.

**lengthA, lengthB**

[float, default: 0.2] Length of the bracket.

**angleA, angleB**

[float, default: 0 degrees] Orientation of the bracket, as a counterclockwise angle. 0 degrees means perpendicular to the line.

**arrow = ']-['**

**class BracketB**(*widthB=1.0*, *lengthB=0.2*, *angleB=0*)

Bases: _Curve

An arrow with an outward square bracket at its end.

**Parameters**

**widthB**

[float, default: 1.0] Width of the bracket.

**lengthB**

[float, default: 0.2] Length of the bracket.

**angleB**

[float, default: 0 degrees] Orientation of the bracket, as a counterclockwise angle. 0 degrees means perpendicular to the line.

**arrow = '-['**

**class BracketCurve**(*widthA=1.0*, *lengthA=0.2*, *angleA=None*)

Bases: _Curve

An arrow with an outward square bracket at its start and a head at the end.

**Parameters**

**widthA**

[float, default: 1.0] Width of the bracket.

**lengthA**

[float, default: 0.2] Length of the bracket.

**angleA**

[float, default: 0 degrees] Orientation of the bracket, as a counterclockwise angle. 0 degrees means perpendicular to the line.

**arrow = ']->'**

**class Curve**

Bases: _Curve

A simple curve without any arrow head.

**Parameters**

**head_length**

[float, default: 0.4] Length of the arrow head, relative to *mutation_size*.

**head_width**

[float, default: 0.2] Width of the arrow head, relative to *mutation_size*.

**widthA, widthB**

[float, default: 1.0] Width of the bracket.

**lengthA, lengthB**

[float, default: 0.2] Length of the bracket.

**angleA, angleB**

[float, default: 0] Orientation of the bracket, as a counterclockwise angle. 0 degrees means perpendicular to the line.

**scaleA, scaleB**

[float, default: *mutation_size*] The scale of the brackets.

**class CurveA**(*head_length=0.4*, *head_width=0.2*, *widthA=1.0*, *widthB=1.0*, *lengthA=0.2*, *lengthB=0.2*, *angleA=0*, *angleB=0*, *scaleA=None*, *scaleB=None*)

Bases: _Curve

An arrow with a head at its start point.

**Parameters**

**head_length**

[float, default: 0.4] Length of the arrow head, relative to *mutation_size*.

**head_width**

[float, default: 0.2] Width of the arrow head, relative to *mutation_size*.

**widthA, widthB**

[float, default: 1.0] Width of the bracket.

**lengthA, lengthB**

[float, default: 0.2] Length of the bracket.

**angleA, angleB**

[float, default: 0] Orientation of the bracket, as a counterclockwise angle. 0 degrees means perpendicular to the line.

**scaleA, scaleB**

[float, default: *mutation_size*] The scale of the brackets.

**arrow = '<-'**

**class CurveAB**(*head_length=0.4*, *head_width=0.2*, *widthA=1.0*, *widthB=1.0*, *lengthA=0.2*, *lengthB=0.2*, *angleA=0*, *angleB=0*, *scaleA=None*, *scaleB=None*)

Bases: _Curve

An arrow with heads both at the start and the end point.

**Parameters**

**head_length**

[float, default: 0.4] Length of the arrow head, relative to *mutation_size*.

**head_width**

[float, default: 0.2] Width of the arrow head, relative to *mutation_size*.

**widthA, widthB**

[float, default: 1.0] Width of the bracket.

**lengthA, lengthB**

[float, default: 0.2] Length of the bracket.

**angleA, angleB**

[float, default: 0] Orientation of the bracket, as a counterclockwise angle. 0 degrees means perpendicular to the line.

**scaleA, scaleB**

[float, default: *mutation_size*] The scale of the brackets.

**arrow = '<->'**

**class CurveB**(*head_length=0.4*, *head_width=0.2*, *widthA=1.0*, *widthB=1.0*, *lengthA=0.2*, *lengthB=0.2*, *angleA=0*, *angleB=0*, *scaleA=None*, *scaleB=None*)

Bases: _Curve

An arrow with a head at its end point.

**Parameters**

**head_length**

[float, default: 0.4] Length of the arrow head, relative to *mutation_size*.

**head_width**

[float, default: 0.2] Width of the arrow head, relative to *mutation_size*.

**widthA, widthB**

[float, default: 1.0] Width of the bracket.

**lengthA, lengthB**

[float, default: 0.2] Length of the bracket.

**angleA, angleB**

[float, default: 0] Orientation of the bracket, as a counterclockwise angle. 0 degrees means perpendicular to the line.

**scaleA, scaleB**

[float, default: *mutation_size*] The scale of the brackets.

**arrow = '->'**

**class CurveBracket**(*widthB=1.0*, *lengthB=0.2*, *angleB=None*)

Bases: _Curve

An arrow with an outward square bracket at its end and a head at the start.

**Parameters**

**widthB**

[float, default: 1.0] Width of the bracket.

**lengthB**

[float, default: 0.2] Length of the bracket.

**angleB**

[float, default: 0 degrees] Orientation of the bracket, as a counterclockwise angle. 0 degrees means perpendicular to the line.

**arrow = '<-['**

**class CurveFilledA**(*head_length=0.4*, *head_width=0.2*, *widthA=1.0*, *widthB=1.0*, *lengthA=0.2*, *lengthB=0.2*, *angleA=0*, *angleB=0*, *scaleA=None*, *scaleB=None*)

Bases: _Curve

An arrow with filled triangle head at the start.

**Parameters**

**head_length**

[float, default: 0.4] Length of the arrow head, relative to *mutation_size*.

**head_width**

[float, default: 0.2] Width of the arrow head, relative to *mutation_size*.

**widthA, widthB**

[float, default: 1.0] Width of the bracket.

**lengthA, lengthB**

[float, default: 0.2] Length of the bracket.

**angleA, angleB**

[float, default: 0] Orientation of the bracket, as a counterclockwise angle. 0 degrees means perpendicular to the line.

**scaleA, scaleB**

[float, default: *mutation_size*] The scale of the brackets.

**arrow = '<|-'**

**class CurveFilledAB**(*head_length=0.4*, *head_width=0.2*, *widthA=1.0*, *widthB=1.0*, *lengthA=0.2*, *lengthB=0.2*, *angleA=0*, *angleB=0*, *scaleA=None*, *scaleB=None*)

Bases: _Curve

An arrow with filled triangle heads at both ends.

**Parameters**

**head_length**

[float, default: 0.4] Length of the arrow head, relative to *mutation_size*.

**head_width**

[float, default: 0.2] Width of the arrow head, relative to *mutation_size*.

**widthA, widthB**

[float, default: 1.0] Width of the bracket.

**lengthA, lengthB**

[float, default: 0.2] Length of the bracket.

**angleA, angleB**

[float, default: 0] Orientation of the bracket, as a counterclockwise angle. 0 degrees means perpendicular to the line.

**scaleA, scaleB**

[float, default: *mutation_size*] The scale of the brackets.

**arrow = '<|-|>'**

**class CurveFilledB**(*head_length=0.4*, *head_width=0.2*, *widthA=1.0*, *widthB=1.0*, *lengthA=0.2*, *lengthB=0.2*, *angleA=0*, *angleB=0*, *scaleA=None*, *scaleB=None*)

Bases: _Curve

An arrow with filled triangle head at the end.

**Parameters**

**head_length**

[float, default: 0.4] Length of the arrow head, relative to *mutation_size*.

**head_width**

[float, default: 0.2] Width of the arrow head, relative to *mutation_size*.

**widthA, widthB**

[float, default: 1.0] Width of the bracket.

**lengthA, lengthB**

[float, default: 0.2] Length of the bracket.

**angleA, angleB**

[float, default: 0] Orientation of the bracket, as a counterclockwise angle. 0 degrees means perpendicular to the line.

**scaleA, scaleB**

[float, default: *mutation_size*] The scale of the brackets.

**arrow = '-|>'**

**class Fancy**(*head_length=0.4*, *head_width=0.4*, *tail_width=0.4*)

Bases: _Base

A fancy arrow. Only works with a quadratic Bézier curve.

**Parameters**

**head_length**

[float, default: 0.4] Length of the arrow head.

**head_width**

[float, default: 0.4] Width of the arrow head.

**tail_width**

[float, default: 0.4] Width of the arrow tail.

**transmute**(*path*, *mutation_size*, *linewidth*)

The transmute method is the very core of the ArrowStyle class and must be overridden in the subclasses. It receives the *path* object along which the arrow will be drawn, and the *mutation_size*, with which the arrow head etc. will be scaled. The *linewidth* may be used to adjust the path so that it does not pass beyond the given points. It returns a tuple of a `Path` instance and a boolean. The boolean value indicate whether the path can be filled or not. The return value can also be a list of paths and list of booleans of the same length.

**class Simple**(*head_length=0.5*, *head_width=0.5*, *tail_width=0.2*)

Bases: `_Base`

A simple arrow. Only works with a quadratic Bézier curve.

**Parameters**

**head_length**

[float, default: 0.5] Length of the arrow head.

**head_width**

[float, default: 0.5] Width of the arrow head.

**tail_width**

[float, default: 0.2] Width of the arrow tail.

**transmute**(*path*, *mutation_size*, *linewidth*)

The transmute method is the very core of the ArrowStyle class and must be overridden in the subclasses. It receives the *path* object along which the arrow will be drawn, and the *mutation_size*, with which the arrow head etc. will be scaled. The *linewidth* may be used to adjust the path so that it does not pass beyond the given points. It returns a tuple of a `Path` instance and a boolean. The boolean value indicate whether the path can be filled or not. The return value can also be a list of paths and list of booleans of the same length.

**class Wedge**(*tail_width=0.3*, *shrink_factor=0.5*)

Bases: `_Base`

Wedge(?) shape. Only works with a quadratic Bézier curve. The start point has a width of the *tail_width* and the end point has a width of 0. At the middle, the width is *shrink_factor*x*tail_width*.

**Parameters**

**tail_width**

[float, default: 0.3] Width of the tail.

**shrink_factor**

[float, default: 0.5] Fraction of the arrow width at the middle point.

**transmute**(*path*, *mutation_size*, *linewidth*)

The transmute method is the very core of the ArrowStyle class and must be overridden in the subclasses. It receives the *path* object along which the arrow will be drawn, and the *mutation_size*, with which the arrow head etc. will be scaled. The *linewidth* may be used to adjust the path so that it does not pass beyond the given points. It returns a tuple of a *Path* instance and a boolean. The boolean value indicate whether the path can be filled or not. The return value can also be a list of paths and list of booleans of the same length.

### Examples using `matplotlib.patches.ArrowStyle`

- *Angle annotations on bracket arrows*
- *Annotation arrow style reference*

## matplotlib.patches.BoxStyle

**class** matplotlib.patches.**BoxStyle**(*stylename*, ***kwargs*)

Bases: `_Style`

*BoxStyle* is a container class which defines several boxstyle classes, which are used for *FancyBoxPatch*.

A style object can be created as:

```
BoxStyle.Round(pad=0.2)
```

or:

```
BoxStyle("Round", pad=0.2)
```

or:

```
BoxStyle("Round, pad=0.2")
```

The following boxstyle classes are defined.

| Class | Name | Attrs |
|---|---|---|
| Square | `square` | pad=0.3 |
| Circle | `circle` | pad=0.3 |
| Ellipse | `ellipse` | pad=0.3 |
| LArrow | `larrow` | pad=0.3 |
| RArrow | `rarrow` | pad=0.3 |
| DArrow | `darrow` | pad=0.3 |
| Round | `round` | pad=0.3, rounding_size=None |
| Round4 | `round4` | pad=0.3, rounding_size=None |
| Sawtooth | `sawtooth` | pad=0.3, tooth_size=None |
| Roundtooth | `roundtooth` | pad=0.3, tooth_size=None |

An instance of a boxstyle class is a callable object, with the signature

```
__call__(self, x0, y0, width, height, mutation_size) -> Path
```

*x0*, *y0*, *width* and *height* specify the location and size of the box to be drawn; *mutation_size* scales the outline properties such as padding.

Return the instance of the subclass with the given style name.

**class Circle**(*pad=0.3*)

Bases: object

A circular box.

**Parameters**

**pad**

[float, default: 0.3] The amount of padding around the original box.

**__call__**(*x0*, *y0*, *width*, *height*, *mutation_size*)

Call self as a function.

**class DArrow**(*pad=0.3*)

Bases: object

A box in the shape of a two-way arrow.

**Parameters**

**pad**

[float, default: 0.3] The amount of padding around the original box.

**__call__**(*x0*, *y0*, *width*, *height*, *mutation_size*)

Call self as a function.

**class Ellipse**(*pad=0.3*)

Bases: object

An elliptical box.

New in version 3.7.

**Parameters**

**pad**

[float, default: 0.3] The amount of padding around the original box.

**__call__**(*x0*, *y0*, *width*, *height*, *mutation_size*)

Call self as a function.

**class LArrow**(*pad=0.3*)

Bases: object

A box in the shape of a left-pointing arrow.

**Parameters**

**pad**

[float, default: 0.3] The amount of padding around the original box.

**__call__**(*x0*, *y0*, *width*, *height*, *mutation_size*)

Call self as a function.

**class RArrow**(*pad=0.3*)

Bases: *LArrow*

A box in the shape of a right-pointing arrow.

**Parameters**

**pad**

[float, default: 0.3] The amount of padding around the original box.

**__call__**(*x0*, *y0*, *width*, *height*, *mutation_size*)

Call self as a function.

**class Round**(*pad=0.3*, *rounding_size=None*)

Bases: object

A box with round corners.

**Parameters**

**pad**

[float, default: 0.3] The amount of padding around the original box.

**rounding_size**

[float, default: *pad*] Radius of the corners.

**__call__**(*x0*, *y0*, *width*, *height*, *mutation_size*)

Call self as a function.

**class Round4**(*pad=0.3*, *rounding_size=None*)

Bases: object

A box with rounded edges.

**Parameters**

**pad**

[float, default: 0.3] The amount of padding around the original box.

**rounding_size**

[float, default: *pad*/2] Rounding of edges.

**__call__**(*x0*, *y0*, *width*, *height*, *mutation_size*)

Call self as a function.

**class Roundtooth**(*pad=0.3*, *tooth_size=None*)

Bases: *Sawtooth*

A box with a rounded sawtooth outline.

**Parameters**

**pad**

[float, default: 0.3] The amount of padding around the original box.

**tooth_size**

[float, default: *pad*/2] Size of the sawtooth.

**__call__**(*x0*, *y0*, *width*, *height*, *mutation_size*)

Call self as a function.

**class Sawtooth**(*pad=0.3*, *tooth_size=None*)

Bases: object

A box with a sawtooth outline.

**Parameters**

**pad**

[float, default: 0.3] The amount of padding around the original box.

**tooth_size**

[float, default: *pad*/2] Size of the sawtooth.

**__call__**(*x0*, *y0*, *width*, *height*, *mutation_size*)

Call self as a function.

**class Square**(*pad=0.3*)

Bases: object

A square box.

**Parameters**

**pad**

[float, default: 0.3] The amount of padding around the original box.

**__call__** (*x0*, *y0*, *width*, *height*, *mutation_size*)

Call self as a function.

### Examples using `matplotlib.patches.BoxStyle`

- *Reference for Matplotlib artists*
- *Drawing fancy boxes*

## matplotlib.patches.Circle

**class** matplotlib.patches.**Circle**(*xy*, *radius=5*, ***kwargs*)

Bases: *Ellipse*

A circle patch.

Create a true circle at center *xy* = (*x*, *y*) with given *radius*.

Unlike *CirclePolygon* which is a polygonal approximation, this uses Bezier splines and is much closer to a scale-free circle.

Valid keyword arguments are:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n |
| *alpha* | unknown |
| *animated* | bool |
| *antialiased* or aa | bool or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* | *color* |
| *edgecolor* or ec | *color* or None |
| *facecolor* or fc | *color* or None |
| *figure* | *Figure* |
| *fill* | bool |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |

Table 74 – continued from previous page

| Property | Description |
|---|---|
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float or None |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *zorder* | float |

**get_radius**()

Return the radius of the circle.

**property radius**

Return the radius of the circle.

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *angle=<UNSET>*, *animated=<UNSET>*, *antialiased=<UNSET>*, *capstyle=<UNSET>*, *center=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *color=<UNSET>*, *edgecolor=<UNSET>*, *facecolor=<UNSET>*, *fill=<UNSET>*, *gid=<UNSET>*, *hatch=<UNSET>*, *height=<UNSET>*, *in_layout=<UNSET>*, *joinstyle=<UNSET>*, *label=<UNSET>*, *linestyle=<UNSET>*, *linewidth=<UNSET>*, *mouseover=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *radius=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *width=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a |
| *alpha* | scalar or None |
| *angle* | float |
| *animated* | bool |
| *antialiased* or aa | bool or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *center* | (float, float) |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |

Table 75 – continued from previous p

| Property | Description |
|---|---|
| *clip_path* | Patch or (Path, Transform) or None |
| *color* | *color* |
| *edgecolor* or ec | *color* or None |
| *facecolor* or fc | *color* or None |
| *figure* | *Figure* |
| *fill* | bool |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *height* | float |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float or None |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *radius* | float |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *width* | float |
| *zorder* | float |

**set_radius**(*radius*)

Set the radius of the circle.

**Parameters**

**radius**

[float]

### Examples using `matplotlib.patches.Circle`

- *Clipping images with patches*
- *Axes box aspect*
- *AnnotationBbox demo*
- *Reference for Matplotlib artists*
- *Dolphins*
- *Mmh Donuts!!!*
- *Circles, Wedges and Polygons*
- *ggplot style sheet*
- *Grayscale style sheet*
- *Style sheets reference*
- *Simple Anchored Artists*
- *Anatomy of a figure*
- *Looking Glass*
- *Anchored Artists*
- *Custom projection*
- *Packed-bubble chart*
- *Draw flat objects in 3D plot*
- *Radar chart (aka spider or star chart)*
- *Transformations Tutorial*
- *Legend guide*
- *Annotations*

## matplotlib.patches.CirclePolygon

**class** matplotlib.patches.**CirclePolygon**(*xy*, *radius=5*, *, *resolution=20*, ***kwargs*)

Bases: *RegularPolygon*

A polygon-approximation of a circle patch.

Create a circle at $xy = (x, y)$ with given *radius*.

This circle is approximated by a regular polygon with *resolution* sides. For a smoother circle drawn with splines, see *Circle*.

Valid keyword arguments are:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n |
| *alpha* | unknown |
| *animated* | bool |
| *antialiased* or aa | bool or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* | *color* |
| *edgecolor* or ec | *color* or None |
| *facecolor* or fc | *color* or None |
| *figure* | *Figure* |
| *fill* | bool |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float or None |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *zorder* | float |

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *antialiased=<UNSET>*, *capstyle=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *color=<UNSET>*, *edgecolor=<UNSET>*, *facecolor=<UNSET>*, *fill=<UNSET>*, *gid=<UNSET>*, *hatch=<UNSET>*, *in_layout=<UNSET>*, *joinstyle=<UNSET>*, *label=<UNSET>*, *linestyle=<UNSET>*, *linewidth=<UNSET>*, *mouseover=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| agg_filter | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a |
| alpha | scalar or None |
| animated | bool |
| antialiased or aa | bool or None |
| capstyle | CapStyle or {'butt', 'projecting', 'round'} |
| clip_box | BboxBase or None |
| clip_on | bool |
| clip_path | Patch or (Path, Transform) or None |
| color | color |
| edgecolor or ec | color or None |
| facecolor or fc | color or None |
| figure | Figure |
| fill | bool |
| gid | str |
| hatch | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| in_layout | bool |
| joinstyle | JoinStyle or {'miter', 'round', 'bevel'} |
| label | object |
| linestyle or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| linewidth or lw | float or None |
| mouseover | bool |
| path_effects | list of AbstractPathEffect |
| picker | None or bool or float or callable |
| rasterized | bool |
| sketch_params | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| transform | Transform |
| url | str |
| visible | bool |
| zorder | float |

### matplotlib.patches.ConnectionPatch

**class** matplotlib.patches.**ConnectionPatch**(*xyA*, *xyB*, *coordsA*, *coordsB=None*, ***, *axesA=None*, *axesB=None*, *arrowstyle='-'*, *connectionstyle='arc3'*, *patchA=None*, *patchB=None*, *shrinkA=0.0*, *shrinkB=0.0*, *mutation_scale=10.0*, *mutation_aspect=None*, *clip_on=False*, ***kwargs*)

Bases: FancyArrowPatch

A patch that connects two points (possibly in different Axes).

Connect point *xyA* in *coordsA* with point *xyB* in *coordsB*.

Valid keys are

| Key | Description |
|---|---|
| arrowstyle | the arrow style |
| connectionstyle | the connection style |
| relpos | default is (0.5, 0.5) |
| patchA | default is bounding box of the text |
| patchB | default is None |
| shrinkA | default is 2 points |
| shrinkB | default is 2 points |
| mutation_scale | default is text size (in points) |
| mutation_aspect | default is 1. |
| ? | any key for `matplotlib.patches.PathPatch` |

*coordsA* and *coordsB* are strings that indicate the coordinates of *xyA* and *xyB*.

| Property | Description |
|---|---|
| 'figure points' | points from the lower left corner of the figure |
| 'figure pixels' | pixels from the lower left corner of the figure |
| 'figure fraction' | 0, 0 is lower left of figure and 1, 1 is upper right |
| 'subfigure points' | points from the lower left corner of the subfigure |
| 'subfigure pixels' | pixels from the lower left corner of the subfigure |
| 'subfigure fraction' | fraction of the subfigure, 0, 0 is lower left. |
| 'axes points' | points from lower left corner of the Axes |
| 'axes pixels' | pixels from lower left corner of the Axes |
| 'axes fraction' | 0, 0 is lower left of Axes and 1, 1 is upper right |
| 'data' | use the coordinate system of the object being annotated (default) |
| 'offset points' | offset (in points) from the *xy* value |
| 'polar' | you can specify *theta*, *r* for the annotation, even in cartesian plots. Note that if you are using a polar Axes, you do not need to specify polar for the coordinate system since that is the native "data" coordinate system. |

Alternatively they can be set to any valid *Transform*.

Note that 'subfigure pixels' and 'figure pixels' are the same for the parent figure, so users who want code that is usable in a subfigure can use 'subfigure pixels'.

**Note:** Using *ConnectionPatch* across two *Axes* instances is not directly compatible with *constrained layout*. Add the artist directly to the *Figure* instead of adding it to a specific Axes, or exclude it from the layout using `con.set_in_layout(False)`.

```
fig, ax = plt.subplots(1, 2, constrained_layout=True)
con = ConnectionPatch(..., axesA=ax[0], axesB=ax[1])
fig.add_artist(con)
```

**draw**(*renderer*)

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

**Parameters**

**renderer**

[*RendererBase* subclass.]

### Notes

This method is overridden in the Artist subclasses.

**get_annotation_clip**()

Return the clipping behavior.

See *set_annotation_clip* for the meaning of the return value.

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *annotation_clip=<UNSET>*, *antialiased=<UNSET>*, *arrowstyle=<UNSET>*, *capstyle=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *color=<UNSET>*, *connectionstyle=<UNSET>*, *edgecolor=<UNSET>*, *facecolor=<UNSET>*, *fill=<UNSET>*, *gid=<UNSET>*, *hatch=<UNSET>*, *in_layout=<UNSET>*, *joinstyle=<UNSET>*, *label=<UNSET>*, *linestyle=<UNSET>*, *linewidth=<UNSET>*, *mouseover=<UNSET>*, *mutation_aspect=<UNSET>*, *mutation_scale=<UNSET>*, *patchA=<UNSET>*, *patchB=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *positions=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a |
| *alpha* | scalar or None |
| *animated* | bool |
| *annotation_clip* | bool or None |
| *antialiased* or aa | bool or None |
| *arrowstyle* | str or *ArrowStyle* |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* | *color* |
| *connectionstyle* | [ 'arc3' \| 'angle3' \| 'angle' \| 'arc' \| 'bar' ] |

Table 78 – continued from previous p

| Property | Description |
|---|---|
| *edgecolor* or ec | *color* or None |
| *facecolor* or fc | *color* or None |
| *figure* | *Figure* |
| *fill* | bool |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float or None |
| *mouseover* | bool |
| *mutation_aspect* | float |
| *mutation_scale* | float |
| *patchA* | *patches.Patch* |
| *patchB* | *patches.Patch* |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *positions* | unknown |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *zorder* | float |

**set_annotation_clip**(*b*)

Set the annotation's clipping behavior.

**Parameters**

**b**

[bool or None]

- True: The annotation will be clipped when `self.xy` is outside the Axes.
- False: The annotation will always be drawn.
- None: The annotation will be clipped when `self.xy` is outside the Axes and `self.xycoords == "data"`.

### Examples using `matplotlib.patches.ConnectionPatch`

- *Bar of pie*
- *Multiple Axes animation*
- *Constrained layout guide*
- *Annotations*

## matplotlib.patches.ConnectionStyle

**class** matplotlib.patches.**ConnectionStyle**(*stylename*, ***kwargs*)

Bases: `_Style`

*ConnectionStyle* is a container class which defines several connectionstyle classes, which is used to create a path between two points. These are mainly used with *FancyArrowPatch*.

A connectionstyle object can be either created as:

```
ConnectionStyle.Arc3(rad=0.2)
```

or:

```
ConnectionStyle("Arc3", rad=0.2)
```

or:

```
ConnectionStyle("Arc3, rad=0.2")
```

The following classes are defined

| Class | Name | Attrs |
|---|---|---|
| Arc3 | `arc3` | rad=0.0 |
| Angle3 | `angle3` | angleA=90, angleB=0 |
| Angle | `angle` | angleA=90, angleB=0, rad=0.0 |
| Arc | `arc` | angleA=0, angleB=0, armA=None, armB=None, rad=0.0 |
| Bar | `bar` | armA=0.0, armB=0.0, fraction=0.3, angle=None |

An instance of any connection style class is a callable object, whose call signature is:

```
__call__(self, posA, posB,
         patchA=None, patchB=None,
         shrinkA=2., shrinkB=2.)
```

and it returns a *Path* instance. *posA* and *posB* are tuples of (x, y) coordinates of the two points to be connected. *patchA* (or *patchB*) is given, the returned path is clipped so that it start (or end) from the boundary of the patch. The path is further shrunk by *shrinkA* (or *shrinkB*) which is given in points.

Return the instance of the subclass with the given style name.

**class Angle**(*angleA=90*, *angleB=0*, *rad=0.0*)

Bases: _Base

Creates a piecewise continuous quadratic Bézier path between two points. The path has a one passing-through point placed at the intersecting point of two lines which cross the start and end point, and have a slope of *angleA* and *angleB*, respectively. The connecting edges are rounded with *rad*.

**Parameters**

**angleA**

[float] Starting angle of the path.

**angleB**

[float] Ending angle of the path.

**rad**

[float] Rounding radius of the edge.

**connect**(*posA*, *posB*)

**class Angle3**(*angleA=90*, *angleB=0*)

Bases: _Base

Creates a simple quadratic Bézier curve between two points. The middle control point is placed at the intersecting point of two lines which cross the start and end point, and have a slope of *angleA* and *angleB*, respectively.

**Parameters**

**angleA**

[float] Starting angle of the path.

**angleB**

[float] Ending angle of the path.

**connect**(*posA*, *posB*)

**class Arc**(*angleA=0*, *angleB=0*, *armA=None*, *armB=None*, *rad=0.0*)

Bases: _Base

Creates a piecewise continuous quadratic Bézier path between two points. The path can have two passing-through points, a point placed at the distance of *armA* and angle of *angleA* from point A, another point with respect to point B. The edges are rounded with *rad*.

**Parameters**

**angleA**

[float] Starting angle of the path.

**angleB**

[float] Ending angle of the path.

**armA**

[float or None] Length of the starting arm.

**armB**

[float or None] Length of the ending arm.

**rad**

[float] Rounding radius of the edges.

**connect**(*posA*, *posB*)

**class Arc3**(*rad=0.0*)

Bases: _Base

Creates a simple quadratic Bézier curve between two points. The curve is created so that the middle control point (C1) is located at the same distance from the start (C0) and end points(C2) and the distance of the C1 to the line connecting C0-C2 is *rad* times the distance of C0-C2.

**Parameters**

**rad**

[float] Curvature of the curve.

**connect**(*posA*, *posB*)

**class Bar**(*armA=0.0*, *armB=0.0*, *fraction=0.3*, *angle=None*)

Bases: _Base

A line with *angle* between A and B with *armA* and *armB*. One of the arms is extended so that they are connected in a right angle. The length of *armA* is determined by (*armA* + *fraction* x AB distance). Same for *armB*.

**Parameters**

**armA**

[float] Minimum length of armA.

**armB**

[float] Minimum length of armB.

**fraction**

[float] A fraction of the distance between two points that will be added to armA and armB.

**angle**

[float or None] Angle of the connecting line (if None, parallel to A and B).

**connect**(*posA*, *posB*)

## matplotlib.patches.Ellipse

**class** matplotlib.patches.**Ellipse**(*xy*, *width*, *height*, *, *angle=0*, ***kwargs*)

Bases: *Patch*

A scale-free ellipse.

**Parameters**

**xy**

[(float, float)] xy coordinates of ellipse centre.

**width**

[float] Total length (diameter) of horizontal axis.

**height**

[float] Total length (diameter) of vertical axis.

**angle**

[float, default: 0] Rotation in degrees anti-clockwise.

### Notes

Valid keyword arguments are:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n |
| *alpha* | unknown |
| *animated* | bool |
| *antialiased* or aa | bool or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* | *color* |
| *edgecolor* or ec | *color* or None |
| *facecolor* or fc | *color* or None |
| *figure* | *Figure* |
| *fill* | bool |

Table 79 – continued from previous page

| Property | Description |
|---|---|
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float or None |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *zorder* | float |

**property angle**

Return the angle of the ellipse.

**property center**

Return the center of the ellipse.

**get_angle**()

Return the angle of the ellipse.

**get_center**()

Return the center of the ellipse.

**get_co_vertices**()

Return the co-vertices coordinates of the ellipse.

The definition can be found here

New in version 3.8.

**get_corners**()

Return the corners of the ellipse bounding box.

The bounding box orientation is moving anti-clockwise from the lower left corner defined before rotation.

**get_height**()

Return the height of the ellipse.

**get_patch_transform**()

Return the *Transform* instance mapping patch coordinates to data coordinates.

For example, one may define a patch of a circle which represents a radius of 5 by providing coordinates for a unit circle, and a transform which scales the coordinates (the patch coordinate) by 5.

**get_path**()

Return the path of the ellipse.

**get_vertices**()

Return the vertices coordinates of the ellipse.

The definition can be found here

New in version 3.8.

**get_width**()

Return the width of the ellipse.

**property height**

Return the height of the ellipse.

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *angle=<UNSET>*, *animated=<UNSET>*, *antialiased=<UNSET>*, *capstyle=<UNSET>*, *center=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *color=<UNSET>*, *edgecolor=<UNSET>*, *facecolor=<UNSET>*, *fill=<UNSET>*, *gid=<UNSET>*, *hatch=<UNSET>*, *height=<UNSET>*, *in_layout=<UNSET>*, *joinstyle=<UNSET>*, *label=<UNSET>*, *linestyle=<UNSET>*, *linewidth=<UNSET>*, *mouseover=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *width=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a |
| *alpha* | scalar or None |
| *angle* | float |
| *animated* | bool |
| *antialiased* or aa | bool or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *center* | (float, float) |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* | *color* |
| *edgecolor* or ec | *color* or None |

Table 80 – continued from previous p

| Property | Description |
|---|---|
| *facecolor* or fc | *color* or None |
| *figure* | *Figure* |
| *fill* | bool |
| *gid* | str |
| *hatch* | {'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *height* | float |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float or None |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *width* | float |
| *zorder* | float |

**set_angle**(*angle*)

Set the angle of the ellipse.

**Parameters**

**angle**

[float]

**set_center**(*xy*)

Set the center of the ellipse.

**Parameters**

**xy**

[(float, float)]

**set_height**(*height*)

Set the height of the ellipse.

**Parameters**

**height**

[float]

**set_width**(*width*)

Set the width of the ellipse.

**Parameters**

**width**

[float]

**property width**

Return the width of the ellipse.

### Examples using `matplotlib.patches.Ellipse`

- *Clipping images with patches*
- *Axes box aspect*
- *Plot a confidence ellipse of a two-dimensional dataset*
- *Scale invariant angle label*
- *Annotating Plots*
- *AnnotationBbox demo*
- *Reference for Matplotlib artists*
- *Dolphins*
- *Ellipse with orientation arrow demo*
- *Ellipse Demo*
- *Hatch demo*
- *Circles, Wedges and Polygons*
- *ggplot style sheet*
- *Grayscale style sheet*
- *Style sheets reference*
- *Anatomy of a figure*
- *Looking Glass*
- *Anchored Artists*
- *Custom projection*
- *Packed-bubble chart*

- *Draw flat objects in 3D plot*
- *Radar chart (aka spider or star chart)*
- *Ellipse with units*
- *Annotate Explain*
- *Simple Annotate01*
- *Transformations Tutorial*
- *Legend guide*
- *Annotations*

### matplotlib.patches.FancyArrow

**class** matplotlib.patches.**FancyArrow**(*x*, *y*, *dx*, *dy*, *, *width=0.001*, *length_includes_head=False*, *head_width=None*, *head_length=None*, *shape='full'*, *overhang=0*, *head_starts_at_zero=False*, ***kwargs*)

Bases: *Polygon*

Like Arrow, but lets you set head width and head height independently.

**Parameters**

**x, y**

[float] The x and y coordinates of the arrow base.

**dx, dy**

[float] The length of the arrow along x and y direction.

**width**

[float, default: 0.001] Width of full arrow tail.

**length_includes_head**

[bool, default: False] True if head is to be counted in calculating the length.

**head_width**

[float or None, default: 3*width] Total width of the full arrow head.

**head_length**

[float or None, default: 1.5*head_width] Length of arrow head.

**shape**

[{'full', 'left', 'right'}, default: 'full'] Draw the left-half, right-half, or full arrow.

**overhang**

[float, default: 0] Fraction that the arrow is swept back (0 overhang means triangular shape). Can be negative or greater than one.

**head_starts_at_zero**

[bool, default: False] If True, the head starts being drawn at coordinate 0 instead of ending at coordinate 0.

****kwargs**

*Patch* properties:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and return |
| *alpha* | unknown |
| *animated* | bool |
| *antialiased* or aa | bool or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* | *color* |
| *edgecolor* or ec | *color* or None |
| *facecolor* or fc | *color* or None |
| *figure* | *Figure* |
| *fill* | bool |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float or None |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *zorder* | float |

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *antialiased=<UNSET>*, *capstyle=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *closed=<UNSET>*, *color=<UNSET>*, *data=<UNSET>*, *edgecolor=<UNSET>*, *facecolor=<UNSET>*, *fill=<UNSET>*, *gid=<UNSET>*, *hatch=<UNSET>*, *in_layout=<UNSET>*, *joinstyle=<UNSET>*, *label=<UNSET>*, *linestyle=<UNSET>*, *linewidth=<UNSET>*, *mouseover=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *xy=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* or aa | bool or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *closed* | bool |
| *color* | *color* |
| *data* | unknown |
| *edgecolor* or ec | *color* or None |
| *facecolor* or fc | *color* or None |
| *figure* | *Figure* |
| *fill* | bool |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float or None |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *xy* | (N, 2) array-like |

Table 82 – continued from previous p

| Property | Description |
|---|---|
| *zorder* | float |

**set_data**(*, *x=None*, *y=None*, *dx=None*, *dy=None*, *width=None*, *head_width=None*, *head_length=None*)

Set *FancyArrow* x, y, dx, dy, width, head_with, and head_length. Values left as None will not be updated.

**Parameters**

**x, y**

[float or None, default: None] The x and y coordinates of the arrow base.

**dx, dy**

[float or None, default: None] The length of the arrow along x and y direction.

**width**

[float or None, default: None] Width of full arrow tail.

**head_width**

[float or None, default: None] Total width of the full arrow head.

**head_length**

[float or None, default: None] Length of arrow head.

### Examples using matplotlib.patches.FancyArrow

- *Arrow guide*

### matplotlib.patches.FancyArrowPatch

**class** matplotlib.patches.**FancyArrowPatch**(*posA=None*, *posB=None*, *, *path=None*, *arrowstyle='simple'*, *connectionstyle='arc3'*, *patchA=None*, *patchB=None*, *shrinkA=2*, *shrinkB=2*, *mutation_scale=1*, *mutation_aspect=1*, ***kwargs*)

Bases: *Patch*

A fancy arrow patch.

It draws an arrow using the *ArrowStyle*. It is primarily used by the *annotate* method. For most purposes, use the annotate method for drawing arrows.

The head and tail positions are fixed at the specified start and end points of the arrow, but the size and shape (in display coordinates) of the arrow does not change when the axis is moved or zoomed.

There are two ways for defining an arrow:

- If *posA* and *posB* are given, a path connecting two points is created according to *connectionstyle*. The path will be clipped with *patchA* and *patchB* and further shrunken by *shrinkA* and *shrinkB*. An arrow is drawn along this resulting path using the *arrowstyle* parameter.
- Alternatively if *path* is provided, an arrow is drawn along this path and *patchA*, *patchB*, *shrinkA*, and *shrinkB* are ignored.

**Parameters**

**posA, posB**

[(float, float), default: None] (x, y) coordinates of arrow tail and arrow head respectively.

**path**

[`Path`, default: None] If provided, an arrow is drawn along this path and *patchA*, *patchB*, *shrinkA*, and *shrinkB* are ignored.

**arrowstyle**

[str or `ArrowStyle`, default: 'simple'] The `ArrowStyle` with which the fancy arrow is drawn. If a string, it should be one of the available arrowstyle names, with optional comma-separated attributes. The optional attributes are meant to be scaled with the *mutation_scale*. The following arrow styles are available:

| Class | Nam | Attrs |
|---|---|---|
| Curve | - | None |
| CurveA | <- | head_length=0.4, head_width=0.2, widthA=1.0, widthB=1.0, lengthA=0.2, lengthB=0.2, angleA=0, angleB=0, scaleA=None, scaleB=None |
| CurveB | -> | head_length=0.4, head_width=0.2, widthA=1.0, widthB=1.0, lengthA=0.2, lengthB=0.2, angleA=0, angleB=0, scaleA=None, scaleB=None |
| CurveA | <-> | head_length=0.4, head_width=0.2, widthA=1.0, widthB=1.0, lengthA=0.2, lengthB=0.2, angleA=0, angleB=0, scaleA=None, scaleB=None |
| Curve-FilledA | <\|- | head_length=0.4, head_width=0.2, widthA=1.0, widthB=1.0, lengthA=0.2, lengthB=0.2, angleA=0, angleB=0, scaleA=None, scaleB=None |
| Curve-FilledB | -\|> | head_length=0.4, head_width=0.2, widthA=1.0, widthB=1.0, lengthA=0.2, lengthB=0.2, angleA=0, angleB=0, scaleA=None, scaleB=None |
| Curve-FilledAl | <\|-\| | head_length=0.4, head_width=0.2, widthA=1.0, widthB=1.0, lengthA=0.2, lengthB=0.2, angleA=0, angleB=0, scaleA=None, scaleB=None |
| Brack-etA | ]- | widthA=1.0, lengthA=0.2, angleA=0 |
| Brack-etB | -[ | widthB=1.0, lengthB=0.2, angleB=0 |
| Brack-etAB | ]-[ | widthA=1.0, lengthA=0.2, angleA=0, widthB=1.0, lengthB=0.2, angleB=0 |
| BarAB | \|-\| | widthA=1.0, angleA=0, widthB=1.0, angleB=0 |
| Brack-etCurve | ]-> | widthA=1.0, lengthA=0.2, angleA=None |
| Curve-Bracket | <-[ | widthB=1.0, lengthB=0.2, angleB=None |
| Sim-ple | sim-ple | head_length=0.5, head_width=0.5, tail_width=0.2 |
| Fancy | fanc | head_length=0.4, head_width=0.4, tail_width=0.4 |
| Wedge | wedg | tail_width=0.3, shrink_factor=0.5 |

**connectionstyle**

[str or *ConnectionStyle* or None, optional, default: 'arc3'] The *ConnectionStyle* with which *posA* and *posB* are connected. If a string, it should be one of the available connectionstyle names, with optional comma-separated attributes. The following connection styles are available:

| Class | Name | Attrs |
|---|---|---|
| Arc3 | `arc3` | rad=0.0 |
| Angle3 | `angle3` | angleA=90, angleB=0 |
| Angle | `angle` | angleA=90, angleB=0, rad=0.0 |
| Arc | `arc` | angleA=0, angleB=0, armA=None, armB=None, rad=0.0 |
| Bar | `bar` | armA=0.0, armB=0.0, fraction=0.3, angle=None |

**patchA, patchB**

[*Patch*, default: None] Head and tail patches, respectively.

**shrinkA, shrinkB**

[float, default: 2] Shrink amount, in points, of the tail and head of the arrow respectively.

**mutation_scale**

[float, default: 1] Value with which attributes of *arrowstyle* (e.g., *head_length*) will be scaled.

**mutation_aspect**

[None or float, default: None] The height of the rectangle will be squeezed by this value before the mutation and the mutated box will be stretched by the inverse of it.

**Other Parameters**

****kwargs**

[*Patch* properties, optional] Here is a list of available *Patch* properties:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and retur |
| *alpha* | unknown |
| *animated* | bool |
| *antialiased* or aa | bool or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* | *color* |
| *edgecolor* or ec | *color* or None |
| *facecolor* or fc | *color* or None |
| *figure* | *Figure* |

Table 83 – continued from previc

| Property | Description |
|---|---|
| *fill* | bool |
| *gid* | str |
| *hatch* | {'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float or None |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *zorder* | float |

In contrast to other patches, the default `capstyle` and `joinstyle` for *FancyArrowPatch* are set to `"round"`.

**draw**(*renderer*)

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

**Parameters**

**renderer**

[*RendererBase* subclass.]

### Notes

This method is overridden in the Artist subclasses.

**get_arrowstyle**()

Return the arrowstyle object.

**get_connectionstyle**()

Return the *ConnectionStyle* used.

**get_mutation_aspect**()

Return the aspect ratio of the bbox mutation.

**get_mutation_scale**()

Return the mutation scale.

**Returns**

**scalar**

**get_path**()

Return the path of the arrow in the data coordinates.

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *antialiased=<UNSET>*, *arrowstyle=<UNSET>*, *capstyle=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *color=<UNSET>*, *connectionstyle=<UNSET>*, *edgecolor=<UNSET>*, *facecolor=<UNSET>*, *fill=<UNSET>*, *gid=<UNSET>*, *hatch=<UNSET>*, *in_layout=<UNSET>*, *joinstyle=<UNSET>*, *label=<UNSET>*, *linestyle=<UNSET>*, *linewidth=<UNSET>*, *mouseover=<UNSET>*, *mutation_aspect=<UNSET>*, *mutation_scale=<UNSET>*, *patchA=<UNSET>*, *patchB=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *positions=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* or aa | bool or None |
| *arrowstyle* | str or *ArrowStyle* |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* | *color* |
| *connectionstyle* | [ 'arc3' \| 'angle3' \| 'angle' \| 'arc' \| 'bar' ] |
| *edgecolor* or ec | *color* or None |
| *facecolor* or fc | *color* or None |
| *figure* | *Figure* |
| *fill* | bool |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |

Table 84 – continued from previous p

| Property | Description |
|---|---|
| *linewidth* or lw | float or None |
| *mouseover* | bool |
| *mutation_aspect* | float |
| *mutation_scale* | float |
| *patchA* | *patches.Patch* |
| *patchB* | *patches.Patch* |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *positions* | unknown |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *zorder* | float |

**set_arrowstyle**(*arrowstyle=None*, ***kwargs*)

Set the arrow style, possibly with further attributes.

Attributes from the previous arrow style are not reused.

Without argument (or with `arrowstyle=None`), the available box styles are returned as a human-readable string.

**Parameters**

**arrowstyle**

[str or *ArrowStyle*] The style of the arrow: either a *ArrowStyle* instance, or a string, which is the style name and optionally comma separated attributes (e.g. "Fancy,head_length=0.2"). Such a string is used to construct a *ArrowStyle* object, as documented in that class.

The following arrow styles are available:

%(ArrowStyle:table_and_accepts)s

****kwargs**

Additional attributes for the arrow style. See the table above for supported parameters.

### Examples

```
set_arrowstyle("Fancy,head_length=0.2")
set_arrowstyle("fancy", head_length=0.2)
```

**set_connectionstyle**(*connectionstyle=None*, ***kwargs*)

Set the connection style, possibly with further attributes.

Attributes from the previous connection style are not reused.

Without argument (or with `connectionstyle=None`), the available box styles are returned as a human-readable string.

**Parameters**

**connectionstyle**

[str or *ConnectionStyle*] The style of the connection: either a *ConnectionStyle* instance, or a string, which is the style name and optionally comma separated attributes (e.g. "Arc,armA=30,rad=10"). Such a string is used to construct a *ConnectionStyle* object, as documented in that class.

The following connection styles are available:

| Class | Name | Attrs |
|---|---|---|
| Arc3 | `arc3` | rad=0.0 |
| Angle3 | `angle3` | angleA=90, angleB=0 |
| Angle | `angle` | angleA=90, angleB=0, rad=0.0 |
| Arc | `arc` | angleA=0, angleB=0, armA=None, armB=None, rad=0.0 |
| Bar | `bar` | armA=0.0, armB=0.0, fraction=0.3, angle=None |

****kwargs**

Additional attributes for the connection style. See the table above for supported parameters.

### Examples

```
set_connectionstyle("Arc,armA=30,rad=10")
set_connectionstyle("arc", armA=30, rad=10)
```

**set_mutation_aspect**(*aspect*)

Set the aspect ratio of the bbox mutation.

**Parameters**

**aspect**

[float]

**set_mutation_scale**(*scale*)

Set the mutation scale.

**Parameters**

**scale**

[float]

**set_patchA**(*patchA*)

Set the tail patch.

**Parameters**

**patchA**

[*patches.Patch*]

**set_patchB**(*patchB*)

Set the head patch.

**Parameters**

**patchB**

[*patches.Patch*]

**set_positions**(*posA*, *posB*)

Set the start and end positions of the connecting path.

**Parameters**

**posA, posB**

[None, tuple] (x, y) coordinates of arrow tail and arrow head respectively. If None use current value.

## Examples using `matplotlib.patches.FancyArrowPatch`

- *Bar of pie*
- *Angle annotations on bracket arrows*
- *Arrow guide*
- *Multiple Axes animation*
- *Connection styles for annotations*

- *Annotations*

### matplotlib.patches.FancyBboxPatch

**class** matplotlib.patches.**FancyBboxPatch**(*xy*, *width*, *height*, *boxstyle='round'*, ***, *mutation_scale=1*, *mutation_aspect=1*, ***kwargs*)

Bases: *Patch*

A fancy box around a rectangle with lower left at $xy = (x, y)$ with specified width and height.

*FancyBboxPatch* is similar to *Rectangle*, but it draws a fancy box around the rectangle. The transformation of the rectangle box to the fancy box is delegated to the style classes defined in *BoxStyle*.

**Parameters**

**xy**

[(float, float)] The lower left corner of the box.

**width**

[float] The width of the box.

**height**

[float] The height of the box.

**boxstyle**

[str or *BoxStyle*] The style of the fancy box. This can either be a *BoxStyle* instance or a string of the style name and optionally comma separated attributes (e.g. "Round, pad=0.2"). This string is passed to *BoxStyle* to construct a *BoxStyle* object. See there for a full documentation.

The following box styles are available:

| Class | Name | Attrs |
|---|---|---|
| Square | square | pad=0.3 |
| Circle | circle | pad=0.3 |
| Ellipse | ellipse | pad=0.3 |
| LArrow | larrow | pad=0.3 |
| RArrow | rarrow | pad=0.3 |
| DArrow | darrow | pad=0.3 |
| Round | round | pad=0.3, rounding_size=None |
| Round4 | round4 | pad=0.3, rounding_size=None |
| Sawtooth | sawtooth | pad=0.3, tooth_size=None |
| Roundtooth | roundtooth | pad=0.3, tooth_size=None |

**mutation_scale**

[float, default: 1] Scaling factor applied to the attributes of the box style (e.g. pad or rounding_size).

**mutation_aspect**

[float, default: 1] The height of the rectangle will be squeezed by this value before the mutation and the mutated box will be stretched by the inverse of it. For example, this allows different horizontal and vertical padding.

**Other Parameters**

****kwargs**

[*Patch* properties]

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and return |
| *alpha* | unknown |
| *animated* | bool |
| *antialiased* or aa | bool or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* | *color* |
| *edgecolor* or ec | *color* or None |
| *facecolor* or fc | *color* or None |
| *figure* | *Figure* |
| *fill* | bool |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float or None |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |

Table 85 – continued from previc

| Property | Description |
|---|---|
| *zorder* | float |

**get_bbox**()

Return the *Bbox*.

**get_boxstyle**()

Return the boxstyle object.

**get_height**()

Return the height of the rectangle.

**get_mutation_aspect**()

Return the aspect ratio of the bbox mutation.

**get_mutation_scale**()

Return the mutation scale.

**get_path**()

Return the mutated path of the rectangle.

**get_width**()

Return the width of the rectangle.

**get_x**()

Return the left coord of the rectangle.

**get_y**()

Return the bottom coord of the rectangle.

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *antialiased=<UNSET>*, *bounds=<UNSET>*, *boxstyle=<UNSET>*, *capstyle=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *color=<UNSET>*, *edgecolor=<UNSET>*, *facecolor=<UNSET>*, *fill=<UNSET>*, *gid=<UNSET>*, *hatch=<UNSET>*, *height=<UNSET>*, *in_layout=<UNSET>*, *joinstyle=<UNSET>*, *label=<UNSET>*, *linestyle=<UNSET>*, *linewidth=<UNSET>*, *mouseover=<UNSET>*, *mutation_aspect=<UNSET>*, *mutation_scale=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *width=<UNSET>*, *x=<UNSET>*, *y=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a |

Table 86 – continued from previous p

| Property | Description |
|---|---|
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* or aa | bool or None |
| *bounds* | unknown |
| *boxstyle* | [ 'square' \| 'circle' \| 'ellipse' \| 'larrow' \| 'rarrow' \| 'darrow' \| 'round' \| 'round4' \| 'sawto |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* | *color* |
| *edgecolor* or ec | *color* or None |
| *facecolor* or fc | *color* or None |
| *figure* | *Figure* |
| *fill* | bool |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *height* | float |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float or None |
| *mouseover* | bool |
| *mutation_aspect* | float |
| *mutation_scale* | float |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *width* | float |
| *x* | float |
| *y* | float |
| *zorder* | float |

**set_bounds**(**args*)

Set the bounds of the rectangle.

Call signatures:

```
set_bounds(left, bottom, width, height)
set_bounds((left, bottom, width, height))
```

**Parameters**

**left, bottom**

[float] The coordinates of the bottom left corner of the rectangle.

**width, height**

[float] The width/height of the rectangle.

**set_boxstyle**(*boxstyle=None*, ***kwargs*)

Set the box style, possibly with further attributes.

Attributes from the previous box style are not reused.

Without argument (or with `boxstyle=None`), the available box styles are returned as a human-readable string.

**Parameters**

**boxstyle**

[str or *BoxStyle*] The style of the box: either a *BoxStyle* instance, or a string, which is the style name and optionally comma separated attributes (e.g. "Round,pad=0.2"). Such a string is used to construct a *BoxStyle* object, as documented in that class.

The following box styles are available:

| Class | Name | Attrs |
|---|---|---|
| Square | `square` | pad=0.3 |
| Circle | `circle` | pad=0.3 |
| Ellipse | `ellipse` | pad=0.3 |
| LArrow | `larrow` | pad=0.3 |
| RArrow | `rarrow` | pad=0.3 |
| DArrow | `darrow` | pad=0.3 |
| Round | `round` | pad=0.3, rounding_size=None |
| Round4 | `round4` | pad=0.3, rounding_size=None |
| Sawtooth | `sawtooth` | pad=0.3, tooth_size=None |
| Roundtooth | `roundtooth` | pad=0.3, tooth_size=None |

****kwargs**

Additional attributes for the box style. See the table above for supported parameters.

### Examples

```
set_boxstyle("Round,pad=0.2")
set_boxstyle("round", pad=0.2)
```

**set_height**(*h*)

Set the rectangle height.

**Parameters**

**h**

[float]

**set_mutation_aspect**(*aspect*)

Set the aspect ratio of the bbox mutation.

**Parameters**

**aspect**

[float]

**set_mutation_scale**(*scale*)

Set the mutation scale.

**Parameters**

**scale**

[float]

**set_width**(*w*)

Set the rectangle width.

**Parameters**

**w**

[float]

**set_x**(*x*)

Set the left coord of the rectangle.

**Parameters**

**x**

[float]

**set_y**(*y*)

Set the bottom coord of the rectangle.

**Parameters**

**y**

[float]

### Examples using `matplotlib.patches.FancyBboxPatch`

- *Reference for Matplotlib artists*
- *Drawing fancy boxes*
- *Annotate Text Arrow*

## matplotlib.patches.Patch

**class** matplotlib.patches.**Patch**(*, *edgecolor=None*, *facecolor=None*, *color=None*, *linewidth=None*, *linestyle=None*, *antialiased=None*, *hatch=None*, *fill=True*, *capstyle=None*, *joinstyle=None*, ***kwargs*)

Bases: *Artist*

A patch is a 2D artist with a face color and an edge color.

If any of *edgecolor*, *facecolor*, *linewidth*, or *antialiased* are *None*, they default to their rc params setting.

The following kwarg properties are supported

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) a |
| *alpha* | unknown |
| *animated* | bool |
| *antialiased* or aa | bool or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* | *color* |
| *edgecolor* or ec | *color* or None |
| *facecolor* or fc | *color* or None |
| *figure* | *Figure* |
| *fill* | bool |
| *gid* | str |

Table 87 – continued from previous page

| Property | Description |
|---|---|
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float or None |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *zorder* | float |

**contains**(*mouseevent*, *radius=None*)

Test whether the mouse event occurred in the patch.

**Parameters**

**mouseevent**

[*MouseEvent*] Where the user clicked.

**radius**

[float, optional] Additional margin on the patch in target coordinates of *Patch.get_transform*. See *Path.contains_point* for further details.

If None, the default value depends on the state of the object:

- If *Artist.get_picker* is a number, the default is that value. This is so that picking works as expected.
- Otherwise if the edge color has a non-zero alpha, the default is half of the linewidth. This is so that all the colored pixels are "in" the patch.
- Finally, if the edge has 0 alpha, the default is 0. This is so that patches without a stroked edge do not have points outside of the filled region report as "in" due to an invisible edge.

**Returns**

**(bool, empty dict)**

**contains_point**(*point*, *radius=None*)

Return whether the given point is inside the patch.

**Parameters**

**point**

[(float, float)] The point (x, y) to check, in target coordinates of `.Patch.get_transform()`. These are display coordinates for patches that are added to a figure or Axes.

**radius**

[float, optional] Additional margin on the patch in target coordinates of *Patch.get_transform*. See *Path.contains_point* for further details.

If `None`, the default value depends on the state of the object:

- If *Artist.get_picker* is a number, the default is that value. This is so that picking works as expected.
- Otherwise if the edge color has a non-zero alpha, the default is half of the linewidth. This is so that all the colored pixels are "in" the patch.
- Finally, if the edge has 0 alpha, the default is 0. This is so that patches without a stroked edge do not have points outside of the filled region report as "in" due to an invisible edge.

**Returns**

**bool**

## Notes

The proper use of this method depends on the transform of the patch. Isolated patches do not have a transform. In this case, the patch creation coordinates and the point coordinates match. The following example checks that the center of a circle is within the circle

```
>>> center = 0, 0
>>> c = Circle(center, radius=1)
>>> c.contains_point(center)
True
```

The convention of checking against the transformed patch stems from the fact that this method is predominantly used to check if display coordinates (e.g. from mouse events) are within the patch. If you want to do the above check with data coordinates, you have to properly transform them first:

```
>>> center = 0, 0
>>> c = Circle(center, radius=3)
>>> plt.gca().add_patch(c)
```

(continues on next page)

(continued from previous page)

```
>>> transformed_interior_point = c.get_data_transform().transform((0,
 ↪ 2))
>>> c.contains_point(transformed_interior_point)
True
```

**contains_points**(*points*, *radius=None*)

Return whether the given points are inside the patch.

**Parameters**

**points**

[(N, 2) array] The points to check, in target coordinates of `self.get_transform()`. These are display coordinates for patches that are added to a figure or Axes. Columns contain x and y values.

**radius**

[float, optional] Additional margin on the patch in target coordinates of *Patch.get_transform*. See *Path.contains_point* for further details.

If `None`, the default value depends on the state of the object:

- If *Artist.get_picker* is a number, the default is that value. This is so that picking works as expected.
- Otherwise if the edge color has a non-zero alpha, the default is half of the linewidth. This is so that all the colored pixels are "in" the patch.
- Finally, if the edge has 0 alpha, the default is 0. This is so that patches without a stroked edge do not have points outside of the filled region report as "in" due to an invisible edge.

**Returns**

**length-N bool array**

## Notes

The proper use of this method depends on the transform of the patch. See the notes on *Patch.contains_point*.

**draw**(*renderer*)

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

**Parameters**

**renderer**

[*RendererBase* subclass.]

### Notes

This method is overridden in the Artist subclasses.

**property fill**
Return whether the patch is filled.

**get_aa**()
Alias for *get_antialiased*.

**get_antialiased**()
Return whether antialiasing is used for drawing.

**get_capstyle**()
Return the capstyle.

**get_data_transform**()
Return the *Transform* mapping data coordinates to physical coordinates.

**get_ec**()
Alias for *get_edgecolor*.

**get_edgecolor**()
Return the edge color.

**get_extents**()
Return the *Patch*'s axis-aligned extents as a *Bbox*.

**get_facecolor**()
Return the face color.

**get_fc**()
Alias for *get_facecolor*.

**get_fill**()
Return whether the patch is filled.

**get_hatch**()
Return the hatching pattern.

**get_joinstyle**()
Return the joinstyle.

**get_linestyle**()
Return the linestyle.

**get_linewidth**()
Return the line width in points.

**get_ls**()
Alias for *get_linestyle*.

**get_lw**()

Alias for *get_linewidth*.

**get_patch_transform**()

Return the *Transform* instance mapping patch coordinates to data coordinates.

For example, one may define a patch of a circle which represents a radius of 5 by providing coordinates for a unit circle, and a transform which scales the coordinates (the patch coordinate) by 5.

**get_path**()

Return the path of this patch.

**get_transform**()

Return the *Transform* applied to the *Patch*.

**get_verts**()

Return a copy of the vertices used in this patch.

If the patch contains Bézier curves, the curves will be interpolated by line segments. To access the curves as curves, use *get_path*.

**get_window_extent**(*renderer=None*)

Get the artist's bounding box in display space.

The bounding box' width and height are nonnegative.

Subclasses should override for inclusion in the bounding box "tight" calculation. Default is to return an empty bounding box at 0, 0.

Be careful when using this function, the results will not update if the artist window extent of the artist changes. The extent can change due to any changes in the transform stack, such as changing the Axes limits, the figure size, or the canvas used (as is done when saving a figure). This can lead to unexpected behavior where interactive figures will look fine on the screen, but will save incorrectly.

**set**(**, agg_filter=<UNSET>, alpha=<UNSET>, animated=<UNSET>, antialiased=<UNSET>, capstyle=<UNSET>, clip_box=<UNSET>, clip_on=<UNSET>, clip_path=<UNSET>, color=<UNSET>, edgecolor=<UNSET>, facecolor=<UNSET>, fill=<UNSET>, gid=<UNSET>, hatch=<UNSET>, in_layout=<UNSET>, joinstyle=<UNSET>, label=<UNSET>, linestyle=<UNSET>, linewidth=<UNSET>, mouseover=<UNSET>, path_effects=<UNSET>, picker=<UNSET>, rasterized=<UNSET>, sketch_params=<UNSET>, snap=<UNSET>, transform=<UNSET>, url=<UNSET>, visible=<UNSET>, zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m |

Table 88 – continued from previous pa

| Property | Description |
|---|---|
| *alpha* | unknown |
| *animated* | bool |
| *antialiased* | bool or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* | *color* |
| *edgecolor* | *color* or None |
| *facecolor* | *color* or None |
| *figure* | *Figure* |
| *fill* | bool |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* | float or None |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *zorder* | float |

**set_aa**(*aa*)

Alias for *set_antialiased*.

**set_alpha**(*alpha*)

Set the alpha value used for blending - not supported on all backends.

**Parameters**

**alpha**

[scalar or None] *alpha* must be within the 0-1 range, inclusive.

**set_antialiased**(*aa*)

Set whether to use antialiased rendering.

**Parameters**

**aa**

[bool or None]

**set_capstyle**(*s*)

Set the *CapStyle*.

The default capstyle is 'round' for *FancyArrowPatch* and 'butt' for all other patches.

**Parameters**

**s**

[*CapStyle* or {'butt', 'projecting', 'round'}]

**set_color**(*c*)

Set both the edgecolor and the facecolor.

**Parameters**

**c**

[*color*]

**See also:**

***Patch.set_facecolor*, *Patch.set_edgecolor***

For setting the edge or face color individually.

**set_ec**(*color*)

Alias for *set_edgecolor*.

**set_edgecolor**(*color*)

Set the patch edge color.

**Parameters**

**color**

[*color* or None]

**set_facecolor**(*color*)

Set the patch face color.

**Parameters**

**color**

[*color* or None]

**set_fc**(*color*)

Alias for *set_facecolor*.

**set_fill**(*b*)

Set whether to fill the patch.

**Parameters**

**b**

[bool]

**set_hatch**(*hatch*)

Set the hatching pattern.

*hatch* can be one of:

```
/   - diagonal hatching
\   - back diagonal
|   - vertical
-   - horizontal
+   - crossed
x   - crossed diagonal
o   - small circle
O   - large circle
.   - dots
*   - stars
```

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

**Parameters**

**hatch**

[{'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}]

**set_joinstyle**(*s*)

Set the *JoinStyle*.

The default joinstyle is 'round' for *FancyArrowPatch* and 'miter' for all other patches.

**Parameters**

**s**

[*JoinStyle* or {'miter', 'round', 'bevel'}]

**set_linestyle**(*ls*)

Set the patch linestyle.

| linestyle | description |
|---|---|
| `'-'` or `'solid'` | solid line |
| `'--'` or `'dashed'` | dashed line |
| `'-.'` or `'dashdot'` | dash-dotted line |
| `':'` or `'dotted'` | dotted line |
| `'none'`, `'None'`, `' '`, or `''` | draw nothing |

Alternatively a dash tuple of the following form can be provided:

```
(offset, onoffseq)
```

where `onoffseq` is an even length tuple of on and off ink in points.

**Parameters**

**ls**

[{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}] The line style.

**set_linewidth**(*w*)

Set the patch linewidth in points.

**Parameters**

**w**

[float or None]

**set_ls**(*ls*)

Alias for *set_linestyle*.

**set_lw**(*w*)

Alias for *set_linewidth*.

**update_from**(*other*)

Copy properties from *other* to *self*.

**zorder = 1**

### Examples using `matplotlib.patches.Patch`

- *Curve with error band*
- *Stairs Demo*
- *Clipping images with patches*
- *Many ways to plot images*
- *Axes box aspect*

- *Controlling view limits using margins and sticky_edges*
- *Axes Zoom Effect*
- *Boxplots*
- *Plot a confidence ellipse of a two-dimensional dataset*
- *Creating boxes from error bars using PatchCollection*
- *Bar of pie*
- *Scale invariant angle label*
- *Angle annotations on bracket arrows*
- *Annotating Plots*
- *Composing Custom Legends*
- *AnnotationBbox demo*
- *Using a text as a Path*
- *Text Rotation Mode*
- *Placing text boxes*
- *Text alignment*
- *List of named colors*
- *Arrow guide*
- *Reference for Matplotlib artists*
- *Compound path*
- *Dolphins*
- *Mmh Donuts!!!*
- *Ellipse with orientation arrow demo*
- *Ellipse Demo*
- *Drawing fancy boxes*
- *Hatch demo*
- *Hatch style reference*
- *Circles, Wedges and Polygons*
- *PathPatch object*
- *Bezier Curve*
- *ggplot style sheet*
- *Grayscale style sheet*
- *Style sheets reference*

- *Inset locator demo*
- *Anatomy of a figure*
- *Firefox*
- *Integral as the area under a curve*
- *Multiple Axes animation*
- *Looking Glass*
- *Path editor*
- *Pick event demo*
- *Poly Editor*
- *Trifinder Event Demo*
- *Viewlims*
- *Anchored Artists*
- *Changing colors of lines intersecting a box*
- *Custom projection*
- *Building histograms using Rectangles and PolyCollections*
- *Matplotlib logo*
- *Packed-bubble chart*
- *SVG filter pie*
- *TickedStroke patheffect*
- *Draw flat objects in 3D plot*
- *Hinton diagrams*
- *Ishikawa Diagram*
- *Radar chart (aka spider or star chart)*
- *SkewT-logP diagram: using transforms and custom projections*
- *Artist tests*
- *Ellipse with units*
- *Menu*
- *Annotate Explain*
- *Simple Annotate01*
- *Path Tutorial*
- *Transformations Tutorial*
- *Legend guide*

- *Specifying colors*
- *Text properties and layout*
- *Annotations*

### matplotlib.patches.PathPatch

**class** matplotlib.patches.**PathPatch**(*path*, ***kwargs*)

Bases: *Patch*

A general polycurve path patch.

*path* is a *Path* object.

Valid keyword arguments are:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n |
| *alpha* | unknown |
| *animated* | bool |
| *antialiased* or aa | bool or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* | *color* |
| *edgecolor* or ec | *color* or None |
| *facecolor* or fc | *color* or None |
| *figure* | *Figure* |
| *fill* | bool |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float or None |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |

Table 89 – continued from previous page

| Property | Description |
|---|---|
| *zorder* | float |

**get_path**()

Return the path of this patch.

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *antialiased=<UNSET>*, *capstyle=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *color=<UNSET>*, *edgecolor=<UNSET>*, *facecolor=<UNSET>*, *fill=<UNSET>*, *gid=<UNSET>*, *hatch=<UNSET>*, *in_layout=<UNSET>*, *joinstyle=<UNSET>*, *label=<UNSET>*, *linestyle=<UNSET>*, *linewidth=<UNSET>*, *mouseover=<UNSET>*, *path=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* or aa | bool or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* | *color* |
| *edgecolor* or ec | *color* or None |
| *facecolor* or fc | *color* or None |
| *figure* | *Figure* |
| *fill* | bool |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float or None |
| *mouseover* | bool |
| *path* | unknown |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |

Table 90 – continued from previous p

| Property | Description |
|---|---|
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *zorder* | float |

**set_path**(*path*)

### Examples using `matplotlib.patches.PathPatch`

- *Curve with error band*
- *Stairs Demo*
- *Many ways to plot images*
- *Box plots with custom fill colors*
- *Using a text as a Path*
- *Reference for Matplotlib artists*
- *Compound path*
- *Dolphins*
- *Mmh Donuts!!!*
- *PathPatch object*
- *Bezier Curve*
- *Firefox*
- *Path editor*
- *Building histograms using Rectangles and PolyCollections*
- *Matplotlib logo*
- *TickedStroke patheffect*
- *Draw flat objects in 3D plot*
- *Path Tutorial*

### matplotlib.patches.StepPatch

**class** matplotlib.patches.**StepPatch**(*values*, *edges*, *, *orientation='vertical'*, *baseline=0*, ***kwargs*)

Bases: *PathPatch*

A path patch describing a stepwise constant function.

By default, the path is not closed and starts and stops at baseline value.

**Parameters**

**values**

[array-like] The step heights.

**edges**

[array-like] The edge positions, with `len(edges) == len(vals) + 1`, between which the curve takes on vals values.

**orientation**

[{'vertical', 'horizontal'}, default: 'vertical'] The direction of the steps. Vertical means that *values* are along the y-axis, and edges are along the x-axis.

**baseline**

[float, array-like or None, default: 0] The bottom value of the bounding edges or when `fill=True`, position of lower edge. If *fill* is True or an array is passed to *baseline*, a closed path is drawn.

****kwargs**

*Patch* properties:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and retur |
| *alpha* | unknown |
| *animated* | bool |
| *antialiased* or aa | bool or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* | *color* |
| *edgecolor* or ec | *color* or None |
| *facecolor* or fc | *color* or None |
| *figure* | *Figure* |
| *fill* | bool |
| *gid* | str |

Table 91 – continued from previc

| Property | Description |
|---|---|
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float or None |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *zorder* | float |

**get_data**()

Get *StepPatch* values, edges and baseline as namedtuple.

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *antialiased=<UNSET>*, *capstyle=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *color=<UNSET>*, *data=<UNSET>*, *edgecolor=<UNSET>*, *facecolor=<UNSET>*, *fill=<UNSET>*, *gid=<UNSET>*, *hatch=<UNSET>*, *in_layout=<UNSET>*, *joinstyle=<UNSET>*, *label=<UNSET>*, *linestyle=<UNSET>*, *linewidth=<UNSET>*, *mouseover=<UNSET>*, *path=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a ( |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* or aa | bool or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* | *color* |
| *data* | 1D array-like or None |

Table 92 – continued from previous p

| Property | Description |
|---|---|
| *edgecolor* or ec | *color* or None |
| *facecolor* or fc | *color* or None |
| *figure* | *Figure* |
| *fill* | bool |
| *gid* | str |
| *hatch* | {'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float or None |
| *mouseover* | bool |
| *path* | unknown |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *zorder* | float |

**set_data**(*values=None*, *edges=None*, *baseline=None*)

Set *StepPatch* values, edges and baseline.

**Parameters**

**values**

[1D array-like or None] Will not update values, if passing None

**edges**

[1D array-like, optional]

**baseline**

[float, 1D array-like or None]

### Examples using `matplotlib.patches.StepPatch`

- *Stairs Demo*

## matplotlib.patches.Polygon

**class** matplotlib.patches.**Polygon**(*xy*, *, *closed=True*, ***kwargs*)

Bases: *Patch*

A general polygon patch.

**Parameters**

**xy**

[(N, 2) array]

**closed**

[bool, default: True] Whether the polygon is closed (i.e., has identical start and end points).

****kwargs**

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and return |
| *alpha* | unknown |
| *animated* | bool |
| *antialiased* or aa | bool or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* | *color* |
| *edgecolor* or ec | *color* or None |
| *facecolor* or fc | *color* or None |
| *figure* | *Figure* |
| *fill* | bool |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float or None |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |

Table 93 – continued from previc

| Property | Description |
|---|---|
| *picker* | None or bool or float or callable |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *zorder* | float |

**get_closed**()

Return whether the polygon is closed.

**get_path**()

Get the *Path* of the polygon.

**get_xy**()

Get the vertices of the path.

**Returns**

**(N, 2) array**

The coordinates of the vertices.

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *antialiased=<UNSET>*, *capstyle=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *closed=<UNSET>*, *color=<UNSET>*, *edgecolor=<UNSET>*, *facecolor=<UNSET>*, *fill=<UNSET>*, *gid=<UNSET>*, *hatch=<UNSET>*, *in_layout=<UNSET>*, *joinstyle=<UNSET>*, *label=<UNSET>*, *linestyle=<UNSET>*, *linewidth=<UNSET>*, *mouseover=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *xy=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* or aa | bool or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |

Table 94 – continued from previous p

| Property | Description |
|---|---|
| *clip_path* | Patch or (Path, Transform) or None |
| *closed* | bool |
| *color* | *color* |
| *edgecolor* or ec | *color* or None |
| *facecolor* or fc | *color* or None |
| *figure* | *Figure* |
| *fill* | bool |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float or None |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *xy* | (N, 2) array-like |
| *zorder* | float |

**set_closed**(*closed*)

Set whether the polygon is closed.

**Parameters**

**closed**

[bool] True if the polygon is closed

**set_xy**(*xy*)

Set the vertices of the polygon.

**Parameters**

**xy**

[(N, 2) array-like] The coordinates of the vertices.

### Notes

Unlike `Path`, we do not ignore the last input vertex. If the polygon is meant to be closed, and the last point of the polygon is not equal to the first, we assume that the user has not explicitly passed a `CLOSEPOLY` vertex, and add it ourselves.

**property xy**

The vertices of the path as a (N, 2) array.

## Examples using `matplotlib.patches.Polygon`

- *Controlling view limits using margins and sticky_edges*
- *Boxplots*
- *Arrow guide*
- *Hatch demo*
- *Circles, Wedges and Polygons*
- *floating_axes features*
- *Integral as the area under a curve*
- *Poly Editor*
- *Trifinder Event Demo*
- *Ishikawa Diagram*
- *Annotations*

## matplotlib.patches.Rectangle

**class** matplotlib.patches.**Rectangle**(*xy*, *width*, *height*, *, *angle=0.0*, *rotation_point='xy'*, ***kwargs*)

Bases: `Patch`

A rectangle defined via an anchor point *xy* and its *width* and *height*.

The rectangle extends from `xy[0]` to `xy[0] + width` in x-direction and from `xy[1]` to `xy[1] + height` in y-direction.

```
:                +------------------+
:                |                  |
:              height               |
:                |                  |
:               (xy)---- width -----+
```

One may picture *xy* as the bottom left corner, but which corner *xy* is actually depends on the direction of the axis and the sign of *width* and *height*; e.g. *xy* would be the bottom right corner if the x-axis was inverted or if *width* was negative.

### Parameters

**xy**

[(float, float)] The anchor point.

**width**

[float] Rectangle width.

**height**

[float] Rectangle height.

**angle**

[float, default: 0] Rotation in degrees anti-clockwise about the rotation point.

**rotation_point**

[{'xy', 'center', (number, number)}, default: 'xy'] If `'xy'`, rotate around the anchor point. If `'center'` rotate around the center. If 2-tuple of number, rotate around this coordinate.

### Other Parameters

****kwargs**

[*Patch* properties]

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and return |
| *alpha* | unknown |
| *animated* | bool |
| *antialiased* or aa | bool or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* | *color* |
| *edgecolor* or ec | *color* or None |
| *facecolor* or fc | *color* or None |
| *figure* | *Figure* |
| *fill* | bool |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float or None |

Table 95 – continued from previc

| Property | Description |
|---|---|
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *zorder* | float |

**get_angle**()

Get the rotation angle in degrees.

**get_bbox**()

Return the *Bbox*.

**get_center**()

Return the centre of the rectangle.

**get_corners**()

Return the corners of the rectangle, moving anti-clockwise from (x0, y0).

**get_height**()

Return the height of the rectangle.

**get_patch_transform**()

Return the *Transform* instance mapping patch coordinates to data coordinates.

For example, one may define a patch of a circle which represents a radius of 5 by providing coordinates for a unit circle, and a transform which scales the coordinates (the patch coordinate) by 5.

**get_path**()

Return the vertices of the rectangle.

**get_width**()

Return the width of the rectangle.

**get_x**()

Return the left coordinate of the rectangle.

**get_xy**()

Return the left and bottom coords of the rectangle as a tuple.

**get_y**()

Return the bottom coordinate of the rectangle.

**property rotation_point**

The rotation point of the patch.

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *angle=<UNSET>*, *animated=<UNSET>*, *antialiased=<UNSET>*, *bounds=<UNSET>*, *capstyle=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *color=<UNSET>*, *edgecolor=<UNSET>*, *facecolor=<UNSET>*, *fill=<UNSET>*, *gid=<UNSET>*, *hatch=<UNSET>*, *height=<UNSET>*, *in_layout=<UNSET>*, *joinstyle=<UNSET>*, *label=<UNSET>*, *linestyle=<UNSET>*, *linewidth=<UNSET>*, *mouseover=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *width=<UNSET>*, *x=<UNSET>*, *xy=<UNSET>*, *y=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a |
| *alpha* | scalar or None |
| *angle* | unknown |
| *animated* | bool |
| *antialiased* or aa | bool or None |
| *bounds* | (left, bottom, width, height) |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* | *color* |
| *edgecolor* or ec | *color* or None |
| *facecolor* or fc | *color* or None |
| *figure* | *Figure* |
| *fill* | bool |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *height* | unknown |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float or None |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |

Table 96 – continued from previous p

| Property | Description |
|---|---|
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *width* | unknown |
| *x* | unknown |
| *xy* | (float, float) |
| *y* | unknown |
| *zorder* | float |

**set_angle**(*angle*)

Set the rotation angle in degrees.

The rotation is performed anti-clockwise around *xy*.

**set_bounds**(**args*)

Set the bounds of the rectangle as *left*, *bottom*, *width*, *height*.

The values may be passed as separate parameters or as a tuple:

```
set_bounds(left, bottom, width, height)
set_bounds((left, bottom, width, height))
```

**set_height**(*h*)

Set the height of the rectangle.

**set_width**(*w*)

Set the width of the rectangle.

**set_x**(*x*)

Set the left coordinate of the rectangle.

**set_xy**(*xy*)

Set the left and bottom coordinates of the rectangle.

**Parameters**

**xy**

[(float, float)]

**set_y**(*y*)

Set the bottom coordinate of the rectangle.

**property xy**

Return the left and bottom coords of the rectangle as a tuple.

### Examples using `matplotlib.patches.Rectangle`

- *Creating boxes from error bars using PatchCollection*
- *Histograms*
- *Text Rotation Mode*
- *Text alignment*
- *List of named colors*
- *Reference for Matplotlib artists*
- *Hatch style reference*
- *Style sheets reference*
- *Inset locator demo*
- *Pick event demo*
- *Viewlims*
- *Changing colors of lines intersecting a box*
- *Matplotlib logo*
- *Hinton diagrams*
- *Fig Axes Customize Simple*
- *Artist tests*
- *Menu*
- *Artist tutorial*
- *Transformations Tutorial*
- *Legend guide*
- *Specifying colors*
- *Text properties and layout*

## matplotlib.patches.RegularPolygon

**class** matplotlib.patches.**RegularPolygon**(*xy*, *numVertices*, *, *radius=5*, *orientation=0*, ***kwargs*)

Bases: `Patch`

A regular polygon patch.

**Parameters**

**xy**

[(float, float)] The center position.

**numVertices**

[int] The number of vertices.

**radius**

[float] The distance from the center to each of the vertices.

**orientation**

[float] The polygon rotation angle (in radians).

****kwargs**

*Patch* properties:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and retur |
| *alpha* | unknown |
| *animated* | bool |
| *antialiased* or aa | bool or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* | *color* |
| *edgecolor* or ec | *color* or None |
| *facecolor* or fc | *color* or None |
| *figure* | *Figure* |
| *fill* | bool |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float or None |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |

Table 97 – continued from previc

| Property | Description |
|---|---|
| *zorder* | float |

**get_patch_transform**()

Return the *Transform* instance mapping patch coordinates to data coordinates.

For example, one may define a patch of a circle which represents a radius of 5 by providing coordinates for a unit circle, and a transform which scales the coordinates (the patch coordinate) by 5.

**get_path**()

Return the path of this patch.

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *antialiased=<UNSET>*, *capstyle=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *color=<UNSET>*, *edgecolor=<UNSET>*, *facecolor=<UNSET>*, *fill=<UNSET>*, *gid=<UNSET>*, *hatch=<UNSET>*, *in_layout=<UNSET>*, *joinstyle=<UNSET>*, *label=<UNSET>*, *linestyle=<UNSET>*, *linewidth=<UNSET>*, *mouseover=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a ( |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* or aa | bool or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* | *color* |
| *edgecolor* or ec | *color* or None |
| *facecolor* or fc | *color* or None |
| *figure* | *Figure* |
| *fill* | bool |
| *gid* | str |
| *hatch* | {'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |

Table 98 – continued from previous p

| Property | Description |
|---|---|
| *linewidth* or lw | float or None |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *zorder* | float |

### Examples using `matplotlib.patches.RegularPolygon`

- *Reference for Matplotlib artists*
- *Radar chart (aka spider or star chart)*

## matplotlib.patches.Shadow

**class** matplotlib.patches.**Shadow**(*patch*, *ox*, *oy*, *, *shade=0.7*, ***kwargs*)

Bases: *Patch*

Create a shadow of the given *patch*.

By default, the shadow will have the same face color as the *patch*, but darkened. The darkness can be controlled by *shade*.

**Parameters**

**patch**

[*Patch*] The patch to create the shadow for.

**ox, oy**

[float] The shift of the shadow in data coordinates, scaled by a factor of dpi/72.

**shade**

[float, default: 0.7] How the darkness of the shadow relates to the original color. If 1, the shadow is black, if 0, the shadow has the same color as the *patch*.

New in version 3.8.

****kwargs**

Properties of the shadow patch. Supported keys are:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and retur |
| *alpha* | unknown |
| *animated* | bool |
| *antialiased* or aa | bool or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* | *color* |
| *edgecolor* or ec | *color* or None |
| *facecolor* or fc | *color* or None |
| *figure* | *Figure* |
| *fill* | bool |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float or None |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *zorder* | float |

**draw**(*renderer*)

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

**Parameters**

**renderer**

[*RendererBase* subclass.]

### Notes

This method is overridden in the Artist subclasses.

**get_patch_transform**()

Return the *Transform* instance mapping patch coordinates to data coordinates.

For example, one may define a patch of a circle which represents a radius of 5 by providing coordinates for a unit circle, and a transform which scales the coordinates (the patch coordinate) by 5.

**get_path**()

Return the path of this patch.

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *antialiased=<UNSET>*, *capstyle=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *color=<UNSET>*, *edgecolor=<UNSET>*, *facecolor=<UNSET>*, *fill=<UNSET>*, *gid=<UNSET>*, *hatch=<UNSET>*, *in_layout=<UNSET>*, *joinstyle=<UNSET>*, *label=<UNSET>*, *linestyle=<UNSET>*, *linewidth=<UNSET>*, *mouseover=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* or aa | bool or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* | *color* |
| *edgecolor* or ec | *color* or None |
| *facecolor* or fc | *color* or None |
| *figure* | *Figure* |
| *fill* | bool |
| *gid* | str |
| *hatch* | {'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float or None |

Table 100 – continued from previous

| Property | Description |
|---|---|
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *zorder* | float |

### Examples using `matplotlib.patches.Shadow`

- *Using a text as a Path*
- *SVG filter pie*

### matplotlib.patches.Wedge

**class** matplotlib.patches.**Wedge**(*center*, *r*, *theta1*, *theta2*, *, *width=None*, ***kwargs*)

Bases: *Patch*

Wedge shaped patch.

A wedge centered at *x*, *y* center with radius *r* that sweeps *theta1* to *theta2* (in degrees). If *width* is given, then a partial wedge is drawn from inner radius *r* - *width* to outer radius *r*.

Valid keyword arguments are:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n |
| *alpha* | unknown |
| *animated* | bool |
| *antialiased* or aa | bool or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* | *color* |
| *edgecolor* or ec | *color* or None |

Table 101 – continued from previous page

| Property | Description |
|---|---|
| *facecolor* or fc | *color* or None |
| *figure* | *Figure* |
| *fill* | bool |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float or None |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *zorder* | float |

**get_path**()

Return the path of this patch.

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *antialiased=<UNSET>*, *capstyle=<UNSET>*, *center=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *color=<UNSET>*, *edgecolor=<UNSET>*, *facecolor=<UNSET>*, *fill=<UNSET>*, *gid=<UNSET>*, *hatch=<UNSET>*, *in_layout=<UNSET>*, *joinstyle=<UNSET>*, *label=<UNSET>*, *linestyle=<UNSET>*, *linewidth=<UNSET>*, *mouseover=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *radius=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *theta1=<UNSET>*, *theta2=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *width=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* or aa | bool or None |

Table 102 – continued from previous

| Property | Description |
|---|---|
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *center* | unknown |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* | *color* |
| *edgecolor* or ec | *color* or None |
| *facecolor* or fc | *color* or None |
| *figure* | *Figure* |
| *fill* | bool |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float or None |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *radius* | unknown |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *theta1* | unknown |
| *theta2* | unknown |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *width* | unknown |
| *zorder* | float |

**set_center**(*center*)

**set_radius**(*radius*)

**set_theta1**(*theta1*)

**set_theta2**(*theta2*)

**set_width**(*width*)

### Examples using `matplotlib.patches.Wedge`

- *Labeling a pie and a donut*
- *Reference for Matplotlib artists*
- *Circles, Wedges and Polygons*
- *SVG filter pie*
- *Ishikawa Diagram*

### Functions

| | |
|---|---|
| `bbox_artist`(artist, renderer[, props, fill]) | A debug function to draw a rectangle around the bounding box returned by an artist's `Artist.get_window_extent` to test whether the artist is returning the correct bbox. |
| `draw_bbox`(bbox, renderer[, color, trans]) | A debug function to draw a rectangle around the bounding box returned by an artist's `Artist.get_window_extent` to test whether the artist is returning the correct bbox. |

### matplotlib.patches.bbox_artist

matplotlib.patches.**bbox_artist**(*artist*, *renderer*, *props=None*, *fill=True*)

A debug function to draw a rectangle around the bounding box returned by an artist's `Artist.get_window_extent` to test whether the artist is returning the correct bbox.

*props* is a dict of rectangle props with the additional property 'pad' that sets the padding around the bbox in points.

### matplotlib.patches.draw_bbox

matplotlib.patches.**draw_bbox**(*bbox*, *renderer*, *color='k'*, *trans=None*)

A debug function to draw a rectangle around the bounding box returned by an artist's `Artist.get_window_extent` to test whether the artist is returning the correct bbox.

### 7.2.38 matplotlib.path

A module for dealing with the polylines used throughout Matplotlib.

The primary class for polyline handling in Matplotlib is *Path*. Almost all vector drawing makes use of *Path*s somewhere in the drawing pipeline.

Whilst a *Path* instance itself cannot be drawn, some *Artist* subclasses, such as *PathPatch* and *PathCollection*, can be used for convenient *Path* visualisation.

**class** matplotlib.path.**Path**(*vertices*, *codes=None*, *_interpolation_steps=1*, *closed=False*, *readonly=False*)

Bases: object

A series of possibly disconnected, possibly closed, line and curve segments.

The underlying storage is made up of two parallel numpy arrays:

- *vertices*: an (N, 2) float array of vertices
- *codes*: an N-length numpy.uint8 array of path codes, or None

These two arrays always have the same length in the first dimension. For example, to represent a cubic curve, you must provide three vertices and three *CURVE4* codes.

The code types are:

- ***STOP***

  [1 vertex (ignored)] A marker for the end of the entire path (currently not required and ignored)

- ***MOVETO***

  [1 vertex] Pick up the pen and move to the given vertex.

- ***LINETO***

  [1 vertex] Draw a line from the current position to the given vertex.

- ***CURVE3***

  [1 control point, 1 endpoint] Draw a quadratic Bézier curve from the current position, with the given control point, to the given end point.

- ***CURVE4***

  [2 control points, 1 endpoint] Draw a cubic Bézier curve from the current position, with the given control points, to the given end point.

- ***CLOSEPOLY***

  [1 vertex (ignored)] Draw a line segment to the start point of the current polyline.

If *codes* is None, it is interpreted as a *MOVETO* followed by a series of *LINETO*.

Users of Path objects should not access the vertices and codes arrays directly. Instead, they should use *iter_segments* or *cleaned* to get the vertex/code pairs. This helps, in particular, to consistently handle the case of *codes* being None.

Some behavior of Path objects can be controlled by rcParams. See the rcParams whose keys start with 'path.'.

---

**Note:** The vertices and codes arrays should be treated as immutable -- there are a number of optimizations and assumptions made up front in the constructor that will not change when the data changes.

---

Create a new path with the given vertices and codes.

**Parameters**

**vertices**

[(N, 2) array-like] The path vertices, as an array, masked array or sequence of pairs. Masked values, if any, will be converted to NaNs, which are then handled correctly by the Agg PathIterator and other consumers of path data, such as *iter_segments()*.

**codes**

[array-like or None, optional] N-length array of integers representing the codes of the path. If not None, codes must be the same length as vertices. If None, *vertices* will be treated as a series of line segments.

**_interpolation_steps**

[int, optional] Used as a hint to certain projections, such as Polar, that this path should be linearly interpolated immediately before drawing. This attribute is primarily an implementation detail and is not intended for public use.

**closed**

[bool, optional] If *codes* is None and closed is True, vertices will be treated as line segments of a closed polygon. Note that the last vertex will then be ignored (as the corresponding code will be set to *CLOSEPOLY*).

**readonly**

[bool, optional] Makes the path behave in an immutable way and sets the vertices and codes as read-only arrays.

**CLOSEPOLY = 79**

**CURVE3 = 3**

**CURVE4 = 4**

**LINETO = 2**

**MOVETO = 1**

**NUM_VERTICES_FOR_CODE = {0: 1, 1: 1, 2: 1, 3: 2, 4: 3, 79: 1}**

A dictionary mapping Path codes to the number of vertices that the code expects.

**STOP = 0**

**classmethod arc**(*theta1*, *theta2*, *n=None*, *is_wedge=False*)

Return a `Path` for the unit circle arc from angles *theta1* to *theta2* (in degrees).

*theta2* is unwrapped to produce the shortest arc within 360 degrees. That is, if *theta2* > *theta1* + 360, the arc will be from *theta1* to *theta2* - 360 and not a full circle plus some extra overlap.

If *n* is provided, it is the number of spline segments to make. If *n* is not provided, the number of spline segments is determined based on the delta between *theta1* and *theta2*.

Masionobe, L. 2003. Drawing an elliptical arc using polylines, quadratic or cubic Bezier curves.

**classmethod circle**(*center=(0.0, 0.0)*, *radius=1.0*, *readonly=False*)

Return a `Path` representing a circle of a given radius and center.

**Parameters**

**center**

[(float, float), default: (0, 0)] The center of the circle.

**radius**

[float, default: 1] The radius of the circle.

**readonly**

[bool] Whether the created path should have the "readonly" argument set when creating the Path instance.

### Notes

The circle is approximated using 8 cubic Bézier curves, as described in

Lancaster, Don. Approximating a Circle or an Ellipse Using Four Bezier Cubic Splines.

**cleaned**(*transform=None*, *remove_nans=False*, *clip=None*, ***, *simplify=False*, *curves=False*, *stroke_width=1.0*, *snap=False*, *sketch=None*)

Return a new `Path` with vertices and codes cleaned according to the parameters.

**See also:**

***Path.iter_segments***

for details of the keyword arguments.

**clip_to_bbox**(*bbox*, *inside=True*)

Clip the path to the given bounding box.

The path must be made up of one or more closed polygons. This algorithm will not behave correctly for unclosed paths.

If *inside* is `True`, clip to the inside of the box, otherwise to the outside of the box.

**code_type**

alias of `uint8`

**property codes**

The list of codes in the *Path* as a 1D array.

Each code is one of *STOP*, *MOVETO*, *LINETO*, *CURVE3*, *CURVE4* or *CLOSEPOLY*. For codes that correspond to more than one vertex (*CURVE3* and *CURVE4*), that code will be repeated so that the length of *vertices* and *codes* is always the same.

**contains_path**(*path*, *transform=None*)

Return whether this (closed) path completely contains the given path.

If *transform* is not `None`, the path will be transformed before checking for containment.

**contains_point**(*point*, *transform=None*, *radius=0.0*)

Return whether the area enclosed by the path contains the given point.

The path is always treated as closed; i.e. if the last code is not *CLOSEPOLY* an implicit segment connecting the last vertex to the first vertex is assumed.

**Parameters**

**point**

[(float, float)] The point (x, y) to check.

**transform**

[*Transform*, optional] If not `None`, *point* will be compared to `self` transformed by *transform*; i.e. for a correct check, *transform* should transform the path into the coordinate system of *point*.

**radius**

[float, default: 0] Additional margin on the path in coordinates of *point*. The path is extended tangentially by *radius/2*; i.e. if you would draw the path with a linewidth of *radius*, all points on the line would still be considered to be contained in the area. Conversely, negative values shrink the area: Points on the imaginary line will be considered outside the area.

**Returns**

**bool**

### Notes

The current algorithm has some limitations:

- The result is undefined for points exactly at the boundary (i.e. at the path shifted by *radius/2*).
- The result is undefined if there is no enclosed area, i.e. all vertices are on a straight line.
- If bounding lines start to cross each other due to *radius* shift, the result is not guaranteed to be correct.

**contains_points**(*points*, *transform=None*, *radius=0.0*)

Return whether the area enclosed by the path contains the given points.

The path is always treated as closed; i.e. if the last code is not `CLOSEPOLY` an implicit segment connecting the last vertex to the first vertex is assumed.

**Parameters**

**points**

[(N, 2) array] The points to check. Columns contain x and y values.

**transform**

[`Transform`, optional] If not `None`, *points* will be compared to `self` transformed by *transform*; i.e. for a correct check, *transform* should transform the path into the coordinate system of *points*.

**radius**

[float, default: 0] Additional margin on the path in coordinates of *points*. The path is extended tangentially by *radius/2*; i.e. if you would draw the path with a linewidth of *radius*, all points on the line would still be considered to be contained in the area. Conversely, negative values shrink the area: Points on the imaginary line will be considered outside the area.

**Returns**

**length-N bool array**

### Notes

The current algorithm has some limitations:

- The result is undefined for points exactly at the boundary (i.e. at the path shifted by *radius/2*).
- The result is undefined if there is no enclosed area, i.e. all vertices are on a straight line.
- If bounding lines start to cross each other due to *radius* shift, the result is not guaranteed to be correct.

**copy**()

Return a shallow copy of the *Path*, which will share the vertices and codes with the source *Path*.

**deepcopy**(*memo=None*)

Return a deepcopy of the *Path*. The *Path* will not be readonly, even if the source *Path* is.

**get_extents**(*transform=None*, ***kwargs*)

Get Bbox of the path.

**Parameters**

**transform**

[*Transform*, optional] Transform to apply to path before computing extents, if any.

****kwargs**

Forwarded to *iter_bezier*.

**Returns**

**matplotlib.transforms.Bbox**

The extents of the path Bbox([[xmin, ymin], [xmax, ymax]])

**static hatch**(*hatchpattern*, *density=6*)

Given a hatch specifier, *hatchpattern*, generates a *Path* that can be used in a repeated hatching pattern. *density* is the number of lines per unit square.

**interpolated**(*steps*)

Return a new path resampled to length N x *steps*.

Codes other than *LINETO* are not handled correctly.

**intersects_bbox**(*bbox*, *filled=True*)

Return whether this path intersects a given *Bbox*.

If *filled* is True, then this also returns True if the path completely encloses the *Bbox* (i.e., the path is treated as filled).

The bounding box is always considered filled.

**intersects_path**(*other*, *filled=True*)

Return whether if this path intersects another given path.

If *filled* is True, then this also returns True if one path completely encloses the other (i.e., the paths are treated as filled).

**iter_bezier**(***kwargs*)

Iterate over each Bézier curve (lines included) in a *Path*.

**Parameters**

****kwargs**

Forwarded to *iter_segments*.

**Yields**

**B**

[*BezierSegment*] The Bézier curves that make up the current path. Note in particular that freestanding points are Bézier curves of order 0, and lines are Bézier curves of order 1 (with two control points).

**code**

[*code_type*] The code describing what kind of curve is being returned. *MOVETO*, *LINETO*, *CURVE3*, and *CURVE4* correspond to Bézier curves with 1, 2, 3, and 4 control points (respectively). *CLOSEPOLY* is a *LINETO* with the control points correctly chosen based on the start/end points of the current stroke.

**iter_segments**(*transform=None*, *remove_nans=True*, *clip=None*, *snap=False*, *stroke_width=1.0*, *simplify=None*, *curves=True*, *sketch=None*)

Iterate over all curve segments in the path.

Each iteration returns a pair `(vertices, code)`, where `vertices` is a sequence of 1-3 coordinate pairs, and `code` is a *Path* code.

Additionally, this method can provide a number of standard cleanups and conversions to the path.

**Parameters**

**transform**

[None or *Transform*] If not None, the given affine transformation will be applied to the path.

**remove_nans**

[bool, optional] Whether to remove all NaNs from the path and skip over them using MOVETO commands.

**clip**

[None or (float, float, float, float), optional] If not None, must be a four-tuple (x1, y1, x2, y2) defining a rectangle in which to clip the path.

**snap**

[None or bool, optional] If True, snap all nodes to pixels; if False, don't snap them. If None, snap if the path contains only segments parallel to the x or y axes, and no more than 1024 of them.

**stroke_width**

[float, optional] The width of the stroke being drawn (used for path snapping).

**simplify**

[None or bool, optional] Whether to simplify the path by removing vertices that do not affect its appearance. If None, use the *should_simplify* attribute. See also *rcParams["path.simplify"]* (default: `True`) and *rcParams["path.simplify_threshold"]* (default: `0.111111111111`).

**curves**

[bool, optional] If True, curve segments will be returned as curve segments. If False, all curves will be converted to line segments.

**sketch**

[None or sequence, optional] If not None, must be a 3-tuple of the form (scale, length, randomness), representing the sketch parameters.

**classmethod make_compound_path**(**args*)

Concatenate a list of *Path*s into a single *Path*, removing all *STOP*s.

**classmethod make_compound_path_from_polys**(*XY*)

Make a compound *Path* object to draw a number of polygons with equal numbers of sides.

**Parameters**

**XY**

[(numpolys, numsides, 2) array]

**property readonly**

`True` if the *Path* is read-only.

**property should_simplify**

`True` if the vertices array should be simplified.

**property simplify_threshold**

The fraction of a pixel difference below which vertices will be simplified out.

**to_polygons**(*transform=None*, *width=0*, *height=0*, *closed_only=True*)

Convert this path to a list of polygons or polylines. Each polygon/polyline is an (N, 2) array of vertices. In other words, each polygon has no *MOVETO* instructions or curves. This is useful for displaying in backends that do not support compound paths or Bézier curves.

If *width* and *height* are both non-zero then the lines will be simplified so that vertices outside of (0, 0), (width, height) will be clipped.

If *closed_only* is `True` (default), only closed polygons, with the last point being the same as the first point, will be returned. Any unclosed polylines in the path will be explicitly closed. If *closed_only* is `False`, any unclosed polygons in the path will be returned as unclosed polygons,

and the closed polygons will be returned explicitly closed by setting the last point to the same as the first point.

**transformed**(*transform*)

Return a transformed copy of the path.

**See also:**

***matplotlib.transforms.TransformedPath***

A specialized path class that will cache the transformed result and automatically update when the transform changes.

**classmethod unit_circle**()

Return the readonly *Path* of the unit circle.

For most cases, *Path.circle()* will be what you want.

**classmethod unit_circle_righthalf**()

Return a *Path* of the right half of a unit circle.

See *Path.circle* for the reference on the approximation used.

**classmethod unit_rectangle**()

Return a *Path* instance of the unit rectangle from (0, 0) to (1, 1).

**classmethod unit_regular_asterisk**(*numVertices*)

Return a *Path* for a unit regular asterisk with the given numVertices and radius of 1.0, centered at (0, 0).

**classmethod unit_regular_polygon**(*numVertices*)

Return a *Path* instance for a unit regular polygon with the given *numVertices* such that the circumscribing circle has radius 1.0, centered at (0, 0).

**classmethod unit_regular_star**(*numVertices*, *innerCircle=0.5*)

Return a *Path* for a unit regular star with the given numVertices and radius of 1.0, centered at (0, 0).

**property vertices**

The vertices of the *Path* as an (N, 2) array.

**classmethod wedge**(*theta1*, *theta2*, *n=None*)

Return a *Path* for the unit circle wedge from angles *theta1* to *theta2* (in degrees).

*theta2* is unwrapped to produce the shortest wedge within 360 degrees. That is, if *theta2* > *theta1* + 360, the wedge will be from *theta1* to *theta2* - 360 and not a full circle plus some extra overlap.

If *n* is provided, it is the number of spline segments to make. If *n* is not provided, the number of spline segments is determined based on the delta between *theta1* and *theta2*.

See *Path.arc* for the reference on the approximation used.

matplotlib.path.**get_path_collection_extents**(*master_transform*, *paths*, *transforms*, *offsets*, *offset_transform*)

Get bounding box of a *PathCollection*s internal objects.

That is, given a sequence of *Path*s, *Transform*s objects, and offsets, as found in a *PathCollection*, return the bounding box that encapsulates all of them.

**Parameters**

**master_transform**

[*Transform*] Global transformation applied to all paths.

**paths**

[list of *Path*]

**transforms**

[list of *Affine2DBase*] If non-empty, this overrides *master_transform*.

**offsets**

[(N, 2) array-like]

**offset_transform**

[*Affine2DBase*] Transform applied to the offsets before offsetting the path.

### Notes

The way that *paths*, *transforms* and *offsets* are combined follows the same method as for collections: each is iterated over independently, so if you have 3 paths (A, B, C), 2 transforms (α, β) and 1 offset (O), their combinations are as follows:

- (A, α, O)
- (B, β, O)
- (C, α, O)

## 7.2.39 matplotlib.patheffects

Defines classes for path effects. The path effects are supported in *Text*, *Line2D* and *Patch*.

**See also:**

*Path effects guide*

**class** matplotlib.patheffects.**AbstractPathEffect**(*offset=(0.0, 0.0)*)

Bases: object

A base class for path effects.

Subclasses should override the `draw_path` method to add effect functionality.

**Parameters**

**offset**

[(float, float), default: (0, 0)] The (x, y) offset to apply to the path, measured in points.

**draw_path**(*renderer*, *gc*, *tpath*, *affine*, *rgbFace=None*)

Derived should override this method. The arguments are the same as *matplotlib.backend_bases.RendererBase.draw_path()* except the first argument is a renderer.

**class** matplotlib.patheffects.**Normal**(*offset=(0.0, 0.0)*)

Bases: *AbstractPathEffect*

The "identity" PathEffect.

The Normal PathEffect's sole purpose is to draw the original artist with no special path effect.

**Parameters**

**offset**

[(float, float), default: (0, 0)] The (x, y) offset to apply to the path, measured in points.

**class** matplotlib.patheffects.**PathEffectRenderer**(*path_effects*, *renderer*)

Bases: *RendererBase*

Implements a Renderer which contains another renderer.

This proxy then intercepts draw calls, calling the appropriate *AbstractPathEffect* draw method.

---

**Note:** Not all methods have been overridden on this RendererBase subclass. It may be necessary to add further methods to extend the PathEffects capabilities further.

---

**Parameters**

**path_effects**

[iterable of *AbstractPathEffect*] The path effects which this renderer represents.

**renderer**

[*RendererBase* subclass]

**close_group**(*s*)

Close a grouping element with label *s*.

Only used by the SVG renderer.

**copy_with_path_effect** (*path_effects*)

**draw_markers** (*gc*, *marker_path*, *marker_trans*, *path*, **args*, ***kwargs*)

Draw a marker at each of *path*'s vertices (excluding control points).

The base (fallback) implementation makes multiple calls to *draw_path*. Backends may want to override this method in order to draw the marker only once and reuse it multiple times.

**Parameters**

**gc**

[*GraphicsContextBase*] The graphics context.

**marker_path**

[*Path*] The path for the marker.

**marker_trans**

[*Transform*] An affine transform applied to the marker.

**path**

[*Path*] The locations to draw the markers.

**trans**

[*Transform*] An affine transform applied to the path.

**rgbFace**

[*color*, optional]

**draw_path** (*gc*, *tpath*, *affine*, *rgbFace=None*)

Draw a *Path* instance using the given affine transform.

**draw_path_collection** (*gc*, *master_transform*, *paths*, **args*, ***kwargs*)

Draw a collection of *paths*.

Each path is first transformed by the corresponding entry in *all_transforms* (a list of (3, 3) matrices) and then by *master_transform*. They are then translated by the corresponding entry in *offsets*, which has been first transformed by *offset_trans*.

*facecolors*, *edgecolors*, *linewidths*, *linestyles*, and *antialiased* are lists that set the corresponding properties.

*offset_position* is unused now, but the argument is kept for backwards compatibility.

The base (fallback) implementation makes multiple calls to *draw_path*. Backends may want to override this in order to render each set of path data only once, and then reference that path multiple times with the different offsets, colors, styles etc. The generator methods `_iter_collection_raw_paths` and `_iter_collection` are provided to help with (and standardize) the implementation across backends. It is highly recommended to use those generators, so that changes to the behavior of *draw_path_collection* can be made globally.

**open_group**(*s*, *gid=None*)

Open a grouping element with label *s* and *gid* (if set) as id.

Only used by the SVG renderer.

**class** matplotlib.patheffects.**PathPatchEffect**(*offset=(0, 0)*, ***kwargs*)

Bases: *AbstractPathEffect*

Draws a *PathPatch* instance whose Path comes from the original PathEffect artist.

**Parameters**

**offset**

[(float, float), default: (0, 0)] The (x, y) offset to apply to the path, in points.

****kwargs**

All keyword arguments are passed through to the *PathPatch* constructor. The properties which cannot be overridden are "path", "clip_box" "transform" and "clip_path".

**draw_path**(*renderer*, *gc*, *tpath*, *affine*, *rgbFace*)

Derived should override this method. The arguments are the same as *matplotlib.backend_bases.RendererBase.draw_path()* except the first argument is a renderer.

**class** matplotlib.patheffects.**SimpleLineShadow**(*offset=(2, -2)*, *shadow_color='k'*, *alpha=0.3*, *rho=0.3*, ***kwargs*)

Bases: *AbstractPathEffect*

A simple shadow via a line.

**Parameters**

**offset**

[(float, float), default: (2, -2)] The (x, y) offset to apply to the path, in points.

**shadow_color**

[*color*, default: 'black'] The shadow color. A value of `None` takes the original artist's color with a scale factor of *rho*.

**alpha**

[float, default: 0.3] The alpha transparency of the created shadow patch.

**rho**

[float, default: 0.3] A scale factor to apply to the rgbFace color if *shadow_color* is `None`.

****kwargs**

Extra keywords are stored and passed through to `AbstractPathEffect._update_gc()`.

**draw_path**(*renderer*, *gc*, *tpath*, *affine*, *rgbFace*)

Overrides the standard draw_path to add the shadow offset and necessary color changes for the shadow.

**class** matplotlib.patheffects.**SimplePatchShadow**(*offset=(2, -2)*, *shadow_rgbFace=None*, *alpha=None*, *rho=0.3*, ***kwargs*)

Bases: *AbstractPathEffect*

A simple shadow via a filled patch.

**Parameters**

**offset**

[(float, float), default: (2, -2)] The (x, y) offset of the shadow in points.

**shadow_rgbFace**

[*color*] The shadow color.

**alpha**

[float, default: 0.3] The alpha transparency of the created shadow patch.

**rho**

[float, default: 0.3] A scale factor to apply to the rgbFace color if *shadow_rgbFace* is not specified.

****kwargs**

Extra keywords are stored and passed through to `AbstractPathEffect._update_gc()`.

**draw_path**(*renderer*, *gc*, *tpath*, *affine*, *rgbFace*)

Overrides the standard draw_path to add the shadow offset and necessary color changes for the shadow.

**class** matplotlib.patheffects.**Stroke**(*offset=(0, 0)*, ***kwargs*)

Bases: *AbstractPathEffect*

A line based PathEffect which re-draws a stroke.

The path will be stroked with its gc updated with the given keyword arguments, i.e., the keyword arguments should be valid gc parameter values.

**draw_path**(*renderer*, *gc*, *tpath*, *affine*, *rgbFace*)

Draw the path with updated gc.

**class** matplotlib.patheffects.**TickedStroke**(*offset=(0, 0)*, *spacing=10.0*, *angle=45.0*, *length=1.4142135623730951*, ***kwargs*)

Bases: *AbstractPathEffect*

A line-based PathEffect which draws a path with a ticked style.

This line style is frequently used to represent constraints in optimization. The ticks may be used to indicate that one side of the line is invalid or to represent a closed boundary of a domain (i.e. a wall or the edge of a pipe).

The spacing, length, and angle of ticks can be controlled.

This line style is sometimes referred to as a hatched line.

See also the *TickedStroke patheffect* example.

**Parameters**

**offset**

[(float, float), default: (0, 0)] The (x, y) offset to apply to the path, in points.

**spacing**

[float, default: 10.0] The spacing between ticks in points.

**angle**

[float, default: 45.0] The angle between the path and the tick in degrees. The angle is measured as if you were an ant walking along the curve, with zero degrees pointing directly ahead, 90 to your left, -90 to your right, and 180 behind you. To change side of the ticks, change sign of the angle.

**length**

[float, default: 1.414] The length of the tick relative to spacing. Recommended length = 1.414 (sqrt(2)) when angle=45, length=1.0 when angle=90 and length=2.0 when angle=60.

****kwargs**

Extra keywords are stored and passed through to `AbstractPathEffect._update_gc()`.

## Examples

See *TickedStroke patheffect*.

**draw_path**(*renderer*, *gc*, *tpath*, *affine*, *rgbFace*)

Draw the path with updated gc.

**class** matplotlib.patheffects.**withSimplePatchShadow**(*offset=(2, -2)*, *shadow_rgbFace=None*, *alpha=None*, *rho=0.3*, ***kwargs*)

Bases: *SimplePatchShadow*

A shortcut PathEffect for applying *SimplePatchShadow* and then drawing the original Artist.

With this class you can use

```
artist.set_path_effects([patheffects.withSimplePatchShadow()])
```

as a shortcut for

```
artist.set_path_effects([patheffects.SimplePatchShadow(),
                         patheffects.Normal()])
```

**Parameters**

**offset**

[(float, float), default: (2, -2)] The (x, y) offset of the shadow in points.

**shadow_rgbFace**

[*color*] The shadow color.

**alpha**

[float, default: 0.3] The alpha transparency of the created shadow patch.

**rho**

[float, default: 0.3] A scale factor to apply to the rgbFace color if *shadow_rgbFace* is not specified.

****kwargs**

Extra keywords are stored and passed through to `AbstractPathEffect._update_gc()`.

**draw_path**(*renderer*, *gc*, *tpath*, *affine*, *rgbFace*)

Overrides the standard draw_path to add the shadow offset and necessary color changes for the shadow.

**class** matplotlib.patheffects.**withStroke**(*offset=(0, 0)*, ***kwargs*)

Bases: *Stroke*

A shortcut PathEffect for applying *Stroke* and then drawing the original Artist.

With this class you can use

```
artist.set_path_effects([patheffects.withStroke()])
```

as a shortcut for

```
artist.set_path_effects([patheffects.Stroke(),
                         patheffects.Normal()])
```

The path will be stroked with its gc updated with the given keyword arguments, i.e., the keyword arguments should be valid gc parameter values.

**draw_path**(*renderer*, *gc*, *tpath*, *affine*, *rgbFace*)

Draw the path with updated gc.

**class** matplotlib.patheffects.**withTickedStroke**(*offset=(0, 0)*, *spacing=10.0*, *angle=45.0*, *length=1.4142135623730951*, ***kwargs*)

Bases: *TickedStroke*

A shortcut PathEffect for applying *TickedStroke* and then drawing the original Artist.

With this class you can use

```
artist.set_path_effects([patheffects.withTickedStroke()])
```

as a shortcut for

```
artist.set_path_effects([patheffects.TickedStroke(),
                         patheffects.Normal()])
```

**Parameters**

**offset**

[(float, float), default: (0, 0)] The (x, y) offset to apply to the path, in points.

**spacing**

[float, default: 10.0] The spacing between ticks in points.

**angle**

[float, default: 45.0] The angle between the path and the tick in degrees. The angle is measured as if you were an ant walking along the curve, with zero degrees pointing directly ahead, 90 to your left, -90 to your right, and 180 behind you. To change side of the ticks, change sign of the angle.

**length**

[float, default: 1.414] The length of the tick relative to spacing. Recommended length = 1.414 (sqrt(2)) when angle=45, length=1.0 when angle=90 and length=2.0 when angle=60.

****kwargs**

Extra keywords are stored and passed through to `AbstractPathEffect._update_gc()`.

### Examples

See *TickedStroke patheffect*.

**draw_path**(*renderer*, *gc*, *tpath*, *affine*, *rgbFace*)

Draw the path with updated gc.

## 7.2.40 matplotlib.pyplot

*matplotlib.pyplot* is a state-based interface to matplotlib. It provides an implicit, MATLAB-like, way of plotting. It also opens figures on your screen, and acts as the figure GUI manager.

pyplot is mainly intended for interactive plots and simple cases of programmatic plot generation:

```
import numpy as np
import matplotlib.pyplot as plt

x = np.arange(0, 5, 0.1)
y = np.sin(x)
plt.plot(x, y)
```

The explicit object-oriented API is recommended for complex plots, though pyplot is still usually used to create the figure and often the Axes in the figure. See *pyplot.figure*, *pyplot.subplots*, and *pyplot.subplot_mosaic* to create figures, and *Axes API* for the plotting methods on an Axes:

```
import numpy as np
import matplotlib.pyplot as plt

x = np.arange(0, 5, 0.1)
y = np.sin(x)
fig, ax = plt.subplots()
ax.plot(x, y)
```

See *Matplotlib Application Interfaces (APIs)* for an explanation of the tradeoffs between the implicit and explicit interfaces.

### Managing Figure and Axes

| | |
|---|---|
| *axes* | Add an Axes to the current figure and make it the current Axes. |
| *cla* | Clear the current Axes. |
| *clf* | Clear the current figure. |
| *close* | Close a figure window. |
| *delaxes* | Remove an *Axes* (defaulting to the current Axes) from its figure. |
| *fignum_exists* | Return whether the figure with the given id exists. |
| *figure* | Create a new figure, or activate an existing figure. |
| *gca* | Get the current Axes. |
| *gcf* | Get the current figure. |
| *get_figlabels* | Return a list of existing figure labels. |
| *get_fignums* | Return a list of existing figure numbers. |
| *sca* | Set the current Axes to *ax* and the current Figure to the parent of *ax*. |
| *subplot* | Add an Axes to the current figure or retrieve an existing Axes. |
| *subplot2grid* | Create a subplot at a specific location inside a regular grid. |
| *subplot_mosaic* | Build a layout of Axes based on ASCII art or nested lists. |
| *subplots* | Create a figure and a set of subplots. |
| *twinx* | Make and return a second Axes that shares the *x*-axis. |
| *twiny* | Make and return a second Axes that shares the *y*-axis. |

### matplotlib.pyplot.axes

matplotlib.pyplot.**axes**(*arg=None*, ***kwargs*)

Add an Axes to the current figure and make it the current Axes.

Call signatures:

```
plt.axes()
plt.axes(rect, projection=None, polar=False, **kwargs)
plt.axes(ax)
```

**Parameters**

**arg**

[None or 4-tuple] The exact behavior of this function depends on the type:

- *None*: A new full window Axes is added using `subplot(**kwargs)`.
- 4-tuple of floats *rect* = `(left, bottom, width, height)`. A new Axes is added with dimensions *rect* in normalized (0, 1) units using *add_axes* on the current figure.

**projection**

[{None, 'aitoff', 'hammer', 'lambert', 'mollweide', 'polar', 'rectilinear', str}, optional] The projection type of the *Axes*. *str* is the name of a custom projection, see *projections*. The default None results in a 'rectilinear' projection.

**polar**

[bool, default: False] If True, equivalent to projection='polar'.

**sharex, sharey**

[*Axes*, optional] Share the x or y *axis* with sharex and/or sharey. The axis will have the same limits, ticks, and scale as the axis of the shared Axes.

**label**

[str] A label for the returned Axes.

### Returns

***Axes*, or a subclass of *Axes***

The returned Axes class depends on the projection used. It is *Axes* if rectilinear projection is used and *projections.polar.PolarAxes* if polar projection is used.

### Other Parameters

****kwargs**

This method also takes the keyword arguments for the returned Axes class. The keyword arguments for the rectilinear Axes class *Axes* can be found in the following table but there might also be other keyword arguments if another projection is used, see the actual Axes class.

| Property | Description |
|---|---|
| *adjustable* | {'box', 'datalim'} |
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi v |
| *alpha* | scalar or None |
| *anchor* | (float, float) or {'C', 'SW', 'S', 'SE', 'E', 'NE', ...} |
| *animated* | bool |
| *aspect* | {'auto', 'equal'} or float |
| *autoscale_on* | bool |
| *autoscalex_on* | unknown |

Table 103 – continued fro

| Property | Description |
|---|---|
| *autoscaley_on* | unknown |
| *axes_locator* | Callable[[Axes, Renderer], Bbox] |
| *axisbelow* | bool or 'line' |
| *box_aspect* | float or None |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *facecolor* or fc | *color* |
| *figure* | *Figure* |
| *forward_navigation_events* | bool or "auto" |
| *frame_on* | bool |
| *gid* | str |
| *in_layout* | bool |
| *label* | object |
| *mouseover* | bool |
| *navigate* | bool |
| *navigate_mode* | unknown |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *position* | [left, bottom, width, height] or *Bbox* |
| *prop_cycle* | *Cycler* |
| *rasterization_zorder* | float or None |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *subplotspec* | unknown |
| *title* | str |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *xbound* | (lower: float, upper: float) |
| *xlabel* | str |
| *xlim* | (left: float, right: float) |
| *xmargin* | float greater than -0.5 |
| *xscale* | unknown |
| *xticklabels* | unknown |
| *xticks* | unknown |
| *ybound* | (lower: float, upper: float) |
| *ylabel* | str |
| *ylim* | (bottom: float, top: float) |
| *ymargin* | float greater than -0.5 |
| *yscale* | unknown |
| *yticklabels* | unknown |
| *yticks* | unknown |

Table 103 – continued fro

| Property | Description |
|---|---|
| *zorder* | float |

**See also:**

***Figure.add_axes***
***pyplot.subplot***
***Figure.add_subplot***
***Figure.subplots***
***pyplot.subplots***

### Examples

```
# Creating a new full window Axes
plt.axes()

# Creating a new Axes with specified dimensions and a grey background
plt.axes((left, bottom, width, height), facecolor='grey')
```

## Examples using `matplotlib.pyplot.axes`

- *Subplots spacings and margins*

## matplotlib.pyplot.cla

matplotlib.pyplot.**cla**()
Clear the current Axes.

## matplotlib.pyplot.clf

matplotlib.pyplot.**clf**()
Clear the current figure.

### matplotlib.pyplot.close

matplotlib.pyplot.**close**(*fig=None*)

Close a figure window.

**Parameters**

**fig**

[None or int or str or *Figure*] The figure to close. There are a number of ways to specify this:

- *None*: the current figure
- *Figure*: the given *Figure* instance
- int: a figure number
- str: a figure name
- 'all': all figures

### Examples using matplotlib.pyplot.close

- *Pong*
- *Multipage PDF*
- *Multiprocessing*
- *Tight layout guide*

### matplotlib.pyplot.delaxes

matplotlib.pyplot.**delaxes**(*ax=None*)

Remove an *Axes* (defaulting to the current Axes) from its figure.

### matplotlib.pyplot.fignum_exists

matplotlib.pyplot.**fignum_exists**(*num*)

Return whether the figure with the given id exists.

**Parameters**

**num**

[int or str] A figure identifier.

**Returns**

**bool**

Whether or not a figure with id *num* exists.

## matplotlib.pyplot.figure

matplotlib.pyplot.**figure**(*num=None*, *figsize=None*, *dpi=None*, ***, *facecolor=None*, *edgecolor=None*, *frameon=True*, *FigureClass=<class 'matplotlib.figure.Figure'>*, *clear=False*, ***kwargs*)

Create a new figure, or activate an existing figure.

**Parameters**

**num**

[int or str or *Figure* or *SubFigure*, optional] A unique identifier for the figure.

If a figure with that identifier already exists, this figure is made active and returned. An integer refers to the `Figure.number` attribute, a string refers to the figure label.

If there is no figure with the identifier or *num* is not given, a new figure is created, made active and returned. If *num* is an int, it will be used for the `Figure.number` attribute, otherwise, an auto-generated integer value is used (starting at 1 and incremented for each new figure). If *num* is a string, the figure label and the window title is set to this value. If num is a `SubFigure`, its parent `Figure` is activated.

**figsize**

[(float, float), default: *rcParams["figure.figsize"]* (default: `[6.4, 4.8]`)] Width, height in inches.

**dpi**

[float, default: *rcParams["figure.dpi"]* (default: `100.0`)] The resolution of the figure in dots-per-inch.

**facecolor**

[*color*, default: *rcParams["figure.facecolor"]* (default: `'white'`)] The background color.

**edgecolor**

[*color*, default: *rcParams["figure.edgecolor"]* (default: `'white'`)] The border color.

**frameon**

[bool, default: True] If False, suppress drawing the figure frame.

**FigureClass**

[subclass of *Figure*] If set, an instance of this subclass will be created, rather than a plain *Figure*.

**clear**

[bool, default: False] If True and the figure already exists, then it is cleared.

**layout**

[{'constrained', 'compressed', 'tight', 'none', *LayoutEngine*, None}, default: None] The layout mechanism for positioning of plot elements to avoid overlapping Axes decorations (labels, ticks, etc). Note that layout managers can measurably slow down figure display.

- 'constrained': The constrained layout solver adjusts Axes sizes to avoid overlapping Axes decorations. Can handle complex plot layouts and colorbars, and is thus recommended.

  See *Constrained layout guide* for examples.

- 'compressed': uses the same algorithm as 'constrained', but removes extra space between fixed-aspect-ratio Axes. Best for simple grids of Axes.

- 'tight': Use the tight layout mechanism. This is a relatively simple algorithm that adjusts the subplot parameters so that decorations do not overlap. See *Figure.set_tight_layout* for further details.

- 'none': Do not use a layout engine.

- A *LayoutEngine* instance. Builtin layout classes are *ConstrainedLayoutEngine* and *TightLayoutEngine*, more easily accessible by 'constrained' and 'tight'. Passing an instance allows third parties to provide their own layout engine.

If not given, fall back to using the parameters *tight_layout* and *constrained_layout*, including their config defaults *rcParams["figure.autolayout"]* (default: `False`) and *rcParams["figure.constrained_layout.use"]* (default: `False`).

****kwargs**

Additional keyword arguments are passed to the *Figure* constructor.

**Returns**

***Figure***

### Notes

A newly created figure is passed to the *new_manager* method or the *new_figure_manager* function provided by the current backend, which install a canvas and a manager on the figure.

Once this is done, *rcParams["figure.hooks"]* (default: `[]`) are called, one at a time, on the figure; these hooks allow arbitrary customization of the figure (e.g., attaching callbacks) or of associated elements (e.g., modifying the toolbar). See *mplcvd -- an example of figure hook* for an example of toolbar customization.

If you are creating many figures, make sure you explicitly call *pyplot.close* on the figures you are not using, because this will enable pyplot to properly clean up the memory.

*rcParams* defines the default values, which can be modified in the matplotlibrc file.

## Examples using `matplotlib.pyplot.figure`

- *Errorbar limit selection*
- *EventCollection Demo*
- *Filled polygon*
- *Scatter plot with histograms*
- *Spectrum representations*
- *Barcode*
- *Figimage Demo*
- *Layer Images*
- *Axes Zoom Effect*
- *Custom Figure subclasses*
- *Resizing Axes with constrained layout*
- *Resizing Axes with tight layout*
- *Figure size in different units*
- *Geographic Projections*
- *Using Gridspec to make multi-column/row subplot layouts*
- *Nested Gridspecs*
- *Managing multiple figures in pyplot*
- *Figure subfigures*
- *Creating multiple subplots using plt.subplots*
- *Plotting cumulative distributions*
- *Error bar rendering on polar axis*

- *Polar legend*
- *Scatter plot on polar axis*
- *Annotation Polar*
- *Arrow Demo*
- *Auto-wrapping text*
- *Text Rotation Mode*
- *The difference between \dfrac and \frac*
- *Annotation arrow style reference*
- *Fonts demo (object-oriented style)*
- *Fonts demo (keyword arguments)*
- *Convert texts to images*
- *Mathtext Examples*
- *Concatenating text objects with different properties*
- *STIX Fonts*
- *Text Commands*
- *Unicode minus*
- *Usetex Baseline Test*
- *Usetex Fonteffects*
- *Reference for Matplotlib artists*
- *Drawing fancy boxes*
- *Hatch demo*
- *Two subplots using pyplot*
- *Axes divider*
- *Demo Axes Grid*
- *Axes Grid2*
- *Showing RGB channels using RGBAxes*
- *Per-row or per-column colorbars*
- *Axes with a fixed physical size*
- *Setting a fixed aspect on ImageGrid cells*
- *Inset locator demo*
- *Make room for ylabel using axes_grid*
- *Parasite Simple2*

- *Simple Axes Divider 1*
- *Simple axes divider 3*
- *Simple ImageGrid*
- *Simple ImageGrid 2*
- *Axis Direction*
- *axis_direction demo*
- *Axis line styles*
- *Curvilinear grid demo*
- *Demo CurveLinear Grid2*
- *floating_axes features*
- *floating_axis demo*
- *Parasite Axes demo*
- *Ticklabel alignment*
- *Ticklabel direction*
- *Simple axis direction*
- *Simple axis tick label and tick directions*
- *Simple Axis Pad*
- *Custom spines with axisartist*
- *Simple Axisline*
- *Simple Axisline3*
- *Anatomy of a figure*
- *Firefox*
- *Shaded & power normalized rendering*
- *Pan/zoom events of overlapping axes*
- *XKCD*
- *The double pendulum problem*
- *Frame grabbing*
- *Rain simulation*
- *Animated 3D random walk*
- *MATPLOTLIB UNCHAINED*
- *Close Event*
- *Interactive functions*

- *Lasso Demo*
- *Adding lines to figures*
- *Hyperlinks*
- *Matplotlib logo*
- *Multipage PDF*
- *SVG Filter Line*
- *SVG filter pie*
- *transforms.offset_copy*
- *Zorder Demo*
- *Plot 2D data on 3D plot*
- *Demo of 3D bar charts*
- *Create 2D bar graphs in different planes*
- *3D box surface plot*
- *Plot contour (level) curves in 3D*
- *Plot contour (level) curves in 3D using the extend3d option*
- *Project contour profiles onto a graph*
- *Filled contours*
- *Project filled contour onto a graph*
- *3D errorbars*
- *Create 3D histogram of 2D data*
- *Parametric curve*
- *Lorenz attractor*
- *2D and 3D Axes in same figure*
- *Automatic text offsetting*
- *Draw flat objects in 3D plot*
- *Generate polygons to fill under 3D line graph*
- *3D quiver plot*
- *Rotating a 3D plot*
- *3D scatterplot*
- *3D plots as subplots*
- *3D surface (solid color)*
- *3D surface (checkerboard)*

- *3D surface with polar coordinates*
- *Text annotations in 3D*
- *Triangular 3D contour plot*
- *Triangular 3D filled contour plot*
- *Triangular 3D surfaces*
- *More triangular 3D surfaces*
- *3D voxel / volumetric plot*
- *3D voxel plot of the NumPy logo*
- *3D voxel / volumetric plot with RGB colors*
- *3D voxel / volumetric plot with cylindrical coordinates*
- *3D wireframe plot*
- *Animate a 3D wireframe plot*
- *Asinh Demo*
- *Left ventricle bullseye*
- *The Sankey class*
- *Long chain of connections using Sankey*
- *Rankine power cycle*
- *SkewT-logP diagram: using transforms and custom projections*
- *Fig Axes Customize Simple*
- *Tick formatters*
- *Ellipse with units*
- *SVG Histogram*
- *Tool Manager*
- *Menu*
- *Rectangle and ellipse selectors*
- *subplot2grid demo*
- *GridSpec demo*
- *Nested GridSpecs*
- *Simple Legend01*
- *Pyplot tutorial*
- *Artist tutorial*
- *Quick start guide*

- *origin and extent in imshow*
- *Path effects guide*
- *Transformations Tutorial*
- *Constrained layout guide*
- *Tight layout guide*
- *Arranging multiple Axes in a Figure*
- *Axis ticks*
- *Complex and semantic figure composition (subplot_mosaic)*
- *Specifying colors*
- *Text in Matplotlib*
- *Text properties and layout*
- *Annotations*
- *Writing mathematical expressions*

### matplotlib.pyplot.gca

matplotlib.pyplot.**gca**()

Get the current Axes.

If there is currently no Axes on this Figure, a new one is created using *Figure.add_subplot*. (To test whether there is currently an Axes on a Figure, check whether `figure.axes` is empty. To test whether there is currently a Figure on the pyplot figure stack, check whether *pyplot.get_fignums()* is empty.)

#### Notes

**Note:** This is the *pyplot wrapper* for *Figure.gca*.

### Examples using `matplotlib.pyplot.gca`

- *Creating annotated heatmaps*
- *Managing multiple figures in pyplot*
- *Scale invariant angle label*
- *Infinite lines*
- *Set and get properties*

- *Hinton diagrams*
- *Tight layout guide*

### matplotlib.pyplot.gcf

matplotlib.pyplot.**gcf**()

Get the current figure.

If there is currently no figure on the pyplot figure stack, a new one is created using `figure()`. (To test whether there is currently a figure on the pyplot figure stack, check whether `get_fignums()` is empty.)

### matplotlib.pyplot.get_figlabels

matplotlib.pyplot.**get_figlabels**()

Return a list of existing figure labels.

### matplotlib.pyplot.get_fignums

matplotlib.pyplot.**get_fignums**()

Return a list of existing figure numbers.

### matplotlib.pyplot.sca

matplotlib.pyplot.**sca**(*ax*)

Set the current Axes to *ax* and the current Figure to the parent of *ax*.

### matplotlib.pyplot.subplot

matplotlib.pyplot.**subplot**(**args*, ***kwargs*)

Add an Axes to the current figure or retrieve an existing Axes.

This is a wrapper of `Figure.add_subplot` which provides additional behavior when working with the implicit API (see the notes section).

Call signatures:

```
subplot(nrows, ncols, index, **kwargs)
subplot(pos, **kwargs)
subplot(**kwargs)
subplot(ax)
```

**Parameters**

***args**

[int, (int, int, *index*), or *SubplotSpec*, default: (1, 1, 1)] The position of the subplot described by one of

- Three integers (*nrows*, *ncols*, *index*). The subplot will take the *index* position on a grid with *nrows* rows and *ncols* columns. *index* starts at 1 in the upper left corner and increases to the right. *index* can also be a two-tuple specifying the (*first*, *last*) indices (1-based, and including *last*) of the subplot, e.g., `fig.add_subplot(3, 1, (1, 2))` makes a subplot that spans the upper 2/3 of the figure.
- A 3-digit integer. The digits are interpreted as if given separately as three single-digit integers, i.e. `fig.add_subplot(235)` is the same as `fig.add_subplot(2, 3, 5)`. Note that this can only be used if there are no more than 9 subplots.
- A *SubplotSpec*.

**projection**

[{None, 'aitoff', 'hammer', 'lambert', 'mollweide', 'polar', 'rectilinear', str}, optional] The projection type of the subplot (*Axes*). *str* is the name of a custom projection, see *projections*. The default None results in a 'rectilinear' projection.

**polar**

[bool, default: False] If True, equivalent to projection='polar'.

**sharex, sharey**

[*Axes*, optional] Share the x or y *axis* with sharex and/or sharey. The axis will have the same limits, ticks, and scale as the axis of the shared Axes.

**label**

[str] A label for the returned Axes.

### Returns

***Axes***

The Axes of the subplot. The returned Axes can actually be an instance of a subclass, such as *projections.polar.PolarAxes* for polar projections.

### Other Parameters

****kwargs**

This method also takes the keyword arguments for the returned Axes base class; except for the *figure* argument. The keyword arguments for the rectilinear base class *Axes* can be found in the following table but there might also be other keyword arguments if another projection is used.

| Property | Description |
|---|---|
| adjustable | {'box', 'datalim'} |
| agg_filter | a filter function, which takes a (m, n, 3) float array and a dpi v |
| alpha | scalar or None |
| anchor | (float, float) or {'C', 'SW', 'S', 'SE', 'E', 'NE', ...} |
| animated | bool |
| aspect | {'auto', 'equal'} or float |
| autoscale_on | bool |
| autoscalex_on | unknown |
| autoscaley_on | unknown |
| axes_locator | Callable[[Axes, Renderer], Bbox] |
| axisbelow | bool or 'line' |
| box_aspect | float or None |
| clip_box | BboxBase or None |
| clip_on | bool |
| clip_path | Patch or (Path, Transform) or None |
| facecolor or fc | color |
| figure | Figure |
| forward_navigation_events | bool or "auto" |
| frame_on | bool |
| gid | str |
| in_layout | bool |
| label | object |
| mouseover | bool |
| navigate | bool |
| navigate_mode | unknown |
| path_effects | list of AbstractPathEffect |
| picker | None or bool or float or callable |
| position | [left, bottom, width, height] or Bbox |
| prop_cycle | Cycler |
| rasterization_zorder | float or None |
| rasterized | bool |
| sketch_params | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| subplotspec | unknown |
| title | str |
| transform | Transform |
| url | str |
| visible | bool |
| xbound | (lower: float, upper: float) |
| xlabel | str |
| xlim | (left: float, right: float) |
| xmargin | float greater than -0.5 |
| xscale | unknown |
| xticklabels | unknown |

Table 104 – continued fro

| Property | Description |
|---|---|
| *xticks* | unknown |
| *ybound* | (lower: float, upper: float) |
| *ylabel* | str |
| *ylim* | (bottom: float, top: float) |
| *ymargin* | float greater than -0.5 |
| *yscale* | unknown |
| *yticklabels* | unknown |
| *yticks* | unknown |
| *zorder* | float |

**See also:**

***Figure.add_subplot***
***pyplot.subplots***
***pyplot.axes***
***Figure.subplots***

## Notes

Creating a new Axes will delete any preexisting Axes that overlaps with it beyond sharing a boundary:

```
import matplotlib.pyplot as plt
# plot a line, implicitly creating a subplot(111)
plt.plot([1, 2, 3])
# now create a subplot which represents the top plot of a grid
# with 2 rows and 1 column. Since this subplot will overlap the
# first, the plot (and its Axes) previously created, will be removed
plt.subplot(211)
```

If you do not want this behavior, use the *Figure.add_subplot* method or the *pyplot.axes* function instead.

If no *kwargs* are passed and there exists an Axes in the location specified by *args* then that Axes will be returned rather than a new Axes being created.

If *kwargs* are passed and there exists an Axes in the location specified by *args*, the projection type is the same, and the *kwargs* match with the existing Axes, then the existing Axes is returned. Otherwise a new Axes is created with the specified parameters. We save a reference to the *kwargs* which we use for this comparison. If any of the values in *kwargs* are mutable we will not detect the case where they are mutated. In these cases we suggest using *Figure.add_subplot* and the explicit Axes API rather than the implicit pyplot API.

### Examples

```
plt.subplot(221)

# equivalent but more general
ax1 = plt.subplot(2, 2, 1)

# add a subplot with no frame
ax2 = plt.subplot(222, frameon=False)

# add a polar subplot
plt.subplot(223, projection='polar')

# add a red subplot that shares the x-axis with ax1
plt.subplot(224, sharex=ax1, facecolor='red')

# delete ax2 from the figure
plt.delaxes(ax2)

# add ax2 to the figure again
plt.subplot(ax2)

# make the first Axes "current" again
plt.subplot(221)
```

### Examples using `matplotlib.pyplot.subplot`

- *Controlling view limits using margins and sticky_edges*
- *Resizing Axes with tight layout*
- *Geographic Projections*
- *Managing multiple figures in pyplot*
- *Sharing axis limits and views*
- *Shared axis*
- *Multiple subplots*
- *Subplots spacings and margins*
- *Bar chart on polar axis*
- *Two subplots using pyplot*
- *MATPLOTLIB UNCHAINED*
- *Customize Rc*
- *transforms.offset_copy*
- *Pyplot tutorial*

- *Constrained layout guide*
- *Tight layout guide*
- *Arranging multiple Axes in a Figure*

### matplotlib.pyplot.subplot2grid

matplotlib.pyplot.**subplot2grid**(*shape*, *loc*, *rowspan=1*, *colspan=1*, *fig=None*, ***kwargs*)

Create a subplot at a specific location inside a regular grid.

**Parameters**

**shape**

[(int, int)] Number of rows and of columns of the grid in which to place axis.

**loc**

[(int, int)] Row number and column number of the axis location within the grid.

**rowspan**

[int, default: 1] Number of rows for the axis to span downwards.

**colspan**

[int, default: 1] Number of columns for the axis to span to the right.

**fig**

[*Figure*, optional] Figure to place the subplot in. Defaults to the current figure.

****kwargs**

Additional keyword arguments are handed to *add_subplot*.

**Returns**

***Axes***

The Axes of the subplot. The returned Axes can actually be an instance of a subclass, such as *projections.polar.PolarAxes* for polar projections.

#### Notes

The following call

```
ax = subplot2grid((nrows, ncols), (row, col), rowspan, colspan)
```

is identical to

```
fig = gcf()
gs = fig.add_gridspec(nrows, ncols)
ax = fig.add_subplot(gs[row:row+rowspan, col:col+colspan])
```

### Examples using `matplotlib.pyplot.subplot2grid`

- *Resizing Axes with tight layout*
- *subplot2grid demo*
- *Constrained layout guide*
- *Tight layout guide*
- *Arranging multiple Axes in a Figure*

## matplotlib.pyplot.subplot_mosaic

matplotlib.pyplot.**subplot_mosaic**(*mosaic*, *, *sharex=False*, *sharey=False*, *width_ratios=None*, *height_ratios=None*, *empty_sentinel='.'*, *subplot_kw=None*, *gridspec_kw=None*, *per_subplot_kw=None*, ***fig_kw*)

Build a layout of Axes based on ASCII art or nested lists.

This is a helper function to build complex GridSpec layouts visually.

See *Complex and semantic figure composition (subplot_mosaic)* for an example and full API documentation

**Parameters**

**mosaic**

[list of list of {hashable or nested} or str] A visual layout of how you want your Axes to be arranged labeled as strings. For example

```
x = [['A panel', 'A panel', 'edge'],
     ['C panel', '.',       'edge']]
```

produces 4 Axes:

- 'A panel' which is 1 row high and spans the first two columns
- 'edge' which is 2 rows high and is on the right edge
- 'C panel' which in 1 row and 1 column wide in the bottom left
- a blank space 1 row and 1 column wide in the bottom center

Any of the entries in the layout can be a list of lists of the same form to create nested layouts.

If input is a str, then it must be of the form

```
'''
AAE
C.E
'''
```

where each character is a column and each line is a row. This only allows only single character Axes labels and does not allow nesting but is very terse.

**sharex, sharey**

[bool, default: False] If True, the x-axis (*sharex*) or y-axis (*sharey*) will be shared among all subplots. In that case, tick label visibility and axis units behave as for *subplots*. If False, each subplot's x- or y-axis will be independent.

**width_ratios**

[array-like of length *ncols*, optional] Defines the relative widths of the columns. Each column gets a relative width of `width_ratios[i] / sum(width_ratios)`. If not given, all columns will have the same width. Convenience for `gridspec_kw={'width_ratios': [...]}`.

**height_ratios**

[array-like of length *nrows*, optional] Defines the relative heights of the rows. Each row gets a relative height of `height_ratios[i] / sum(height_ratios)`. If not given, all rows will have the same height. Convenience for `gridspec_kw={'height_ratios': [...]}`.

**empty_sentinel**

[object, optional] Entry in the layout to mean "leave this space empty". Defaults to `'.'`. Note, if *layout* is a string, it is processed via `inspect.cleandoc` to remove leading white space, which may interfere with using white-space as the empty sentinel.

**subplot_kw**

[dict, optional] Dictionary with keywords passed to the *Figure.add_subplot* call used to create each subplot. These values may be overridden by values in *per_subplot_kw*.

**per_subplot_kw**

[dict, optional] A dictionary mapping the Axes identifiers or tuples of identifiers to a dictionary of keyword arguments to be passed to the *Figure.add_subplot* call used to create each subplot. The values in these dictionaries have precedence over the values in *subplot_kw*.

If *mosaic* is a string, and thus all keys are single characters, it is possible to use a single string instead of a tuple as keys; i.e. `"AB"` is equivalent to `("A", "B")`.

New in version 3.7.

**gridspec_kw**

[dict, optional] Dictionary with keywords passed to the *GridSpec* constructor used to create the grid the subplots are placed on.

****fig_kw**

All additional keyword arguments are passed to the *pyplot.figure* call.

**Returns**

**fig**

[*Figure*] The new figure

**dict[label, Axes]**

A dictionary mapping the labels to the Axes objects. The order of the Axes is left-to-right and top-to-bottom of their position in the total layout.

## Examples using `matplotlib.pyplot.subplot_mosaic`

- *Power spectral density (PSD)*
- *Many ways to plot images*
- *Figure size in different units*
- *Histogram bins, density, and weight*
- *Labelling subplots*
- *Primary 3D view planes*
- *MRI with EEG*
- *Spine placement*
- *mplcvd -- an example of figure hook*
- *Radio Buttons*
- *Quick start guide*
- *Legend guide*
- *Arranging multiple Axes in a Figure*
- *Axis scales*
- *Complex and semantic figure composition (subplot_mosaic)*

### matplotlib.pyplot.subplots

matplotlib.pyplot.**subplots**(*nrows=1*, *ncols=1*, *, *sharex=False*, *sharey=False*, *squeeze=True*, *width_ratios=None*, *height_ratios=None*, *subplot_kw=None*, *gridspec_kw=None*, ***fig_kw*)

Create a figure and a set of subplots.

This utility wrapper makes it convenient to create common layouts of subplots, including the enclosing figure object, in a single call.

**Parameters**

**nrows, ncols**

[int, default: 1] Number of rows/columns of the subplot grid.

**sharex, sharey**

[bool or {'none', 'all', 'row', 'col'}, default: False] Controls sharing of properties among x (*sharex*) or y (*sharey*) axes:

- True or 'all': x- or y-axis will be shared among all subplots.
- False or 'none': each subplot x- or y-axis will be independent.
- 'row': each subplot row will share an x- or y-axis.
- 'col': each subplot column will share an x- or y-axis.

When subplots have a shared x-axis along a column, only the x tick labels of the bottom subplot are created. Similarly, when subplots have a shared y-axis along a row, only the y tick labels of the first column subplot are created. To later turn other subplots' ticklabels on, use *tick_params*.

When subplots have a shared axis that has units, calling *set_units* will update each axis with the new units.

**squeeze**

[bool, default: True]

- If True, extra dimensions are squeezed out from the returned array of *Axes*:
  - if only one subplot is constructed (nrows=ncols=1), the resulting single Axes object is returned as a scalar.
  - for Nx1 or 1xM subplots, the returned object is a 1D numpy object array of Axes objects.
  - for NxM, subplots with N>1 and M>1 are returned as a 2D array.
- If False, no squeezing at all is done: the returned Axes object is always a 2D array containing Axes instances, even if it ends up being 1x1.

**width_ratios**

[array-like of length *ncols*, optional] Defines the relative widths of the columns. Each column gets a relative width of `width_ratios[i] / sum(width_ratios)`. If not given, all columns will have the same width. Equivalent to `gridspec_kw={'width_ratios': [...]}`.

**height_ratios**

[array-like of length *nrows*, optional] Defines the relative heights of the rows. Each row gets a relative height of `height_ratios[i] / sum(height_ratios)`. If not given, all rows will have the same height. Convenience for `gridspec_kw={'height_ratios': [...]}`.

**subplot_kw**

[dict, optional] Dict with keywords passed to the *add_subplot* call used to create each subplot.

**gridspec_kw**

[dict, optional] Dict with keywords passed to the *GridSpec* constructor used to create the grid the subplots are placed on.

****fig_kw**

All additional keyword arguments are passed to the *pyplot.figure* call.

**Returns**

**fig**

[*Figure*]

**ax**

[*Axes* or array of Axes] *ax* can be either a single *Axes* object, or an array of Axes objects if more than one subplot was created. The dimensions of the resulting array can be controlled with the squeeze keyword, see above.

Typical idioms for handling the return value are:

```
# using the variable ax for single a Axes
fig, ax = plt.subplots()

# using the variable axs for multiple Axes
fig, axs = plt.subplots(2, 2)

# using tuple unpacking for multiple Axes
fig, (ax1, ax2) = plt.subplots(1, 2)
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2)
```

The names `ax` and pluralized `axs` are preferred over `axes` because for the latter it's not clear if it refers to a single *Axes* instance or a collection of these.

**See also:**

*pyplot.figure*
*pyplot.subplot*
*pyplot.axes*
*Figure.subplots*
*Figure.add_subplot*

### Examples

```
# First create some toy data:
x = np.linspace(0, 2*np.pi, 400)
y = np.sin(x**2)

# Create just a figure and only one subplot
fig, ax = plt.subplots()
ax.plot(x, y)
ax.set_title('Simple plot')

# Create two subplots and unpack the output array immediately
f, (ax1, ax2) = plt.subplots(1, 2, sharey=True)
ax1.plot(x, y)
ax1.set_title('Sharing Y axis')
ax2.scatter(x, y)

# Create four polar Axes and access them through the returned array
fig, axs = plt.subplots(2, 2, subplot_kw=dict(projection="polar"))
axs[0, 0].plot(x, y)
axs[1, 1].scatter(x, y)

# Share a X axis with each column of subplots
plt.subplots(2, 2, sharex='col')

# Share a Y axis with each row of subplots
plt.subplots(2, 2, sharey='row')

# Share both X and Y axes with all subplots
plt.subplots(2, 2, sharex='all', sharey='all')

# Note that this is the same as
plt.subplots(2, 2, sharex=True, sharey=True)

# Create figure number 10 with a single subplot
# and clears it if it already exists.
fig, ax = plt.subplots(num=10, clear=True)
```

### Examples using `matplotlib.pyplot.subplots`

- *Bar color demo*
- *Bar Label Demo*
- *Stacked bar chart*
- *Grouped bar chart with labels*
- *Horizontal bar chart*
- *Broken Barh*
- *Plotting categorical variables*
- *Plotting the coherence of two signals*
- *Cross spectral density (CSD)*
- *Curve with error band*
- *Errorbar subsampling*
- *Eventplot demo*
- *Filled polygon*
- *Fill Between and Alpha*
- *Filling the area between lines*
- *Fill Betweenx Demo*
- *Hatch-filled histograms*
- *Bar chart with gradients*
- *Hat graph*
- *Discrete distribution as horizontal bar chart*
- *Customizing dashed line styles*
- *Lines with a ticked patheffect*
- *Linestyles*
- *Marker reference*
- *Markevery Demo*
- *Multicolored lines*
- *Mapping marker properties to multivariate data*
- *Power spectral density (PSD)*
- *Scatter Demo2*
- *Marker examples*

- *Scatter plots with a legend*
- *Simple Plot*
- *Shade regions defined by a logical mask using fill_between*
- *Stackplots and streamgraphs*
- *Stairs Demo*
- *Creating a timeline with lines, dates, and text*
- *hlines and vlines*
- *Cross- and auto-correlation*
- *Affine transform of an image*
- *Wind Barbs*
- *Interactive Adjustment of Colormap Range*
- *Colormap normalizations*
- *Colormap normalizations SymLogNorm*
- *Contour Corner Mask*
- *Contour Demo*
- *Contour Image*
- *Contour Label Demo*
- *Contourf demo*
- *Contourf Hatching*
- *Contourf and log color scale*
- *Contouring the solution space of optimizations*
- *BboxImage Demo*
- *Creating annotated heatmaps*
- *Image antialiasing*
- *Clipping images with patches*
- *Many ways to plot images*
- *Image Masked*
- *Image nonuniform*
- *Blend transparency with color in 2D images*
- *Modifying the coordinate formatter*
- *Interpolations for imshow*
- *Contour plot of irregularly spaced data*

- *Multiple images*
- *pcolor images*
- *pcolormesh grids and shading*
- *pcolormesh*
- *Streamplot*
- *QuadMesh Demo*
- *Advanced quiver and quiverkey functions*
- *Quiver Simple Demo*
- *Shading example*
- *Spectrogram*
- *Spy Demos*
- *Tricontour Demo*
- *Tricontour Smooth Delaunay*
- *Tricontour Smooth User*
- *Trigradient Demo*
- *Triinterp Demo*
- *Tripcolor Demo*
- *Triplot Demo*
- *Watermark image*
- *Aligning Labels and Titles*
- *Programmatically controlling subplot adjustment*
- *Axes box aspect*
- *Axes Demo*
- *Controlling view limits using margins and sticky_edges*
- *Axes Props*
- *axhspan Demo*
- *Equal axis aspect ratio*
- *Axis Label Position*
- *Broken Axis*
- *Resizing Axes with constrained layout*
- *Resizing Axes with tight layout*
- *Different scales on the same Axes*

- *Figure size in different units*
- *Figure labels: suptitle, supxlabel, supylabel*
- *Creating adjacent subplots*
- *Combining two subplots using subplots and GridSpec*
- *Inverted axis*
- *Secondary Axis*
- *Figure subfigures*
- *Multiple subplots*
- *Creating multiple subplots using plt.subplots*
- *Plots with different scales*
- *Zoom region inset Axes*
- *Percentiles as horizontal bar chart*
- *Artist customization in box plots*
- *Box plots with custom fill colors*
- *Boxplots*
- *Box plot vs. violin plot comparison*
- *Separate calculation and plotting of boxplots*
- *Plot a confidence ellipse of a two-dimensional dataset*
- *Violin plot customization*
- *Errorbar function*
- *Different ways of specifying error bars*
- *Including upper and lower limits in error bars*
- *Creating boxes from error bars using PatchCollection*
- *Hexagonal binned plot*
- *Histograms*
- *Bihistogram*
- *Demo of the histogram function's different histtype settings*
- *The histogram (hist) function with multiple data sets*
- *Histogram bins, density, and weight*
- *Producing multiple histograms side by side*
- *Time Series Histogram*
- *Violin plot basics*

- *Pie charts*
- *Bar of pie*
- *Nested pie charts*
- *Labeling a pie and a donut*
- *Polar plot*
- *Accented text*
- *Align y-labels*
- *Scale invariant angle label*
- *Angle annotations on bracket arrows*
- *Annotate Transform*
- *Annotating a plot*
- *Annotating Plots*
- *Composing Custom Legends*
- *Date tick labels*
- *AnnotationBbox demo*
- *Using a text as a Path*
- *Labeling ticks using engineering notation*
- *Figure legend demo*
- *Configuring the font family*
- *Using ttf font files*
- *Font table*
- *Legend using pre-defined labels*
- *Legend Demo*
- *Artist within an artist*
- *Mathtext*
- *Math fontfamily*
- *Multiline*
- *Placing text boxes*
- *Rendering math equations using TeX*
- *Text alignment*
- *Text Rotation Relative To Line*
- *Title positioning*

- *Text watermark*
- *Color Demo*
- *Color by y-value*
- *Colors in the default property cycle*
- *Colorbar*
- *Colormap reference*
- *Creating a colormap from a list of colors*
- *Selecting individual colors from a colormap*
- *List of named colors*
- *Ways to set a color's alpha value*
- *Arrow guide*
- *Line, Poly and RegularPoly Collection with autoscaling*
- *Compound path*
- *Dolphins*
- *Mmh Donuts!!!*
- *Ellipse with orientation arrow demo*
- *Ellipse Collection*
- *Ellipse Demo*
- *Drawing fancy boxes*
- *Hatch style reference*
- *Plotting multiple lines with a LineCollection*
- *Circles, Wedges and Polygons*
- *PathPatch object*
- *Bezier Curve*
- *Bayesian Methods for Hackers style sheet*
- *Dark background style sheet*
- *FiveThirtyEight style sheet*
- *ggplot style sheet*
- *Grayscale style sheet*
- *Style sheets reference*
- *Anchored Direction Arrow*
- *HBoxDivider and VBoxDivider demo*

- *Showing RGB channels using RGBAxes*
- *Colorbar with AxesDivider*
- *Controlling the position and size of colorbars with Inset Axes*
- *Inset locator demo*
- *Inset locator demo 2*
- *Scatter Histogram (Locatable Axes)*
- *Simple Anchored Artists*
- *Integral as the area under a curve*
- *Stock prices over 32 years*
- *Decay*
- *Animated histogram*
- *pyplot animation*
- *The Bayes update*
- *Animated image using a precomputed list of images*
- *Multiple Axes animation*
- *Pausing and Resuming an Animation*
- *Animated line plot*
- *Animated scatter saved as GIF*
- *Oscilloscope*
- *Mouse move and click events*
- *Cross-hair cursor*
- *Data browser*
- *Figure/Axes enter and leave events*
- *Scroll event*
- *Keypress event*
- *Legend picking*
- *Looking Glass*
- *Path editor*
- *Pick event demo*
- *Pick event demo 2*
- *Poly Editor*
- *Pong*

- *Resampling Data*
- *Timers*
- *Trifinder Event Demo*
- *Viewlims*
- *Zoom Window*
- *Anchored Artists*
- *Changing colors of lines intersecting a box*
- *Manual Contour*
- *Coords Report*
- *Custom projection*
- *AGG filter*
- *Ribbon Box*
- *Findobj Demo*
- *Building histograms using Rectangles and PolyCollections*
- *Plotting with keywords*
- *Multiprocessing*
- *Packed-bubble chart*
- *Patheffect Demo*
- *Rasterization for vector graphics*
- *TickedStroke patheffect*
- *Zorder Demo*
- *Custom hillshading in a 3D surface plot*
- *3D plot projection types*
- *3D stem*
- *3D surface (colormap)*
- *3D wireframe plots in one direction*
- *Loglog Aspect*
- *Log Bar*
- *Log Demo*
- *Logit Demo*
- *Exploring normalizations*
- *Scales*

- *Log Axis*
- *Symlog Demo*
- *Hillshading*
- *Anscombe's quartet*
- *Ishikawa Diagram*
- *Radar chart (aka spider or star chart)*
- *Topographic hillshading*
- *Spines*
- *Dropped spines*
- *Multiple y-axis with Spines*
- *Centered spines with arrows*
- *Automatically setting tick positions*
- *Centering labels between ticks*
- *Colorbar Tick Labelling*
- *Custom Ticker*
- *Formatting date ticks using ConciseDateFormatter*
- *Date Demo Convert*
- *Placing date ticks using recurrence rules*
- *Date tick locators and formatters*
- *Custom tick formatter for time series*
- *Date Precision and Epochs*
- *Dollar ticks*
- *Major and minor ticks*
- *Multilevel (nested) ticks*
- *The default tick formatter*
- *Tick locators*
- *Set default y-axis tick labels on the right*
- *Setting tick labels from a list of values*
- *Move x-axis tick labels to the top*
- *Rotating custom tick labels*
- *Fixing too many ticks*
- *Annotation with units*

- *Artist tests*
- *Bar demo with units*
- *Group barchart with units*
- *Evans test*
- *Radian ticks*
- *Inches and Centimeters*
- *Unit handling*
- *pyplot with GTK3*
- *pyplot with GTK4*
- *SVG Tooltip*
- *Annotated cursor*
- *Buttons*
- *Check buttons*
- *Cursor*
- *Lasso Selector*
- *Mouse Cursor*
- *Multicursor*
- *Select indices from a collection using polygon selector*
- *Polygon Selector*
- *Thresholding an Image with RangeSlider*
- *Slider*
- *Snapping Sliders to Discrete Values*
- *Span Selector*
- *Textbox*
- *Annotate Explain*
- *Annotate Text Arrow*
- *Connection styles for annotations*
- *Custom box styles*
- *PGF fonts*
- *PGF preamble*
- *PGF texsystem*
- *Simple Annotate01*

- *Simple Legend02*
- *The Lifecycle of a Plot*
- *Artist tutorial*
- *plot(x, y)*
- *scatter(x, y)*
- *bar(x, height)*
- *stem(x, y)*
- *fill_between(x, y1, y2)*
- *stackplot(x, y)*
- *stairs(values)*
- *hist(x)*
- *boxplot(X)*
- *errorbar(x, y, yerr, xerr)*
- *violinplot(D)*
- *eventplot(D)*
- *hist2d(x, y)*
- *hexbin(x, y, C)*
- *pie(x)*
- *ecdf(x)*
- *imshow(Z)*
- *pcolormesh(X, Y, Z)*
- *contour(X, Y, Z)*
- *contourf(X, Y, Z)*
- *barbs(X, Y, U, V)*
- *quiver(X, Y, U, V)*
- *streamplot(X, Y, U, V)*
- *tricontour(x, y, z)*
- *tricontourf(x, y, z)*
- *tripcolor(x, y, z)*
- *triplot(x, y)*
- *scatter(xs, ys, zs)*
- *plot_surface(X, Y, Z)*

- *plot_trisurf(x, y, z)*
- *voxels([x, y, z], filled)*
- *plot_wireframe(X, Y, Z)*
- *Quick start guide*
- *Animations using Matplotlib*
- *Faster rendering by using blitting*
- *Styling with cycler*
- *Path Tutorial*
- *Transformations Tutorial*
- *Legend guide*
- *Constrained layout guide*
- *Tight layout guide*
- *Arranging multiple Axes in a Figure*
- *Autoscaling Axis*
- *Axis scales*
- *Axis ticks*
- *Plotting dates and strings*
- *Placing colorbars*
- *Specifying colors*
- *Customized Colorbars Tutorial*
- *Creating Colormaps in Matplotlib*
- *Colormap normalization*
- *Choosing Colormaps in Matplotlib*
- *Text in Matplotlib*
- *Annotations*

### matplotlib.pyplot.twinx

matplotlib.pyplot.**twinx**(*ax=None*)

Make and return a second Axes that shares the *x*-axis. The new Axes will overlay *ax* (or the current Axes if *ax* is *None*), and its ticks will be on the right.

#### Examples

*Plots with different scales*

### Examples using `matplotlib.pyplot.twinx`

- *Plots with different scales*
- *Percentiles as horizontal bar chart*

## matplotlib.pyplot.twiny

`matplotlib.pyplot.`**`twiny`**(*ax=None*)

Make and return a second Axes that shares the *y*-axis. The new Axes will overlay *ax* (or the current Axes if *ax* is *None*), and its ticks will be on the top.

#### Examples

*Plots with different scales*

### Examples using `matplotlib.pyplot.twiny`

- *Plots with different scales*

### Adding data to the plot

### Basic

| | |
|---|---|
| *plot* | Plot y versus x as lines and/or markers. |
| *errorbar* | Plot y versus x as lines and/or markers with attached errorbars. |
| *scatter* | A scatter plot of *y* vs. |
| *plot_date* | [*Deprecated*] Plot coercing the axis to treat floats as dates. |
| *step* | Make a step plot. |
| *loglog* | Make a plot with log scaling on both the x- and y-axis. |
| *semilogx* | Make a plot with log scaling on the x-axis. |
| *semilogy* | Make a plot with log scaling on the y-axis. |
| *fill_between* | Fill the area between two horizontal curves. |
| *fill_betweenx* | Fill the area between two vertical curves. |
| *bar* | Make a bar plot. |
| *barh* | Make a horizontal bar plot. |
| *bar_label* | Label a bar plot. |
| *stem* | Create a stem plot. |
| *eventplot* | Plot identical parallel lines at the given positions. |
| *pie* | Plot a pie chart. |
| *stackplot* | Draw a stacked area plot or a streamgraph. |
| *broken_barh* | Plot a horizontal sequence of rectangles. |
| *vlines* | Plot vertical lines at each *x* from *ymin* to *ymax*. |
| *hlines* | Plot horizontal lines at each *y* from *xmin* to *xmax*. |
| *fill* | Plot filled polygons. |
| *polar* | Make a polar plot. |

### matplotlib.pyplot.plot

matplotlib.pyplot.**plot**(**args*, *scalex=True*, *scaley=True*, *data=None*, ***kwargs*)

Plot y versus x as lines and/or markers.

Call signatures:

```
plot([x], y, [fmt], *, data=None, **kwargs)
plot([x], y, [fmt], [x2], y2, [fmt2], ..., **kwargs)
```

The coordinates of the points or line nodes are given by *x*, *y*.

The optional parameter *fmt* is a convenient way for defining basic formatting like color, marker and linestyle. It's a shortcut string notation described in the *Notes* section below.

```
>>> plot(x, y)         # plot x and y using default line style and color
>>> plot(x, y, 'bo')   # plot x and y using blue circle markers
>>> plot(y)            # plot y using x as index array 0..N-1
>>> plot(y, 'r+')      # ditto, but with red plusses
```

You can use *Line2D* properties as keyword arguments for more control on the appearance. Line properties and *fmt* can be mixed. The following two calls yield identical results:

```
>>> plot(x, y, 'go--', linewidth=2, markersize=12)
>>> plot(x, y, color='green', marker='o', linestyle='dashed',
...      linewidth=2, markersize=12)
```

When conflicting with *fmt*, keyword arguments take precedence.

**Plotting labelled data**

There's a convenient way for plotting objects with labelled data (i.e. data that can be accessed by index `obj['y']`). Instead of giving the data in *x* and *y*, you can provide the object in the *data* parameter and just give the labels for *x* and *y*:

```
>>> plot('xlabel', 'ylabel', data=obj)
```

All indexable objects are supported. This could e.g. be a `dict`, a `pandas.DataFrame` or a structured numpy array.

**Plotting multiple sets of data**

There are various ways to plot multiple sets of data.

- The most straight forward way is just to call *plot* multiple times. Example:

  ```
  >>> plot(x1, y1, 'bo')
  >>> plot(x2, y2, 'go')
  ```

- If *x* and/or *y* are 2D arrays a separate data set will be drawn for every column. If both *x* and *y* are 2D, they must have the same shape. If only one of them is 2D with shape (N, m) the other must have length N and will be used for every data set m.

  Example:

  ```
  >>> x = [1, 2, 3]
  >>> y = np.array([[1, 2], [3, 4], [5, 6]])
  >>> plot(x, y)
  ```

  is equivalent to:

  ```
  >>> for col in range(y.shape[1]):
  ...     plot(x, y[:, col])
  ```

- The third way is to specify multiple sets of *[x]*, *y*, *[fmt]* groups:

  ```
  >>> plot(x1, y1, 'g^', x2, y2, 'g-')
  ```

In this case, any additional keyword argument applies to all datasets. Also, this syntax cannot be combined with the *data* parameter.

By default, each line is assigned a different style specified by a 'style cycle'. The *fmt* and line property parameters are only necessary if you want explicit deviations from these defaults. Alternatively, you can also change the style cycle using *rcParams["axes.prop_cycle"]* (default: `cycler('color', ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728', '#9467bd', '#8c564b', '#e377c2', '#7f7f7f', '#bcbd22', '#17becf'])`).

**Parameters**

**x, y**

[array-like or scalar] The horizontal / vertical coordinates of the data points. *x* values are optional and default to `range(len(y))`.

Commonly, these parameters are 1D arrays.

They can also be scalars, or two-dimensional (in that case, the columns represent separate data sets).

These arguments cannot be passed as keywords.

**fmt**

[str, optional] A format string, e.g. 'ro' for red circles. See the *Notes* section for a full description of the format strings.

Format strings are just an abbreviation for quickly setting basic line properties. All of these and more can also be controlled by keyword arguments.

This argument cannot be passed as keyword.

**data**

[indexable object, optional] An object with labelled data. If given, provide the label names to plot in *x* and *y*.

---

**Note:** Technically there's a slight ambiguity in calls where the second label is a valid *fmt*. `plot('n', 'o', data=obj)` could be `plt(x, y)` or `plt(y, fmt)`. In such cases, the former interpretation is chosen, but a warning is issued. You may suppress the warning by adding an empty format string `plot('n', 'o', '', data=obj)`.

---

**Returns**

**list of *Line2D***

A list of lines representing the plotted data.

**Other Parameters**

**scalex, scaley**

[bool, default: True] These parameters determine if the view limits are adapted to the data limits. The values are passed on to *autoscale_view*.

****kwargs**

[*Line2D* properties, optional] *kwargs* are used to specify properties like a line label (for auto legends), linewidth, antialiasing, marker face color. Example:

```
>>> plot([1, 2, 3], [1, 2, 3], 'go-', label='line 1',␣
↪linewidth=2)
>>> plot([1, 2, 3], [1, 4, 9], 'rs', label='line 2')
```

If you specify multiple lines with one plot call, the kwargs apply to all those lines. In case the label object is iterable, each element is used as labels for each set of data.

Here is a list of available *Line2D* properties:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi va |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* or aa | bool |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* or c | *color* |
| *dash_capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *dash_joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *dashes* | sequence of floats (on/off ink in points) or (None, None) |
| *data* | (2, N) array or two 1D arrays |
| *drawstyle* or ds | {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: ' |
| *figure* | *Figure* |
| *fillstyle* | {'full', 'left', 'right', 'bottom', 'top', 'none'} |
| *gapcolor* | *color* or None |
| *gid* | str |
| *in_layout* | bool |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float |
| *marker* | marker style string, *Path* or *MarkerStyle* |
| *markeredgecolor* or mec | *color* |
| *markeredgewidth* or mew | float |
| *markerfacecolor* or mfc | *color* |
| *markerfacecoloralt* or mfcalt | *color* |
| *markersize* or ms | float |

Table 105 – continued from

| Property | Description |
|---|---|
| markevery | None or int or (int, int) or slice or list[int] or float or (float, floa |
| mouseover | bool |
| path_effects | list of AbstractPathEffect |
| picker | float or callable[[Artist, Event], tuple[bool, dict]] |
| pickradius | float |
| rasterized | bool |
| sketch_params | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| solid_capstyle | CapStyle or {'butt', 'projecting', 'round'} |
| solid_joinstyle | JoinStyle or {'miter', 'round', 'bevel'} |
| transform | unknown |
| url | str |
| visible | bool |
| xdata | 1D array |
| ydata | 1D array |
| zorder | float |

**See also:**

***scatter***

XY scatter plot with markers of varying size and/or color ( sometimes also called bubble chart).

## Notes

**Note:** This is the *pyplot wrapper* for `axes.Axes.plot`.

### Format Strings

A format string consists of a part for color, marker and line:

```
fmt = '[marker][line][color]'
```

Each of them is optional. If not provided, the value from the style cycle is used. Exception: If `line` is given, but no `marker`, the data will be a line without markers.

Other combinations such as `[color][marker][line]` are also supported, but note that their parsing may be ambiguous.

### Markers

| character | description |
|---|---|
| '.' | point marker |
| ',' | pixel marker |
| 'o' | circle marker |
| 'v' | triangle_down marker |
| '^' | triangle_up marker |
| '<' | triangle_left marker |
| '>' | triangle_right marker |
| '1' | tri_down marker |
| '2' | tri_up marker |
| '3' | tri_left marker |
| '4' | tri_right marker |
| '8' | octagon marker |
| 's' | square marker |
| 'p' | pentagon marker |
| 'P' | plus (filled) marker |
| '*' | star marker |
| 'h' | hexagon1 marker |
| 'H' | hexagon2 marker |
| '+' | plus marker |
| 'x' | x marker |
| 'X' | x (filled) marker |
| 'D' | diamond marker |
| 'd' | thin_diamond marker |
| '\|' | vline marker |
| '_' | hline marker |

**Line Styles**

| character | description |
|---|---|
| '-' | solid line style |
| '--' | dashed line style |
| '-.' | dash-dot line style |
| ':' | dotted line style |

Example format strings:

```
'b'    # blue markers with default shape
'or'   # red circles
'-g'   # green solid line
'--'   # dashed line with default color
'^k:'  # black triangle_up markers connected by a dotted line
```

**Colors**

The supported color abbreviations are the single letter codes

| character | color |
|---|---|
| `'b'` | blue |
| `'g'` | green |
| `'r'` | red |
| `'c'` | cyan |
| `'m'` | magenta |
| `'y'` | yellow |
| `'k'` | black |
| `'w'` | white |

and the `'CN'` colors that index into the default property cycle.

If the color is the only part of the format string, you can additionally use any *matplotlib.colors* spec, e.g. full names (`'green'`) or hex strings (`'#008000'`).

### Examples using `matplotlib.pyplot.plot`

- *Plotting masked and NaN values*
- *Scatter Masked*
- *Simple Plot*
- *Stairs Demo*
- *Step Demo*
- *Triinterp Demo*
- *Custom Figure subclasses*
- *Managing multiple figures in pyplot*
- *Shared axis*
- *Multiple subplots*
- *Polar plot*
- *Polar legend*
- *Align y-labels*
- *Legend using pre-defined labels*
- *Controlling style of text and labels using a dictionary*
- *Title positioning*
- *Color by y-value*
- *Dolphins*
- *Solarized Light stylesheet*

- *Infinite lines*
- *Simple plot*
- *Text and mathtext using pyplot*
- *Multiple lines using pyplot*
- *Two subplots using pyplot*
- *Frame grabbing*
- *Coords Report*
- *Customize Rc*
- *Findobj Demo*
- *Multipage PDF*
- *Print Stdout*
- *Set and get properties*
- *transforms.offset_copy*
- *Zorder Demo*
- *Custom scale*
- *Placing date ticks using recurrence rules*
- *CanvasAgg demo*
- *Tool Manager*
- *Pyplot tutorial*
- *Quick start guide*
- *Customizing Matplotlib with style sheets and rcParams*
- *Path effects guide*

### matplotlib.pyplot.errorbar

matplotlib.pyplot.**errorbar**(*x*, *y*, *yerr=None*, *xerr=None*, *fmt=''*, *ecolor=None*, *elinewidth=None*, *capsize=None*, *barsabove=False*, *lolims=False*, *uplims=False*, *xlolims=False*, *xuplims=False*, *errorevery=1*, *capthick=None*, ***, *data=None*, ***kwargs*)

Plot y versus x as lines and/or markers with attached errorbars.

*x*, *y* define the data locations, *xerr*, *yerr* define the errorbar sizes. By default, this draws the data markers/lines as well as the errorbars. Use fmt='none' to draw errorbars without any data markers.

New in version 3.7: Caps and error lines are drawn in polar coordinates on polar plots.

**Parameters**

**x, y**

[float or array-like] The data positions.

**xerr, yerr**

[float or array-like, shape(N,) or shape(2, N), optional] The errorbar sizes:

- scalar: Symmetric +/- values for all data points.
- shape(N,): Symmetric +/-values for each data point.
- shape(2, N): Separate - and + values for each bar. First row contains the lower errors, the second row contains the upper errors.
- *None*: No errorbar.

All values must be >= 0.

See *Different ways of specifying error bars* for an example on the usage of `xerr` and `yerr`.

**fmt**

[str, default: ''] The format for the data points / data lines. See `plot` for details.

Use 'none' (case-insensitive) to plot errorbars without any data markers.

**ecolor**

[*color*, default: None] The color of the errorbar lines. If None, use the color of the line connecting the markers.

**elinewidth**

[float, default: None] The linewidth of the errorbar lines. If None, the linewidth of the current style is used.

**capsize**

[float, default: `rcParams["errorbar.capsize"]` (default: `0.0`)] The length of the error bar caps in points.

**capthick**

[float, default: None] An alias to the keyword argument *markeredgewidth* (a.k.a. *mew*). This setting is a more sensible name for the property that controls the thickness of the error bar cap in points. For backwards compatibility, if *mew* or *markeredgewidth* are given, then they will over-ride *capthick*. This may change in future releases.

**barsabove**

[bool, default: False] If True, will plot the errorbars above the plot symbols. Default is below.

**lolims, uplims, xlolims, xuplims**

[bool or array-like, default: False] These arguments can be used to indicate that a value gives only upper/lower limits. In that case a caret symbol is used to indicate

this. *lims*-arguments may be scalars, or array-likes of the same length as *xerr* and *yerr*. To use limits with inverted axes, `set_xlim` or `set_ylim` must be called before `errorbar()`. Note the tricky parameter names: setting e.g. *lolims* to True means that the y-value is a *lower* limit of the True value, so, only an *upward*-pointing arrow will be drawn!

**errorevery**

[int or (int, int), default: 1] draws error bars on a subset of the data. *errorevery* =N draws error bars on the points (x[::N], y[::N]). *errorevery* =(start, N) draws error bars on the points (x[start::N], y[start::N]). e.g. errorevery=(6, 3) adds error bars to the data at (x[6], x[9], x[12], x[15], ...). Used to avoid overlapping error bars when two series share x-axis values.

**Returns**

***ErrorbarContainer***

The container contains:

- plotline: `Line2D` instance of x, y plot markers and/or line.
- caplines: A tuple of `Line2D` instances of the error bar caps.
- barlinecols: A tuple of `LineCollection` with the horizontal and vertical error ranges.

**Other Parameters**

**data**

[indexable object, optional] If given, the following parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception):

*x*, *y*, *xerr*, *yerr*

****kwargs**

All other keyword arguments are passed on to the `plot` call drawing the markers. For example, this code makes big red squares with thick green edges:

```
x, y, yerr = rand(3, 10)
errorbar(x, y, yerr, marker='s', mfc='red',
         mec='green', ms=20, mew=4)
```

where *mfc*, *mec*, *ms* and *mew* are aliases for the longer property names, *markerfacecolor*, *markeredgecolor*, *markersize* and *markeredgewidth*.

Valid kwargs for the marker properties are:

- *dashes*
- *dash_capstyle*
- *dash_joinstyle*

- *drawstyle*
- *fillstyle*
- *linestyle*
- *marker*
- *markeredgecolor*
- *markeredgewidth*
- *markerfacecolor*
- *markerfacecoloralt*
- *markersize*
- *markevery*
- *solid_capstyle*
- *solid_joinstyle*

Refer to the corresponding *Line2D* property for more details:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi va |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* or aa | bool |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* or c | *color* |
| *dash_capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *dash_joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *dashes* | sequence of floats (on/off ink in points) or (None, None) |
| *data* | (2, N) array or two 1D arrays |
| *drawstyle* or ds | {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: |
| *figure* | *Figure* |
| *fillstyle* | {'full', 'left', 'right', 'bottom', 'top', 'none'} |
| *gapcolor* | *color* or None |
| *gid* | str |
| *in_layout* | bool |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float |
| *marker* | marker style string, *Path* or *MarkerStyle* |
| *markeredgecolor* or mec | *color* |
| *markeredgewidth* or mew | float |
| *markerfacecolor* or mfc | *color* |

Table 106 – continued from

| Property | Description |
|---|---|
| markerfacecoloralt or mfcalt | color |
| markersize or ms | float |
| markevery | None or int or (int, int) or slice or list[int] or float or (float, floa |
| mouseover | bool |
| path_effects | list of AbstractPathEffect |
| picker | float or callable[[Artist, Event], tuple[bool, dict]] |
| pickradius | float |
| rasterized | bool |
| sketch_params | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| solid_capstyle | CapStyle or {'butt', 'projecting', 'round'} |
| solid_joinstyle | JoinStyle or {'miter', 'round', 'bevel'} |
| transform | unknown |
| url | str |
| visible | bool |
| xdata | 1D array |
| ydata | 1D array |
| zorder | float |

### Notes

**Note:** This is the *pyplot wrapper* for axes.Axes.errorbar.

### Examples using `matplotlib.pyplot.errorbar`

- *Errorbar limit selection*
- *Errorbar function*
- *Different ways of specifying error bars*
- *Including upper and lower limits in error bars*
- *Creating boxes from error bars using PatchCollection*
- *Error bar rendering on polar axis*

## matplotlib.pyplot.scatter

matplotlib.pyplot.**scatter**(*x*, *y*, *s=None*, *c=None*, *marker=None*, *cmap=None*, *norm=None*, *vmin=None*, *vmax=None*, *alpha=None*, *linewidths=None*, ***, *edgecolors=None*, *plotnonfinite=False*, *data=None*, ***kwargs*)

A scatter plot of *y* vs. *x* with varying marker size and/or color.

**Parameters**

**x, y**

[float or array-like, shape (n, )] The data positions.

**s**

[float or array-like, shape (n, ), optional] The marker size in points**2 (typographic points are 1/72 in.). Default is `rcParams['lines.markersize'] ** 2`.

The linewidth and edgecolor can visually interact with the marker size, and can lead to artifacts if the marker size is smaller than the linewidth.

If the linewidth is greater than 0 and the edgecolor is anything but *'none'*, then the effective size of the marker will be increased by half the linewidth because the stroke will be centered on the edge of the shape.

To eliminate the marker edge either set *linewidth=0* or *edgecolor='none'*.

**c**

[array-like or list of *color* or *color*, optional] The marker colors. Possible values:

- A scalar or sequence of n numbers to be mapped to colors using *cmap* and *norm*.
- A 2D array in which the rows are RGB or RGBA.
- A sequence of colors of length n.
- A single color format string.

Note that *c* should not be a single numeric RGB or RGBA sequence because that is indistinguishable from an array of values to be colormapped. If you want to specify the same RGB or RGBA value for all points, use a 2D array with a single row. Otherwise, value-matching will have precedence in case of a size matching with *x* and *y*.

If you wish to specify a single color for all points prefer the *color* keyword argument.

Defaults to `None`. In that case the marker color is determined by the value of *color*, *facecolor* or *facecolors*. In case those are not specified or `None`, the marker color is determined by the next color of the `Axes`' current "shape and fill" color cycle. This cycle defaults to *rcParams["axes.prop_cycle"]* (default: `cycler('color', ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728', '#9467bd', '#8c564b', '#e377c2', '#7f7f7f', '#bcbd22', '#17becf'])`).

**marker**

[*MarkerStyle*, default: *rcParams["scatter.marker"]* (default: 'o')] The marker style. *marker* can be either an instance of the class or the text shorthand for a particular marker. See *matplotlib.markers* for more information about marker styles.

**cmap**

[str or *Colormap*, default: *rcParams["image.cmap"]* (default: 'viridis')] The Colormap instance or registered colormap name used to map scalar data to colors.

This parameter is ignored if *c* is RGB(A).

**norm**

[str or *Normalize*, optional] The normalization method used to scale scalar data to the [0, 1] range before mapping to colors using *cmap*. By default, a linear scaling is used, mapping the lowest value to 0 and the highest to 1.

If given, this can be one of the following:

- An instance of *Normalize* or one of its subclasses (see *Colormap normalization*).
- A scale name, i.e. one of "linear", "log", "symlog", "logit", etc. For a list of available scales, call *matplotlib.scale.get_scale_names()*. In that case, a suitable *Normalize* subclass is dynamically generated and instantiated.

This parameter is ignored if *c* is RGB(A).

**vmin, vmax**

[float, optional] When using scalar data and no explicit *norm*, *vmin* and *vmax* define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data. It is an error to use *vmin*/*vmax* when a *norm* instance is given (but using a str *norm* name together with *vmin*/*vmax* is acceptable).

This parameter is ignored if *c* is RGB(A).

**alpha**

[float, default: None] The alpha blending value, between 0 (transparent) and 1 (opaque).

**linewidths**

[float or array-like, default: *rcParams["lines.linewidth"]* (default: 1.5)] The linewidth of the marker edges. Note: The default *edgecolors* is 'face'. You may want to change this as well.

**edgecolors**

[{'face', 'none', *None*} or *color* or list of *color*, default: `rcParams["scatter.edgecolors"]` (default: `'face'`)] The edge color of the marker. Possible values:

- 'face': The edge color will always be the same as the face color.
- 'none': No patch boundary will be drawn.
- A color or sequence of colors.

For non-filled markers, *edgecolors* is ignored. Instead, the color is determined like with 'face', i.e. from *c*, *colors*, or *facecolors*.

**plotnonfinite**

[bool, default: False] Whether to plot points with nonfinite *c* (i.e. `inf`, `-inf` or `nan`). If `True` the points are drawn with the *bad* colormap color (see `Colormap.set_bad`).

**Returns**

***PathCollection***

**Other Parameters**

**data**

[indexable object, optional] If given, the following parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception):

*x*, *y*, *s*, *linewidths*, *edgecolors*, *c*, *facecolor*, *facecolors*, *color*

****kwargs**

[`Collection` properties]

**See also:**

***plot***

To plot scatter plots when markers are identical in size and color.

## Notes

**Note:** This is the *pyplot wrapper* for `axes.Axes.scatter`.

- The `plot` function will be faster for scatterplots where markers don't vary in size or color.
- Any or all of *x*, *y*, *s*, and *c* may be masked arrays, in which case all masks will be combined and only unmasked points will be plotted.

- Fundamentally, scatter works with 1D arrays; *x*, *y*, *s*, and *c* may be input as N-D arrays, but within scatter they will be flattened. The exception is *c*, which will be flattened only if its size matches the size of *x* and *y*.

### Examples using `matplotlib.pyplot.scatter`

- *Scatter Masked*
- *Scatter plots with a legend*
- *Scatter plot on polar axis*
- *Scatter plot*
- *Hyperlinks*
- *Pyplot tutorial*

## matplotlib.pyplot.plot_date

matplotlib.pyplot.**plot_date**(*x*, *y*, *fmt='o'*, *tz=None*, *xdate=True*, *ydate=False*, ***, *data=None*, ***kwargs*)

[*Deprecated*] Plot coercing the axis to treat floats as dates.

Deprecated since version 3.9: This method exists for historic reasons and will be removed in version 3.11.

- `datetime`-like data should directly be plotted using *plot*.
- If you need to plot plain numeric data as *Matplotlib date format* or need to set a timezone, call `ax.xaxis.axis_date` / `ax.yaxis.axis_date` before *plot*. See *Axis.axis_date*.

Similar to *plot*, this plots *y* vs. *x* as lines or markers. However, the axis labels are formatted as dates depending on *xdate* and *ydate*. Note that *plot* will work with `datetime` and `numpy.datetime64` objects without resorting to this method.

**Parameters**

**x, y**

[array-like] The coordinates of the data points. If *xdate* or *ydate* is *True*, the respective values *x* or *y* are interpreted as *Matplotlib dates*.

**fmt**

[str, optional] The plot format string. For details, see the corresponding parameter in *plot*.

**tz**

[timezone string or `datetime.tzinfo`, default: *rcParams["timezone"]* (default: `'UTC'`)] The time zone to use in labeling dates.

**xdate**

[bool, default: True] If *True*, the *x*-axis will be interpreted as Matplotlib dates.

**ydate**

[bool, default: False] If *True*, the *y*-axis will be interpreted as Matplotlib dates.

**Returns**

**list of *Line2D***

Objects representing the plotted data.

**Other Parameters**

**data**

[indexable object, optional] If given, the following parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception):

*x*, *y*

****kwargs**

Keyword arguments control the *Line2D* properties:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi va |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* or aa | bool |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* or c | *color* |
| *dash_capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *dash_joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *dashes* | sequence of floats (on/off ink in points) or (None, None) |
| *data* | (2, N) array or two 1D arrays |
| *drawstyle* or ds | {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: |
| *figure* | *Figure* |
| *fillstyle* | {'full', 'left', 'right', 'bottom', 'top', 'none'} |
| *gapcolor* | *color* or None |
| *gid* | str |
| *in_layout* | bool |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float |
| *marker* | marker style string, *Path* or *MarkerStyle* |

Table 107 – continued from

| Property | Description |
|---|---|
| *markeredgecolor* or mec | *color* |
| *markeredgewidth* or mew | float |
| *markerfacecolor* or mfc | *color* |
| *markerfacecoloralt* or mfcalt | *color* |
| *markersize* or ms | float |
| *markevery* | None or int or (int, int) or slice or list[int] or float or (float, floa |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | float or callable[[Artist, Event], tuple[bool, dict]] |
| *pickradius* | float |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *solid_capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *solid_joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *transform* | unknown |
| *url* | str |
| *visible* | bool |
| *xdata* | 1D array |
| *ydata* | 1D array |
| *zorder* | float |

**See also:**

***matplotlib.dates***

Helper functions on dates.

***matplotlib.dates.date2num***

Convert dates to num.

***matplotlib.dates.num2date***

Convert num to dates.

***matplotlib.dates.drange***

Create an equally spaced sequence of dates.

### Notes

**Note:** This is the *pyplot wrapper* for `axes.Axes.plot_date`.

If you are using custom date tickers and formatters, it may be necessary to set the formatters/locators after the call to `plot_date`. `plot_date` will set the default tick locator to `AutoDateLocator` (if the tick locator is not already set to a `DateLocator` instance) and the default tick formatter to `AutoDateFormatter` (if the tick formatter is not already set to a `DateFormatter` instance).

Deprecated since version 3.9: Use plot instead.

## matplotlib.pyplot.step

matplotlib.pyplot.**step**(*x*, *y*, **args*, *where='pre'*, *data=None*, ***kwargs*)

Make a step plot.

Call signatures:

```
step(x, y, [fmt], *, data=None, where='pre', **kwargs)
step(x, y, [fmt], x2, y2, [fmt2], ..., *, where='pre', **kwargs)
```

This is just a thin wrapper around `plot` which changes some formatting options. Most of the concepts and parameters of plot can be used here as well.

**Note:** This method uses a standard plot with a step drawstyle: The *x* values are the reference positions and steps extend left/right/both directions depending on *where*.

For the common case where you know the values and edges of the steps, use `stairs` instead.

### Parameters

**x**

[array-like] 1D sequence of x positions. It is assumed, but not checked, that it is uniformly increasing.

**y**

[array-like] 1D sequence of y levels.

**fmt**

[str, optional] A format string, e.g. 'g' for a green line. See `plot` for a more detailed description.

Note: While full format strings are accepted, it is recommended to only specify the color. Line styles are currently ignored (use the keyword argument *linestyle*

instead). Markers are accepted and plotted on the given positions, however, this is a rarely needed feature for step plots.

**where**

[{'pre', 'post', 'mid'}, default: 'pre'] Define where the steps should be placed:

- 'pre': The y value is continued constantly to the left from every *x* position, i.e. the interval `(x[i-1], x[i]]` has the value `y[i]`.
- 'post': The y value is continued constantly to the right from every *x* position, i.e. the interval `[x[i], x[i+1])` has the value `y[i]`.
- 'mid': Steps occur half-way between the *x* positions.

**data**

[indexable object, optional] An object with labelled data. If given, provide the label names to plot in *x* and *y*.

****kwargs**

Additional parameters are the same as those for *plot*.

**Returns**

**list of *Line2D***

Objects representing the plotted data.

### Notes

**Note:** This is the *pyplot wrapper* for *axes.Axes.step*.

### Examples using `matplotlib.pyplot.step`

- *Stairs Demo*
- *Step Demo*

## matplotlib.pyplot.loglog

matplotlib.pyplot.**loglog**(**args*, ***kwargs*)

Make a plot with log scaling on both the x- and y-axis.

Call signatures:

```
loglog([x], y, [fmt], data=None, **kwargs)
loglog([x], y, [fmt], [x2], y2, [fmt2], ..., **kwargs)
```

This is just a thin wrapper around *plot* which additionally changes both the x-axis and the y-axis to log scaling. All the concepts and parameters of plot can be used here as well.

The additional parameters *base*, *subs* and *nonpositive* control the x/y-axis properties. They are just forwarded to *Axes.set_xscale* and *Axes.set_yscale*. To use different properties on the x-axis and the y-axis, use e.g. `ax.set_xscale("log", base=10); ax.set_yscale("log", base=2)`.

**Parameters**

**base**

[float, default: 10] Base of the logarithm.

**subs**

[sequence, optional] The location of the minor ticks. If *None*, reasonable locations are automatically chosen depending on the number of decades in the plot. See *Axes.set_xscale*/*Axes.set_yscale* for details.

**nonposition**

[{'mask', 'clip'}, default: 'clip'] Non-positive values can be masked as invalid, or clipped to a very small positive number.

****kwargs**

All parameters supported by *plot*.

**Returns**

**list of *Line2D***

Objects representing the plotted data.

## Notes

**Note:** This is the *pyplot wrapper* for *axes.Axes.loglog*.

## matplotlib.pyplot.semilogx

matplotlib.pyplot.**semilogx**(**args*, ***kwargs*)

Make a plot with log scaling on the x-axis.

Call signatures:

```
semilogx([x], y, [fmt], data=None, **kwargs)
semilogx([x], y, [fmt], [x2], y2, [fmt2], ..., **kwargs)
```

This is just a thin wrapper around *plot* which additionally changes the x-axis to log scaling. All the concepts and parameters of plot can be used here as well.

The additional parameters *base*, *subs*, and *nonpositive* control the x-axis properties. They are just forwarded to *Axes.set_xscale*.

**Parameters**

**base**

[float, default: 10] Base of the x logarithm.

**subs**

[array-like, optional] The location of the minor xticks. If *None*, reasonable locations are automatically chosen depending on the number of decades in the plot. See *Axes.set_xscale* for details.

**nonpositive**

[{'mask', 'clip'}, default: 'clip'] Non-positive values in x can be masked as invalid, or clipped to a very small positive number.

****kwargs**

All parameters supported by *plot*.

**Returns**

**list of *Line2D***

Objects representing the plotted data.

## Notes

**Note:** This is the *pyplot wrapper* for *axes.Axes.semilogx*.

## matplotlib.pyplot.semilogy

matplotlib.pyplot.**semilogy**(**args*, ***kwargs*)

Make a plot with log scaling on the y-axis.

Call signatures:

```
semilogy([x], y, [fmt], data=None, **kwargs)
semilogy([x], y, [fmt], [x2], y2, [fmt2], ..., **kwargs)
```

This is just a thin wrapper around *plot* which additionally changes the y-axis to log scaling. All the concepts and parameters of plot can be used here as well.

The additional parameters *base*, *subs*, and *nonposition* control the y-axis properties. They are just forwarded to `Axes.set_yscale`.

**Parameters**

**base**

[float, default: 10] Base of the y logarithm.

**subs**

[array-like, optional] The location of the minor yticks. If *None*, reasonable locations are automatically chosen depending on the number of decades in the plot. See `Axes.set_yscale` for details.

**nonpositive**

[{'mask', 'clip'}, default: 'clip'] Non-positive values in y can be masked as invalid, or clipped to a very small positive number.

****kwargs**

All parameters supported by `plot`.

**Returns**

**list of `Line2D`**

Objects representing the plotted data.

## Notes

**Note:** This is the *pyplot wrapper* for `axes.Axes.semilogy`.

## matplotlib.pyplot.fill_between

matplotlib.pyplot.**fill_between**(*x*, *y1*, *y2=0*, *where=None*, *interpolate=False*, *step=None*, *, *data=None*, ***kwargs*)

Fill the area between two horizontal curves.

The curves are defined by the points (*x*, *y1*) and (*x*, *y2*). This creates one or multiple polygons describing the filled area.

You may exclude some horizontal sections from filling using *where*.

By default, the edges connect the given points directly. Use *step* if the filling should be a step function, i.e. constant in between *x*.

**Parameters**

**x**

[array (length N)] The x coordinates of the nodes defining the curves.

**y1**

[array (length N) or scalar] The y coordinates of the nodes defining the first curve.

**y2**

[array (length N) or scalar, default: 0] The y coordinates of the nodes defining the second curve.

**where**

[array of bool (length N), optional] Define *where* to exclude some horizontal regions from being filled. The filled regions are defined by the coordinates `x[where]`. More precisely, fill between `x[i]` and `x[i+1]` if `where[i] and where[i+1]`. Note that this definition implies that an isolated *True* value between two *False* values in *where* will not result in filling. Both sides of the *True* position remain unfilled due to the adjacent *False* values.

**interpolate**

[bool, default: False] This option is only relevant if *where* is used and the two curves are crossing each other.

Semantically, *where* is often used for *y1* > *y2* or similar. By default, the nodes of the polygon defining the filled region will only be placed at the positions in the *x* array. Such a polygon cannot describe the above semantics close to the intersection. The x-sections containing the intersection are simply clipped.

Setting *interpolate* to *True* will calculate the actual intersection point and extend the filled region up to this point.

**step**

[{'pre', 'post', 'mid'}, optional] Define *step* if the filling should be a step function, i.e. constant in between *x*. The value determines where the step will occur:

- 'pre': The y value is continued constantly to the left from every *x* position, i.e. the interval `(x[i-1], x[i]]` has the value `y[i]`.
- 'post': The y value is continued constantly to the right from every *x* position, i.e. the interval `[x[i], x[i+1])` has the value `y[i]`.
- 'mid': Steps occur half-way between the *x* positions.

**Returns**

***PolyCollection***

A *PolyCollection* containing the plotted polygons.

**Other Parameters**

**data**

[indexable object, optional] If given, the following parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception):

*x*, *y1*, *y2*, *where*

****kwargs**

All other keyword arguments are passed on to *PolyCollection*. They control the *Polygon* properties:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a d |
| *alpha* | array-like or scalar or None |
| *animated* | bool |
| *antialiased* or aa or antialiaseds | bool or list of bools |
| *array* | array-like or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clim* | (vmin: float, vmax: float) |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *cmap* | *Colormap* or str or None |
| *color* | *color* or list of RGBA tuples |
| *edgecolor* or ec or edgecolors | *color* or list of *color* or 'face' |
| *facecolor* or facecolors or fc | *color* or list of *color* |
| *figure* | *Figure* |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or dashes or linestyles or ls | str or tuple or list thereof |
| *linewidth* or linewidths or lw | float or list of floats |
| *mouseover* | bool |
| *norm* | *Normalize* or str or None |
| *offset_transform* or transOffset | *Transform* |
| *offsets* | (N, 2) or (2,) array-like |
| *path_effects* | list of *AbstractPathEffect* |
| *paths* | list of array-like |
| *picker* | None or bool or float or callable |
| *pickradius* | float |
| *rasterized* | bool |
| sizes | numpy.ndarray or None |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |

Table 108 – continued fr

| Property | Description |
|---|---|
| *url* | str |
| *urls* | list of str or None |
| *verts* | list of array-like |
| *verts_and_codes* | unknown |
| *visible* | bool |
| *zorder* | float |

**See also:**

***fill_between***

Fill between two sets of y-values.

***fill_betweenx***

Fill between two sets of x-values.

### Notes

**Note:** This is the *pyplot wrapper* for *axes.Axes.fill_between*.

### Examples using `matplotlib.pyplot.fill_between`

- *Filling the area between lines*
- *Hatch-filled histograms*

## matplotlib.pyplot.fill_betweenx

matplotlib.pyplot.**fill_betweenx**(*y*, *x1*, *x2=0*, *where=None*, *step=None*, *interpolate=False*, ***, *data=None*, ***kwargs*)

Fill the area between two vertical curves.

The curves are defined by the points (*y*, *x1*) and (*y*, *x2*). This creates one or multiple polygons describing the filled area.

You may exclude some vertical sections from filling using *where*.

By default, the edges connect the given points directly. Use *step* if the filling should be a step function, i.e. constant in between *y*.

**Parameters**

**y**

[array (length N)] The y coordinates of the nodes defining the curves.

**x1**

[array (length N) or scalar] The x coordinates of the nodes defining the first curve.

**x2**

[array (length N) or scalar, default: 0] The x coordinates of the nodes defining the second curve.

**where**

[array of bool (length N), optional] Define *where* to exclude some vertical regions from being filled. The filled regions are defined by the coordinates `y[where]`. More precisely, fill between `y[i]` and `y[i+1]` if `where[i] and where[i+1]`. Note that this definition implies that an isolated *True* value between two *False* values in *where* will not result in filling. Both sides of the *True* position remain unfilled due to the adjacent *False* values.

**interpolate**

[bool, default: False] This option is only relevant if *where* is used and the two curves are crossing each other.

Semantically, *where* is often used for *x1* > *x2* or similar. By default, the nodes of the polygon defining the filled region will only be placed at the positions in the *y* array. Such a polygon cannot describe the above semantics close to the intersection. The y-sections containing the intersection are simply clipped.

Setting *interpolate* to *True* will calculate the actual intersection point and extend the filled region up to this point.

**step**

[{'pre', 'post', 'mid'}, optional] Define *step* if the filling should be a step function, i.e. constant in between *y*. The value determines where the step will occur:

- 'pre': The y value is continued constantly to the left from every *x* position, i.e. the interval `(x[i-1], x[i]]` has the value `y[i]`.
- 'post': The y value is continued constantly to the right from every *x* position, i.e. the interval `[x[i], x[i+1])` has the value `y[i]`.
- 'mid': Steps occur half-way between the *x* positions.

**Returns**

***PolyCollection***

A *PolyCollection* containing the plotted polygons.

**Other Parameters**

**data**

[indexable object, optional] If given, the following parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception):

*y*, *x1*, *x2*, *where*

****kwargs**

All other keyword arguments are passed on to *PolyCollection*. They control the *Polygon* properties:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a d |
| *alpha* | array-like or scalar or None |
| *animated* | bool |
| *antialiased* or aa or antialiaseds | bool or list of bools |
| *array* | array-like or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clim* | (vmin: float, vmax: float) |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *cmap* | *Colormap* or str or None |
| *color* | *color* or list of RGBA tuples |
| *edgecolor* or ec or edgecolors | *color* or list of *color* or 'face' |
| *facecolor* or facecolors or fc | *color* or list of *color* |
| *figure* | *Figure* |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or dashes or linestyles or ls | str or tuple or list thereof |
| *linewidth* or linewidths or lw | float or list of floats |
| *mouseover* | bool |
| *norm* | *Normalize* or str or None |
| *offset_transform* or transOffset | *Transform* |
| *offsets* | (N, 2) or (2,) array-like |
| *path_effects* | list of *AbstractPathEffect* |
| *paths* | list of array-like |
| *picker* | None or bool or float or callable |
| *pickradius* | float |
| *rasterized* | bool |
| sizes | numpy.ndarray or None |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |

Table 109 – continued fr

| Property | Description |
|---|---|
| *url* | str |
| *urls* | list of str or None |
| *verts* | list of array-like |
| *verts_and_codes* | unknown |
| *visible* | bool |
| *zorder* | float |

**See also:**

***fill_between***

Fill between two sets of y-values.

***fill_betweenx***

Fill between two sets of x-values.

### Notes

**Note:** This is the *pyplot wrapper* for *axes.Axes.fill_betweenx*.

## Examples using `matplotlib.pyplot.fill_betweenx`

- *Hatch-filled histograms*

## matplotlib.pyplot.bar

matplotlib.pyplot.**bar**(*x*, *height*, *width=0.8*, *bottom=None*, *, *align='center'*, *data=None*, ***kwargs*)

Make a bar plot.

The bars are positioned at *x* with the given *align*ment. Their dimensions are given by *height* and *width*. The vertical baseline is *bottom* (default 0).

Many parameters can take either a single value applying to all bars or a sequence of values, one for each bar.

**Parameters**

**x**

[float or array-like] The x coordinates of the bars. See also *align* for the alignment of the bars to the coordinates.

**height**

[float or array-like] The height(s) of the bars.

Note that if *bottom* has units (e.g. datetime), *height* should be in units that are a difference from the value of *bottom* (e.g. timedelta).

**width**

[float or array-like, default: 0.8] The width(s) of the bars.

Note that if *x* has units (e.g. datetime), then *width* should be in units that are a difference (e.g. timedelta) around the *x* values.

**bottom**

[float or array-like, default: 0] The y coordinate(s) of the bottom side(s) of the bars.

Note that if *bottom* has units, then the y-axis will get a Locator and Formatter appropriate for the units (e.g. dates, or categorical).

**align**

[{'center', 'edge'}, default: 'center'] Alignment of the bars to the *x* coordinates:

- 'center': Center the base on the *x* positions.
- 'edge': Align the left edges of the bars with the *x* positions.

To align the bars on the right edge pass a negative *width* and `align='edge'`.

**Returns**

***BarContainer***

Container with all the bars and optionally errorbars.

**Other Parameters**

**color**

[*color* or list of *color*, optional] The colors of the bar faces.

**edgecolor**

[*color* or list of *color*, optional] The colors of the bar edges.

**linewidth**

[float or array-like, optional] Width of the bar edge(s). If 0, don't draw edges.

**tick_label**

[str or list of str, optional] The tick labels of the bars. Default: None (Use default numeric labels.)

**label**

[str or list of str, optional] A single label is attached to the resulting *BarContainer* as a label for the whole dataset. If a list is provided, it must be the same length as *x* and labels the individual bars. Repeated labels are not de-duplicated and will cause repeated label entries, so this is best used when bars also differ in style (e.g., by passing a list to *color*.)

**xerr, yerr**

[float or array-like of shape(N,) or shape(2, N), optional] If not *None*, add horizontal / vertical errorbars to the bar tips. The values are +/- sizes relative to the data:

- scalar: symmetric +/- values for all bars
- shape(N,): symmetric +/- values for each bar
- shape(2, N): Separate - and + values for each bar. First row contains the lower errors, the second row contains the upper errors.
- *None*: No errorbar. (Default)

See *Different ways of specifying error bars* for an example on the usage of *xerr* and *yerr*.

**ecolor**

[*color* or list of *color*, default: 'black'] The line color of the errorbars.

**capsize**

[float, default: `rcParams["errorbar.capsize"]` (default: `0.0`)] The length of the error bar caps in points.

**error_kw**

[dict, optional] Dictionary of keyword arguments to be passed to the *errorbar* method. Values of *ecolor* or *capsize* defined here take precedence over the independent keyword arguments.

**log**

[bool, default: False] If *True*, set the y-axis to be log scale.

**data**

[indexable object, optional] If given, all parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception).

****kwargs**

[*Rectangle* properties]

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and return |
| *alpha* | scalar or None |
| *angle* | unknown |
| *animated* | bool |
| *antialiased* or aa | bool or None |
| *bounds* | (left, bottom, width, height) |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* | *color* |
| *edgecolor* or ec | *color* or None |
| *facecolor* or fc | *color* or None |
| *figure* | *Figure* |
| *fill* | bool |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *height* | unknown |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float or None |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *width* | unknown |
| *x* | unknown |
| *xy* | (float, float) |
| *y* | unknown |
| *zorder* | float |

**See also:**

***barh***

Plot a horizontal bar plot.

### Notes

**Note:** This is the *pyplot wrapper* for `axes.Axes.bar`.

Stacked bars can be achieved by passing individual *bottom* values per bar. See *Stacked bar chart*.

### Examples using `matplotlib.pyplot.bar`

- *Bar color demo*
- *Bar Label Demo*
- *Stacked bar chart*
- *Grouped bar chart with labels*
- *Hat graph*
- *Percentiles as horizontal bar chart*
- *Bar of pie*
- *Nested pie charts*
- *Bar chart on polar axis*
- *Hatch demo*
- *Table Demo*
- *Pyplot tutorial*

## matplotlib.pyplot.barh

matplotlib.pyplot.**barh**(*y*, *width*, *height=0.8*, *left=None*, ***, *align='center'*, *data=None*, ***kwargs*)

Make a horizontal bar plot.

The bars are positioned at *y* with the given *align*ment. Their dimensions are given by *width* and *height*. The horizontal baseline is *left* (default 0).

Many parameters can take either a single value applying to all bars or a sequence of values, one for each bar.

**Parameters**

**y**

[float or array-like] The y coordinates of the bars. See also *align* for the alignment of the bars to the coordinates.

**width**

[float or array-like] The width(s) of the bars.

Note that if *left* has units (e.g. datetime), *width* should be in units that are a difference from the value of *left* (e.g. timedelta).

**height**

[float or array-like, default: 0.8] The heights of the bars.

Note that if *y* has units (e.g. datetime), then *height* should be in units that are a difference (e.g. timedelta) around the *y* values.

**left**

[float or array-like, default: 0] The x coordinates of the left side(s) of the bars.

Note that if *left* has units, then the x-axis will get a Locator and Formatter appropriate for the units (e.g. dates, or categorical).

**align**

[{'center', 'edge'}, default: 'center'] Alignment of the base to the *y* coordinates*:

- 'center': Center the bars on the *y* positions.
- 'edge': Align the bottom edges of the bars with the *y* positions.

To align the bars on the top edge pass a negative *height* and `align='edge'`.

**Returns**

***BarContainer***

Container with all the bars and optionally errorbars.

**Other Parameters**

**color**

[*color* or list of *color*, optional] The colors of the bar faces.

**edgecolor**

[*color* or list of *color*, optional] The colors of the bar edges.

**linewidth**

[float or array-like, optional] Width of the bar edge(s). If 0, don't draw edges.

**tick_label**

[str or list of str, optional] The tick labels of the bars. Default: None (Use default numeric labels.)

**label**

[str or list of str, optional] A single label is attached to the resulting *BarContainer* as a label for the whole dataset. If a list is provided, it must be the same length as *y* and labels the individual bars. Repeated labels are not de-duplicated and will cause repeated label entries, so this is best used when bars also differ in style (e.g., by passing a list to *color*.)

**xerr, yerr**

[float or array-like of shape(N,) or shape(2, N), optional] If not *None*, add horizontal / vertical errorbars to the bar tips. The values are +/- sizes relative to the data:

- scalar: symmetric +/- values for all bars
- shape(N,): symmetric +/- values for each bar
- shape(2, N): Separate - and + values for each bar. First row contains the lower errors, the second row contains the upper errors.
- *None*: No errorbar. (default)

See *Different ways of specifying error bars* for an example on the usage of *xerr* and *yerr*.

**ecolor**

[*color* or list of *color*, default: 'black'] The line color of the errorbars.

**capsize**

[float, default: *rcParams["errorbar.capsize"]* (default: `0.0`)] The length of the error bar caps in points.

**error_kw**

[dict, optional] Dictionary of keyword arguments to be passed to the *errorbar* method. Values of *ecolor* or *capsize* defined here take precedence over the independent keyword arguments.

**log**

[bool, default: False] If `True`, set the x-axis to be log scale.

**data**

[indexable object, optional] If given, all parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception).

****kwargs**

[*Rectangle* properties]

| Property | Description |
|---|---|
| agg_filter | a filter function, which takes a (m, n, 3) float array and a dpi value, and return |
| alpha | scalar or None |
| angle | unknown |
| animated | bool |
| antialiased or aa | bool or None |
| bounds | (left, bottom, width, height) |
| capstyle | CapStyle or {'butt', 'projecting', 'round'} |
| clip_box | BboxBase or None |
| clip_on | bool |
| clip_path | Patch or (Path, Transform) or None |
| color | color |
| edgecolor or ec | color or None |
| facecolor or fc | color or None |
| figure | Figure |
| fill | bool |
| gid | str |
| hatch | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| height | unknown |
| in_layout | bool |
| joinstyle | JoinStyle or {'miter', 'round', 'bevel'} |
| label | object |
| linestyle or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| linewidth or lw | float or None |
| mouseover | bool |
| path_effects | list of AbstractPathEffect |
| picker | None or bool or float or callable |
| rasterized | bool |
| sketch_params | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| transform | Transform |
| url | str |
| visible | bool |
| width | unknown |
| x | unknown |
| xy | (float, float) |
| y | unknown |
| zorder | float |

**See also:**

**bar**

Plot a vertical bar plot.

### Notes

**Note:** This is the *pyplot wrapper* for `axes.Axes.barh`.

Stacked bars can be achieved by passing individual *left* values per bar. See *Discrete distribution as horizontal bar chart*.

## Examples using `matplotlib.pyplot.barh`

- *Bar Label Demo*
- *Discrete distribution as horizontal bar chart*
- *Producing multiple histograms side by side*

## matplotlib.pyplot.bar_label

matplotlib.pyplot.**bar_label**(*container*, *labels=None*, ***, *fmt='%g'*, *label_type='edge'*, *padding=0*, ***kwargs*)

Label a bar plot.

Adds labels to bars in the given `BarContainer`. You may need to adjust the axis limits to fit the labels.

**Parameters**

**container**

[`BarContainer`] Container with all the bars and optionally errorbars, likely returned from `bar` or `barh`.

**labels**

[array-like, optional] A list of label texts, that should be displayed. If not given, the label texts will be the data values formatted with *fmt*.

**fmt**

[str or callable, default: '%g'] An unnamed %-style or {}-style format string for the label or a function to call with the value as the first argument. When *fmt* is a string and can be interpreted in both formats, %-style takes precedence over {}-style.

New in version 3.7: Support for {}-style format string and callables.

**label_type**

[{'edge', 'center'}, default: 'edge'] The label type. Possible values:

- 'edge': label placed at the end-point of the bar segment, and the value displayed will be the position of that end-point.

- 'center': label placed in the center of the bar segment, and the value displayed will be the length of that segment. (useful for stacked bars, i.e., *Bar Label Demo*)

**padding**

[float, default: 0] Distance of label from the end of the bar, in points.

****kwargs**

Any remaining keyword arguments are passed through to `Axes.annotate`. The alignment parameters ( *horizontalalignment* / *ha*, *verticalalignment* / *va*) are not supported because the labels are automatically aligned to the bars.

**Returns**

**list of `Annotation`**

A list of `Annotation` instances for the labels.

### Notes

**Note:** This is the *pyplot wrapper* for `axes.Axes.bar_label`.

### Examples using `matplotlib.pyplot.bar_label`

- *Bar Label Demo*
- *Grouped bar chart with labels*
- *Discrete distribution as horizontal bar chart*
- *Percentiles as horizontal bar chart*

## matplotlib.pyplot.stem

matplotlib.pyplot.**stem**(**args*, *linefmt=None*, *markerfmt=None*, *basefmt=None*, *bottom=0*, *label=None*, *orientation='vertical'*, *data=None*)

Create a stem plot.

A stem plot draws lines perpendicular to a baseline at each location *locs* from the baseline to *heads*, and places a marker there. For vertical stem plots (the default), the *locs* are *x* positions, and the *heads* are *y* values. For horizontal stem plots, the *locs* are *y* positions, and the *heads* are *x* values.

Call signature:

```
stem([locs,] heads, linefmt=None, markerfmt=None, basefmt=None)
```

The *locs*-positions are optional. *linefmt* may be provided as positional, but all other formats must be provided as keyword arguments.

**Parameters**

**locs**

[array-like, default: (0, 1, ..., len(heads) - 1)] For vertical stem plots, the x-positions of the stems. For horizontal stem plots, the y-positions of the stems.

**heads**

[array-like] For vertical stem plots, the y-values of the stem heads. For horizontal stem plots, the x-values of the stem heads.

**linefmt**

[str, optional] A string defining the color and/or linestyle of the vertical lines:

| Character | Line Style |
|---|---|
| `'-'` | solid line |
| `'--'` | dashed line |
| `'-.'` | dash-dot line |
| `':'` | dotted line |

Default: 'C0-', i.e. solid line with the first color of the color cycle.

Note: Markers specified through this parameter (e.g. 'x') will be silently ignored. Instead, markers should be specified using *markerfmt*.

**markerfmt**

[str, optional] A string defining the color and/or shape of the markers at the stem heads. If the marker is not given, use the marker 'o', i.e. filled circles. If the color is not given, use the color from *linefmt*.

**basefmt**

[str, default: 'C3-' ('C2-' in classic mode)] A format string defining the properties of the baseline.

**orientation**

[{'vertical', 'horizontal'}, default: 'vertical'] The orientation of the stems.

**bottom**

[float, default: 0] The y/x-position of the baseline (depending on *orientation*).

**label**

[str, optional] The label to use for the stems in legends.

**data**

[indexable object, optional] If given, all parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception).

**Returns**

***StemContainer***

The container may be treated like a tuple (*markerline*, *stemlines*, *baseline*)

### Notes

**Note:** This is the *pyplot wrapper* for *axes.Axes.stem*.

**See also:**

The MATLAB function stem which inspired this method.

### Examples using `matplotlib.pyplot.stem`

- *Stem Plot*

## matplotlib.pyplot.eventplot

matplotlib.pyplot.**eventplot**(*positions*, *orientation='horizontal'*, *lineoffsets=1*, *linelengths=1*, *linewidths=None*, *colors=None*, *alpha=None*, *linestyles='solid'*, ***, *data=None*, ***kwargs*)

Plot identical parallel lines at the given positions.

This type of plot is commonly used in neuroscience for representing neural events, where it is usually called a spike raster, dot raster, or raster plot.

However, it is useful in any situation where you wish to show the timing or position of multiple sets of discrete events, such as the arrival times of people to a business on each day of the month or the date of hurricanes each year of the last century.

**Parameters**

**positions**

[array-like or list of array-like] A 1D array-like defines the positions of one sequence of events.

Multiple groups of events may be passed as a list of array-likes. Each group can be styled independently by passing lists of values to *lineoffsets*, *linelengths*, *linewidths*, *colors* and *linestyles*.

Note that *positions* can be a 2D array, but in practice different event groups usually have different counts so that one will use a list of different-length arrays rather than a 2D array.

**orientation**

[{'horizontal', 'vertical'}, default: 'horizontal'] The direction of the event sequence:

- 'horizontal': the events are arranged horizontally. The indicator lines are vertical.
- 'vertical': the events are arranged vertically. The indicator lines are horizontal.

**lineoffsets**

[float or array-like, default: 1] The offset of the center of the lines from the origin, in the direction orthogonal to *orientation*.

If *positions* is 2D, this can be a sequence with length matching the length of *positions*.

**linelengths**

[float or array-like, default: 1] The total height of the lines (i.e. the lines stretches from `lineoffset - linelength/2` to `lineoffset + linelength/2`).

If *positions* is 2D, this can be a sequence with length matching the length of *positions*.

**linewidths**

[float or array-like, default: *rcParams["lines.linewidth"]* (default: `1.5`)] The line width(s) of the event lines, in points.

If *positions* is 2D, this can be a sequence with length matching the length of *positions*.

**colors**

[*color* or list of color, default: *rcParams["lines.color"]* (default: `'C0'`)] The color(s) of the event lines.

If *positions* is 2D, this can be a sequence with length matching the length of *positions*.

**alpha**

[float or array-like, default: 1] The alpha blending value(s), between 0 (transparent) and 1 (opaque).

If *positions* is 2D, this can be a sequence with length matching the length of *positions*.

**linestyles**

[str or tuple or list of such values, default: 'solid'] Default is 'solid'. Valid strings are ['solid', 'dashed', 'dashdot', 'dotted', '-', '--', '-.', ':']. Dash tuples should be of the form:

```
(offset, onoffseq),
```

where *onoffseq* is an even length tuple of on and off ink in points.

If *positions* is 2D, this can be a sequence with length matching the length of *positions*.

**data**

[indexable object, optional] If given, the following parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception):

*positions*, *lineoffsets*, *linelengths*, *linewidths*, *colors*, *linestyles*

****kwargs**

Other keyword arguments are line collection properties. See *LineCollection* for a list of the valid properties.

**Returns**

**list of *EventCollection***

The *EventCollection* that were added.

## Notes

---

**Note:** This is the *pyplot wrapper* for *axes.Axes.eventplot*.

---

For *linelengths*, *linewidths*, *colors*, *alpha* and *linestyles*, if only a single value is given, that value is applied to all lines. If an array-like is given, it must have the same length as *positions*, and each value will be applied to the corresponding row of the array.

## Examples

### matplotlib.pyplot.pie

matplotlib.pyplot.**pie**(*x*, *explode=None*, *labels=None*, *colors=None*, *autopct=None*, *pctdistance=0.6*, *shadow=False*, *labeldistance=1.1*, *startangle=0*, *radius=1*, *counterclock=True*, *wedgeprops=None*, *textprops=None*, *center=(0, 0)*, *frame=False*, *rotatelabels=False*, ***, *normalize=True*, *hatch=None*, *data=None*)

Plot a pie chart.

Make a pie chart of array *x*. The fractional area of each wedge is given by `x/sum(x)`.

The wedges are plotted counterclockwise, by default starting from the x-axis.

**Parameters**

**x**

[1D array-like] The wedge sizes.

**explode**

[array-like, default: None] If not *None*, is a `len(x)` array which specifies the fraction of the radius with which to offset each wedge.

**labels**

[list, default: None] A sequence of strings providing the labels for each wedge

**colors**

[*color* or list of *color*, default: None] A sequence of colors through which the pie chart will cycle. If *None*, will use the colors in the currently active cycle.

**hatch**

[str or list, default: None] Hatching pattern applied to all pie wedges or sequence of patterns through which the chart will cycle. For a list of valid patterns, see *Hatch style reference*.

New in version 3.7.

**autopct**

[None or str or callable, default: None] If not *None*, *autopct* is a string or function used to label the wedges with their numeric value. The label will be placed inside the wedge. If *autopct* is a format string, the label will be `fmt % pct`. If *autopct* is a function, then it will be called.

**pctdistance**

[float, default: 0.6] The relative distance along the radius at which the text generated by *autopct* is drawn. To draw the text outside the pie, set *pctdistance* > 1. This parameter is ignored if *autopct* is `None`.

**labeldistance**

[float or None, default: 1.1] The relative distance along the radius at which the labels are drawn. To draw the labels inside the pie, set *labeldistance* < 1. If set to `None`, labels are not drawn but are still stored for use in *legend*.

**shadow**

[bool or dict, default: False] If bool, whether to draw a shadow beneath the pie. If dict, draw a shadow passing the properties in the dict to *Shadow*.

New in version 3.8: *shadow* can be a dict.

**startangle**

[float, default: 0 degrees] The angle by which the start of the pie is rotated, counterclockwise from the x-axis.

**radius**

[float, default: 1] The radius of the pie.

**counterclock**

[bool, default: True] Specify fractions direction, clockwise or counterclockwise.

**wedgeprops**

[dict, default: None] Dict of arguments passed to each *patches.Wedge* of the pie. For example, `wedgeprops = {'linewidth': 3}` sets the width of the wedge border lines equal to 3. By default, `clip_on=False`. When there is a conflict between these properties and other keywords, properties passed to *wedgeprops* take precedence.

**textprops**

[dict, default: None] Dict of arguments to pass to the text objects.

**center**

[(float, float), default: (0, 0)] The coordinates of the center of the chart.

**frame**

[bool, default: False] Plot Axes frame with the chart if true.

**rotatelabels**

[bool, default: False] Rotate each label to the angle of the corresponding slice if true.

**normalize**

[bool, default: True] When *True*, always make a full pie by normalizing x so that `sum(x) == 1`. *False* makes a partial pie if `sum(x) <= 1` and raises a `ValueError` for `sum(x) > 1`.

**data**

[indexable object, optional] If given, the following parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception):

*x*, *explode*, *labels*, *colors*

**Returns**

**patches**

[list] A sequence of `matplotlib.patches.Wedge` instances

**texts**

[list] A list of the label `Text` instances.

**autotexts**

[list] A list of `Text` instances for the numeric labels. This will only be returned if the parameter *autopct* is not *None*.

### Notes

**Note:** This is the *pyplot wrapper* for `axes.Axes.pie`.

The pie chart will probably look best if the figure and Axes are square, or the Axes aspect is equal. This method sets the aspect ratio of the axis to "equal". The Axes aspect ratio can be controlled with `Axes.set_aspect`.

### Examples using `matplotlib.pyplot.pie`

- *Pie charts*
- *Bar of pie*
- *Nested pie charts*
- *Labeling a pie and a donut*

## matplotlib.pyplot.stackplot

matplotlib.pyplot.**stackplot**(*x*, **args*, *labels=()*, *colors=None*, *hatch=None*, *baseline='zero'*, *data=None*, ***kwargs*)

Draw a stacked area plot or a streamgraph.

**Parameters**

**x**

[(N,) array-like]

**y**

[(M, N) array-like] The data is assumed to be unstacked. Each of the following calls is legal:

```
stackplot(x, y)           # where y has shape (M, N)
stackplot(x, y1, y2, y3)  # where y1, y2, y3, y4 have␣
↪length N
```

**baseline**

[{'zero', 'sym', 'wiggle', 'weighted_wiggle'}] Method used to calculate the baseline:

- `'zero'`: Constant zero baseline, i.e. a simple stacked plot.
- `'sym'`: Symmetric around zero and is sometimes called 'ThemeRiver'.
- `'wiggle'`: Minimizes the sum of the squared slopes.
- `'weighted_wiggle'`: Does the same but weights to account for size of each layer. It is also called 'Streamgraph'-layout. More details can be found at http://leebyron.com/streamgraph/.

**labels**

[list of str, optional] A sequence of labels to assign to each data series. If unspecified, then no labels will be applied to artists.

**colors**

[list of *color*, optional] A sequence of colors to be cycled through and used to color the stacked areas. The sequence need not be exactly the same length as the number of provided *y*, in which case the colors will repeat from the beginning.

If not specified, the colors from the Axes property cycle will be used.

**hatch**

[list of str, default: None] A sequence of hatching styles. See *Hatch style reference*. The sequence will be cycled through for filling the stacked areas from bottom to top. It need not be exactly the same length as the number of provided *y*, in which case the styles will repeat from the beginning.

**data**

[indexable object, optional] If given, all parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception).

****kwargs**

All other keyword arguments are passed to *`Axes.fill_between`*.

**Returns**

**list of *`PolyCollection`***

A list of *PolyCollection* instances, one for each element in the stacked area plot.

### Notes

**Note:** This is the *pyplot wrapper* for *axes.Axes.stackplot*.

## matplotlib.pyplot.broken_barh

matplotlib.pyplot.**broken_barh**(*xranges*, *yrange*, *, *data=None*, ***kwargs*)

Plot a horizontal sequence of rectangles.

A rectangle is drawn for each element of *xranges*. All rectangles have the same vertical position and size defined by *yrange*.

**Parameters**

**xranges**

[sequence of tuples (*xmin*, *xwidth*)] The x-positions and extents of the rectangles. For each tuple (*xmin*, *xwidth*) a rectangle is drawn from *xmin* to *xmin* + *xwidth*.

**yrange**

[(*ymin*, *yheight*)] The y-position and extent for all the rectangles.

**Returns**

***PolyCollection***

**Other Parameters**

**data**

[indexable object, optional] If given, all parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception).

****kwargs**

[*PolyCollection* properties] Each *kwarg* can be either a single argument applying to all rectangles, e.g.:

```
facecolors='black'
```

or a sequence of arguments over which is cycled, e.g.:

```
facecolors=('black', 'blue')
```

would create interleaving black and blue rectangles.

Supported keywords:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a d |
| *alpha* | array-like or scalar or None |
| *animated* | bool |
| *antialiased* or aa or antialiaseds | bool or list of bools |
| *array* | array-like or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clim* | (vmin: float, vmax: float) |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *cmap* | *Colormap* or str or None |
| *color* | *color* or list of RGBA tuples |
| *edgecolor* or ec or edgecolors | *color* or list of *color* or 'face' |
| *facecolor* or facecolors or fc | *color* or list of *color* |
| *figure* | *Figure* |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or dashes or linestyles or ls | str or tuple or list thereof |
| *linewidth* or linewidths or lw | float or list of floats |
| *mouseover* | bool |
| *norm* | *Normalize* or str or None |
| *offset_transform* or transOffset | *Transform* |
| *offsets* | (N, 2) or (2,) array-like |
| *path_effects* | list of *AbstractPathEffect* |
| *paths* | list of array-like |
| *picker* | None or bool or float or callable |
| *pickradius* | float |
| *rasterized* | bool |
| sizes | numpy.ndarray or None |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *urls* | list of str or None |
| *verts* | list of array-like |
| *verts_and_codes* | unknown |
| *visible* | bool |
| *zorder* | float |

### Notes

**Note:** This is the *pyplot wrapper* for `axes.Axes.broken_barh`.

## matplotlib.pyplot.vlines

matplotlib.pyplot.**vlines**(*x*, *ymin*, *ymax*, *colors=None*, *linestyles='solid'*, *label=''*, ***, *data=None*, ***kwargs*)

Plot vertical lines at each *x* from *ymin* to *ymax*.

**Parameters**

**x**

[float or array-like] x-indexes where to plot the lines.

**ymin, ymax**

[float or array-like] Respective beginning and end of each line. If scalars are provided, all lines will have the same length.

**colors**

[*color* or list of color, default: *rcParams["lines.color"]* (default: `'C0'`)]

**linestyles**

[{'solid', 'dashed', 'dashdot', 'dotted'}, default: 'solid']

**label**

[str, default: '']

**Returns**

***LineCollection***

**Other Parameters**

**data**

[indexable object, optional] If given, the following parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception):

*x*, *ymin*, *ymax*, *colors*

****kwargs**

[*LineCollection* properties.]

**See also:**

***hlines***

horizontal lines

***axvline***

vertical line across the Axes

### Notes

**Note:** This is the *pyplot wrapper* for *axes.Axes.vlines*.

### Examples using `matplotlib.pyplot.vlines`

- *Violin plot customization*

### matplotlib.pyplot.hlines

matplotlib.pyplot.**hlines**(*y*, *xmin*, *xmax*, *colors=None*, *linestyles='solid'*, *label=''*, ***, *data=None*, ***kwargs*)

Plot horizontal lines at each *y* from *xmin* to *xmax*.

**Parameters**

**y**

[float or array-like] y-indexes where to plot the lines.

**xmin, xmax**

[float or array-like] Respective beginning and end of each line. If scalars are provided, all lines will have the same length.

**colors**

[*color* or list of color , default: *rcParams["lines.color"]* (default: `'C0'`)]

**linestyles**

[{'solid', 'dashed', 'dashdot', 'dotted'}, default: 'solid']

**label**

[str, default: '']

**Returns**

*LineCollection*

**Other Parameters**

**data**

[indexable object, optional] If given, the following parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception):

*y*, *xmin*, *xmax*, *colors*

****kwargs**

[*LineCollection* properties.]

**See also:**

*vlines*

vertical lines

*axhline*

horizontal line across the Axes

## Notes

**Note:** This is the *pyplot wrapper* for *axes.Axes.hlines*.

## matplotlib.pyplot.fill

matplotlib.pyplot.**fill**(**args*, *data=None*, ***kwargs*)

Plot filled polygons.

**Parameters**

***args**

[sequence of x, y, [color]] Each polygon is defined by the lists of *x* and *y* positions of its nodes, optionally followed by a *color* specifier. See *matplotlib.colors* for supported color specifiers. The standard color cycle is used for polygons without a color specifier.

You can plot multiple polygons by providing multiple *x*, *y*, *[color]* groups.

For example, each of the following is legal:

```
ax.fill(x, y)                    # a polygon with default␣
 ↪color
ax.fill(x, y, "b")               # a blue polygon
ax.fill(x, y, x2, y2)            # two polygons
ax.fill(x, y, "b", x2, y2, "r")  # a blue and a red polygon
```

**data**

[indexable object, optional] An object with labelled data. If given, provide the label names to plot in *x* and *y*, e.g.:

```
ax.fill("time", "signal",
        data={"time": [0, 1, 2], "signal": [0, 1, 0]})
```

**Returns**

**list of *Polygon***

**Other Parameters**

****kwargs**

[*Polygon* properties]

## Notes

**Note:** This is the *pyplot wrapper* for *axes.Axes.fill*.

Use *fill_between()* if you would like to fill the region between two curves.

## Examples using `matplotlib.pyplot.fill`

- *Filled polygon*
- *Interactive functions*
- *Fill Spiral*

### matplotlib.pyplot.polar

matplotlib.pyplot.**polar**(**args*, ***kwargs*)

Make a polar plot.

call signature:

```
polar(theta, r, **kwargs)
```

Multiple *theta*, *r* arguments are supported, with format strings, as in *plot*.

### Examples using `matplotlib.pyplot.polar`

- *transforms.offset_copy*

### Spans

| | |
|---|---|
| *axhline* | Add a horizontal line across the Axes. |
| *axhspan* | Add a horizontal span (rectangle) across the Axes. |
| *axvline* | Add a vertical line across the Axes. |
| *axvspan* | Add a vertical span (rectangle) across the Axes. |
| *axline* | Add an infinitely long straight line. |

### matplotlib.pyplot.axhline

matplotlib.pyplot.**axhline**(*y=0*, *xmin=0*, *xmax=1*, ***kwargs*)

Add a horizontal line across the Axes.

**Parameters**

**y**

[float, default: 0] y position in data coordinates of the horizontal line.

**xmin**

[float, default: 0] Should be between 0 and 1, 0 being the far left of the plot, 1 the far right of the plot.

**xmax**

[float, default: 1] Should be between 0 and 1, 0 being the far left of the plot, 1 the far right of the plot.

**Returns**

*Line2D*

**Other Parameters**

****kwargs**

Valid keyword arguments are *Line2D* properties, except for 'transform':

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi va |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* or aa | bool |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* or c | *color* |
| *dash_capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *dash_joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *dashes* | sequence of floats (on/off ink in points) or (None, None) |
| *data* | (2, N) array or two 1D arrays |
| *drawstyle* or ds | {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: |
| *figure* | *Figure* |
| *fillstyle* | {'full', 'left', 'right', 'bottom', 'top', 'none'} |
| *gapcolor* | *color* or None |
| *gid* | str |
| *in_layout* | bool |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float |
| *marker* | marker style string, *Path* or *MarkerStyle* |
| *markeredgecolor* or mec | *color* |
| *markeredgewidth* or mew | float |
| *markerfacecolor* or mfc | *color* |
| *markerfacecoloralt* or mfcalt | *color* |
| *markersize* or ms | float |
| *markevery* | None or int or (int, int) or slice or list[int] or float or (float, floa |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | float or callable[[Artist, Event], tuple[bool, dict]] |
| *pickradius* | float |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *solid_capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *solid_joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |

Table 113 – continued from

| Property | Description |
|---|---|
| *transform* | unknown |
| *url* | str |
| *visible* | bool |
| *xdata* | 1D array |
| *ydata* | 1D array |
| *zorder* | float |

**See also:**

***hlines***

Add horizontal lines in data coordinates.

***axhspan***

Add a horizontal span (rectangle) across the axis.

***axline***

Add a line with an arbitrary slope.

## Notes

**Note:** This is the *pyplot wrapper* for *axes.Axes.axhline*.

## Examples

- draw a thick red hline at 'y' = 0 that spans the xrange:

```
>>> axhline(linewidth=4, color='r')
```

- draw a default hline at 'y' = 1 that spans the xrange:

```
>>> axhline(y=1)
```

- draw a default hline at 'y' = .5 that spans the middle half of the xrange:

```
>>> axhline(y=.5, xmin=0.25, xmax=0.75)
```

### Examples using `matplotlib.pyplot.axhline`

- *Colors in the default property cycle*
- *Infinite lines*
- *Zorder Demo*

## matplotlib.pyplot.axhspan

matplotlib.pyplot.**axhspan**(*ymin*, *ymax*, *xmin=0*, *xmax=1*, ***kwargs*)

Add a horizontal span (rectangle) across the Axes.

The rectangle spans from *ymin* to *ymax* vertically, and, by default, the whole x-axis horizontally. The x-span can be set using *xmin* (default: 0) and *xmax* (default: 1) which are in axis units; e.g. `xmin = 0.5` always refers to the middle of the x-axis regardless of the limits set by *set_xlim*.

**Parameters**

**ymin**

[float] Lower y-coordinate of the span, in data units.

**ymax**

[float] Upper y-coordinate of the span, in data units.

**xmin**

[float, default: 0] Lower x-coordinate of the span, in x-axis (0-1) units.

**xmax**

[float, default: 1] Upper x-coordinate of the span, in x-axis (0-1) units.

**Returns**

***Polygon***

Horizontal span (rectangle) from (xmin, ymin) to (xmax, ymax).

**Other Parameters**

****kwargs**

[*Polygon* properties]

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and return |
| *alpha* | scalar or None |
| *animated* | bool |

Table 114 – continued from previ

| Property | Description |
|---|---|
| *antialiased* or aa | bool or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *closed* | bool |
| *color* | *color* |
| *edgecolor* or ec | *color* or None |
| *facecolor* or fc | *color* or None |
| *figure* | *Figure* |
| *fill* | bool |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float or None |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *xy* | (N, 2) array-like |
| *zorder* | float |

**See also:**

***axvspan***

Add a vertical span across the Axes.

### Notes

**Note:** This is the *pyplot wrapper* for `axes.Axes.axhspan`.

## matplotlib.pyplot.axvline

matplotlib.pyplot.**axvline**(*x=0*, *ymin=0*, *ymax=1*, ***kwargs*)

Add a vertical line across the Axes.

**Parameters**

**x**

[float, default: 0] x position in data coordinates of the vertical line.

**ymin**

[float, default: 0] Should be between 0 and 1, 0 being the bottom of the plot, 1 the top of the plot.

**ymax**

[float, default: 1] Should be between 0 and 1, 0 being the bottom of the plot, 1 the top of the plot.

**Returns**

***Line2D***

**Other Parameters**

****kwargs**

Valid keyword arguments are `Line2D` properties, except for 'transform':

| Property | Description |
|---|---|
| `agg_filter` | a filter function, which takes a (m, n, 3) float array and a dpi va |
| `alpha` | scalar or None |
| `animated` | bool |
| `antialiased` or aa | bool |
| `clip_box` | `BboxBase` or None |
| `clip_on` | bool |
| `clip_path` | Patch or (Path, Transform) or None |
| `color` or c | *color* |
| `dash_capstyle` | `CapStyle` or {'butt', 'projecting', 'round'} |
| `dash_joinstyle` | `JoinStyle` or {'miter', 'round', 'bevel'} |

Table 115 – continued from

| Property | Description |
|---|---|
| *dashes* | sequence of floats (on/off ink in points) or (None, None) |
| *data* | (2, N) array or two 1D arrays |
| *drawstyle* or ds | {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: |
| *figure* | *Figure* |
| *fillstyle* | {'full', 'left', 'right', 'bottom', 'top', 'none'} |
| *gapcolor* | *color* or None |
| *gid* | str |
| *in_layout* | bool |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float |
| *marker* | marker style string, *Path* or *MarkerStyle* |
| *markeredgecolor* or mec | *color* |
| *markeredgewidth* or mew | float |
| *markerfacecolor* or mfc | *color* |
| *markerfacecoloralt* or mfcalt | *color* |
| *markersize* or ms | float |
| *markevery* | None or int or (int, int) or slice or list[int] or float or (float, floa |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | float or callable[[Artist, Event], tuple[bool, dict]] |
| *pickradius* | float |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *solid_capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *solid_joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *transform* | unknown |
| *url* | str |
| *visible* | bool |
| *xdata* | 1D array |
| *ydata* | 1D array |
| *zorder* | float |

**See also:**

***vlines***

Add vertical lines in data coordinates.

***axvspan***

Add a vertical span (rectangle) across the axis.

***axline***

Add a line with an arbitrary slope.

### Notes

**Note:** This is the *pyplot wrapper* for `axes.Axes.axvline`.

### Examples

- draw a thick red vline at $x = 0$ that spans the yrange:

```
>>> axvline(linewidth=4, color='r')
```

- draw a default vline at $x = 1$ that spans the yrange:

```
>>> axvline(x=1)
```

- draw a default vline at $x = .5$ that spans the middle half of the yrange:

```
>>> axvline(x=.5, ymin=0.25, ymax=0.75)
```

## Examples using `matplotlib.pyplot.axvline`

- *Colors in the default property cycle*
- *Infinite lines*

## matplotlib.pyplot.axvspan

matplotlib.pyplot.**axvspan**(*xmin*, *xmax*, *ymin=0*, *ymax=1*, ***kwargs*)

Add a vertical span (rectangle) across the Axes.

The rectangle spans from *xmin* to *xmax* horizontally, and, by default, the whole y-axis vertically. The y-span can be set using *ymin* (default: 0) and *ymax* (default: 1) which are in axis units; e.g. `ymin = 0.5` always refers to the middle of the y-axis regardless of the limits set by `set_ylim`.

**Parameters**

**xmin**

[float] Lower x-coordinate of the span, in data units.

**xmax**

[float] Upper x-coordinate of the span, in data units.

**ymin**

[float, default: 0] Lower y-coordinate of the span, in y-axis units (0-1).

**ymax**

[float, default: 1] Upper y-coordinate of the span, in y-axis units (0-1).

**Returns**

*Polygon*

Vertical span (rectangle) from (xmin, ymin) to (xmax, ymax).

**Other Parameters**

****kwargs**

[*Polygon* properties]

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and retur |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* or aa | bool or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *closed* | bool |
| *color* | *color* |
| *edgecolor* or ec | *color* or None |
| *facecolor* or fc | *color* or None |
| *figure* | *Figure* |
| *fill* | bool |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float or None |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |

Table 116 – continued from previ

| Property | Description |
|---|---|
| *xy* | (N, 2) array-like |
| *zorder* | float |

**See also:**

***axhspan***

Add a horizontal span across the Axes.

### Notes

**Note:** This is the *pyplot wrapper* for *axes.Axes.axvspan*.

### Examples

Draw a vertical, green, translucent rectangle from x = 1.25 to x = 1.55 that spans the yrange of the Axes.

```
>>> axvspan(1.25, 1.55, facecolor='g', alpha=0.5)
```

## matplotlib.pyplot.axline

matplotlib.pyplot.**axline**(*xy1*, *xy2=None*, *, *slope=None*, ***kwargs*)

Add an infinitely long straight line.

The line can be defined either by two points *xy1* and *xy2*, or by one point *xy1* and a *slope*.

This draws a straight line "on the screen", regardless of the x and y scales, and is thus also suitable for drawing exponential decays in semilog plots, power laws in loglog plots, etc. However, *slope* should only be used with linear scales; It has no clear meaning for all other scales, and thus the behavior is undefined. Please specify the line using the points *xy1*, *xy2* for non-linear scales.

The *transform* keyword argument only applies to the points *xy1*, *xy2*. The *slope* (if given) is always in data coordinates. This can be used e.g. with `ax.transAxes` for drawing grid lines with a fixed slope.

**Parameters**

**xy1, xy2**

[(float, float)] Points for the line to pass through. Either *xy2* or *slope* has to be given.

**slope**

[float, optional] The slope of the line. Either *xy2* or *slope* has to be given.

**Returns**

*AxLine*

**Other Parameters**

****kwargs**

Valid kwargs are *Line2D* properties

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi va |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* or aa | bool |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* or c | *color* |
| *dash_capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *dash_joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *dashes* | sequence of floats (on/off ink in points) or (None, None) |
| *data* | (2, N) array or two 1D arrays |
| *drawstyle* or ds | {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: |
| *figure* | *Figure* |
| *fillstyle* | {'full', 'left', 'right', 'bottom', 'top', 'none'} |
| *gapcolor* | *color* or None |
| *gid* | str |
| *in_layout* | bool |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float |
| *marker* | marker style string, *Path* or *MarkerStyle* |
| *markeredgecolor* or mec | *color* |
| *markeredgewidth* or mew | float |
| *markerfacecolor* or mfc | *color* |
| *markerfacecoloralt* or mfcalt | *color* |
| *markersize* or ms | float |
| *markevery* | None or int or (int, int) or slice or list[int] or float or (float, floa |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | float or callable[[Artist, Event], tuple[bool, dict]] |
| *pickradius* | float |

Table 117 – continued from

| Property | Description |
|---|---|
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *solid_capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *solid_joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *transform* | unknown |
| *url* | str |
| *visible* | bool |
| *xdata* | 1D array |
| *ydata* | 1D array |
| *zorder* | float |

**See also:**

***axhline***

for horizontal lines

***axvline***

for vertical lines

### Notes

**Note:** This is the *pyplot wrapper* for *axes.Axes.axline*.

### Examples

Draw a thick red line passing through (0, 0) and (1, 1):

```
>>> axline((0, 0), (1, 1), linewidth=4, color='r')
```

## Examples using `matplotlib.pyplot.axline`

- *Infinite lines*
- *Anscombe's quartet*

### Spectral

| | |
|---|---|
| *acorr* | Plot the autocorrelation of *x*. |
| *angle_spectrum* | Plot the angle spectrum. |
| *cohere* | Plot the coherence between *x* and *y*. |
| *csd* | Plot the cross-spectral density. |
| *magnitude_spectrum* | Plot the magnitude spectrum. |
| *phase_spectrum* | Plot the phase spectrum. |
| *psd* | Plot the power spectral density. |
| *specgram* | Plot a spectrogram. |
| *xcorr* | Plot the cross correlation between *x* and *y*. |

### matplotlib.pyplot.acorr

matplotlib.pyplot.**acorr**(*x*, *, *data=None*, ***kwargs*)

Plot the autocorrelation of *x*.

**Parameters**

**x**

[array-like] Not run through Matplotlib's unit conversion, so this should be a unitless array.

**detrend**

[callable, default: *mlab.detrend_none* (no detrending)] A detrending function applied to *x*. It must have the signature

```
detrend(x: np.ndarray) -> np.ndarray
```

**normed**

[bool, default: True] If `True`, input vectors are normalised to unit length.

**usevlines**

[bool, default: True] Determines the plot style.

If `True`, vertical lines are plotted from 0 to the acorr value using *Axes.vlines*. Additionally, a horizontal line is plotted at y=0 using *Axes.axhline*.

If `False`, markers are plotted at the acorr values using *Axes.plot*.

**maxlags**

[int, default: 10] Number of lags to show. If `None`, will return all `2 * len(x) - 1` lags.

**Returns**

**lags**

[array (length `2*maxlags+1`)] The lag vector.

**c**

[array (length `2*maxlags+1`)] The auto correlation vector.

**line**

[*LineCollection* or *Line2D*] *Artist* added to the Axes of the correlation:

- *LineCollection* if *usevlines* is True.
- *Line2D* if *usevlines* is False.

**b**

[*Line2D* or None] Horizontal line at 0 if *usevlines* is True None *usevlines* is False.

**Other Parameters**

**linestyle**

[*Line2D* property, optional] The linestyle for plotting the data points. Only used if *usevlines* is `False`.

**marker**

[str, default: 'o'] The marker for plotting the data points. Only used if *usevlines* is `False`.

**data**

[indexable object, optional] If given, the following parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception):

*x*

****kwargs**

Additional parameters are passed to *Axes.vlines* and *Axes.axhline* if *usevlines* is `True`; otherwise they are passed to *Axes.plot*.

## Notes

**Note:** This is the *pyplot wrapper* for *axes.Axes.acorr*.

The cross correlation is performed with `numpy.correlate` with `mode = "full"`.

### Examples using `matplotlib.pyplot.acorr`

- *Cross- and auto-correlation*

## matplotlib.pyplot.angle_spectrum

matplotlib.pyplot.**angle_spectrum**(*x*, *Fs=None*, *Fc=None*, *window=None*, *pad_to=None*, *sides=None*, *, *data=None*, **kwargs*)

Plot the angle spectrum.

Compute the angle spectrum (wrapped phase spectrum) of *x*. Data is padded to a length of *pad_to* and the windowing function *window* is applied to the signal.

**Parameters**

**x**

[1-D array or sequence] Array or sequence containing the data.

**Fs**

[float, default: 2] The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, *freqs*, in cycles per time unit.

**window**

[callable or ndarray, default: *window_hanning*] A function or a vector of length *NFFT*. To create window vectors see *window_hanning*, *window_none*, `numpy.blackman`, `numpy.hamming`, `numpy.bartlett`, `scipy.signal`, `scipy.signal.get_window`, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

**sides**

[{'default', 'onesided', 'twosided'}, optional] Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

**pad_to**

[int, optional] The number of points to which the data segment is padded when performing the FFT. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to `fft`. The default is None, which sets *pad_to* equal to the length of the input signal (i.e. no padding).

**Fc**

[int, default: 0] The center frequency of *x*, which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

### Returns

**spectrum**

[1-D array] The values for the angle spectrum in radians (real valued).

**freqs**

[1-D array] The frequencies corresponding to the elements in *spectrum*.

**line**

[*Line2D*] The line created by this function.

### Other Parameters

**data**

[indexable object, optional] If given, the following parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception):

*x*

****kwargs**

Keyword arguments control the *Line2D* properties:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi va |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* or aa | bool |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* or c | *color* |
| *dash_capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *dash_joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *dashes* | sequence of floats (on/off ink in points) or (None, None) |
| *data* | (2, N) array or two 1D arrays |
| *drawstyle* or ds | {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: |
| *figure* | *Figure* |
| *fillstyle* | {'full', 'left', 'right', 'bottom', 'top', 'none'} |
| *gapcolor* | *color* or None |
| *gid* | str |
| *in_layout* | bool |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float |
| *marker* | marker style string, *Path* or *MarkerStyle* |

Table 118 – continued from

| Property | Description |
|---|---|
| markeredgecolor or mec | color |
| markeredgewidth or mew | float |
| markerfacecolor or mfc | color |
| markerfacecoloralt or mfcalt | color |
| markersize or ms | float |
| markevery | None or int or (int, int) or slice or list[int] or float or (float, floa |
| mouseover | bool |
| path_effects | list of AbstractPathEffect |
| picker | float or callable[[Artist, Event], tuple[bool, dict]] |
| pickradius | float |
| rasterized | bool |
| sketch_params | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| solid_capstyle | CapStyle or {'butt', 'projecting', 'round'} |
| solid_joinstyle | JoinStyle or {'miter', 'round', 'bevel'} |
| transform | unknown |
| url | str |
| visible | bool |
| xdata | 1D array |
| ydata | 1D array |
| zorder | float |

**See also:**

***magnitude_spectrum***

Plots the magnitudes of the corresponding frequencies.

***phase_spectrum***

Plots the unwrapped version of this function.

***specgram***

Can plot the angle spectrum of segments within the signal in a colormap.

## Notes

**Note:** This is the *pyplot wrapper* for `axes.Axes.angle_spectrum`.

### matplotlib.pyplot.cohere

matplotlib.pyplot.**cohere**(*x*, *y*, *NFFT=256*, *Fs=2*, *Fc=0*, *detrend=<function detrend_none>*, *window=<function window_hanning>*, *noverlap=0*, *pad_to=None*, *sides='default'*, *scale_by_freq=None*, ***, *data=None*, ***kwargs*)

Plot the coherence between *x* and *y*.

Coherence is the normalized cross spectral density:

$$C_{xy} = \frac{|P_{xy}|^2}{P_{xx}P_{yy}}$$

**Parameters**

**Fs**

[float, default: 2] The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, *freqs*, in cycles per time unit.

**window**

[callable or ndarray, default: `window_hanning`] A function or a vector of length *NFFT*. To create window vectors see `window_hanning`, `window_none`, `numpy.blackman`, `numpy.hamming`, `numpy.bartlett`, `scipy.signal`, `scipy.signal.get_window`, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

**sides**

[{'default', 'onesided', 'twosided'}, optional] Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

**pad_to**

[int, optional] The number of points to which the data segment is padded when performing the FFT. This can be different from *NFFT*, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to `fft`. The default is None, which sets *pad_to* equal to *NFFT*

**NFFT**

[int, default: 256] The number of data points used in each block for the FFT. A power 2 is most efficient. This should *NOT* be used to get zero padding, or the scaling of the result will be incorrect; use *pad_to* for this instead.

**detrend**

[{'none', 'mean', 'linear'} or callable, default: 'none'] The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in

MATLAB, where the *detrend* parameter is a vector, in Matplotlib it is a function. The *mlab* module defines *detrend_none*, *detrend_mean*, and *detrend_linear*, but you can use a custom function as well. You can also use a string to choose one of the functions: 'none' calls *detrend_none*. 'mean' calls *detrend_mean*. 'linear' calls *detrend_linear*.

**scale_by_freq**

[bool, default: True] Whether the resulting density values should be scaled by the scaling frequency, which gives density in units of 1/Hz. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

**noverlap**

[int, default: 0 (no overlap)] The number of points of overlap between blocks.

**Fc**

[int, default: 0] The center frequency of *x*, which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

**Returns**

**Cxy**

[1-D array] The coherence vector.

**freqs**

[1-D array] The frequencies for the elements in *Cxy*.

**Other Parameters**

**data**

[indexable object, optional] If given, the following parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception):

*x*, *y*

****kwargs**

Keyword arguments control the *Line2D* properties:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi va |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* or aa | bool |
| *clip_box* | *BboxBase* or None |

Table 119 – continued from

| Property | Description |
|---|---|
| clip_on | bool |
| clip_path | Patch or (Path, Transform) or None |
| color or c | color |
| dash_capstyle | CapStyle or {'butt', 'projecting', 'round'} |
| dash_joinstyle | JoinStyle or {'miter', 'round', 'bevel'} |
| dashes | sequence of floats (on/off ink in points) or (None, None) |
| data | (2, N) array or two 1D arrays |
| drawstyle or ds | {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: |
| figure | Figure |
| fillstyle | {'full', 'left', 'right', 'bottom', 'top', 'none'} |
| gapcolor | color or None |
| gid | str |
| in_layout | bool |
| label | object |
| linestyle or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| linewidth or lw | float |
| marker | marker style string, Path or MarkerStyle |
| markeredgecolor or mec | color |
| markeredgewidth or mew | float |
| markerfacecolor or mfc | color |
| markerfacecoloralt or mfcalt | color |
| markersize or ms | float |
| markevery | None or int or (int, int) or slice or list[int] or float or (float, floa |
| mouseover | bool |
| path_effects | list of AbstractPathEffect |
| picker | float or callable[[Artist, Event], tuple[bool, dict]] |
| pickradius | float |
| rasterized | bool |
| sketch_params | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| solid_capstyle | CapStyle or {'butt', 'projecting', 'round'} |
| solid_joinstyle | JoinStyle or {'miter', 'round', 'bevel'} |
| transform | unknown |
| url | str |
| visible | bool |
| xdata | 1D array |
| ydata | 1D array |
| zorder | float |

### Notes

**Note:** This is the *pyplot wrapper* for `axes.Axes.cohere`.

### References

Bendat & Piersol -- Random Data: Analysis and Measurement Procedures, John Wiley & Sons (1986)

## matplotlib.pyplot.csd

matplotlib.pyplot.**csd**(*x*, *y*, *NFFT=None*, *Fs=None*, *Fc=None*, *detrend=None*, *window=None*, *noverlap=None*, *pad_to=None*, *sides=None*, *scale_by_freq=None*, *return_line=None*, *, *data=None*, **kwargs*)

Plot the cross-spectral density.

The cross spectral density $P_{xy}$ by Welch's average periodogram method. The vectors $x$ and $y$ are divided into *NFFT* length segments. Each segment is detrended by function *detrend* and windowed by function *window*. *noverlap* gives the length of the overlap between segments. The product of the direct FFTs of $x$ and $y$ are averaged over each segment to compute $P_{xy}$, with a scaling to correct for power loss due to windowing.

If len($x$) < *NFFT* or len($y$) < *NFFT*, they will be zero padded to *NFFT*.

**Parameters**

**x, y**

[1-D arrays or sequences] Arrays or sequences containing the data.

**Fs**

[float, default: 2] The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, *freqs*, in cycles per time unit.

**window**

[callable or ndarray, default: `window_hanning`] A function or a vector of length *NFFT*. To create window vectors see `window_hanning`, `window_none`, `numpy.blackman`, `numpy.hamming`, `numpy.bartlett`, `scipy.signal`, `scipy.signal.get_window`, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

**sides**

[{'default', 'onesided', 'twosided'}, optional] Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

**pad_to**

[int, optional] The number of points to which the data segment is padded when performing the FFT. This can be different from *NFFT*, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to `fft`. The default is None, which sets *pad_to* equal to *NFFT*

**NFFT**

[int, default: 256] The number of data points used in each block for the FFT. A power 2 is most efficient. This should *NOT* be used to get zero padding, or the scaling of the result will be incorrect; use *pad_to* for this instead.

**detrend**

[{'none', 'mean', 'linear'} or callable, default: 'none'] The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the *detrend* parameter is a vector, in Matplotlib it is a function. The `mlab` module defines `detrend_none`, `detrend_mean`, and `detrend_linear`, but you can use a custom function as well. You can also use a string to choose one of the functions: 'none' calls `detrend_none`. 'mean' calls `detrend_mean`. 'linear' calls `detrend_linear`.

**scale_by_freq**

[bool, default: True] Whether the resulting density values should be scaled by the scaling frequency, which gives density in units of 1/Hz. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

**noverlap**

[int, default: 0 (no overlap)] The number of points of overlap between segments.

**Fc**

[int, default: 0] The center frequency of *x*, which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

**return_line**

[bool, default: False] Whether to include the line object plotted in the returned values.

**Returns**

**Pxy**

[1-D array] The values for the cross spectrum $P_{xy}$ before scaling (complex valued).

**freqs**

[1-D array] The frequencies corresponding to the elements in *Pxy*.

**line**

[*Line2D*] The line created by this function. Only returned if *return_line* is True.

**Other Parameters**

**data**

[indexable object, optional] If given, the following parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception):

*x*, *y*

****kwargs**

Keyword arguments control the *Line2D* properties:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi va |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* or aa | bool |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* or c | *color* |
| *dash_capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *dash_joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *dashes* | sequence of floats (on/off ink in points) or (None, None) |
| *data* | (2, N) array or two 1D arrays |
| *drawstyle* or ds | {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: |
| *figure* | *Figure* |
| *fillstyle* | {'full', 'left', 'right', 'bottom', 'top', 'none'} |
| *gapcolor* | *color* or None |
| *gid* | str |
| *in_layout* | bool |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float |
| *marker* | marker style string, *Path* or *MarkerStyle* |
| *markeredgecolor* or mec | *color* |
| *markeredgewidth* or mew | float |
| *markerfacecolor* or mfc | *color* |
| *markerfacecoloralt* or mfcalt | *color* |
| *markersize* or ms | float |
| *markevery* | None or int or (int, int) or slice or list[int] or float or (float, floa |
| *mouseover* | bool |

Table 120 – continued from

| Property | Description |
|---|---|
| path_effects | list of AbstractPathEffect |
| picker | float or callable[[Artist, Event], tuple[bool, dict]] |
| pickradius | float |
| rasterized | bool |
| sketch_params | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| solid_capstyle | CapStyle or {'butt', 'projecting', 'round'} |
| solid_joinstyle | JoinStyle or {'miter', 'round', 'bevel'} |
| transform | unknown |
| url | str |
| visible | bool |
| xdata | 1D array |
| ydata | 1D array |
| zorder | float |

**See also:**

***psd***

is equivalent to setting `y = x`.

### Notes

**Note:** This is the *pyplot wrapper* for `axes.Axes.csd`.

For plotting, the power is plotted as $10\log_{10}(P_{xy})$ for decibels, though $P_{xy}$ itself is returned.

### References

Bendat & Piersol -- Random Data: Analysis and Measurement Procedures, John Wiley & Sons (1986)

## matplotlib.pyplot.magnitude_spectrum

matplotlib.pyplot.**magnitude_spectrum**(*x*, *Fs=None*, *Fc=None*, *window=None*, *pad_to=None*, *sides=None*, *scale=None*, ***, *data=None*, ***kwargs*)

Plot the magnitude spectrum.

Compute the magnitude spectrum of *x*. Data is padded to a length of *pad_to* and the windowing function *window* is applied to the signal.

**Parameters**

**x**

[1-D array or sequence] Array or sequence containing the data.

**Fs**

[float, default: 2] The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, *freqs*, in cycles per time unit.

**window**

[callable or ndarray, default: *window_hanning*] A function or a vector of length *NFFT*. To create window vectors see *window_hanning*, *window_none*, numpy.blackman, numpy.hamming, numpy.bartlett, scipy.signal, scipy.signal.get_window, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

**sides**

[{'default', 'onesided', 'twosided'}, optional] Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

**pad_to**

[int, optional] The number of points to which the data segment is padded when performing the FFT. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to fft. The default is None, which sets *pad_to* equal to the length of the input signal (i.e. no padding).

**scale**

[{'default', 'linear', 'dB'}] The scaling of the values in the *spec*. 'linear' is no scaling. 'dB' returns the values in dB scale, i.e., the dB amplitude (20 * log10). 'default' is 'linear'.

**Fc**

[int, default: 0] The center frequency of *x*, which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

**Returns**

**spectrum**

[1-D array] The values for the magnitude spectrum before scaling (real valued).

**freqs**

[1-D array] The frequencies corresponding to the elements in *spectrum*.

**line**

[*Line2D*] The line created by this function.

**Other Parameters**

**data**

[indexable object, optional] If given, the following parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception):

*x*

****kwargs**

Keyword arguments control the *Line2D* properties:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi va |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* or aa | bool |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* or c | *color* |
| *dash_capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *dash_joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *dashes* | sequence of floats (on/off ink in points) or (None, None) |
| *data* | (2, N) array or two 1D arrays |
| *drawstyle* or ds | {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: |
| *figure* | *Figure* |
| *fillstyle* | {'full', 'left', 'right', 'bottom', 'top', 'none'} |
| *gapcolor* | *color* or None |
| *gid* | str |
| *in_layout* | bool |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float |
| *marker* | marker style string, *Path* or *MarkerStyle* |
| *markeredgecolor* or mec | *color* |
| *markeredgewidth* or mew | float |
| *markerfacecolor* or mfc | *color* |
| *markerfacecoloralt* or mfcalt | *color* |
| *markersize* or ms | float |
| *markevery* | None or int or (int, int) or slice or list[int] or float or (float, floa |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |

Table 121 – continued from

| Property | Description |
|---|---|
| *picker* | float or callable[[Artist, Event], tuple[bool, dict]] |
| *pickradius* | float |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *solid_capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *solid_joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *transform* | unknown |
| *url* | str |
| *visible* | bool |
| *xdata* | 1D array |
| *ydata* | 1D array |
| *zorder* | float |

**See also:**

***psd***

Plots the power spectral density.

***angle_spectrum***

Plots the angles of the corresponding frequencies.

***phase_spectrum***

Plots the phase (unwrapped angle) of the corresponding frequencies.

***specgram***

Can plot the magnitude spectrum of segments within the signal in a colormap.

### Notes

**Note:** This is the *pyplot wrapper* for *axes.Axes.magnitude_spectrum*.

## matplotlib.pyplot.phase_spectrum

matplotlib.pyplot.**phase_spectrum**(*x*, *Fs=None*, *Fc=None*, *window=None*, *pad_to=None*, *sides=None*, *, *data=None*, ***kwargs*)

Plot the phase spectrum.

Compute the phase spectrum (unwrapped angle spectrum) of *x*. Data is padded to a length of *pad_to* and the windowing function *window* is applied to the signal.

**Parameters**

**x**

[1-D array or sequence] Array or sequence containing the data

**Fs**

[float, default: 2] The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, *freqs*, in cycles per time unit.

**window**

[callable or ndarray, default: *window_hanning*] A function or a vector of length *NFFT*. To create window vectors see *window_hanning*, *window_none*, numpy.blackman, numpy.hamming, numpy.bartlett, scipy.signal, scipy.signal.get_window, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

**sides**

[{'default', 'onesided', 'twosided'}, optional] Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

**pad_to**

[int, optional] The number of points to which the data segment is padded when performing the FFT. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to fft. The default is None, which sets *pad_to* equal to the length of the input signal (i.e. no padding).

**Fc**

[int, default: 0] The center frequency of *x*, which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

**Returns**

**spectrum**

[1-D array] The values for the phase spectrum in radians (real valued).

**freqs**

[1-D array] The frequencies corresponding to the elements in *spectrum*.

**line**

[*Line2D*] The line created by this function.

**Other Parameters**

**data**

[indexable object, optional] If given, the following parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception):

*x*

****kwargs**

Keyword arguments control the *Line2D* properties:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi va |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* or aa | bool |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* or c | *color* |
| *dash_capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *dash_joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *dashes* | sequence of floats (on/off ink in points) or (None, None) |
| *data* | (2, N) array or two 1D arrays |
| *drawstyle* or ds | {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: |
| *figure* | *Figure* |
| *fillstyle* | {'full', 'left', 'right', 'bottom', 'top', 'none'} |
| *gapcolor* | *color* or None |
| *gid* | str |
| *in_layout* | bool |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float |
| *marker* | marker style string, *Path* or *MarkerStyle* |
| *markeredgecolor* or mec | *color* |
| *markeredgewidth* or mew | float |
| *markerfacecolor* or mfc | *color* |
| *markerfacecoloralt* or mfcalt | *color* |
| *markersize* or ms | float |
| *markevery* | None or int or (int, int) or slice or list[int] or float or (float, floa |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | float or callable[[Artist, Event], tuple[bool, dict]] |
| *pickradius* | float |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *solid_capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |

Table 122 – continued from

| Property | Description |
|---|---|
| solid_joinstyle | JoinStyle or {'miter', 'round', 'bevel'} |
| transform | unknown |
| url | str |
| visible | bool |
| xdata | 1D array |
| ydata | 1D array |
| zorder | float |

**See also:**

**magnitude_spectrum**

Plots the magnitudes of the corresponding frequencies.

**angle_spectrum**

Plots the wrapped version of this function.

**specgram**

Can plot the phase spectrum of segments within the signal in a colormap.

### Notes

**Note:** This is the *pyplot wrapper* for axes.Axes.phase_spectrum.

## matplotlib.pyplot.psd

matplotlib.pyplot.**psd**(*x*, *NFFT=None*, *Fs=None*, *Fc=None*, *detrend=None*, *window=None*, *noverlap=None*, *pad_to=None*, *sides=None*, *scale_by_freq=None*, *return_line=None*, ***, *data=None*, ***kwargs*)

Plot the power spectral density.

The power spectral density $P_{xx}$ by Welch's average periodogram method. The vector $x$ is divided into *NFFT* length segments. Each segment is detrended by function *detrend* and windowed by function *window*. *noverlap* gives the length of the overlap between segments. The $|\mathrm{fft}(i)|^2$ of each segment $i$ are averaged to compute $P_{xx}$, with a scaling to correct for power loss due to windowing.

If len($x$) < *NFFT*, it will be zero padded to *NFFT*.

**Parameters**

**x**

[1-D array or sequence] Array or sequence containing the data

**Fs**

[float, default: 2] The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, *freqs*, in cycles per time unit.

**window**

[callable or ndarray, default: *window_hanning*] A function or a vector of length *NFFT*. To create window vectors see *window_hanning*, *window_none*, numpy.blackman, numpy.hamming, numpy.bartlett, scipy.signal, scipy.signal.get_window, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

**sides**

[{'default', 'onesided', 'twosided'}, optional] Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

**pad_to**

[int, optional] The number of points to which the data segment is padded when performing the FFT. This can be different from *NFFT*, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to fft. The default is None, which sets *pad_to* equal to *NFFT*

**NFFT**

[int, default: 256] The number of data points used in each block for the FFT. A power 2 is most efficient. This should *NOT* be used to get zero padding, or the scaling of the result will be incorrect; use *pad_to* for this instead.

**detrend**

[{'none', 'mean', 'linear'} or callable, default: 'none'] The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the *detrend* parameter is a vector, in Matplotlib it is a function. The *mlab* module defines *detrend_none*, *detrend_mean*, and *detrend_linear*, but you can use a custom function as well. You can also use a string to choose one of the functions: 'none' calls *detrend_none*. 'mean' calls *detrend_mean*. 'linear' calls *detrend_linear*.

**scale_by_freq**

[bool, default: True] Whether the resulting density values should be scaled by the scaling frequency, which gives density in units of 1/Hz. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

**noverlap**

[int, default: 0 (no overlap)] The number of points of overlap between segments.

**Fc**

[int, default: 0] The center frequency of *x*, which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

**return_line**

[bool, default: False] Whether to include the line object plotted in the returned values.

**Returns**

**Pxx**

[1-D array] The values for the power spectrum $P_{xx}$ before scaling (real valued).

**freqs**

[1-D array] The frequencies corresponding to the elements in *Pxx*.

**line**

[`Line2D`] The line created by this function. Only returned if *return_line* is True.

**Other Parameters**

**data**

[indexable object, optional] If given, the following parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception):

*x*

****kwargs**

Keyword arguments control the `Line2D` properties:

| Property | Description |
|---|---|
| `agg_filter` | a filter function, which takes a (m, n, 3) float array and a dpi va |
| `alpha` | scalar or None |
| `animated` | bool |
| `antialiased` or aa | bool |
| `clip_box` | `BboxBase` or None |
| `clip_on` | bool |
| `clip_path` | Patch or (Path, Transform) or None |
| `color` or c | *color* |
| `dash_capstyle` | `CapStyle` or {'butt', 'projecting', 'round'} |
| `dash_joinstyle` | `JoinStyle` or {'miter', 'round', 'bevel'} |
| `dashes` | sequence of floats (on/off ink in points) or (None, None) |
| `data` | (2, N) array or two 1D arrays |
| `drawstyle` or ds | {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: |

Table 123 – continued from

| Property | Description |
|---|---|
| *figure* | *Figure* |
| *fillstyle* | {'full', 'left', 'right', 'bottom', 'top', 'none'} |
| *gapcolor* | *color* or None |
| *gid* | str |
| *in_layout* | bool |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float |
| *marker* | marker style string, *Path* or *MarkerStyle* |
| *markeredgecolor* or mec | *color* |
| *markeredgewidth* or mew | float |
| *markerfacecolor* or mfc | *color* |
| *markerfacecoloralt* or mfcalt | *color* |
| *markersize* or ms | float |
| *markevery* | None or int or (int, int) or slice or list[int] or float or (float, floa |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | float or callable[[Artist, Event], tuple[bool, dict]] |
| *pickradius* | float |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *solid_capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *solid_joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *transform* | unknown |
| *url* | str |
| *visible* | bool |
| *xdata* | 1D array |
| *ydata* | 1D array |
| *zorder* | float |

**See also:**

***specgram***

Differs in the default overlap; in not returning the mean of the segment periodograms; in returning the times of the segments; and in plotting a colormap instead of a line.

***magnitude_spectrum***

Plots the magnitude spectrum.

***csd***

Plots the spectral density between two signals.

### Notes

**Note:** This is the *pyplot wrapper* for `axes.Axes.psd`.

For plotting, the power is plotted as $10\log_{10}(P_{xx})$ for decibels, though *Pxx* itself is returned.

### References

Bendat & Piersol -- Random Data: Analysis and Measurement Procedures, John Wiley & Sons (1986)

## matplotlib.pyplot.specgram

matplotlib.pyplot.**specgram**(*x*, *NFFT=None*, *Fs=None*, *Fc=None*, *detrend=None*, *window=None*, *noverlap=None*, *cmap=None*, *xextent=None*, *pad_to=None*, *sides=None*, *scale_by_freq=None*, *mode=None*, *scale=None*, *vmin=None*, *vmax=None*, *, *data=None*, **kwargs*)

Plot a spectrogram.

Compute and plot a spectrogram of data in *x*. Data are split into *NFFT* length segments and the spectrum of each section is computed. The windowing function *window* is applied to each segment, and the amount of overlap of each segment is specified with *noverlap*. The spectrogram is plotted as a colormap (using imshow).

**Parameters**

**x**

[1-D array or sequence] Array or sequence containing the data.

**Fs**

[float, default: 2] The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, *freqs*, in cycles per time unit.

**window**

[callable or ndarray, default: `window_hanning`] A function or a vector of length *NFFT*. To create window vectors see `window_hanning`, `window_none`, `numpy.blackman`, `numpy.hamming`, `numpy.bartlett`, `scipy.signal`, `scipy.signal.get_window`, etc. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

**sides**

[{'default', 'onesided', 'twosided'}, optional] Which sides of the spectrum to return. 'default' is one-sided for real data and two-sided for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

**pad_to**

[int, optional] The number of points to which the data segment is padded when performing the FFT. This can be different from *NFFT*, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to `fft`. The default is None, which sets *pad_to* equal to *NFFT*

**NFFT**

[int, default: 256] The number of data points used in each block for the FFT. A power 2 is most efficient. This should *NOT* be used to get zero padding, or the scaling of the result will be incorrect; use *pad_to* for this instead.

**detrend**

[{'none', 'mean', 'linear'} or callable, default: 'none'] The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the *detrend* parameter is a vector, in Matplotlib it is a function. The `mlab` module defines `detrend_none`, `detrend_mean`, and `detrend_linear`, but you can use a custom function as well. You can also use a string to choose one of the functions: 'none' calls `detrend_none`. 'mean' calls `detrend_mean`. 'linear' calls `detrend_linear`.

**scale_by_freq**

[bool, default: True] Whether the resulting density values should be scaled by the scaling frequency, which gives density in units of 1/Hz. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

**mode**

[{'default', 'psd', 'magnitude', 'angle', 'phase'}] What sort of spectrum to use. Default is 'psd', which takes the power spectral density. 'magnitude' returns the magnitude spectrum. 'angle' returns the phase spectrum without unwrapping. 'phase' returns the phase spectrum with unwrapping.

**noverlap**

[int, default: 128] The number of points of overlap between blocks.

**scale**

[{'default', 'linear', 'dB'}] The scaling of the values in the *spec*. 'linear' is no scaling. 'dB' returns the values in dB scale. When *mode* is 'psd', this is dB power (10 * log10). Otherwise, this is dB amplitude (20 * log10). 'default' is 'dB' if *mode* is 'psd' or 'magnitude' and 'linear' otherwise. This must be 'linear' if *mode* is 'angle' or 'phase'.

**Fc**

[int, default: 0] The center frequency of *x*, which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and

downsampled to baseband.

**cmap**

[*Colormap*, default: *rcParams["image.cmap"]* (default: 'viridis')]

**xextent**

[*None* or (xmin, xmax)] The image extent along the x-axis. The default sets *xmin* to the left border of the first bin (*spectrum* column) and *xmax* to the right border of the last bin. Note that for *noverlap>0* the width of the bins is smaller than those of the segments.

**data**

[indexable object, optional] If given, the following parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception):

*x*

****kwargs**

Additional keyword arguments are passed on to *imshow* which makes the specgram image. The origin keyword argument is not supported.

**Returns**

**spectrum**

[2D array] Columns are the periodograms of successive segments.

**freqs**

[1-D array] The frequencies corresponding to the rows in *spectrum*.

**t**

[1-D array] The times corresponding to midpoints of segments (i.e., the columns in *spectrum*).

**im**

[*AxesImage*] The image created by imshow containing the spectrogram.

**See also:**

***psd***

Differs in the default overlap; in returning the mean of the segment periodograms; in not returning times; and in generating a line plot instead of colormap.

***magnitude_spectrum***

A single spectrum, similar to having a single segment when *mode* is 'magnitude'. Plots a line instead of a colormap.

***angle_spectrum***

A single spectrum, similar to having a single segment when *mode* is 'angle'. Plots a line instead of a colormap.

***phase_spectrum***

A single spectrum, similar to having a single segment when *mode* is 'phase'. Plots a line instead of a colormap.

### Notes

---

**Note:** This is the *pyplot wrapper* for `axes.Axes.specgram`.

---

The parameters *detrend* and *scale_by_freq* do only apply when *mode* is set to 'psd'.

### Examples using `matplotlib.pyplot.specgram`

- *Spectrogram*

## matplotlib.pyplot.xcorr

matplotlib.pyplot.**xcorr**(*x*, *y*, *normed=True*, *detrend=<function detrend_none>*, *usevlines=True*, *maxlags=10*, ***, *data=None*, ***kwargs*)

Plot the cross correlation between *x* and *y*.

The correlation with lag k is defined as $\sum_n x[n+k] \cdot y^*[n]$, where $y^*$ is the complex conjugate of $y$.

**Parameters**

**x, y**

[array-like of length n] Neither *x* nor *y* are run through Matplotlib's unit conversion, so these should be unit-less arrays.

**detrend**

[callable, default: `mlab.detrend_none` (no detrending)] A detrending function applied to *x* and *y*. It must have the signature

```
detrend(x: np.ndarray) -> np.ndarray
```

**normed**

[bool, default: True] If `True`, input vectors are normalised to unit length.

**usevlines**

[bool, default: True] Determines the plot style.

If `True`, vertical lines are plotted from 0 to the xcorr value using *Axes.vlines*. Additionally, a horizontal line is plotted at y=0 using *Axes.axhline*.

If `False`, markers are plotted at the xcorr values using *Axes.plot*.

**maxlags**

[int, default: 10] Number of lags to show. If None, will return all `2 * len(x) - 1` lags.

**Returns**

**lags**

[array (length `2*maxlags+1`)] The lag vector.

**c**

[array (length `2*maxlags+1`)] The auto correlation vector.

**line**

[*LineCollection* or *Line2D*] *Artist* added to the Axes of the correlation:

- *LineCollection* if *usevlines* is True.
- *Line2D* if *usevlines* is False.

**b**

[*Line2D* or None] Horizontal line at 0 if *usevlines* is True None *usevlines* is False.

**Other Parameters**

**linestyle**

[*Line2D* property, optional] The linestyle for plotting the data points. Only used if *usevlines* is `False`.

**marker**

[str, default: 'o'] The marker for plotting the data points. Only used if *usevlines* is `False`.

**data**

[indexable object, optional] If given, the following parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception):

*x*, *y*

****kwargs**

Additional parameters are passed to *Axes.vlines* and *Axes.axhline* if *usevlines* is `True`; otherwise they are passed to *Axes.plot*.

### Notes

**Note:** This is the *pyplot wrapper* for `axes.Axes.xcorr`.

The cross correlation is performed with `numpy.correlate` with `mode = "full"`.

## Examples using `matplotlib.pyplot.xcorr`

- *Cross- and auto-correlation*

## Statistics

| | |
|---|---|
| `ecdf` | Compute and plot the empirical cumulative distribution function of *x*. |
| `boxplot` | Draw a box and whisker plot. |
| `violinplot` | Make a violin plot. |

## matplotlib.pyplot.ecdf

matplotlib.pyplot.**ecdf**(*x*, *weights=None*, ***, *complementary=False*, *orientation='vertical'*, *compress=False*, *data=None*, ***kwargs*)

Compute and plot the empirical cumulative distribution function of *x*.

New in version 3.8.

**Parameters**

**x**

[1d array-like] The input data. Infinite entries are kept (and move the relevant end of the ecdf from 0/1), but NaNs and masked values are errors.

**weights**

[1d array-like or None, default: None] The weights of the entries; must have the same shape as *x*. Weights corresponding to NaN data points are dropped, and then the remaining weights are normalized to sum to 1. If unset, all entries have the same weight.

**complementary**

[bool, default: False] Whether to plot a cumulative distribution function, which increases from 0 to 1 (the default), or a complementary cumulative distribution function, which decreases from 1 to 0.

**orientation**

[{"vertical", "horizontal"}, default: "vertical"] Whether the entries are plotted along the x-axis ("vertical", the default) or the y-axis ("horizontal"). This parameter takes the same values as in *hist*.

**compress**

[bool, default: False] Whether multiple entries with the same values are grouped together (with a summed weight) before plotting. This is mainly useful if *x* contains many identical data points, to decrease the rendering complexity of the plot. If *x* contains no duplicate points, this has no effect and just uses some time and memory.

**Returns**

***Line2D***

**Other Parameters**

**data**

[indexable object, optional] If given, the following parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception):

*x*, *weights*

****kwargs**

Keyword arguments control the *Line2D* properties:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi va |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* or aa | bool |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* or c | *color* |
| *dash_capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *dash_joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *dashes* | sequence of floats (on/off ink in points) or (None, None) |
| *data* | (2, N) array or two 1D arrays |
| *drawstyle* or ds | {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: |
| *figure* | *Figure* |
| *fillstyle* | {'full', 'left', 'right', 'bottom', 'top', 'none'} |
| *gapcolor* | *color* or None |
| *gid* | str |

Table 124 – continued from

| Property | Description |
|---|---|
| *in_layout* | bool |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float |
| *marker* | marker style string, *Path* or *MarkerStyle* |
| *markeredgecolor* or mec | *color* |
| *markeredgewidth* or mew | float |
| *markerfacecolor* or mfc | *color* |
| *markerfacecoloralt* or mfcalt | *color* |
| *markersize* or ms | float |
| *markevery* | None or int or (int, int) or slice or list[int] or float or (float, floa |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | float or callable[[Artist, Event], tuple[bool, dict]] |
| *pickradius* | float |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *solid_capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *solid_joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *transform* | unknown |
| *url* | str |
| *visible* | bool |
| *xdata* | 1D array |
| *ydata* | 1D array |
| *zorder* | float |

## Notes

**Note:** This is the *pyplot wrapper* for *axes.Axes.ecdf*.

The ecdf plot can be thought of as a cumulative histogram with one bin per data entry; i.e. it reports on the entire dataset without any arbitrary binning.

If *x* contains NaNs or masked entries, either remove them first from the array (if they should not taken into account), or replace them by -inf or +inf (if they should be sorted at the beginning or the end of the array).

### Examples using `matplotlib.pyplot.ecdf`

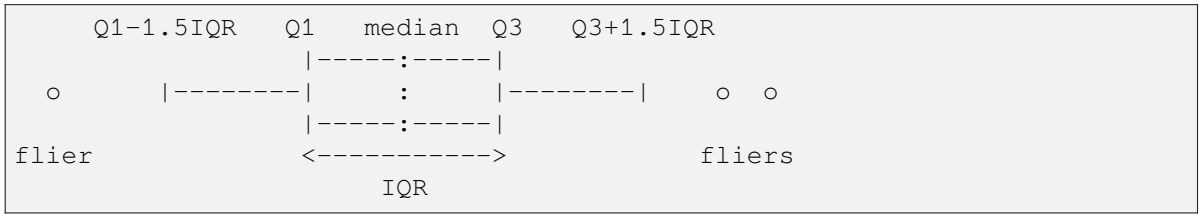
- *Plotting cumulative distributions*

### matplotlib.pyplot.boxplot

matplotlib.pyplot.**boxplot**(*x*, *notch=None*, *sym=None*, *vert=None*, *whis=None*, *positions=None*, *widths=None*, *patch_artist=None*, *bootstrap=None*, *usermedians=None*, *conf_intervals=None*, *meanline=None*, *showmeans=None*, *showcaps=None*, *showbox=None*, *showfliers=None*, *boxprops=None*, *tick_labels=None*, *flierprops=None*, *medianprops=None*, *meanprops=None*, *capprops=None*, *whiskerprops=None*, *manage_ticks=True*, *autorange=False*, *zorder=None*, *capwidths=None*, *label=None*, *, *data=None*)

Draw a box and whisker plot.

The box extends from the first quartile (Q1) to the third quartile (Q3) of the data, with a line at the median. The whiskers extend from the box to the farthest data point lying within 1.5x the inter-quartile range (IQR) from the box. Flier points are those past the end of the whiskers. See https://en.wikipedia.org/wiki/Box_plot for reference.




**Parameters**

**x**

[Array or a sequence of vectors.] The input data. If a 2D array, a boxplot is drawn for each column in *x*. If a sequence of 1D arrays, a boxplot is drawn for each array in *x*.

**notch**

[bool, default: `rcParams["boxplot.notch"]` (default: `False`)] Whether to draw a notched boxplot (`True`), or a rectangular boxplot (`False`). The notches represent the confidence interval (CI) around the median. The documentation for *bootstrap* describes how the locations of the notches are computed by default, but their locations may also be overridden by setting the *conf_intervals* parameter.

---

**Note:** In cases where the values of the CI are less than the lower quartile or greater than the upper quartile, the notches will extend beyond the box, giving it

a distinctive "flipped" appearance. This is expected behavior and consistent with other statistical visualization packages.

---

**sym**

[str, optional] The default symbol for flier points. An empty string ('') hides the fliers. If `None`, then the fliers default to 'b+'. More control is provided by the *flierprops* parameter.

**vert**

[bool, default: *rcParams["boxplot.vertical"]* (default: `True`)] If `True`, draws vertical boxes. If `False`, draw horizontal boxes.

**whis**

[float or (float, float), default: 1.5] The position of the whiskers.

If a float, the lower whisker is at the lowest datum above `Q1 - whis*(Q3-Q1)`, and the upper whisker at the highest datum below `Q3 + whis*(Q3-Q1)`, where Q1 and Q3 are the first and third quartiles. The default value of `whis = 1.5` corresponds to Tukey's original definition of boxplots.

If a pair of floats, they indicate the percentiles at which to draw the whiskers (e.g., (5, 95)). In particular, setting this to (0, 100) results in whiskers covering the whole range of the data.

In the edge case where `Q1 == Q3`, *whis* is automatically set to (0, 100) (cover the whole range of the data) if *autorange* is True.

Beyond the whiskers, data are considered outliers and are plotted as individual points.

**bootstrap**

[int, optional] Specifies whether to bootstrap the confidence intervals around the median for notched boxplots. If *bootstrap* is None, no bootstrapping is performed, and notches are calculated using a Gaussian-based asymptotic approximation (see McGill, R., Tukey, J.W., and Larsen, W.A., 1978, and Kendall and Stuart, 1967). Otherwise, bootstrap specifies the number of times to bootstrap the median to determine its 95% confidence intervals. Values between 1000 and 10000 are recommended.

**usermedians**

[1D array-like, optional] A 1D array-like of length `len(x)`. Each entry that is not `None` forces the value of the median for the corresponding dataset. For entries that are `None`, the medians are computed by Matplotlib as normal.

**conf_intervals**

[array-like, optional] A 2D array-like of shape `(len(x), 2)`. Each entry that is not None forces the location of the corresponding notch (which is only drawn if *notch* is `True`). For entries that are `None`, the notches are computed by the method specified by the other parameters (e.g., *bootstrap*).

**positions**

[array-like, optional] The positions of the boxes. The ticks and limits are automatically set to match the positions. Defaults to `range(1, N+1)` where N is the number of boxes to be drawn.

**widths**

[float or array-like] The widths of the boxes. The default is 0.5, or `0.15*(distance between extreme positions)`, if that is smaller.

**patch_artist**

[bool, default: *rcParams["boxplot.patchartist"]* (default: `False`)] If `False` produces boxes with the Line2D artist. Otherwise, boxes are drawn with Patch artists.

**tick_labels**

[list of str, optional] The tick labels of each boxplot. Ticks are always placed at the box *positions*. If *tick_labels* is given, the ticks are labelled accordingly. Otherwise, they keep their numeric values.

Changed in version 3.9: Renamed from *labels*, which is deprecated since 3.9 and will be removed in 3.11.

**manage_ticks**

[bool, default: True] If True, the tick locations and labels will be adjusted to match the boxplot positions.

**autorange**

[bool, default: False] When `True` and the data are distributed such that the 25th and 75th percentiles are equal, *whis* is set to (0, 100) such that the whisker ends are at the minimum and maximum of the data.

**meanline**

[bool, default: *rcParams["boxplot.meanline"]* (default: `False`)] If `True` (and *showmeans* is `True`), will try to render the mean as a line spanning the full width of the box according to *meanprops* (see below). Not recommended if *shownotches* is also True. Otherwise, means will be shown as points.

**zorder**

[float, default: `Line2D.zorder = 2`] The zorder of the boxplot.

**Returns**

**dict**

A dictionary mapping each component of the boxplot to a list of the *Line2D* instances created. That dictionary has the following keys (assuming vertical boxplots):

- `boxes`: the main body of the boxplot showing the quartiles and the median's confidence intervals if enabled.
- `medians`: horizontal lines at the median of each box.
- `whiskers`: the vertical lines extending to the most extreme, non-outlier data points.
- `caps`: the horizontal lines at the ends of the whiskers.
- `fliers`: points representing data that extend beyond the whiskers (fliers).
- `means`: points or lines representing the means.

**Other Parameters**

**showcaps**

[bool, default: *rcParams["boxplot.showcaps"]* (default: `True`)] Show the caps on the ends of whiskers.

**showbox**

[bool, default: *rcParams["boxplot.showbox"]* (default: `True`)] Show the central box.

**showfliers**

[bool, default: *rcParams["boxplot.showfliers"]* (default: `True`)] Show the outliers beyond the caps.

**showmeans**

[bool, default: *rcParams["boxplot.showmeans"]* (default: `False`)] Show the arithmetic means.

**capprops**

[dict, default: None] The style of the caps.

**capwidths**

[float or array, default: None] The widths of the caps.

**boxprops**

[dict, default: None] The style of the box.

**whiskerprops**

[dict, default: None] The style of the whiskers.

**flierprops**

[dict, default: None] The style of the fliers.

**medianprops**

[dict, default: None] The style of the median.

**meanprops**

[dict, default: None] The style of the mean.

**label**

[str or list of str, optional] Legend labels. Use a single string when all boxes have the same style and you only want a single legend entry for them. Use a list of strings to label all boxes individually. To be distinguishable, the boxes should be styled individually, which is currently only possible by modifying the returned artists, see e.g. *Boxplots*.

In the case of a single string, the legend entry will technically be associated with the first box only. By default, the legend will show the median line (`result["medians"]`); if *patch_artist* is True, the legend will show the box `Patch` artists (`result["boxes"]`) instead.

New in version 3.9.

**data**

[indexable object, optional] If given, all parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception).

**See also:**

***Axes.bxp***

Draw a boxplot from pre-computed statistics.

***violinplot***

Draw an estimate of the probability density function.

## Notes

**Note:** This is the *pyplot wrapper* for `axes.Axes.boxplot`.

## Examples using `matplotlib.pyplot.boxplot`

- *Artist customization in box plots*
- *Box plots with custom fill colors*
- *Boxplots*
- *Box plot vs. violin plot comparison*

### matplotlib.pyplot.violinplot

matplotlib.pyplot.**violinplot**(*dataset*, *positions=None*, *vert=True*, *widths=0.5*, *showmeans=False*, *showextrema=True*, *showmedians=False*, *quantiles=None*, *points=100*, *bw_method=None*, *side='both'*, ***, *data=None*)

Make a violin plot.

Make a violin plot for each column of *dataset* or each vector in sequence *dataset*. Each filled area extends to represent the entire data range, with optional lines at the mean, the median, the minimum, the maximum, and user-specified quantiles.

**Parameters**

**dataset**

[Array or a sequence of vectors.] The input data.

**positions**

[array-like, default: [1, 2, ..., n]] The positions of the violins; i.e. coordinates on the x-axis for vertical violins (or y-axis for horizontal violins).

**vert**

[bool, default: True.] If true, creates a vertical violin plot. Otherwise, creates a horizontal violin plot.

**widths**

[float or array-like, default: 0.5] The maximum width of each violin in units of the *positions* axis. The default is 0.5, which is half the available space when using default *positions*.

**showmeans**

[bool, default: False] Whether to show the mean with a line.

**showextrema**

[bool, default: True] Whether to show extrema with a line.

**showmedians**

[bool, default: False] Whether to show the median with a line.

**quantiles**

[array-like, default: None] If not None, set a list of floats in interval [0, 1] for each violin, which stands for the quantiles that will be rendered for that violin.

**points**

[int, default: 100] The number of points to evaluate each of the gaussian kernel density estimations at.

**bw_method**

[{'scott', 'silverman'} or float or callable, default: 'scott'] The method used to calculate the estimator bandwidth. If a float, this will be used directly as `kde.factor`. If a callable, it should take a *matplotlib.mlab.GaussianKDE* instance as its only parameter and return a float.

**side**

[{'both', 'low', 'high'}, default: 'both'] 'both' plots standard violins. 'low'/'high' only plots the side below/above the positions value.

**data**

[indexable object, optional] If given, the following parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception):

*dataset*

**Returns**

**dict**

A dictionary mapping each component of the violinplot to a list of the corresponding collection instances created. The dictionary has the following keys:

- `bodies`: A list of the *PolyCollection* instances containing the filled area of each violin.
- `cmeans`: A *LineCollection* instance that marks the mean values of each of the violin's distribution.
- `cmins`: A *LineCollection* instance that marks the bottom of each violin's distribution.
- `cmaxes`: A *LineCollection* instance that marks the top of each violin's distribution.
- `cbars`: A *LineCollection* instance that marks the centers of each violin's distribution.
- `cmedians`: A *LineCollection* instance that marks the median values of each of the violin's distribution.
- `cquantiles`: A *LineCollection* instance created to identify the quantile values of each of the violin's distribution.

**See also:**

***Axes.violin***

Draw a violin from pre-computed statistics.

***boxplot***

Draw a box and whisker plot.

### Notes

**Note:** This is the *pyplot wrapper* for `axes.Axes.violinplot`.

### Examples using `matplotlib.pyplot.violinplot`

- *Box plot vs. violin plot comparison*
- *Violin plot customization*
- *Violin plot basics*

## Binned

| | |
|---|---|
| `hexbin` | Make a 2D hexagonal binning plot of points *x*, *y*. |
| `hist` | Compute and plot a histogram. |
| `hist2d` | Make a 2D histogram plot. |
| `stairs` | Draw a stepwise constant function as a line or a filled plot. |

### matplotlib.pyplot.hexbin

matplotlib.pyplot.**hexbin**(*x*, *y*, *C=None*, *gridsize=100*, *bins=None*, *xscale='linear'*, *yscale='linear'*, *extent=None*, *cmap=None*, *norm=None*, *vmin=None*, *vmax=None*, *alpha=None*, *linewidths=None*, *edgecolors='face'*, *reduce_C_function=<function mean>*, *mincnt=None*, *marginals=False*, ***, *data=None*, ***kwargs*)

Make a 2D hexagonal binning plot of points *x*, *y*.

If *C* is *None*, the value of the hexagon is determined by the number of points in the hexagon. Otherwise, *C* specifies values at the coordinate (x[i], y[i]). For each hexagon, these values are reduced using *reduce_C_function*.

**Parameters**

**x, y**

[array-like] The data positions. *x* and *y* must be of the same length.

**C**

[array-like, optional] If given, these values are accumulated in the bins. Otherwise, every point has a value of 1. Must be of the same length as *x* and *y*.

**gridsize**

[int or (int, int), default: 100] If a single int, the number of hexagons in the *x*-direction. The number of hexagons in the *y*-direction is chosen such that the hexagons are approximately regular.

Alternatively, if a tuple (*nx*, *ny*), the number of hexagons in the *x*-direction and the *y*-direction. In the *y*-direction, counting is done along vertically aligned hexagons, not along the zig-zag chains of hexagons; see the following illustration.

To get approximately regular hexagons, choose $n_x = \sqrt{3}\,n_y$.

**bins**

['log' or int or sequence, default: None] Discretization of the hexagon values.

- If *None*, no binning is applied; the color of each hexagon directly corresponds to its count value.
- If 'log', use a logarithmic scale for the colormap. Internally, $log_{10}(i+1)$ is used to determine the hexagon color. This is equivalent to `norm=LogNorm()`.
- If an integer, divide the counts in the specified number of bins, and color the hexagons accordingly.
- If a sequence of values, the values of the lower bound of the bins to be used.

**xscale**

[{'linear', 'log'}, default: 'linear'] Use a linear or log10 scale on the horizontal axis.

**yscale**

[{'linear', 'log'}, default: 'linear'] Use a linear or log10 scale on the vertical axis.

**mincnt**

[int >= 0, default: *None*] If not *None*, only display cells with at least *mincnt* number of points in the cell.

**marginals**

[bool, default: *False*] If marginals is *True*, plot the marginal density as colormapped rectangles along the bottom of the x-axis and left of the y-axis.

**extent**

[4-tuple of float, default: *None*] The limits of the bins (xmin, xmax, ymin, ymax). The default assigns the limits based on *gridsize*, *x*, *y*, *xscale* and *yscale*.

If *xscale* or *yscale* is set to 'log', the limits are expected to be the exponent for a power of 10. E.g. for x-limits of 1 and 50 in 'linear' scale and y-limits of 10 and 1000 in 'log' scale, enter (1, 50, 1, 3).

**Returns**

*PolyCollection*

A *PolyCollection* defining the hexagonal bins.

- *PolyCollection.get_offsets* contains a Mx2 array containing the x, y positions of the M hexagon centers.
- *PolyCollection.get_array* contains the values of the M hexagons.

If *marginals* is *True*, horizontal bar and vertical bar (both PolyCollections) will be attached to the return collection as attributes *hbar* and *vbar*.

**Other Parameters**

**cmap**

[str or *Colormap*, default: *rcParams["image.cmap"]* (default: `'viridis'`)] The Colormap instance or registered colormap name used to map scalar data to colors.

**norm**

[str or *Normalize*, optional] The normalization method used to scale scalar data to the [0, 1] range before mapping to colors using *cmap*. By default, a linear scaling is used, mapping the lowest value to 0 and the highest to 1.

If given, this can be one of the following:

- An instance of *Normalize* or one of its subclasses (see *Colormap normalization*).
- A scale name, i.e. one of "linear", "log", "symlog", "logit", etc. For a list of available scales, call *matplotlib.scale.get_scale_names()*. In that case, a suitable *Normalize* subclass is dynamically generated and instantiated.

**vmin, vmax**

[float, optional] When using scalar data and no explicit *norm*, *vmin* and *vmax* define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data. It is an error to use *vmin*/*vmax* when a *norm* instance is given (but using a `str` *norm* name together with *vmin*/*vmax* is acceptable).

**alpha**

[float between 0 and 1, optional] The alpha blending value, between 0 (transparent) and 1 (opaque).

**linewidths**

[float, default: *None*] If *None*, defaults to *rcParams["patch.linewidth"]* (default: `1.0`).

**edgecolors**

[{'face', 'none', *None*} or color, default: 'face'] The color of the hexagon edges. Possible values are:

- 'face': Draw the edges in the same color as the fill color.
- 'none': No edges are drawn. This can sometimes lead to unsightly unpainted pixels between the hexagons.
- *None*: Draw outlines in the default color.
- An explicit color.

**reduce_C_function**

[callable, default: `numpy.mean`] The function to aggregate *C* within the bins. It is ignored if *C* is not given. This must have the signature:

```
def reduce_C_function(C: array) -> float
```

Commonly used functions are:

- `numpy.mean`: average of the points
- `numpy.sum`: integral of the point values
- `numpy.amax`: value taken from the largest point

By default will only reduce cells with at least 1 point because some reduction functions (such as `numpy.amax`) will error/warn with empty input. Changing *mincnt* will adjust the cutoff, and if set to 0 will pass empty input to the reduction function.

**data**

[indexable object, optional] If given, the following parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception):

*x*, *y*, *C*

****kwargs**

[*PolyCollection* properties] All other keyword arguments are passed on to *PolyCollection*:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a d |
| *alpha* | array-like or scalar or None |
| *animated* | bool |
| *antialiased* or aa or antialiaseds | bool or list of bools |
| *array* | array-like or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clim* | (vmin: float, vmax: float) |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |

Table 125 – continued fr

| Property | Description |
|---|---|
| *clip_path* | Patch or (Path, Transform) or None |
| *cmap* | *Colormap* or str or None |
| *color* | *color* or list of RGBA tuples |
| *edgecolor* or ec or edgecolors | *color* or list of *color* or 'face' |
| *facecolor* or facecolors or fc | *color* or list of *color* |
| *figure* | *Figure* |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or dashes or linestyles or ls | str or tuple or list thereof |
| *linewidth* or linewidths or lw | float or list of floats |
| *mouseover* | bool |
| *norm* | *Normalize* or str or None |
| *offset_transform* or transOffset | *Transform* |
| *offsets* | (N, 2) or (2,) array-like |
| *path_effects* | list of *AbstractPathEffect* |
| *paths* | list of array-like |
| *picker* | None or bool or float or callable |
| *pickradius* | float |
| *rasterized* | bool |
| sizes | numpy.ndarray or None |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *urls* | list of str or None |
| *verts* | list of array-like |
| *verts_and_codes* | unknown |
| *visible* | bool |
| *zorder* | float |

**See also:**

***hist2d***

2D histogram rectangular bins

### Notes

**Note:** This is the *pyplot wrapper* for `axes.Axes.hexbin`.

## Examples using `matplotlib.pyplot.hexbin`

- *Hexagonal binned plot*

## matplotlib.pyplot.hist

matplotlib.pyplot.**hist**(*x*, *bins=None*, *range=None*, *density=False*, *weights=None*, *cumulative=False*, *bottom=None*, *histtype='bar'*, *align='mid'*, *orientation='vertical'*, *rwidth=None*, *log=False*, *color=None*, *label=None*, *stacked=False*, ***, *data=None*, ***kwargs*)

Compute and plot a histogram.

This method uses `numpy.histogram` to bin the data in *x* and count the number of values in each bin, then draws the distribution either as a `BarContainer` or `Polygon`. The *bins*, *range*, *density*, and *weights* parameters are forwarded to `numpy.histogram`.

If the data has already been binned and counted, use `bar` or `stairs` to plot the distribution:

```
counts, bins = np.histogram(x)
plt.stairs(counts, bins)
```

Alternatively, plot pre-computed bins and counts using `hist()` by treating each bin as a single point with a weight equal to its count:

```
plt.hist(bins[:-1], bins, weights=counts)
```

The data input *x* can be a singular array, a list of datasets of potentially different lengths ([*x0*, *x1*, ...]), or a 2D ndarray in which each column is a dataset. Note that the ndarray form is transposed relative to the list form. If the input is an array, then the return value is a tuple (*n*, *bins*, *patches*); if the input is a sequence of arrays, then the return value is a tuple ([*n0*, *n1*, ...], *bins*, [*patches0*, *patches1*, ...]).

Masked arrays are not supported.

**Parameters**

**x**

[(n,) array or sequence of (n,) arrays] Input values, this takes either a single array or a sequence of arrays which are not required to be of the same length.

**bins**

[int or sequence or str, default: `rcParams["hist.bins"]` (default: `10`)] If *bins* is an integer, it defines the number of equal-width bins in the range.

If *bins* is a sequence, it defines the bin edges, including the left edge of the first bin and the right edge of the last bin; in this case, bins may be unequally spaced. All but the last (righthand-most) bin is half-open. In other words, if *bins* is:

```
[1, 2, 3, 4]
```

then the first bin is `[1, 2)` (including 1, but excluding 2) and the second `[2, 3)`. The last bin, however, is `[3, 4]`, which *includes* 4.

If *bins* is a string, it is one of the binning strategies supported by `numpy.histogram_bin_edges`: 'auto', 'fd', 'doane', 'scott', 'stone', 'rice', 'sturges', or 'sqrt'.

**range**

[tuple or None, default: None] The lower and upper range of the bins. Lower and upper outliers are ignored. If not provided, *range* is `(x.min(), x.max())`. Range has no effect if *bins* is a sequence.

If *bins* is a sequence or *range* is specified, autoscaling is based on the specified bin range instead of the range of x.

**density**

[bool, default: False] If `True`, draw and return a probability density: each bin will display the bin's raw count divided by the total number of counts *and the bin width* (`density = counts / (sum(counts) * np.diff(bins))`), so that the area under the histogram integrates to 1 (`np.sum(density * np.diff(bins)) == 1`).

If *stacked* is also `True`, the sum of the histograms is normalized to 1.

**weights**

[(n,) array-like or None, default: None] An array of weights, of the same shape as *x*. Each value in *x* only contributes its associated weight towards the bin count (instead of 1). If *density* is `True`, the weights are normalized, so that the integral of the density over the range remains 1.

**cumulative**

[bool or -1, default: False] If `True`, then a histogram is computed where each bin gives the counts in that bin plus all bins for smaller values. The last bin gives the total number of datapoints.

If *density* is also `True` then the histogram is normalized such that the last bin equals 1.

If *cumulative* is a number less than 0 (e.g., -1), the direction of accumulation is reversed. In this case, if *density* is also `True`, then the histogram is normalized such that the first bin equals 1.

**bottom**

[array-like, scalar, or None, default: None] Location of the bottom of each bin, i.e. bins are drawn from `bottom` to `bottom + hist(x, bins)` If a scalar, the bottom of each bin is shifted by the same amount. If an array, each bin is shifted independently and the length of bottom must match the number of bins. If None, defaults to 0.

**histtype**

[{'bar', 'barstacked', 'step', 'stepfilled'}, default: 'bar'] The type of histogram to draw.

- 'bar' is a traditional bar-type histogram. If multiple data are given the bars are arranged side by side.
- 'barstacked' is a bar-type histogram where multiple data are stacked on top of each other.
- 'step' generates a lineplot that is by default unfilled.
- 'stepfilled' generates a lineplot that is by default filled.

**align**

[{'left', 'mid', 'right'}, default: 'mid'] The horizontal alignment of the histogram bars.

- 'left': bars are centered on the left bin edges.
- 'mid': bars are centered between the bin edges.
- 'right': bars are centered on the right bin edges.

**orientation**

[{'vertical', 'horizontal'}, default: 'vertical'] If 'horizontal', *barh* will be used for bar-type histograms and the *bottom* kwarg will be the left edges.

**rwidth**

[float or None, default: None] The relative width of the bars as a fraction of the bin width. If `None`, automatically compute the width.

Ignored if *histtype* is 'step' or 'stepfilled'.

**log**

[bool, default: False] If `True`, the histogram axis will be set to a log scale.

**color**

[*color* or list of *color* or None, default: None] Color or sequence of colors, one per dataset. Default (`None`) uses the standard line color sequence.

**label**

[str or list of str, optional] String, or sequence of strings to match multiple datasets. Bar charts yield multiple patches per dataset, but only the first gets the label, so that *legend* will work as expected.

**stacked**

[bool, default: False] If `True`, multiple data are stacked on top of each other If `False` multiple data are arranged side by side if histtype is 'bar' or on top of each other if histtype is 'step'

**Returns**

**n**

[array or list of arrays] The values of the histogram bins. See *density* and *weights* for a description of the possible semantics. If input *x* is an array, then this is an array of length *nbins*. If input is a sequence of arrays `[data1, data2, ...]`, then this is a list of arrays with the values of the histograms for each of the arrays in the same order. The dtype of the array *n* (or of its element arrays) will always be float even if no weighting or normalization is used.

**bins**

[array] The edges of the bins. Length nbins + 1 (nbins left edges and right edge of last bin). Always a single array even when multiple data sets are passed in.

**patches**

[*BarContainer* or list of a single *Polygon* or list of such objects] Container of individual artists used to create the histogram or list of such containers if there are multiple input datasets.

**Other Parameters**

**data**

[indexable object, optional] If given, the following parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception):

*x*, *weights*

****kwargs**

*Patch* properties

**See also:**

***hist2d***

2D histogram with rectangular bins

***hexbin***

2D histogram with hexagonal bins

***stairs***

Plot a pre-computed histogram

***bar***

Plot a pre-computed histogram

### Notes

**Note:** This is the *pyplot wrapper* for `axes.Axes.hist`.

For large numbers of bins (>1000), plotting can be significantly accelerated by using `stairs` to plot a pre-computed histogram (`plt.stairs(*np.histogram(data))`), or by setting *histtype* to 'step' or 'stepfilled' rather than 'bar' or 'barstacked'.

### Examples using `matplotlib.pyplot.hist`

- *Histograms*
- *Plotting cumulative distributions*
- *Demo of the histogram function's different histtype settings*
- *The histogram (hist) function with multiple data sets*
- *Histogram bins, density, and weight*
- *Text and mathtext using pyplot*
- *Animated histogram*
- *SVG Histogram*
- *Pyplot tutorial*
- *Image tutorial*

## matplotlib.pyplot.hist2d

`matplotlib.pyplot.`**`hist2d`**(*x*, *y*, *bins=10*, *range=None*, *density=False*, *weights=None*, *cmin=None*, *cmax=None*, *, *data=None*, **kwargs*)

Make a 2D histogram plot.

**Parameters**

**x, y**

[array-like, shape (n, )] Input values

**bins**

[None or int or [int, int] or array-like or [array, array]] The bin specification:

- If int, the number of bins for the two dimensions (`nx = ny = bins`).

- If `[int, int]`, the number of bins in each dimension (`nx, ny = bins`).
- If array-like, the bin edges for the two dimensions (`x_edges = y_edges = bins`).
- If `[array, array]`, the bin edges in each dimension (`x_edges, y_edges = bins`).

The default value is 10.

**range**

[array-like shape(2, 2), optional] The leftmost and rightmost edges of the bins along each dimension (if not specified explicitly in the bins parameters): `[[xmin, xmax], [ymin, ymax]]`. All values outside of this range will be considered outliers and not tallied in the histogram.

**density**

[bool, default: False] Normalize histogram. See the documentation for the *density* parameter of *hist* for more details.

**weights**

[array-like, shape (n, ), optional] An array of values w_i weighing each sample (x_i, y_i).

**cmin, cmax**

[float, default: None] All bins that has count less than *cmin* or more than *cmax* will not be displayed (set to NaN before passing to *pcolormesh*) and these count values in the return value count histogram will also be set to nan upon return.

**Returns**

**h**

[2D array] The bi-dimensional histogram of samples x and y. Values in x are histogrammed along the first dimension and values in y are histogrammed along the second dimension.

**xedges**

[1D array] The bin edges along the x-axis.

**yedges**

[1D array] The bin edges along the y-axis.

**image**

[*QuadMesh*]

**Other Parameters**

**cmap**

[str or *Colormap*, default: *rcParams["image.cmap"]* (default: `'viridis'`)] The Colormap instance or registered colormap name used to map scalar data to colors.

**norm**

[str or *Normalize*, optional] The normalization method used to scale scalar data to the [0, 1] range before mapping to colors using *cmap*. By default, a linear scaling is used, mapping the lowest value to 0 and the highest to 1.

If given, this can be one of the following:

- An instance of *Normalize* or one of its subclasses (see *Colormap normalization*).
- A scale name, i.e. one of "linear", "log", "symlog", "logit", etc. For a list of available scales, call *matplotlib.scale.get_scale_names()*. In that case, a suitable *Normalize* subclass is dynamically generated and instantiated.

**vmin, vmax**

[float, optional] When using scalar data and no explicit *norm*, *vmin* and *vmax* define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data. It is an error to use *vmin*/*vmax* when a *norm* instance is given (but using a `str` *norm* name together with *vmin*/*vmax* is acceptable).

**alpha**

[`0 <= scalar <= 1` or `None`, optional] The alpha blending value.

**data**

[indexable object, optional] If given, the following parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception):

*x*, *y*, *weights*

****kwargs**

Additional parameters are passed along to the *pcolormesh* method and *QuadMesh* constructor.

**See also:**

***hist***

1D histogram plotting

***hexbin***

2D histogram with hexagonal bins

### Notes

**Note:** This is the *pyplot wrapper* for `axes.Axes.hist2d`.

- Currently `hist2d` calculates its own axis limits, and any limits previously set are ignored.
- Rendering the histogram with a logarithmic color scale is accomplished by passing a `colors.LogNorm` instance to the *norm* keyword argument. Likewise, power-law normalization (similar in effect to gamma correction) can be accomplished with `colors.PowerNorm`.

## Examples using `matplotlib.pyplot.hist2d`

- *Histograms*
- *Exploring normalizations*

## matplotlib.pyplot.stairs

matplotlib.pyplot.**stairs**(*values*, *edges=None*, ***, *orientation='vertical'*, *baseline=0*, *fill=False*, *data=None*, ***kwargs*)

Draw a stepwise constant function as a line or a filled plot.

*edges* define the x-axis positions of the steps. *values* the function values between these steps. Depending on *fill*, the function is drawn either as a continuous line with vertical segments at the edges, or as a filled area.

**Parameters**

**values**

[array-like] The step heights.

**edges**

[array-like] The step positions, with `len(edges) == len(vals) + 1`, between which the curve takes on vals values.

**orientation**

[{'vertical', 'horizontal'}, default: 'vertical'] The direction of the steps. Vertical means that *values* are along the y-axis, and edges are along the x-axis.

**baseline**

[float, array-like or None, default: 0] The bottom value of the bounding edges or when `fill=True`, position of lower edge. If *fill* is True or an array is passed to *baseline*, a closed path is drawn.

**fill**

[bool, default: False] Whether the area under the step curve should be filled.

**Returns**

**StepPatch**

[*StepPatch*]

**Other Parameters**

**data**

[indexable object, optional] If given, all parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception).

****kwargs**

*StepPatch* properties

## Notes

**Note:** This is the *pyplot wrapper* for *axes.Axes.stairs*.

## Examples using `matplotlib.pyplot.stairs`

- *Stairs Demo*

## Contours

| | |
|---|---|
| *clabel* | Label a contour plot. |
| *contour* | Plot contour lines. |
| *contourf* | Plot filled contours. |

### matplotlib.pyplot.clabel

matplotlib.pyplot.**clabel**(*CS*, *levels=None*, ***kwargs*)

Label a contour plot.

Adds labels to line contours in given *ContourSet*.

**Parameters**

**CS**

[*ContourSet* instance] Line contours to label.

**levels**

[array-like, optional] A list of level values, that should be labeled. The list must be a subset of `CS.levels`. If not given, all levels are labeled.

****kwargs**

All other parameters are documented in *clabel*.

#### Notes

**Note:** This is the *pyplot wrapper* for *axes.Axes.clabel*.

### Examples using `matplotlib.pyplot.clabel`

- *Contour Demo*
- *Contour Label Demo*
- *Contourf demo*
- *Interactive functions*

### matplotlib.pyplot.contour

matplotlib.pyplot.**contour**(**args*, *data=None*, ***kwargs*)

Plot contour lines.

Call signature:

```
contour([X, Y,] Z, [levels], **kwargs)
```

*contour* and *contourf* draw contour lines and filled contours, respectively. Except as noted, function signatures and return values are the same for both versions.

**Parameters**

**X, Y**

[array-like, optional] The coordinates of the values in *Z*.

*X* and *Y* must both be 2D with the same shape as *Z* (e.g. created via `numpy.meshgrid`), or they must both be 1-D such that `len(X) == N` is the number of columns in *Z* and `len(Y) == M` is the number of rows in *Z*.

*X* and *Y* must both be ordered monotonically.

If not given, they are assumed to be integer indices, i.e. `X = range(N), Y = range(M)`.

**Z**

[(M, N) array-like] The height values over which the contour is drawn. Color-mapping is controlled by *cmap*, *norm*, *vmin*, and *vmax*.

**levels**

[int or array-like, optional] Determines the number and positions of the contour lines / regions.

If an int *n*, use *MaxNLocator*, which tries to automatically choose no more than *n+1* "nice" contour levels between minimum and maximum numeric values of *Z*.

If array-like, draw contour lines at the specified levels. The values must be in increasing order.

**Returns**

*QuadContourSet*

**Other Parameters**

**corner_mask**

[bool, default: *rcParams["contour.corner_mask"]* (default: `True`)] Enable/disable corner masking, which only has an effect if *Z* is a masked array. If `False`, any quad touching a masked point is masked out. If `True`, only the triangular corners of quads nearest those points are always masked out, other triangular corners comprising three unmasked points are contoured as usual.

**colors**

[*color* or list of *color*, optional] The colors of the levels, i.e. the lines for *contour* and the areas for *contourf*.

The sequence is cycled for the levels in ascending order. If the sequence is shorter than the number of levels, it's repeated.

As a shortcut, single color strings may be used in place of one-element lists, i.e. `'red'` instead of `['red']` to color all levels with the same color. This shortcut does only work for color strings, not for other ways of specifying colors.

By default (value *None*), the colormap specified by *cmap* will be used.

**alpha**

[float, default: 1] The alpha blending value, between 0 (transparent) and 1 (opaque).

**cmap**

[str or *Colormap*, default: *rcParams["image.cmap"]* (default: `'viridis'`)] The Colormap instance or registered colormap name used to map scalar data to colors.

This parameter is ignored if *colors* is set.

**norm**

[str or *Normalize*, optional] The normalization method used to scale scalar data to the [0, 1] range before mapping to colors using *cmap*. By default, a linear scaling is used, mapping the lowest value to 0 and the highest to 1.

If given, this can be one of the following:

- An instance of *Normalize* or one of its subclasses (see *Colormap normalization*).
- A scale name, i.e. one of "linear", "log", "symlog", "logit", etc. For a list of available scales, call *matplotlib.scale.get_scale_names()*. In that case, a suitable *Normalize* subclass is dynamically generated and instantiated.

This parameter is ignored if *colors* is set.

**vmin, vmax**

[float, optional] When using scalar data and no explicit *norm*, *vmin* and *vmax* define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data. It is an error to use *vmin*/*vmax* when a *norm* instance is given (but using a `str` *norm* name together with *vmin*/*vmax* is acceptable).

If *vmin* or *vmax* are not given, the default color scaling is based on *levels*.

This parameter is ignored if *colors* is set.

**origin**

[{*None*, 'upper', 'lower', 'image'}, default: None] Determines the orientation and exact position of *Z* by specifying the position of `Z[0, 0]`. This is only relevant, if *X*, *Y* are not given.

- *None*: `Z[0, 0]` is at X=0, Y=0 in the lower left corner.
- 'lower': `Z[0, 0]` is at X=0.5, Y=0.5 in the lower left corner.
- 'upper': `Z[0, 0]` is at X=N+0.5, Y=0.5 in the upper left corner.
- 'image': Use the value from *rcParams["image.origin"]* (default: `'upper'`).

**extent**

[(x0, x1, y0, y1), optional] If *origin* is not *None*, then *extent* is interpreted as in *imshow*: it gives the outer pixel boundaries. In this case, the position of Z[0, 0] is the center of the pixel, not a corner. If *origin* is *None*, then (*x0*, *y0*) is the position of Z[0, 0], and (*x1*, *y1*) is the position of Z[-1, -1].

This argument is ignored if *X* and *Y* are specified in the call to contour.

**locator**

[ticker.Locator subclass, optional] The locator is used to determine the contour levels if they are not given explicitly via *levels*. Defaults to *MaxNLocator*.

**extend**

[{'neither', 'both', 'min', 'max'}, default: 'neither'] Determines the `contourf`-coloring of values that are outside the *levels* range.

If 'neither', values outside the *levels* range are not colored. If 'min', 'max' or 'both', color the values below, above or below and above the *levels* range.

Values below `min(levels)` and above `max(levels)` are mapped to the under/over values of the *Colormap*. Note that most colormaps do not have dedicated colors for these by default, so that the over and under values are the edge values of the colormap. You may want to set these values explicitly using *Colormap.set_under* and *Colormap.set_over*.

---

**Note:** An existing *QuadContourSet* does not get notified if properties of its colormap are changed. Therefore, an explicit call `QuadContourSet.changed()` is needed after modifying the colormap. The explicit call can be left out, if a colorbar is assigned to the *QuadContourSet* because it internally calls `QuadContourSet.changed()`.

---

Example:

```
x = np.arange(1, 10)
y = x.reshape(-1, 1)
h = x * y

cs = plt.contourf(h, levels=[10, 30, 50],
    colors=['#808080', '#A0A0A0', '#C0C0C0'], extend='both
 ↪')
cs.cmap.set_over('red')
cs.cmap.set_under('blue')
cs.changed()
```

**xunits, yunits**

[registered units, optional] Override axis units by specifying an instance of a *matplotlib.units.ConversionInterface*.

**antialiased**

[bool, optional] Enable antialiasing, overriding the defaults. For filled contours, the default is *False*. For line contours, it is taken from *rcParams["lines.antialiased"]* (default: True).

**nchunk**

[int >= 0, optional] If 0, no subdivision of the domain. Specify a positive integer to divide the domain into subdomains of *nchunk* by *nchunk* quads. Chunking reduces the maximum length of polygons generated by the contouring algorithm which reduces the rendering workload passed on to the backend and also requires slightly less RAM. It can however introduce rendering artifacts at chunk boundaries depending on the backend, the *antialiased* flag and value of *alpha*.

**linewidths**

[float or array-like, default: *rcParams["contour.linewidth"]* (default: None)] *Only applies to* *contour*.

The line width of the contour lines.

If a number, all levels will be plotted with this linewidth.

If a sequence, the levels in ascending order will be plotted with the linewidths in the order specified.

If None, this falls back to *rcParams["lines.linewidth"]* (default: 1.5).

**linestyles**

[{*None*, 'solid', 'dashed', 'dashdot', 'dotted'}, optional] *Only applies to* *contour*.

If *linestyles* is *None*, the default is 'solid' unless the lines are monochrome. In that case, negative contours will instead take their linestyle from the *negative_linestyles* argument.

*linestyles* can also be an iterable of the above strings specifying a set of linestyles to be used. If this iterable is shorter than the number of contour levels it will be repeated as necessary.

**negative_linestyles**

[{*None*, 'solid', 'dashed', 'dashdot', 'dotted'}, optional] *Only applies to* *contour*.

If *linestyles* is *None* and the lines are monochrome, this argument specifies the line style for negative contours.

If *negative_linestyles* is *None*, the default is taken from *rcParams["contour.negative_linestyles"]*.

*negative_linestyles* can also be an iterable of the above strings specifying a set of linestyles to be used. If this iterable is shorter than the number of contour levels it will be repeated as necessary.

**hatches**

[list[str], optional] *Only applies to* *contourf*.

A list of cross hatch patterns to use on the filled areas. If None, no hatching will be added to the contour.

**algorithm**

[{'mpl2005', 'mpl2014', 'serial', 'threaded'}, optional] Which contouring algorithm to use to calculate the contour lines and polygons. The algorithms are implemented in ContourPy, consult the ContourPy documentation for further information.

The default is taken from *rcParams["contour.algorithm"]* (default: `'mpl2014'`).

**clip_path**

[*Patch* or *Path* or *TransformedPath*] Set the clip path. See *set_clip_path*.

New in version 3.8.

**data**

[indexable object, optional] If given, all parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception).

## Notes

**Note:** This is the *pyplot wrapper* for *axes.Axes.contour*.

1. *contourf* differs from the MATLAB version in that it does not draw the polygon edges. To draw edges, add line contours with calls to *contour*.
2. *contourf* fills intervals that are closed at the top; that is, for boundaries *z1* and *z2*, the filled region is:

```
z1 < Z <= z2
```

   except for the lowest interval, which is closed on both sides (i.e. it includes the lowest value).

3. *contour* and *contourf* use a marching squares algorithm to compute contour locations. More information can be found in ContourPy documentation.

### Examples using `matplotlib.pyplot.contour`

- *Contour Corner Mask*
- *Contour Demo*
- *Contour Image*
- *Contour Label Demo*
- *Contourf demo*
- *Contourf Hatching*
- *Blend transparency with color in 2D images*
- *Contour plot of irregularly spaced data*
- *Interactive functions*

## matplotlib.pyplot.contourf

matplotlib.pyplot.**contourf**(**args*, *data=None*, ***kwargs*)

Plot filled contours.

Call signature:

```
contourf([X, Y,] Z, [levels], **kwargs)
```

*contour* and *contourf* draw contour lines and filled contours, respectively. Except as noted, function signatures and return values are the same for both versions.

**Parameters**

**X, Y**

[array-like, optional] The coordinates of the values in *Z*.

*X* and *Y* must both be 2D with the same shape as *Z* (e.g. created via `numpy.meshgrid`), or they must both be 1-D such that `len(X) == N` is the number of columns in *Z* and `len(Y) == M` is the number of rows in *Z*.

*X* and *Y* must both be ordered monotonically.

If not given, they are assumed to be integer indices, i.e. `X = range(N), Y = range(M)`.

**Z**

[(M, N) array-like] The height values over which the contour is drawn. Color-mapping is controlled by *cmap*, *norm*, *vmin*, and *vmax*.

**levels**

[int or array-like, optional] Determines the number and positions of the contour lines / regions.

If an int *n*, use *MaxNLocator*, which tries to automatically choose no more than *n+1* "nice" contour levels between minimum and maximum numeric values of *Z*.

If array-like, draw contour lines at the specified levels. The values must be in increasing order.

**Returns**

*QuadContourSet*

**Other Parameters**

**corner_mask**

[bool, default: *rcParams["contour.corner_mask"]* (default: `True`)] Enable/disable corner masking, which only has an effect if *Z* is a masked array. If `False`, any quad touching a masked point is masked out. If `True`, only the triangular corners of quads nearest those points are always masked out, other triangular corners comprising three unmasked points are contoured as usual.

**colors**

[*color* or list of *color*, optional] The colors of the levels, i.e. the lines for *contour* and the areas for *contourf*.

The sequence is cycled for the levels in ascending order. If the sequence is shorter than the number of levels, it's repeated.

As a shortcut, single color strings may be used in place of one-element lists, i.e. `'red'` instead of `['red']` to color all levels with the same color. This shortcut does only work for color strings, not for other ways of specifying colors.

By default (value *None*), the colormap specified by *cmap* will be used.

**alpha**

[float, default: 1] The alpha blending value, between 0 (transparent) and 1 (opaque).

**cmap**

[str or *Colormap*, default: *rcParams["image.cmap"]* (default: `'viridis'`)] The Colormap instance or registered colormap name used to map scalar data to colors.

This parameter is ignored if *colors* is set.

**norm**

[str or *Normalize*, optional] The normalization method used to scale scalar data to the [0, 1] range before mapping to colors using *cmap*. By default, a linear scaling is used, mapping the lowest value to 0 and the highest to 1.

If given, this can be one of the following:

- An instance of *Normalize* or one of its subclasses (see *Colormap normalization*).
- A scale name, i.e. one of "linear", "log", "symlog", "logit", etc. For a list of available scales, call *matplotlib.scale.get_scale_names()*. In that case, a suitable *Normalize* subclass is dynamically generated and instantiated.

This parameter is ignored if *colors* is set.

**vmin, vmax**

[float, optional] When using scalar data and no explicit *norm*, *vmin* and *vmax* define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data. It is an error to use *vmin*/*vmax* when a *norm* instance is given (but using a `str` *norm* name together with *vmin*/*vmax* is acceptable).

If *vmin* or *vmax* are not given, the default color scaling is based on *levels*.

This parameter is ignored if *colors* is set.

**origin**

[{*None*, 'upper', 'lower', 'image'}, default: None] Determines the orientation and exact position of *Z* by specifying the position of `Z[0, 0]`. This is only relevant, if *X*, *Y* are not given.

- *None*: `Z[0, 0]` is at X=0, Y=0 in the lower left corner.
- 'lower': `Z[0, 0]` is at X=0.5, Y=0.5 in the lower left corner.
- 'upper': `Z[0, 0]` is at X=N+0.5, Y=0.5 in the upper left corner.
- 'image': Use the value from *rcParams["image.origin"]* (default: `'upper'`).

**extent**

[(x0, x1, y0, y1), optional] If *origin* is not *None*, then *extent* is interpreted as in *imshow*: it gives the outer pixel boundaries. In this case, the position of Z[0, 0] is the center of the pixel, not a corner. If *origin* is *None*, then (*x0*, *y0*) is the position of Z[0, 0], and (*x1*, *y1*) is the position of Z[-1, -1].

This argument is ignored if *X* and *Y* are specified in the call to contour.

**locator**

[ticker.Locator subclass, optional] The locator is used to determine the contour levels if they are not given explicitly via *levels*. Defaults to *MaxNLocator*.

**extend**

[{'neither', 'both', 'min', 'max'}, default: 'neither'] Determines the `contourf`-coloring of values that are outside the *levels* range.

If 'neither', values outside the *levels* range are not colored. If 'min', 'max' or 'both', color the values below, above or below and above the *levels* range.

Values below `min(levels)` and above `max(levels)` are mapped to the under/over values of the *Colormap*. Note that most colormaps do not have dedicated colors for these by default, so that the over and under values are the edge values of the colormap. You may want to set these values explicitly using *Colormap.set_under* and *Colormap.set_over*.

---

**Note:** An existing *QuadContourSet* does not get notified if properties of its colormap are changed. Therefore, an explicit call `QuadContourSet.changed()` is needed after modifying the colormap. The explicit call can be left out, if a colorbar is assigned to the *QuadContourSet* because it internally calls `QuadContourSet.changed()`.

---

Example:

```
x = np.arange(1, 10)
y = x.reshape(-1, 1)
h = x * y

cs = plt.contourf(h, levels=[10, 30, 50],
    colors=['#808080', '#A0A0A0', '#C0C0C0'], extend='both
 ↪')
cs.cmap.set_over('red')
cs.cmap.set_under('blue')
cs.changed()
```

**xunits, yunits**

[registered units, optional] Override axis units by specifying an instance of a *matplotlib.units.ConversionInterface*.

**antialiased**

[bool, optional] Enable antialiasing, overriding the defaults. For filled contours, the default is *False*. For line contours, it is taken from *rcParams["lines.antialiased"]* (default: `True`).

**nchunk**

[int >= 0, optional] If 0, no subdivision of the domain. Specify a positive integer to divide the domain into subdomains of *nchunk* by *nchunk* quads. Chunking reduces the maximum length of polygons generated by the contouring algorithm which reduces the rendering workload passed on to the backend and also requires slightly less RAM. It can however introduce rendering artifacts at chunk boundaries depending on the backend, the *antialiased* flag and value of *alpha*.

**linewidths**

[float or array-like, default: *rcParams["contour.linewidth"]* (default: `None`)] *Only applies to* *contour*.

The line width of the contour lines.

If a number, all levels will be plotted with this linewidth.

If a sequence, the levels in ascending order will be plotted with the linewidths in the order specified.

If None, this falls back to *rcParams["lines.linewidth"]* (default: 1.5).

**linestyles**

[{*None*, 'solid', 'dashed', 'dashdot', 'dotted'}, optional] *Only applies to* *contour*.

If *linestyles* is *None*, the default is 'solid' unless the lines are monochrome. In that case, negative contours will instead take their linestyle from the *negative_linestyles* argument.

*linestyles* can also be an iterable of the above strings specifying a set of linestyles to be used. If this iterable is shorter than the number of contour levels it will be repeated as necessary.

**negative_linestyles**

[{*None*, 'solid', 'dashed', 'dashdot', 'dotted'}, optional] *Only applies to* *contour*.

If *linestyles* is *None* and the lines are monochrome, this argument specifies the line style for negative contours.

If *negative_linestyles* is *None*, the default is taken from *rcParams["contour.negative_linestyles"]*.

*negative_linestyles* can also be an iterable of the above strings specifying a set of linestyles to be used. If this iterable is shorter than the number of contour levels it will be repeated as necessary.

**hatches**

[list[str], optional] *Only applies to* *contourf*.

A list of cross hatch patterns to use on the filled areas. If None, no hatching will be added to the contour.

**algorithm**

[{'mpl2005', 'mpl2014', 'serial', 'threaded'}, optional] Which contouring algorithm to use to calculate the contour lines and polygons. The algorithms are implemented in ContourPy, consult the ContourPy documentation for further information.

The default is taken from *rcParams["contour.algorithm"]* (default: 'mpl2014').

**clip_path**

[*Patch* or *Path* or *TransformedPath*] Set the clip path. See *set_clip_path*.

New in version 3.8.

**data**

[indexable object, optional] If given, all parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception).

## Notes

**Note:** This is the *pyplot wrapper* for `axes.Axes.contourf`.

1. `contourf` differs from the MATLAB version in that it does not draw the polygon edges. To draw edges, add line contours with calls to `contour`.
2. `contourf` fills intervals that are closed at the top; that is, for boundaries *z1* and *z2*, the filled region is:

```
z1 < Z <= z2
```

   except for the lowest interval, which is closed on both sides (i.e. it includes the lowest value).

3. `contour` and `contourf` use a marching squares algorithm to compute contour locations. More information can be found in ContourPy documentation.

## Examples using `matplotlib.pyplot.contourf`

- *Contour Corner Mask*
- *Contourf demo*
- *Contourf Hatching*
- *Contourf and log color scale*
- *Contour plot of irregularly spaced data*
- *pcolormesh*
- *Triinterp Demo*

### 2D arrays

| | |
|---|---|
| *imshow* | Display data as an image, i.e., on a 2D regular raster. |
| *matshow* | Display a 2D array as a matrix in a new figure window. |
| *pcolor* | Create a pseudocolor plot with a non-regular rectangular grid. |
| *pcolormesh* | Create a pseudocolor plot with a non-regular rectangular grid. |
| *spy* | Plot the sparsity pattern of a 2D array. |
| *figimage* | Add a non-resampled image to the figure. |

### matplotlib.pyplot.imshow

matplotlib.pyplot.**imshow**(*X*, *cmap=None*, *norm=None*, ***, *aspect=None*, *interpolation=None*, *alpha=None*, *vmin=None*, *vmax=None*, *origin=None*, *extent=None*, *interpolation_stage=None*, *filternorm=True*, *filterrad=4.0*, *resample=None*, *url=None*, *data=None*, ***kwargs*)

Display data as an image, i.e., on a 2D regular raster.

The input may either be actual RGB(A) data, or 2D scalar data, which will be rendered as a pseudocolor image. For displaying a grayscale image, set up the colormapping using the parameters `cmap='gray', vmin=0, vmax=255`.

The number of pixels used to render an image is set by the Axes size and the figure *dpi*. This can lead to aliasing artifacts when the image is resampled, because the displayed image size will usually not match the size of *X* (see *Image antialiasing*). The resampling can be controlled via the *interpolation* parameter and/or *rcParams["image.interpolation"]* (default: `'antialiased'`).

**Parameters**

**X**

[array-like or PIL image] The image data. Supported array shapes are:

- (M, N): an image with scalar data. The values are mapped to colors using normalization and a colormap. See parameters *norm*, *cmap*, *vmin*, *vmax*.
- (M, N, 3): an image with RGB values (0-1 float or 0-255 int).
- (M, N, 4): an image with RGBA values (0-1 float or 0-255 int), i.e. including transparency.

The first two dimensions (M, N) define the rows and columns of the image.

Out-of-range RGB(A) values are clipped.

**cmap**

[str or *Colormap*, default: *rcParams["image.cmap"]* (default: `'viridis'`)] The Colormap instance or registered colormap name used to map scalar data to colors.

This parameter is ignored if *X* is RGB(A).

**norm**

[str or *Normalize*, optional] The normalization method used to scale scalar data to the [0, 1] range before mapping to colors using *cmap*. By default, a linear scaling is used, mapping the lowest value to 0 and the highest to 1.

If given, this can be one of the following:

- An instance of *Normalize* or one of its subclasses (see *Colormap normalization*).
- A scale name, i.e. one of "linear", "log", "symlog", "logit", etc. For a list of available scales, call *matplotlib.scale.get_scale_names()*. In that case, a suitable *Normalize* subclass is dynamically generated and instantiated.

This parameter is ignored if *X* is RGB(A).

**vmin, vmax**

[float, optional] When using scalar data and no explicit *norm*, *vmin* and *vmax* define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data. It is an error to use *vmin*/*vmax* when a *norm* instance is given (but using a `str` *norm* name together with *vmin*/*vmax* is acceptable).

This parameter is ignored if *X* is RGB(A).

**aspect**

[{'equal', 'auto'} or float or None, default: None] The aspect ratio of the Axes. This parameter is particularly relevant for images since it determines whether data pixels are square.

This parameter is a shortcut for explicitly calling *Axes.set_aspect*. See there for further details.

- 'equal': Ensures an aspect ratio of 1. Pixels will be square (unless pixel sizes are explicitly made non-square in data coordinates using *extent*).
- 'auto': The Axes is kept fixed and the aspect is adjusted so that the data fit in the Axes. In general, this will result in non-square pixels.

Normally, None (the default) means to use *rcParams["image.aspect"]* (default: `'equal'`). However, if the image uses a transform that does not contain the axes data transform, then None means to not modify the axes aspect at all (in that case, directly call *Axes.set_aspect* if desired).

**interpolation**

[str, default: *rcParams["image.interpolation"]* (default: 'antialiased')] The interpolation method used.

Supported values are 'none', 'antialiased', 'nearest', 'bilinear', 'bicubic', 'spline16', 'spline36', 'hanning', 'hamming', 'hermite', 'kaiser', 'quadric', 'catrom', 'gaussian', 'bessel', 'mitchell', 'sinc', 'lanczos', 'blackman'.

The data *X* is resampled to the pixel size of the image on the figure canvas, using the interpolation method to either up- or downsample the data.

If *interpolation* is 'none', then for the ps, pdf, and svg backends no down- or upsampling occurs, and the image data is passed to the backend as a native image. Note that different ps, pdf, and svg viewers may display these raw pixels differently. On other backends, 'none' is the same as 'nearest'.

If *interpolation* is the default 'antialiased', then 'nearest' interpolation is used if the image is upsampled by more than a factor of three (i.e. the number of display pixels is at least three times the size of the data array). If the upsampling rate is smaller than 3, or the image is downsampled, then 'hanning' interpolation is used to act as an anti-aliasing filter, unless the image happens to be upsampled by exactly a factor of two or one.

See *Interpolations for imshow* for an overview of the supported interpolation methods, and *Image antialiasing* for a discussion of image antialiasing.

Some interpolation methods require an additional radius parameter, which can be set by *filterrad*. Additionally, the antigrain image resize filter is controlled by the parameter *filternorm*.

**interpolation_stage**

[{'data', 'rgba'}, default: 'data'] If 'data', interpolation is carried out on the data provided by the user. If 'rgba', the interpolation is carried out after the colormapping has been applied (visual interpolation).

**alpha**

[float or array-like, optional] The alpha blending value, between 0 (transparent) and 1 (opaque). If *alpha* is an array, the alpha blending values are applied pixel by pixel, and *alpha* must have the same shape as *X*.

**origin**

[{'upper', 'lower'}, default: *rcParams["image.origin"]* (default: 'upper')] Place the [0, 0] index of the array in the upper left or lower left corner of the Axes. The convention (the default) 'upper' is typically used for matrices and images.

Note that the vertical axis points upward for 'lower' but downward for 'upper'.

See the *origin and extent in imshow* tutorial for examples and a more detailed description.

**extent**

[floats (left, right, bottom, top), optional] The bounding box in data coordinates that the image will fill. These values may be unitful and match the units of the Axes. The image is stretched individually along x and y to fill the box.

The default extent is determined by the following conditions. Pixels have unit size in data coordinates. Their centers are on integer coordinates, and their center coordinates range from 0 to columns-1 horizontally and from 0 to rows-1 vertically.

Note that the direction of the vertical axis and thus the default values for top and bottom depend on *origin*:

- For `origin == 'upper'` the default is `(-0.5, numcols-0.5, numrows-0.5, -0.5)`.
- For `origin == 'lower'` the default is `(-0.5, numcols-0.5, -0.5, numrows-0.5)`.

See the *origin and extent in imshow* tutorial for examples and a more detailed description.

**filternorm**

[bool, default: True] A parameter for the antigrain image resize filter (see the antigrain documentation). If *filternorm* is set, the filter normalizes integer values and corrects the rounding errors. It doesn't do anything with the source floating point values, it corrects only integers according to the rule of 1.0 which means that any sum of pixel weights must be equal to 1.0. So, the filter function must produce a graph of the proper shape.

**filterrad**

[float > 0, default: 4.0] The filter radius for filters that have a radius parameter, i.e. when interpolation is one of: 'sinc', 'lanczos' or 'blackman'.

**resample**

[bool, default: *rcParams["image.resample"]* (default: `True`)] When *True*, use a full resampling method. When *False*, only resample when the output image is larger than the input image.

**url**

[str, optional] Set the url of the created *AxesImage*. See *Artist.set_url*.

**Returns**

***AxesImage***

**Other Parameters**

**data**

[indexable object, optional] If given, all parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception).

****kwargs**

[*Artist* properties] These parameters are passed on to the constructor of the *AxesImage* artist.

**See also:**

***matshow***

Plot a matrix or an array as an image.

## Notes

---

**Note:** This is the *pyplot wrapper* for *axes.Axes.imshow*.

---

Unless *extent* is used, pixel centers will be located at integer coordinates. In other words: the origin will coincide with the center of pixel (0, 0).

There are two common representations for RGB images with an alpha channel:

- Straight (unassociated) alpha: R, G, and B channels represent the color of the pixel, disregarding its opacity.
- Premultiplied (associated) alpha: R, G, and B channels represent the color of the pixel, adjusted for its opacity by multiplication.

*imshow* expects RGB images adopting the straight (unassociated) alpha representation.

## Examples using `matplotlib.pyplot.imshow`

- *Affine transform of an image*
- *Barcode*
- *Contour Image*
- *Creating annotated heatmaps*
- *Clipping images with patches*
- *Many ways to plot images*
- *Image Masked*
- *Blend transparency with color in 2D images*
- *Interpolations for imshow*
- *Layer Images*
- *Visualize matrices with matshow*
- *Multiple images*

- *pcolor images*
- *Shading example*
- *Subplots spacings and margins*
- *Colorbar*
- *Creating a colormap from a list of colors*
- *Dolphins*
- *Hyperlinks*
- *Image tutorial*
- *Tight layout guide*

### matplotlib.pyplot.matshow

matplotlib.pyplot.**matshow**(*A*, *fignum=None*, ***kwargs*)

Display a 2D array as a matrix in a new figure window.

The origin is set at the upper left hand corner. The indexing is `(row, column)` so that the first index runs vertically and the second index runs horizontally in the figure:

```
A[0, 0]   … A[0, M-1]
   ⋮              ⋮
A[N-1, 0] … A[N-1, M-1]
```

The aspect ratio of the figure window is that of the array, unless this would make an excessively short or narrow figure.

Tick labels for the xaxis are placed on top.

**Parameters**

**A**

[2D array-like] The matrix to be displayed.

**fignum**

[None or int] If *None*, create a new, appropriately sized figure window.

If 0, use the current Axes (creating one if there is none, without ever adjusting the figure size).

Otherwise, create a new Axes on the figure with the given number (creating it at the appropriate size if it does not exist, but not adjusting the figure size otherwise). Note that this will be drawn on top of any preexisting Axes on the figure.

**Returns**

***AxesImage***

**Other Parameters**

****kwargs**

[*imshow* arguments]

## Examples using `matplotlib.pyplot.matshow`

- *Visualize matrices with matshow*

## matplotlib.pyplot.pcolor

matplotlib.pyplot.**pcolor**(**args*, *shading=None*, *alpha=None*, *norm=None*, *cmap=None*, *vmin=None*, *vmax=None*, *data=None*, ***kwargs*)

Create a pseudocolor plot with a non-regular rectangular grid.

Call signature:

```
pcolor([X, Y,] C, **kwargs)
```

*X* and *Y* can be used to specify the corners of the quadrilaterals.

---

**Hint:** `pcolor()` can be very slow for large arrays. In most cases you should use the similar but much faster *pcolormesh* instead. See *Differences between pcolor() and pcolormesh()* for a discussion of the differences.

---

**Parameters**

**C**

[2D array-like] The color-mapped values. Color-mapping is controlled by *cmap*, *norm*, *vmin*, and *vmax*.

**X, Y**

[array-like, optional] The coordinates of the corners of quadrilaterals of a pcolormesh:

```
(X[i+1, j], Y[i+1, j])        (X[i+1, j+1], Y[i+1, j+1])
                       ●------●
                       |      |
                       ●------●
    (X[i, j], Y[i, j])        (X[i, j+1], Y[i, j+1])
```

Note that the column index corresponds to the x-coordinate, and the row index corresponds to y. For details, see the *Notes* section below.

If `shading='flat'` the dimensions of *X* and *Y* should be one greater than those of *C*, and the quadrilateral is colored due to the value at `C[i, j]`. If *X*, *Y* and *C* have equal dimensions, a warning will be raised and the last row and column of *C* will be ignored.

If `shading='nearest'`, the dimensions of *X* and *Y* should be the same as those of *C* (if not, a ValueError will be raised). The color `C[i, j]` will be centered on `(X[i, j], Y[i, j])`.

If *X* and/or *Y* are 1-D arrays or column vectors they will be expanded as needed into the appropriate 2D arrays, making a rectangular grid.

**shading**

[{'flat', 'nearest', 'auto'}, default: *rcParams["pcolor.shading"]* (default: `'auto'`)] The fill style for the quadrilateral. Possible values:

- 'flat': A solid color is used for each quad. The color of the quad (i, j), (i+1, j), (i, j+1), (i+1, j+1) is given by `C[i, j]`. The dimensions of *X* and *Y* should be one greater than those of *C*; if they are the same as *C*, then a deprecation warning is raised, and the last row and column of *C* are dropped.
- 'nearest': Each grid point will have a color centered on it, extending halfway between the adjacent grid centers. The dimensions of *X* and *Y* must be the same as *C*.
- 'auto': Choose 'flat' if dimensions of *X* and *Y* are one larger than *C*. Choose 'nearest' if dimensions are the same.

See *pcolormesh grids and shading* for more description.

**cmap**

[str or *Colormap*, default: *rcParams["image.cmap"]* (default: `'viridis'`)] The Colormap instance or registered colormap name used to map scalar data to colors.

**norm**

[str or *Normalize*, optional] The normalization method used to scale scalar data to the [0, 1] range before mapping to colors using *cmap*. By default, a linear scaling is used, mapping the lowest value to 0 and the highest to 1.

If given, this can be one of the following:

- An instance of *Normalize* or one of its subclasses (see *Colormap normalization*).
- A scale name, i.e. one of "linear", "log", "symlog", "logit", etc. For a list of available scales, call *matplotlib.scale.get_scale_names()*. In that case, a suitable *Normalize* subclass is dynamically generated and instantiated.

**vmin, vmax**

[float, optional] When using scalar data and no explicit *norm*, *vmin* and *vmax* define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data. It is an error to use *vmin*/*vmax* when a *norm* instance is given (but using a `str` *norm* name together with *vmin*/*vmax* is acceptable).

**edgecolors**

[{'none', None, 'face', color, color sequence}, optional] The color of the edges. Defaults to 'none'. Possible values:

- 'none' or '': No edge.
- *None*: *rcParams["patch.edgecolor"]* (default: `'black'`) will be used. Note that currently *rcParams["patch.force_edgecolor"]* (default: `False`) has to be True for this to work.
- 'face': Use the adjacent face color.
- A color or sequence of colors will set the edge color.

The singular form *edgecolor* works as an alias.

**alpha**

[float, default: None] The alpha blending value of the face color, between 0 (transparent) and 1 (opaque). Note: The edgecolor is currently not affected by this.

**snap**

[bool, default: False] Whether to snap the mesh to pixel boundaries.

**Returns**

***matplotlib.collections.PolyQuadMesh***

**Other Parameters**

**antialiaseds**

[bool, default: False] The default *antialiaseds* is False if the default *edgecolors*="none" is used. This eliminates artificial lines at patch boundaries, and works regardless of the value of alpha. If *edgecolors* is not "none", then the default *antialiaseds* is taken from *rcParams["patch.antialiased"]* (default: `True`). Stroking the edges may be preferred if *alpha* is 1, but will cause artifacts otherwise.

**data**

[indexable object, optional] If given, all parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception).

****kwargs**

Additionally, the following arguments are allowed. They are passed along to the *PolyQuadMesh* constructor:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a d |
| *alpha* | array-like or scalar or None |
| *animated* | bool |
| *antialiased* or aa or antialiaseds | bool or list of bools |
| *array* | array-like or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clim* | (vmin: float, vmax: float) |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *cmap* | *Colormap* or str or None |
| *color* | *color* or list of RGBA tuples |
| *edgecolor* or ec or edgecolors | *color* or list of *color* or 'face' |
| *facecolor* or facecolors or fc | *color* or list of *color* |
| *figure* | *Figure* |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or dashes or linestyles or ls | str or tuple or list thereof |
| *linewidth* or linewidths or lw | float or list of floats |
| *mouseover* | bool |
| *norm* | *Normalize* or str or None |
| *offset_transform* or transOffset | *Transform* |
| *offsets* | (N, 2) or (2,) array-like |
| *path_effects* | list of *AbstractPathEffect* |
| *paths* | list of array-like |
| *picker* | None or bool or float or callable |
| *pickradius* | float |
| *rasterized* | bool |
| sizes | numpy.ndarray or None |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *urls* | list of str or None |
| *verts* | list of array-like |
| *verts_and_codes* | unknown |
| *visible* | bool |
| *zorder* | float |

**See also:**

***pcolormesh***

for an explanation of the differences between pcolor and pcolormesh.

***imshow***

If *X* and *Y* are each equidistant, *imshow* can be a faster alternative.

### Notes

**Note:** This is the *pyplot wrapper* for *axes.Axes.pcolor*.

**Masked arrays**

*X*, *Y* and *C* may be masked arrays. If either `C[i, j]`, or one of the vertices surrounding `C[i, j]` (*X* or *Y* at `[i, j]`, `[i+1, j]`, `[i, j+1]`, `[i+1, j+1]`) is masked, nothing is plotted.

**Grid orientation**

The grid orientation follows the standard matrix convention: An array *C* with shape (nrows, ncolumns) is plotted with the column number as *X* and the row number as *Y*.

## Examples using `matplotlib.pyplot.pcolor`

- *pcolor images*
- *Controlling view limits using margins and sticky_edges*

## matplotlib.pyplot.pcolormesh

matplotlib.pyplot.**pcolormesh**(**args*, *alpha=None*, *norm=None*, *cmap=None*, *vmin=None*, *vmax=None*, *shading=None*, *antialiased=False*, *data=None*, ***kwargs*)

Create a pseudocolor plot with a non-regular rectangular grid.

Call signature:

```
pcolormesh([X, Y,] C, **kwargs)
```

*X* and *Y* can be used to specify the corners of the quadrilaterals.

**Hint:** *pcolormesh* is similar to *pcolor*. It is much faster and preferred in most cases. For a detailed discussion on the differences see *Differences between pcolor() and pcolormesh()*.

**Parameters**

**C**

[array-like] The mesh data. Supported array shapes are:

- (M, N) or M*N: a mesh with scalar data. The values are mapped to colors using normalization and a colormap. See parameters *norm*, *cmap*, *vmin*, *vmax*.
- (M, N, 3): an image with RGB values (0-1 float or 0-255 int).
- (M, N, 4): an image with RGBA values (0-1 float or 0-255 int), i.e. including transparency.

The first two dimensions (M, N) define the rows and columns of the mesh data.

**X, Y**

[array-like, optional] The coordinates of the corners of quadrilaterals of a pcolormesh:

```
(X[i+1, j], Y[i+1, j])        (X[i+1, j+1], Y[i+1, j+1])
                       ●──────●
                       │      │
                       ●──────●
    (X[i, j], Y[i, j])        (X[i, j+1], Y[i, j+1])
```

Note that the column index corresponds to the x-coordinate, and the row index corresponds to y. For details, see the *Notes* section below.

If `shading='flat'` the dimensions of *X* and *Y* should be one greater than those of *C*, and the quadrilateral is colored due to the value at `C[i, j]`. If *X*, *Y* and *C* have equal dimensions, a warning will be raised and the last row and column of *C* will be ignored.

If `shading='nearest'` or `'gouraud'`, the dimensions of *X* and *Y* should be the same as those of *C* (if not, a ValueError will be raised). For `'nearest'` the color `C[i, j]` is centered on `(X[i, j], Y[i, j])`. For `'gouraud'`, a smooth interpolation is carried out between the quadrilateral corners.

If *X* and/or *Y* are 1-D arrays or column vectors they will be expanded as needed into the appropriate 2D arrays, making a rectangular grid.

**cmap**

[str or *Colormap*, default: *rcParams["image.cmap"]* (default: `'viridis'`)] The Colormap instance or registered colormap name used to map scalar data to colors.

**norm**

[str or *Normalize*, optional] The normalization method used to scale scalar data to the [0, 1] range before mapping to colors using *cmap*. By default, a linear scaling is used, mapping the lowest value to 0 and the highest to 1.

If given, this can be one of the following:

- An instance of *Normalize* or one of its subclasses (see *Colormap normalization*).
- A scale name, i.e. one of "linear", "log", "symlog", "logit", etc. For a list of available scales, call *matplotlib.scale.get_scale_names()*. In that case, a suitable *Normalize* subclass is dynamically generated and instantiated.

**vmin, vmax**

[float, optional] When using scalar data and no explicit *norm*, *vmin* and *vmax* define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data. It is an error to use *vmin*/*vmax* when a *norm* instance is given (but using a `str` *norm* name together with *vmin*/*vmax* is acceptable).

**edgecolors**

[{'none', None, 'face', color, color sequence}, optional] The color of the edges. Defaults to 'none'. Possible values:

- 'none' or '': No edge.
- *None*: *rcParams["patch.edgecolor"]* (default: `'black'`) will be used. Note that currently *rcParams["patch.force_edgecolor"]* (default: `False`) has to be True for this to work.
- 'face': Use the adjacent face color.
- A color or sequence of colors will set the edge color.

The singular form *edgecolor* works as an alias.

**alpha**

[float, default: None] The alpha blending value, between 0 (transparent) and 1 (opaque).

**shading**

[{'flat', 'nearest', 'gouraud', 'auto'}, optional] The fill style for the quadrilateral; defaults to *rcParams["pcolor.shading"]* (default: `'auto'`). Possible values:

- 'flat': A solid color is used for each quad. The color of the quad (i, j), (i+1, j), (i, j+1), (i+1, j+1) is given by `C[i, j]`. The dimensions of *X* and *Y* should be one greater than those of *C*; if they are the same as *C*, then a deprecation warning is raised, and the last row and column of *C* are dropped.
- 'nearest': Each grid point will have a color centered on it, extending halfway between the adjacent grid centers. The dimensions of *X* and *Y* must be the same as *C*.
- 'gouraud': Each quad will be Gouraud shaded: The color of the corners (i', j') are given by `C[i', j']`. The color values of the area in between is interpolated

from the corner values. The dimensions of *X* and *Y* must be the same as *C*. When Gouraud shading is used, *edgecolors* is ignored.

- 'auto': Choose 'flat' if dimensions of *X* and *Y* are one larger than *C*. Choose 'nearest' if dimensions are the same.

See *pcolormesh grids and shading* for more description.

**snap**

[bool, default: False] Whether to snap the mesh to pixel boundaries.

**rasterized**

[bool, optional] Rasterize the pcolormesh when drawing vector graphics. This can speed up rendering and produce smaller files for large data sets. See also *Rasterization for vector graphics*.

**Returns**

***matplotlib.collections.QuadMesh***

**Other Parameters**

**data**

[indexable object, optional] If given, all parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception).

****kwargs**

Additionally, the following arguments are allowed. They are passed along to the *QuadMesh* constructor:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a d |
| *alpha* | array-like or scalar or None |
| *animated* | bool |
| *antialiased* or aa or antialiaseds | bool or list of bools |
| `array` | array-like |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clim* | (vmin: float, vmax: float) |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *cmap* | *Colormap* or str or None |
| *color* | *color* or list of RGBA tuples |
| *edgecolor* or ec or edgecolors | *color* or list of *color* or 'face' |
| *facecolor* or facecolors or fc | *color* or list of *color* |
| *figure* | *Figure* |

Table 127 – continued fr

| Property | Description |
|---|---|
| *gid* | str |
| *hatch* | {'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or dashes or linestyles or ls | str or tuple or list thereof |
| *linewidth* or linewidths or lw | float or list of floats |
| *mouseover* | bool |
| *norm* | *Normalize* or str or None |
| *offset_transform* or transOffset | *Transform* |
| *offsets* | (N, 2) or (2,) array-like |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *pickradius* | float |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *urls* | list of str or None |
| *visible* | bool |
| *zorder* | float |

**See also:**

***pcolor***

An alternative implementation with slightly different features. For a detailed discussion on the differences see *Differences between pcolor() and pcolormesh()*.

***imshow***

If *X* and *Y* are each equidistant, *imshow* can be a faster alternative.

## Notes

**Note:** This is the *pyplot wrapper* for *axes.Axes.pcolormesh*.

### Masked arrays

*C* may be a masked array. If `C[i, j]` is masked, the corresponding quadrilateral will be transparent. Masking of *X* and *Y* is not supported. Use *pcolor* if you need this functionality.

### Grid orientation

The grid orientation follows the standard matrix convention: An array *C* with shape (nrows, ncolumns) is plotted with the column number as *X* and the row number as *Y*.

**Differences between pcolor() and pcolormesh()**

Both methods are used to create a pseudocolor plot of a 2D array using quadrilaterals.

The main difference lies in the created object and internal data handling: While `pcolor` returns a `PolyQuadMesh`, `pcolormesh` returns a `QuadMesh`. The latter is more specialized for the given purpose and thus is faster. It should almost always be preferred.

There is also a slight difference in the handling of masked arrays. Both `pcolor` and `pcolormesh` support masked arrays for *C*. However, only `pcolor` supports masked arrays for *X* and *Y*. The reason lies in the internal handling of the masked values. `pcolor` leaves out the respective polygons from the PolyQuadMesh. `pcolormesh` sets the facecolor of the masked elements to transparent. You can see the difference when using edgecolors. While all edges are drawn irrespective of masking in a QuadMesh, the edge between two adjacent masked quadrilaterals in `pcolor` is not drawn as the corresponding polygons do not exist in the PolyQuadMesh. Because PolyQuadMesh draws each individual polygon, it also supports applying hatches and linestyles to the collection.

Another difference is the support of Gouraud shading in `pcolormesh`, which is not available with `pcolor`.

### Examples using `matplotlib.pyplot.pcolormesh`

- *pcolor images*
- *pcolormesh grids and shading*
- *pcolormesh*
- *QuadMesh Demo*
- *Time Series Histogram*
- *Rasterization for vector graphics*

## matplotlib.pyplot.spy

matplotlib.pyplot.**spy**(*Z*, *precision=0*, *marker=None*, *markersize=None*, *aspect='equal'*, *origin='upper'*, ***kwargs*)

Plot the sparsity pattern of a 2D array.

This visualizes the non-zero values of the array.

Two plotting styles are available: image and marker. Both are available for full arrays, but only the marker style works for `scipy.sparse.spmatrix` instances.

**Image style**

If *marker* and *markersize* are *None*, `imshow` is used. Any extra remaining keyword arguments are passed to this method.

#### Marker style

If *Z* is a `scipy.sparse.spmatrix` or *marker* or *markersize* are *None*, a *Line2D* object will be returned with the value of marker determining the marker type, and any remaining keyword arguments passed to *plot*.

**Parameters**

**Z**

[(M, N) array-like] The array to be plotted.

**precision**

[float or 'present', default: 0] If *precision* is 0, any non-zero value will be plotted. Otherwise, values of $|Z| > precision$ will be plotted.

For `scipy.sparse.spmatrix` instances, you can also pass 'present'. In this case any value present in the array will be plotted, even if it is identically zero.

**aspect**

[{'equal', 'auto', None} or float, default: 'equal'] The aspect ratio of the Axes. This parameter is particularly relevant for images since it determines whether data pixels are square.

This parameter is a shortcut for explicitly calling *Axes.set_aspect*. See there for further details.

- 'equal': Ensures an aspect ratio of 1. Pixels will be square.
- 'auto': The Axes is kept fixed and the aspect is adjusted so that the data fit in the Axes. In general, this will result in non-square pixels.
- *None*: Use *rcParams["image.aspect"]* (default: `'equal'`).

**origin**

[{'upper', 'lower'}, default: *rcParams["image.origin"]* (default: `'upper'`)] Place the [0, 0] index of the array in the upper left or lower left corner of the Axes. The convention 'upper' is typically used for matrices and images.

**Returns**

***AxesImage* or *Line2D***

The return type depends on the plotting style (see above).

**Other Parameters**

****kwargs**

The supported additional parameters depend on the plotting style.

For the image style, you can pass the following additional parameters of *imshow*:

- *cmap*

- *alpha*
- *url*
- any *Artist* properties (passed on to the *AxesImage*)

For the marker style, you can pass any *Line2D* property except for *linestyle*:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi va |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* or aa | bool |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* or c | *color* |
| *dash_capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *dash_joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *dashes* | sequence of floats (on/off ink in points) or (None, None) |
| *data* | (2, N) array or two 1D arrays |
| *drawstyle* or ds | {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: ' |
| *figure* | *Figure* |
| *fillstyle* | {'full', 'left', 'right', 'bottom', 'top', 'none'} |
| *gapcolor* | *color* or None |
| *gid* | str |
| *in_layout* | bool |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float |
| *marker* | marker style string, *Path* or *MarkerStyle* |
| *markeredgecolor* or mec | *color* |
| *markeredgewidth* or mew | float |
| *markerfacecolor* or mfc | *color* |
| *markerfacecoloralt* or mfcalt | *color* |
| *markersize* or ms | float |
| *markevery* | None or int or (int, int) or slice or list[int] or float or (float, floa |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | float or callable[[Artist, Event], tuple[bool, dict]] |
| *pickradius* | float |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *solid_capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *solid_joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *transform* | unknown |

Table 128 – continued from

| Property | Description |
|---|---|
| *url* | str |
| *visible* | bool |
| *xdata* | 1D array |
| *ydata* | 1D array |
| *zorder* | float |

### Notes

**Note:** This is the *pyplot wrapper* for *axes.Axes.spy*.

### Examples using `matplotlib.pyplot.spy`

- *Spy Demos*

## matplotlib.pyplot.figimage

matplotlib.pyplot.**figimage**(*X*, *xo=0*, *yo=0*, *alpha=None*, *norm=None*, *cmap=None*, *vmin=None*, *vmax=None*, *origin=None*, *resize=False*, ***kwargs*)

Add a non-resampled image to the figure.

The image is attached to the lower or upper left corner depending on *origin*.

**Parameters**

**X**

The image data. This is an array of one of the following shapes:

- (M, N): an image with scalar data. Color-mapping is controlled by *cmap*, *norm*, *vmin*, and *vmax*.
- (M, N, 3): an image with RGB values (0-1 float or 0-255 int).
- (M, N, 4): an image with RGBA values (0-1 float or 0-255 int), i.e. including transparency.

**xo, yo**

[int] The *x*/*y* image offset in pixels.

**alpha**

[None or float] The alpha blending value.

**cmap**

[str or *Colormap*, default: *rcParams["image.cmap"]* (default: `'viridis'`)] The Colormap instance or registered colormap name used to map scalar data to colors.

This parameter is ignored if *X* is RGB(A).

**norm**

[str or *Normalize*, optional] The normalization method used to scale scalar data to the [0, 1] range before mapping to colors using *cmap*. By default, a linear scaling is used, mapping the lowest value to 0 and the highest to 1.

If given, this can be one of the following:

- An instance of *Normalize* or one of its subclasses (see *Colormap normalization*).
- A scale name, i.e. one of "linear", "log", "symlog", "logit", etc. For a list of available scales, call *matplotlib.scale.get_scale_names()*. In that case, a suitable *Normalize* subclass is dynamically generated and instantiated.

This parameter is ignored if *X* is RGB(A).

**vmin, vmax**

[float, optional] When using scalar data and no explicit *norm*, *vmin* and *vmax* define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data. It is an error to use *vmin*/*vmax* when a *norm* instance is given (but using a `str` *norm* name together with *vmin*/*vmax* is acceptable).

This parameter is ignored if *X* is RGB(A).

**origin**

[{'upper', 'lower'}, default: *rcParams["image.origin"]* (default: `'upper'`)] Indicates where the [0, 0] index of the array is in the upper left or lower left corner of the Axes.

**resize**

[bool] If *True*, resize the figure to match the given image size.

**Returns**

***matplotlib.image.FigureImage***

**Other Parameters**

****kwargs**

Additional kwargs are *Artist* kwargs passed on to *FigureImage*.

### Notes

**Note:** This is the *pyplot wrapper* for `Figure.figimage`.

figimage complements the Axes image (`imshow`) which will be resampled to fit the current Axes. If you want a resampled image to fill the entire figure, you can define an `Axes` with extent [0, 0, 1, 1].

### Examples

```
f = plt.figure()
nx = int(f.get_figwidth() * f.dpi)
ny = int(f.get_figheight() * f.dpi)
data = np.random.random((ny, nx))
f.figimage(data)
plt.show()
```

## Examples using `matplotlib.pyplot.figimage`

- *Figimage Demo*

## Unstructured triangles

| | |
|---|---|
| `triplot` | Draw an unstructured triangular grid as lines and/or markers. |
| `tripcolor` | Create a pseudocolor plot of an unstructured triangular grid. |
| `tricontour` | Draw contour lines on an unstructured triangular grid. |
| `tricontourf` | Draw contour regions on an unstructured triangular grid. |

## matplotlib.pyplot.triplot

matplotlib.pyplot.**triplot**(**args*, ***kwargs*)

Draw an unstructured triangular grid as lines and/or markers.

Call signatures:

```
triplot(triangulation, ...)
triplot(x, y, [triangles], *, [mask=mask], ...)
```

The triangular grid can be specified either by passing a *Triangulation* object as the first parameter, or by passing the points *x*, *y* and optionally the *triangles* and a *mask*. If neither of *triangulation* or *triangles* are given, the triangulation is calculated on the fly.

**Parameters**

**triangulation**

[*Triangulation*] An already created triangular grid.

**x, y, triangles, mask**

Parameters defining the triangular grid. See *Triangulation*. This is mutually exclusive with specifying *triangulation*.

**other_parameters**

All other args and kwargs are forwarded to *plot*.

**Returns**

**lines**

[*Line2D*] The drawn triangles edges.

**markers**

[*Line2D*] The drawn marker nodes.

## Notes

**Note:** This is the *pyplot wrapper* for *axes.Axes.triplot*.

## Examples using `matplotlib.pyplot.triplot`

- *Tricontour Smooth Delaunay*
- *Trigradient Demo*
- *Triinterp Demo*
- *Triplot Demo*

## matplotlib.pyplot.tripcolor

matplotlib.pyplot.**tripcolor**(**args*, *alpha=1.0*, *norm=None*, *cmap=None*, *vmin=None*, *vmax=None*, *shading='flat'*, *facecolors=None*, ***kwargs*)

Create a pseudocolor plot of an unstructured triangular grid.

Call signatures:

```
tripcolor(triangulation, c, *, ...)
tripcolor(x, y, c, *, [triangles=triangles], [mask=mask], ...)
```

The triangular grid can be specified either by passing a *Triangulation* object as the first parameter, or by passing the points *x*, *y* and optionally the *triangles* and a *mask*. See *Triangulation* for an explanation of these parameters.

It is possible to pass the triangles positionally, i.e. `tripcolor(x, y, triangles, c, ...)`. However, this is discouraged. For more clarity, pass *triangles* via keyword argument.

If neither of *triangulation* or *triangles* are given, the triangulation is calculated on the fly. In this case, it does not make sense to provide colors at the triangle faces via *c* or *facecolors* because there are multiple possible triangulations for a group of points and you don't know which triangles will be constructed.

**Parameters**

**triangulation**

[*Triangulation*] An already created triangular grid.

**x, y, triangles, mask**

Parameters defining the triangular grid. See *Triangulation*. This is mutually exclusive with specifying *triangulation*.

**c**

[array-like] The color values, either for the points or for the triangles. Which one is automatically inferred from the length of *c*, i.e. does it match the number of points or the number of triangles. If there are the same number of points and triangles in the triangulation it is assumed that color values are defined at points; to force the use of color values at triangles use the keyword argument `facecolors=c` instead of just `c`. This parameter is position-only.

**facecolors**

[array-like, optional] Can be used alternatively to *c* to specify colors at the triangle faces. This parameter takes precedence over *c*.

**shading**

[{'flat', 'gouraud'}, default: 'flat'] If 'flat' and the color values *c* are defined at points, the color values used for each triangle are from the mean c of the triangle's three points. If *shading* is 'gouraud' then color values must be defined at points.

**other_parameters**

All other parameters are the same as for *pcolor*.

### Notes

**Note:** This is the *pyplot wrapper* for *axes.Axes.tripcolor*.

## Examples using `matplotlib.pyplot.tripcolor`

- *Tripcolor Demo*

## matplotlib.pyplot.tricontour

matplotlib.pyplot.**tricontour**(**args*, ***kwargs*)

Draw contour lines on an unstructured triangular grid.

Call signatures:

```
tricontour(triangulation, z, [levels], ...)
tricontour(x, y, z, [levels], *, [triangles=triangles], [mask=mask], ...)
```

The triangular grid can be specified either by passing a *Triangulation* object as the first parameter, or by passing the points *x*, *y* and optionally the *triangles* and a *mask*. See *Triangulation* for an explanation of these parameters. If neither of *triangulation* or *triangles* are given, the triangulation is calculated on the fly.

It is possible to pass *triangles* positionally, i.e. `tricontour(x, y, triangles, z, ...)`. However, this is discouraged. For more clarity, pass *triangles* via keyword argument.

**Parameters**

**triangulation**

[*Triangulation*, optional] An already created triangular grid.

**x, y, triangles, mask**

Parameters defining the triangular grid. See *Triangulation*. This is mutually exclusive with specifying *triangulation*.

**z**

[array-like] The height values over which the contour is drawn. Color-mapping is controlled by *cmap*, *norm*, *vmin*, and *vmax*.

**Note:** All values in $z$ must be finite. Hence, nan and inf values must either be removed or *set_mask* be used.

**levels**

[int or array-like, optional] Determines the number and positions of the contour lines / regions.

If an int *n*, use *MaxNLocator*, which tries to automatically choose no more than *n+1* "nice" contour levels between between minimum and maximum numeric values of *Z*.

If array-like, draw contour lines at the specified levels. The values must be in increasing order.

**Returns**

***TriContourSet***

**Other Parameters**

**colors**

[*color* or list of *color*, optional] The colors of the levels, i.e., the contour lines.

The sequence is cycled for the levels in ascending order. If the sequence is shorter than the number of levels, it is repeated.

As a shortcut, single color strings may be used in place of one-element lists, i.e. `'red'` instead of `['red']` to color all levels with the same color. This shortcut does only work for color strings, not for other ways of specifying colors.

By default (value *None*), the colormap specified by *cmap* will be used.

**alpha**

[float, default: 1] The alpha blending value, between 0 (transparent) and 1 (opaque).

**cmap**

[str or *Colormap*, default: *rcParams["image.cmap"]* (default: `'viridis'`)] The Colormap instance or registered colormap name used to map scalar data to colors.

This parameter is ignored if *colors* is set.

**norm**

[str or *Normalize*, optional] The normalization method used to scale scalar data to the [0, 1] range before mapping to colors using *cmap*. By default, a linear scaling is used, mapping the lowest value to 0 and the highest to 1.

If given, this can be one of the following:

- An instance of *Normalize* or one of its subclasses (see *Colormap normalization*).
- A scale name, i.e. one of "linear", "log", "symlog", "logit", etc. For a list of available scales, call *matplotlib.scale.get_scale_names()*. In that case, a suitable *Normalize* subclass is dynamically generated and instantiated.

This parameter is ignored if *colors* is set.

**vmin, vmax**

[float, optional] When using scalar data and no explicit *norm*, *vmin* and *vmax* define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data. It is an error to use *vmin*/*vmax* when a *norm* instance is given (but using a `str` *norm* name together with *vmin*/*vmax* is acceptable).

If *vmin* or *vmax* are not given, the default color scaling is based on *levels*.

This parameter is ignored if *colors* is set.

**origin**

[{*None*, 'upper', 'lower', 'image'}, default: None] Determines the orientation and exact position of *z* by specifying the position of `z[0, 0]`. This is only relevant, if *X*, *Y* are not given.

- *None*: `z[0, 0]` is at X=0, Y=0 in the lower left corner.
- 'lower': `z[0, 0]` is at X=0.5, Y=0.5 in the lower left corner.
- 'upper': `z[0, 0]` is at X=N+0.5, Y=0.5 in the upper left corner.
- 'image': Use the value from *rcParams["image.origin"]* (default: `'upper'`).

**extent**

[(x0, x1, y0, y1), optional] If *origin* is not *None*, then *extent* is interpreted as in *imshow*: it gives the outer pixel boundaries. In this case, the position of z[0, 0] is the center of the pixel, not a corner. If *origin* is *None*, then (*x0*, *y0*) is the position of z[0, 0], and (*x1*, *y1*) is the position of z[-1, -1].

This argument is ignored if *X* and *Y* are specified in the call to contour.

**locator**

[ticker.Locator subclass, optional] The locator is used to determine the contour levels if they are not given explicitly via *levels*. Defaults to *MaxNLocator*.

**extend**

[{'neither', 'both', 'min', 'max'}, default: 'neither'] Determines the `tricontour`-coloring of values that are outside the *levels* range.

If 'neither', values outside the *levels* range are not colored. If 'min', 'max' or 'both', color the values below, above or below and above the *levels* range.

Values below `min(levels)` and above `max(levels)` are mapped to the under/over values of the *Colormap*. Note that most colormaps do not have dedicated colors for these by default, so that the over and under values are the edge values of the colormap. You may want to set these values explicitly using *Colormap.set_under* and *Colormap.set_over*.

---

**Note:** An existing *TriContourSet* does not get notified if properties of its colormap are changed. Therefore, an explicit call to *ContourSet.changed()* is needed after modifying the colormap. The explicit call can be left out, if a colorbar is assigned to the *TriContourSet* because it internally calls *ContourSet.changed()*.

---

**xunits, yunits**

[registered units, optional] Override axis units by specifying an instance of a *matplotlib.units.ConversionInterface*.

**antialiased**

[bool, optional] Enable antialiasing, overriding the defaults. For filled contours, the default is *True*. For line contours, it is taken from *rcParams["lines.antialiased"]* (default: `True`).

**linewidths**

[float or array-like, default: *rcParams["contour.linewidth"]* (default: `None`)] The line width of the contour lines.

If a number, all levels will be plotted with this linewidth.

If a sequence, the levels in ascending order will be plotted with the linewidths in the order specified.

If None, this falls back to *rcParams["lines.linewidth"]* (default: `1.5`).

**linestyles**

[{*None*, 'solid', 'dashed', 'dashdot', 'dotted'}, optional] If *linestyles* is *None*, the default is 'solid' unless the lines are monochrome. In that case, negative contours will take their linestyle from *rcParams["contour.negative_linestyle"]* (default: `'dashed'`) setting.

*linestyles* can also be an iterable of the above strings specifying a set of linestyles to be used. If this iterable is shorter than the number of contour levels it will be repeated as necessary.

### Notes

**Note:** This is the *pyplot wrapper* for `axes.Axes.tricontour`.

## Examples using `matplotlib.pyplot.tricontour`

- *Contour plot of irregularly spaced data*
- *Tricontour Smooth Delaunay*
- *Tricontour Smooth User*
- *Trigradient Demo*

## matplotlib.pyplot.tricontourf

matplotlib.pyplot.**tricontourf**(**args*, ***kwargs*)

Draw contour regions on an unstructured triangular grid.

Call signatures:

```
tricontourf(triangulation, z, [levels], ...)
tricontourf(x, y, z, [levels], *, [triangles=triangles], [mask=mask], ...
 ↪)
```

The triangular grid can be specified either by passing a `Triangulation` object as the first parameter, or by passing the points *x*, *y* and optionally the *triangles* and a *mask*. See `Triangulation` for an explanation of these parameters. If neither of *triangulation* or *triangles* are given, the triangulation is calculated on the fly.

It is possible to pass *triangles* positionally, i.e. `tricontourf(x, y, triangles, z, ...)`. However, this is discouraged. For more clarity, pass *triangles* via keyword argument.

**Parameters**

**triangulation**

[`Triangulation`, optional] An already created triangular grid.

**x, y, triangles, mask**

Parameters defining the triangular grid. See `Triangulation`. This is mutually exclusive with specifying *triangulation*.

**z**

[array-like] The height values over which the contour is drawn. Color-mapping is controlled by *cmap*, *norm*, *vmin*, and *vmax*.

**Note:** All values in $z$ must be finite. Hence, nan and inf values must either be removed or *set_mask* be used.

**levels**

[int or array-like, optional] Determines the number and positions of the contour lines / regions.

If an int *n*, use *MaxNLocator*, which tries to automatically choose no more than *n+1* "nice" contour levels between between minimum and maximum numeric values of *Z*.

If array-like, draw contour lines at the specified levels. The values must be in increasing order.

**Returns**

***TriContourSet***

**Other Parameters**

**colors**

[*color* or list of *color*, optional] The colors of the levels, i.e., the contour regions.

The sequence is cycled for the levels in ascending order. If the sequence is shorter than the number of levels, it is repeated.

As a shortcut, single color strings may be used in place of one-element lists, i.e. `'red'` instead of `['red']` to color all levels with the same color. This shortcut does only work for color strings, not for other ways of specifying colors.

By default (value *None*), the colormap specified by *cmap* will be used.

**alpha**

[float, default: 1] The alpha blending value, between 0 (transparent) and 1 (opaque).

**cmap**

[str or *Colormap*, default: *rcParams["image.cmap"]* (default: `'viridis'`)] The Colormap instance or registered colormap name used to map scalar data to colors.

This parameter is ignored if *colors* is set.

**norm**

[str or *Normalize*, optional] The normalization method used to scale scalar data to the [0, 1] range before mapping to colors using *cmap*. By default, a linear scaling is used, mapping the lowest value to 0 and the highest to 1.

If given, this can be one of the following:

- An instance of *Normalize* or one of its subclasses (see *Colormap normalization*).
- A scale name, i.e. one of "linear", "log", "symlog", "logit", etc. For a list of available scales, call *matplotlib.scale.get_scale_names()*. In that case, a suitable *Normalize* subclass is dynamically generated and instantiated.

This parameter is ignored if *colors* is set.

**vmin, vmax**

[float, optional] When using scalar data and no explicit *norm*, *vmin* and *vmax* define the data range that the colormap covers. By default, the colormap covers the complete value range of the supplied data. It is an error to use *vmin*/*vmax* when a *norm* instance is given (but using a `str` *norm* name together with *vmin*/*vmax* is acceptable).

If *vmin* or *vmax* are not given, the default color scaling is based on *levels*.

This parameter is ignored if *colors* is set.

**origin**

[{*None*, 'upper', 'lower', 'image'}, default: None] Determines the orientation and exact position of *z* by specifying the position of `z[0, 0]`. This is only relevant, if *X*, *Y* are not given.

- *None*: `z[0, 0]` is at X=0, Y=0 in the lower left corner.
- 'lower': `z[0, 0]` is at X=0.5, Y=0.5 in the lower left corner.
- 'upper': `z[0, 0]` is at X=N+0.5, Y=0.5 in the upper left corner.
- 'image': Use the value from *rcParams["image.origin"]* (default: `'upper'`).

**extent**

[(x0, x1, y0, y1), optional] If *origin* is not *None*, then *extent* is interpreted as in *imshow*: it gives the outer pixel boundaries. In this case, the position of z[0, 0] is the center of the pixel, not a corner. If *origin* is *None*, then (*x0*, *y0*) is the position of z[0, 0], and (*x1*, *y1*) is the position of z[-1, -1].

This argument is ignored if *X* and *Y* are specified in the call to contour.

**locator**

[ticker.Locator subclass, optional] The locator is used to determine the contour levels if they are not given explicitly via *levels*. Defaults to *MaxNLocator*.

**extend**

[{'neither', 'both', 'min', 'max'}, default: 'neither'] Determines the `tricontourf`-coloring of values that are outside the *levels* range.

If 'neither', values outside the *levels* range are not colored. If 'min', 'max' or 'both', color the values below, above or below and above the *levels* range.

Values below `min(levels)` and above `max(levels)` are mapped to the under/over values of the *Colormap*. Note that most colormaps do not have dedicated colors for these by default, so that the over and under values are the edge values of the colormap. You may want to set these values explicitly using *Colormap.set_under* and *Colormap.set_over*.

---

**Note:** An existing *TriContourSet* does not get notified if properties of its colormap are changed. Therefore, an explicit call to *ContourSet.changed()* is needed after modifying the colormap. The explicit call can be left out, if a colorbar is assigned to the *TriContourSet* because it internally calls *ContourSet.changed()*.

---

**xunits, yunits**

[registered units, optional] Override axis units by specifying an instance of a *matplotlib.units.ConversionInterface*.

**antialiased**

[bool, optional] Enable antialiasing, overriding the defaults. For filled contours, the default is *True*. For line contours, it is taken from *rcParams["lines.antialiased"]* (default: `True`).

**hatches**

[list[str], optional] A list of crosshatch patterns to use on the filled areas. If None, no hatching will be added to the contour.

## Notes

---

**Note:** This is the *pyplot wrapper* for *axes.Axes.tricontourf*.

---

*tricontourf* fills intervals that are closed at the top; that is, for boundaries *z1* and *z2*, the filled region is:

```
z1 < Z <= z2
```

except for the lowest interval, which is closed on both sides (i.e. it includes the lowest value).

### Examples using `matplotlib.pyplot.tricontourf`

- *Contour plot of irregularly spaced data*
- *Tricontour Demo*
- *Tricontour Smooth Delaunay*
- *Tricontour Smooth User*
- *Triinterp Demo*

### Text and annotations

| | |
|---|---|
| *annotate* | Annotate the point *xy* with text *text*. |
| *text* | Add text to the Axes. |
| *figtext* | Add text to figure. |
| *table* | Add a table to an *Axes*. |
| *arrow* | Add an arrow to the Axes. |
| *figlegend* | Place a legend on the figure. |
| *legend* | Place a legend on the Axes. |

### matplotlib.pyplot.annotate

matplotlib.pyplot.**annotate**(*text*, *xy*, *xytext=None*, *xycoords='data'*, *textcoords=None*, *arrowprops=None*, *annotation_clip=None*, ***kwargs*)

Annotate the point *xy* with text *text*.

In the simplest form, the text is placed at *xy*.

Optionally, the text can be displayed in another position *xytext*. An arrow pointing from the text to the annotated point *xy* can then be added by defining *arrowprops*.

**Parameters**

**text**

[str] The text of the annotation.

**xy**

[(float, float)] The point *(x, y)* to annotate. The coordinate system is determined by *xycoords*.

**xytext**

[(float, float), default: *xy*] The position *(x, y)* to place the text at. The coordinate system is determined by *textcoords*.

**xycoords**

[single or two-tuple of str or *Artist* or *Transform* or callable, default: 'data'] The coordinate system that *xy* is given in. The following types of values are supported:

- One of the following strings:

| Value | Description |
|---|---|
| 'figure points' | Points from the lower left of the figure |
| 'figure pixels' | Pixels from the lower left of the figure |
| 'figure fraction' | Fraction of figure from lower left |
| 'subfigure points' | Points from the lower left of the subfigure |
| 'subfigure pixels' | Pixels from the lower left of the subfigure |
| 'subfigure fraction' | Fraction of subfigure from lower left |
| 'axes points' | Points from lower left corner of the Axes |
| 'axes pixels' | Pixels from lower left corner of the Axes |
| 'axes fraction' | Fraction of Axes from lower left |
| 'data' | Use the coordinate system of the object being annotated (default) |
| 'polar' | *(theta, r)* if not native 'data' coordinates |

  Note that 'subfigure pixels' and 'figure pixels' are the same for the parent figure, so users who want code that is usable in a subfigure can use 'subfigure pixels'.

- An *Artist*: *xy* is interpreted as a fraction of the artist's *Bbox*. E.g. *(0, 0)* would be the lower left corner of the bounding box and *(0.5, 1)* would be the center top of the bounding box.

- A *Transform* to transform *xy* to screen coordinates.

- A function with one of the following signatures:

```
def transform(renderer) -> Bbox
def transform(renderer) -> Transform
```

  where *renderer* is a *RendererBase* subclass.

  The result of the function is interpreted like the *Artist* and *Transform* cases above.

- A tuple *(xcoords, ycoords)* specifying separate coordinate systems for *x* and *y*. *xcoords* and *ycoords* must each be of one of the above described types.

See *Advanced annotation* for more details.

**textcoords**

[single or two-tuple of str or *Artist* or *Transform* or callable, default: value of *xycoords*] The coordinate system that *xytext* is given in.

All *xycoords* values are valid as well as the following strings:

| Value | Description |
|---|---|
| 'offset points' | Offset, in points, from the *xy* value |
| 'offset pixels' | Offset, in pixels, from the *xy* value |
| 'offset fontsize' | Offset, relative to fontsize, from the *xy* value |

**arrowprops**

[dict, optional] The properties used to draw a *FancyArrowPatch* arrow between the positions *xy* and *xytext*. Defaults to None, i.e. no arrow is drawn.

For historical reasons there are two different ways to specify arrows, "simple" and "fancy":

**Simple arrow:**

If *arrowprops* does not contain the key 'arrowstyle' the allowed keys are:

| Key | Description |
|---|---|
| width | The width of the arrow in points |
| headwidth | The width of the base of the arrow head in points |
| headlength | The length of the arrow head in points |
| shrink | Fraction of total length to shrink from both ends |
| ? | Any *FancyArrowPatch* property |

The arrow is attached to the edge of the text box, the exact position (corners or centers) depending on where it's pointing to.

**Fancy arrow:**

This is used if 'arrowstyle' is provided in the *arrowprops*.

Valid keys are the following *FancyArrowPatch* parameters:

| Key | Description |
|---|---|
| arrowstyle | The arrow style |
| connectionstyle | The connection style |
| relpos | See below; default is (0.5, 0.5) |
| patchA | Default is bounding box of the text |
| patchB | Default is None |
| shrinkA | In points. Default is 2 points |
| shrinkB | In points. Default is 2 points |
| mutation_scale | Default is text size (in points) |
| mutation_aspect | Default is 1 |
| ? | Any `FancyArrowPatch` property |

The exact starting point position of the arrow is defined by *relpos*. It's a tuple of relative coordinates of the text box, where (0, 0) is the lower left corner and (1, 1) is the upper right corner. Values <0 and >1 are supported and specify points outside the text box. By default (0.5, 0.5), so the starting point is centered in the text box.

**annotation_clip**

[bool or None, default: None] Whether to clip (i.e. not draw) the annotation when the annotation point *xy* is outside the Axes area.

- If *True*, the annotation will be clipped when *xy* is outside the Axes.
- If *False*, the annotation will always be drawn.
- If *None*, the annotation will be clipped when *xy* is outside the Axes and *xycoords* is 'data'.

****kwargs**

Additional kwargs are passed to `Text`.

**Returns**

`Annotation`

**See also:**

*Annotations*

### Notes

**Note:** This is the *pyplot wrapper* for `axes.Axes.annotate`.

### Examples using `matplotlib.pyplot.annotate`

- *Hat graph*
- *Scale invariant angle label*
- *Annotate Transform*
- *Annotating a plot*
- *Annotation Polar*
- *Pyplot tutorial*

## matplotlib.pyplot.text

`matplotlib.pyplot.`**`text`**(*x*, *y*, *s*, *fontdict=None*, ***kwargs*)

Add text to the Axes.

Add the text *s* to the Axes at location *x*, *y* in data coordinates, with a default `horizontalalignment` on the `left` and `verticalalignment` at the `baseline`. See *Text alignment*.

**Parameters**

**x, y**

[float] The position to place the text. By default, this is in data coordinates. The coordinate system can be changed using the *transform* parameter.

**s**

[str] The text.

**fontdict**

[dict, default: None]

**Discouraged**

The use of *fontdict* is discouraged. Parameters should be passed as individual keyword arguments or using dictionary-unpacking `text(..., **fontdict)`.

A dictionary to override the default text properties. If fontdict is None, the defaults are determined by `rcParams`.

### Returns

***Text***

The created *Text* instance.

### Other Parameters

****kwargs**

[*Text* properties.] Other miscellaneous text parameters.

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array a |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* | bool |
| *backgroundcolor* | *color* |
| *bbox* | dict with properties for *patches.FancyBboxPa* |
| *clip_box* | unknown |
| *clip_on* | unknown |
| *clip_path* | unknown |
| *color* or c | *color* |
| *figure* | *Figure* |
| *fontfamily* or family or fontname | {FONTNAME, 'serif', 'sans-serif', 'cursive', 'fantasy' |
| *fontproperties* or font or font_properties | *font_manager.FontProperties* or *str* or |
| *fontsize* or size | float or {'xx-small', 'x-small', 'small', 'medium', 'large |
| *fontstretch* or stretch | {a numeric value in range 0-1000, 'ultra-condensed', |
| *fontstyle* or style | {'normal', 'italic', 'oblique'} |
| *fontvariant* or variant | {'normal', 'small-caps'} |
| *fontweight* or weight | {a numeric value in range 0-1000, 'ultralight', 'light', |
| *gid* | str |
| *horizontalalignment* or ha | {'left', 'center', 'right'} |
| *in_layout* | bool |
| *label* | object |
| *linespacing* | float (multiple of font size) |
| *math_fontfamily* | str |
| *mouseover* | bool |
| *multialignment* or ma | {'left', 'right', 'center'} |
| *parse_math* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *position* | (float, float) |
| *rasterized* | bool |
| *rotation* | float or {'vertical', 'horizontal'} |
| *rotation_mode* | {None, 'default', 'anchor'} |
| *sketch_params* | (scale: float, length: float, randomness: float) |

Table 129 -

| Property | Description |
|---|---|
| *snap* | bool or None |
| *text* | object |
| *transform* | *Transform* |
| *transform_rotates_text* | bool |
| *url* | str |
| *usetex* | bool, default: *rcParams["text.usetex"]* (de |
| *verticalalignment* or va | {'baseline', 'bottom', 'center', 'center_baseline', 'top'} |
| *visible* | bool |
| *wrap* | bool |
| *x* | float |
| *y* | float |
| *zorder* | float |

### Notes

**Note:** This is the *pyplot wrapper* for *axes.Axes.text*.

### Examples

Individual keyword arguments can be used to override any given parameter:

```
>>> text(x, y, s, fontsize=12)
```

The default transform specifies that text is in data coords, alternatively, you can specify text in axis coords ((0, 0) is lower-left and (1, 1) is upper-right). The example below places text in the center of the Axes:

```
>>> text(0.5, 0.5, 'matplotlib', horizontalalignment='center',
...      verticalalignment='center', transform=ax.transAxes)
```

You can put a rectangular box around the text instance (e.g., to set a background color) by using the keyword *bbox*. *bbox* is a dictionary of *Rectangle* properties. For example:

```
>>> text(x, y, s, bbox=dict(facecolor='red', alpha=0.5))
```

### Examples using `matplotlib.pyplot.text`

- *Figure size in different units*
- *Auto-wrapping text*
- *Text Rotation Mode*
- *Styling text boxes*
- *Controlling style of text and labels using a dictionary*
- *Text and mathtext using pyplot*
- *Close Event*
- *transforms.offset_copy*
- *Anscombe's quartet*
- *Pyplot tutorial*
- *Path effects guide*

## matplotlib.pyplot.figtext

`matplotlib.pyplot.`**`figtext`**(*x*, *y*, *s*, *fontdict=None*, ***kwargs*)

Add text to figure.

**Parameters**

**x, y**

[float] The position to place the text. By default, this is in figure coordinates, floats in [0, 1]. The coordinate system can be changed using the *transform* keyword.

**s**

[str] The text string.

**fontdict**

[dict, optional] A dictionary to override the default text properties. If not given, the defaults are determined by `rcParams["font.*"]`. Properties passed as *kwargs* override the corresponding ones given in *fontdict*.

**Returns**

***Text***

**Other Parameters**

****kwargs**

[*Text* properties] Other miscellaneous text parameters.

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array a |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* | bool |
| *backgroundcolor* | *color* |
| *bbox* | dict with properties for *patches.FancyBboxPa* |
| *clip_box* | unknown |
| *clip_on* | unknown |
| *clip_path* | unknown |
| *color* or c | *color* |
| *figure* | *Figure* |
| *fontfamily* or family or fontname | {FONTNAME, 'serif', 'sans-serif', 'cursive', 'fantasy' |
| *fontproperties* or font or font_properties | *font_manager.FontProperties* or *str* or |
| *fontsize* or size | float or {'xx-small', 'x-small', 'small', 'medium', 'large |
| *fontstretch* or stretch | {a numeric value in range 0-1000, 'ultra-condensed', |
| *fontstyle* or style | {'normal', 'italic', 'oblique'} |
| *fontvariant* or variant | {'normal', 'small-caps'} |
| *fontweight* or weight | {a numeric value in range 0-1000, 'ultralight', 'light', |
| *gid* | str |
| *horizontalalignment* or ha | {'left', 'center', 'right'} |
| *in_layout* | bool |
| *label* | object |
| *linespacing* | float (multiple of font size) |
| *math_fontfamily* | str |
| *mouseover* | bool |
| *multialignment* or ma | {'left', 'right', 'center'} |
| *parse_math* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *position* | (float, float) |
| *rasterized* | bool |
| *rotation* | float or {'vertical', 'horizontal'} |
| *rotation_mode* | {None, 'default', 'anchor'} |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *text* | object |
| *transform* | *Transform* |
| *transform_rotates_text* | bool |
| *url* | str |
| *usetex* | bool, default: *rcParams["text.usetex"]* (de |
| *verticalalignment* or va | {'baseline', 'bottom', 'center', 'center_baseline', 'top'} |

Table 130 -

| Property | Description |
|---|---|
| visible | bool |
| wrap | bool |
| x | float |
| y | float |
| zorder | float |

**See also:**

**Axes.text**
**pyplot.text**

### Notes

**Note:** This is the *pyplot wrapper* for Figure.text.

## matplotlib.pyplot.table

matplotlib.pyplot.**table**(*cellText=None*, *cellColours=None*, *cellLoc='right'*, *colWidths=None*, *rowLabels=None*, *rowColours=None*, *rowLoc='left'*, *colLabels=None*, *colColours=None*, *colLoc='center'*, *loc='bottom'*, *bbox=None*, *edges='closed'*, ***kwargs*)

Add a table to an Axes.

At least one of *cellText* or *cellColours* must be specified. These parameters must be 2D lists, in which the outer lists define the rows and the inner list define the column values per row. Each row must have the same number of elements.

The table can optionally have row and column headers, which are configured using *rowLabels*, *rowColours*, *rowLoc* and *colLabels*, *colColours*, *colLoc* respectively.

For finer grained control over tables, use the Table class and add it to the Axes with Axes.add_table.

**Parameters**

**cellText**

[2D list of str, optional] The texts to place into the table cells.

*Note*: Line breaks in the strings are currently not accounted for and will result in the text exceeding the cell boundaries.

**cellColours**

[2D list of *color*, optional] The background colors of the cells.

**cellLoc**

[{'right', 'center', 'left'}] The alignment of the text within the cells.

**colWidths**

[list of float, optional] The column widths in units of the axes. If not given, all columns will have a width of *1 / ncols*.

**rowLabels**

[list of str, optional] The text of the row header cells.

**rowColours**

[list of *color*, optional] The colors of the row header cells.

**rowLoc**

[{'left', 'center', 'right'}] The text alignment of the row header cells.

**colLabels**

[list of str, optional] The text of the column header cells.

**colColours**

[list of *color*, optional] The colors of the column header cells.

**colLoc**

[{'center', 'left', 'right'}] The text alignment of the column header cells.

**loc**

[str, default: 'bottom'] The position of the cell with respect to *ax*. This must be one of the `codes`.

**bbox**

[`Bbox` or [xmin, ymin, width, height], optional] A bounding box to draw the table into. If this is not *None*, this overrides *loc*.

**edges**

[{'closed', 'open', 'horizontal', 'vertical'} or substring of 'BRTL'] The cell edges to be drawn with a line. See also `visible_edges`.

**Returns**

***Table***

The created table.

**Other Parameters**

****kwargs**

`Table` properties.

| Property | Description |
|---|---|
| agg_fil | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image |
| alpha | scalar or None |
| animated | bool |
| clip_bo | BboxBase or None |
| clip_on | bool |
| clip_pa | Patch or (Path, Transform) or None |
| figure | Figure |
| fontsize | float |
| gid | str |
| in_layo | bool |
| label | object |
| mouseov | bool |
| path_ef | list of AbstractPathEffect |
| picker | None or bool or float or callable |
| rasterized | bool |
| sketch_ | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| transform | Transform |
| url | str |
| visible | bool |
| zorder | float |

## Notes

**Note:** This is the *pyplot wrapper* for `axes.Axes.table`.

### Examples using `matplotlib.pyplot.table`

- *Table Demo*

### matplotlib.pyplot.arrow

`matplotlib.pyplot.`**`arrow`**(*x*, *y*, *dx*, *dy*, ***kwargs*)

Add an arrow to the Axes.

This draws an arrow from `(x, y)` to `(x+dx, y+dy)`.

**Parameters**

**x, y**

[float] The x and y coordinates of the arrow base.

**dx, dy**

[float] The length of the arrow along x and y direction.

**width**

[float, default: 0.001] Width of full arrow tail.

**length_includes_head**

[bool, default: False] True if head is to be counted in calculating the length.

**head_width**

[float or None, default: 3*width] Total width of the full arrow head.

**head_length**

[float or None, default: 1.5*head_width] Length of arrow head.

**shape**

[{'full', 'left', 'right'}, default: 'full'] Draw the left-half, right-half, or full arrow.

**overhang**

[float, default: 0] Fraction that the arrow is swept back (0 overhang means triangular shape). Can be negative or greater than one.

**head_starts_at_zero**

[bool, default: False] If True, the head starts being drawn at coordinate 0 instead of ending at coordinate 0.

****kwargs**

*Patch* properties:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and return |
| *alpha* | unknown |
| *animated* | bool |
| *antialiased* or aa | bool or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* | *color* |
| *edgecolor* or ec | *color* or None |
| *facecolor* or fc | *color* or None |
| *figure* | *Figure* |
| *fill* | bool |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float or None |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *zorder* | float |

**Returns**

***FancyArrow***

The created *FancyArrow* object.

### Notes

**Note:** This is the *pyplot wrapper* for *axes.Axes.arrow*.

The resulting arrow is affected by the Axes aspect ratio and limits. This may produce an arrow whose head is not square with its stem. To create an arrow whose head is square with its stem, use *annotate()* for example:

```
>>> ax.annotate("", xy=(0.5, 0.5), xytext=(0, 0),
...             arrowprops=dict(arrowstyle="->"))
```

## matplotlib.pyplot.figlegend

matplotlib.pyplot.**figlegend**(**args*, ***kwargs*)

Place a legend on the figure.

Call signatures:

```
figlegend()
figlegend(handles, labels)
figlegend(handles=handles)
figlegend(labels)
```

The call signatures correspond to the following different ways to use this method:

**1. Automatic detection of elements to be shown in the legend**

The elements to be added to the legend are automatically determined, when you do not pass in any extra arguments.

In this case, the labels are taken from the artist. You can specify them either at artist creation or by calling the *set_label()* method on the artist:

```
plt.plot([1, 2, 3], label='Inline label')
plt.figlegend()
```

or:

```
line, = plt.plot([1, 2, 3])
line.set_label('Label via method')
plt.figlegend()
```

Specific lines can be excluded from the automatic legend element selection by defining a label starting with an underscore. This is default for all artists, so calling *Figure.legend* without any arguments and without setting the labels manually will result in no legend being drawn.

**2. Explicitly listing the artists and labels in the legend**

For full control of which artists have a legend entry, it is possible to pass an iterable of legend artists followed by an iterable of legend labels respectively:

```
plt.figlegend([line1, line2, line3], ['label1', 'label2', 'label3'])
```

**3. Explicitly listing the artists in the legend**

This is similar to 2, but the labels are taken from the artists' label properties. Example:

```
line1, = ax1.plot([1, 2, 3], label='label1')
line2, = ax2.plot([1, 2, 3], label='label2')
plt.figlegend(handles=[line1, line2])
```

**4. Labeling existing plot elements**

---

**Discouraged**

This call signature is discouraged, because the relation between plot elements and labels is only implicit by their order and can easily be mixed up.

---

To make a legend for all artists on all Axes, call this function with an iterable of strings, one for each legend item. For example:

```
fig, (ax1, ax2) = plt.subplots(1, 2)
ax1.plot([1, 3, 5], color='blue')
ax2.plot([2, 4, 6], color='red')
plt.figlegend(['the blues', 'the reds'])
```

**Parameters**

**handles**

[list of *Artist*, optional] A list of Artists (lines, patches) to be added to the legend. Use this together with *labels*, if you need full control on what is shown in the legend and the automatic mechanism described above is not sufficient.

The length of handles and labels should be the same in this case. If they are not, they are truncated to the smaller length.

**labels**

[list of str, optional] A list of labels to show next to the artists. Use this together with *handles*, if you need full control on what is shown in the legend and the automatic mechanism described above is not sufficient.

**Returns**

***Legend***

**Other Parameters**

**loc**

[str or pair of floats, default: 'upper right'] The location of the legend.

The strings `'upper left'`, `'upper right'`, `'lower left'`, `'lower right'` place the legend at the corresponding corner of the figure.

The strings `'upper center'`, `'lower center'`, `'center left'`, `'center right'` place the legend at the center of the corresponding edge of the figure.

The string `'center'` places the legend at the center of the figure.

The location can also be a 2-tuple giving the coordinates of the lower-left corner of the legend in figure coordinates (in which case *bbox_to_anchor* will be ignored).

For back-compatibility, `'center right'` (but no other location) can also be spelled `'right'`, and each "string" location can also be given as a numeric value:

| Location String | Location Code |
|---|---|
| 'best' (Axes only) | 0 |
| 'upper right' | 1 |
| 'upper left' | 2 |
| 'lower left' | 3 |
| 'lower right' | 4 |
| 'right' | 5 |
| 'center left' | 6 |
| 'center right' | 7 |
| 'lower center' | 8 |
| 'upper center' | 9 |
| 'center' | 10 |

If a figure is using the constrained layout manager, the string codes of the *loc* keyword argument can get better layout behaviour using the prefix 'outside'. There is ambiguity at the corners, so 'outside upper right' will make space for the legend above the rest of the axes in the layout, and 'outside right upper' will make space on the right side of the layout. In addition to the values of *loc* listed above, we have 'outside right upper', 'outside right lower', 'outside left upper', and 'outside left lower'. See *Legend guide* for more details.

**bbox_to_anchor**

[*BboxBase*, 2-tuple, or 4-tuple of floats] Box that is used to position the legend in conjunction with *loc*. Defaults to `axes.bbox` (if called as a method to *Axes.legend*) or `figure.bbox` (if *Figure.legend*). This argument allows arbitrary placement of the legend.

Bbox coordinates are interpreted in the coordinate system given by *bbox_transform*, with the default transform Axes or Figure coordinates, depending on which `legend` is called.

If a 4-tuple or *BboxBase* is given, then it specifies the bbox `(x, y, width, height)` that the legend is placed in. To put the legend in the best location in

the bottom right quadrant of the Axes (or figure):

```
loc='best', bbox_to_anchor=(0.5, 0., 0.5, 0.5)
```

A 2-tuple `(x, y)` places the corner of the legend specified by *loc* at x, y. For example, to put the legend's upper right-hand corner in the center of the Axes (or figure) the following keywords can be used:

```
loc='upper right', bbox_to_anchor=(0.5, 0.5)
```

**ncols**

[int, default: 1] The number of columns that the legend has.

For backward compatibility, the spelling *ncol* is also supported but it is discouraged. If both are given, *ncols* takes precedence.

**prop**

[None or *FontProperties* or dict] The font properties of the legend. If None (default), the current *matplotlib.rcParams* will be used.

**fontsize**

[int or {'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large'}] The font size of the legend. If the value is numeric the size will be the absolute font size in points. String values are relative to the current default font size. This argument is only used if *prop* is not specified.

**labelcolor**

[str or list, default: *rcParams["legend.labelcolor"]* (default: `'None'`)] The color of the text in the legend. Either a valid color string (for example, 'red'), or a list of color strings. The labelcolor can also be made to match the color of the line or marker using 'linecolor', 'markerfacecolor' (or 'mfc'), or 'markeredgecolor' (or 'mec').

Labelcolor can be set globally using *rcParams["legend.labelcolor"]* (default: `'None'`). If None, use *rcParams["text.color"]* (default: `'black'`).

**numpoints**

[int, default: *rcParams["legend.numpoints"]* (default: `1`)] The number of marker points in the legend when creating a legend entry for a *Line2D* (line).

**scatterpoints**

[int, default: *rcParams["legend.scatterpoints"]* (default: `1`)] The number of marker points in the legend when creating a legend entry for a *PathCollection* (scatter plot).

**scatteryoffsets**

[iterable of floats, default: `[0.375, 0.5, 0.3125]`] The vertical offset (relative to the font size) for the markers created for a scatter plot legend entry. 0.0 is

at the base the legend text, and 1.0 is at the top. To draw all markers at the same height, set to `[0.5]`.

**markerscale**

[float, default: *rcParams["legend.markerscale"]* (default: `1.0`)] The relative size of legend markers compared to the originally drawn ones.

**markerfirst**

[bool, default: True] If *True*, legend marker is placed to the left of the legend label. If *False*, legend marker is placed to the right of the legend label.

**reverse**

[bool, default: False] If *True*, the legend labels are displayed in reverse order from the input. If *False*, the legend labels are displayed in the same order as the input.

New in version 3.7.

**frameon**

[bool, default: *rcParams["legend.frameon"]* (default: `True`)] Whether the legend should be drawn on a patch (frame).

**fancybox**

[bool, default: *rcParams["legend.fancybox"]* (default: `True`)] Whether round edges should be enabled around the *FancyBboxPatch* which makes up the legend's background.

**shadow**

[None, bool or dict, default: *rcParams["legend.shadow"]* (default: `False`)] Whether to draw a shadow behind the legend. The shadow can be configured using *Patch* keywords. Customization via *rcParams["legend.shadow"]* (default: `False`) is currently not supported.

**framealpha**

[float, default: *rcParams["legend.framealpha"]* (default: `0.8`)] The alpha transparency of the legend's background. If *shadow* is activated and *framealpha* is `None`, the default value is ignored.

**facecolor**

["inherit" or color, default: *rcParams["legend.facecolor"]* (default: `'inherit'`)] The legend's background color. If `"inherit"`, use *rcParams["axes.facecolor"]* (default: `'white'`).

**edgecolor**

["inherit" or color, default: *rcParams["legend.edgecolor"]* (default: `'0.8'`)] The legend's background patch edge color. If `"inherit"`, use *rcParams["axes.edgecolor"]* (default: `'black'`).

**mode**

[{"expand", None}] If *mode* is set to `"expand"` the legend will be horizontally expanded to fill the Axes area (or *bbox_to_anchor* if defines the legend's size).

**bbox_transform**

[None or *Transform*] The transform for the bounding box (*bbox_to_anchor*). For a value of `None` (default) the Axes' `transAxes` transform will be used.

**title**

[str or None] The legend's title. Default is no title (`None`).

**title_fontproperties**

[None or *FontProperties* or dict] The font properties of the legend's title. If None (default), the *title_fontsize* argument will be used if present; if *title_fontsize* is also None, the current *rcParams["legend.title_fontsize"]* (default: `None`) will be used.

**title_fontsize**

[int or {'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large'}, default: *rcParams["legend.title_fontsize"]* (default: `None`)] The font size of the legend's title. Note: This cannot be combined with *title_fontproperties*. If you want to set the fontsize alongside other font properties, use the *size* parameter in *title_fontproperties*.

**alignment**

[{'center', 'left', 'right'}, default: 'center'] The alignment of the legend title and the box of entries. The entries are aligned as a single block, so that markers always lined up.

**borderpad**

[float, default: *rcParams["legend.borderpad"]* (default: `0.4`)] The fractional whitespace inside the legend border, in font-size units.

**labelspacing**

[float, default: *rcParams["legend.labelspacing"]* (default: `0.5`)] The vertical space between the legend entries, in font-size units.

**handlelength**

[float, default: *rcParams["legend.handlelength"]* (default: `2.0`)] The length of the legend handles, in font-size units.

**handleheight**

[float, default: *rcParams["legend.handleheight"]* (default: `0.7`)] The height of the legend handles, in font-size units.

**handletextpad**

[float, default: *rcParams["legend.handletextpad"]* (default: `0.8`)] The pad between the legend handle and text, in font-size units.

**borderaxespad**

[float, default: *rcParams["legend.borderaxespad"]* (default: `0.5`)] The pad between the Axes and legend border, in font-size units.

**columnspacing**

[float, default: *rcParams["legend.columnspacing"]* (default: `2.0`)] The spacing between columns, in font-size units.

**handler_map**

[dict or None] The custom dictionary mapping instances or types to a legend handler. This *handler_map* updates the default handler map found at *matplotlib.legend.Legend.get_legend_handler_map*.

**draggable**

[bool, default: False] Whether the legend can be dragged with the mouse.

**See also:**

***Axes.legend***

### Notes

Some artists are not supported by this function. See *Legend guide* for details.

## matplotlib.pyplot.legend

matplotlib.pyplot.**legend**(**args*, ***kwargs*)

Place a legend on the Axes.

Call signatures:

```
legend()
legend(handles, labels)
legend(handles=handles)
legend(labels)
```

The call signatures correspond to the following different ways to use this method:

**1. Automatic detection of elements to be shown in the legend**

The elements to be added to the legend are automatically determined, when you do not pass in any extra arguments.

In this case, the labels are taken from the artist. You can specify them either at artist creation or by calling the *set_label()* method on the artist:

```
ax.plot([1, 2, 3], label='Inline label')
ax.legend()
```

or:

```
line, = ax.plot([1, 2, 3])
line.set_label('Label via method')
ax.legend()
```

---

**Note:** Specific artists can be excluded from the automatic legend element selection by using a label starting with an underscore, "_". A string starting with an underscore is the default label for all artists, so calling *Axes.legend* without any arguments and without setting the labels manually will result in a `UserWarning` and an empty legend being drawn.

---

**2. Explicitly listing the artists and labels in the legend**

For full control of which artists have a legend entry, it is possible to pass an iterable of legend artists followed by an iterable of legend labels respectively:

```
ax.legend([line1, line2, line3], ['label1', 'label2', 'label3'])
```

**3. Explicitly listing the artists in the legend**

This is similar to 2, but the labels are taken from the artists' label properties. Example:

```
line1, = ax.plot([1, 2, 3], label='label1')
line2, = ax.plot([1, 2, 3], label='label2')
ax.legend(handles=[line1, line2])
```

**4. Labeling existing plot elements**

---

**Discouraged**

This call signature is discouraged, because the relation between plot elements and labels is only implicit by their order and can easily be mixed up.

---

To make a legend for all artists on an Axes, call this function with an iterable of strings, one for each legend item. For example:

```
ax.plot([1, 2, 3])
ax.plot([5, 6, 7])
ax.legend(['First line', 'Second line'])
```

**Parameters**

**handles**

[list of (*Artist* or tuple of *Artist*), optional] A list of Artists (lines, patches) to be added to the legend. Use this together with *labels*, if you need full control on what is shown in the legend and the automatic mechanism described above is not sufficient.

The length of handles and labels should be the same in this case. If they are not, they are truncated to the smaller length.

If an entry contains a tuple, then the legend handler for all Artists in the tuple will be placed alongside a single label.

**labels**

[list of str, optional] A list of labels to show next to the artists. Use this together with *handles*, if you need full control on what is shown in the legend and the automatic mechanism described above is not sufficient.

**Returns**

*Legend*

**Other Parameters**

**loc**

[str or pair of floats, default: *rcParams["legend.loc"]* (default: 'best')] The location of the legend.

The strings 'upper left', 'upper right', 'lower left', 'lower right' place the legend at the corresponding corner of the axes.

The strings 'upper center', 'lower center', 'center left', 'center right' place the legend at the center of the corresponding edge of the axes.

The string 'center' places the legend at the center of the axes.

The string 'best' places the legend at the location, among the nine locations defined so far, with the minimum overlap with other drawn artists. This option can be quite slow for plots with large amounts of data; your plotting speed may benefit from providing a specific location.

The location can also be a 2-tuple giving the coordinates of the lower-left corner of the legend in axes coordinates (in which case *bbox_to_anchor* will be ignored).

For back-compatibility, 'center right' (but no other location) can also be spelled 'right', and each "string" location can also be given as a numeric value:

| Location String | Location Code |
|---|---|
| 'best' (Axes only) | 0 |
| 'upper right' | 1 |
| 'upper left' | 2 |
| 'lower left' | 3 |
| 'lower right' | 4 |
| 'right' | 5 |
| 'center left' | 6 |
| 'center right' | 7 |
| 'lower center' | 8 |
| 'upper center' | 9 |
| 'center' | 10 |

**bbox_to_anchor**

[*BboxBase*, 2-tuple, or 4-tuple of floats] Box that is used to position the legend in conjunction with *loc*. Defaults to `axes.bbox` (if called as a method to *Axes.legend*) or `figure.bbox` (if *Figure.legend*). This argument allows arbitrary placement of the legend.

Bbox coordinates are interpreted in the coordinate system given by *bbox_transform*, with the default transform Axes or Figure coordinates, depending on which `legend` is called.

If a 4-tuple or *BboxBase* is given, then it specifies the bbox `(x, y, width, height)` that the legend is placed in. To put the legend in the best location in the bottom right quadrant of the Axes (or figure):

```
loc='best', bbox_to_anchor=(0.5, 0., 0.5, 0.5)
```

A 2-tuple `(x, y)` places the corner of the legend specified by *loc* at x, y. For example, to put the legend's upper right-hand corner in the center of the Axes (or figure) the following keywords can be used:

```
loc='upper right', bbox_to_anchor=(0.5, 0.5)
```

**ncols**

[int, default: 1] The number of columns that the legend has.

For backward compatibility, the spelling *ncol* is also supported but it is discouraged. If both are given, *ncols* takes precedence.

**prop**

[None or *FontProperties* or dict] The font properties of the legend. If None (default), the current *matplotlib.rcParams* will be used.

**fontsize**

[int or {'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large'}] The font size of the legend. If the value is numeric the size will be the absolute font size in points. String values are relative to the current default font size. This argument is only used if *prop* is not specified.

**labelcolor**

[str or list, default: *rcParams["legend.labelcolor"]* (default: 'None')] The color of the text in the legend. Either a valid color string (for example, 'red'), or a list of color strings. The labelcolor can also be made to match the color of the line or marker using 'linecolor', 'markerfacecolor' (or 'mfc'), or 'markeredgecolor' (or 'mec').

Labelcolor can be set globally using *rcParams["legend.labelcolor"]* (default: 'None'). If None, use *rcParams["text.color"]* (default: 'black').

**numpoints**

[int, default: *rcParams["legend.numpoints"]* (default: 1)] The number of marker points in the legend when creating a legend entry for a *Line2D* (line).

**scatterpoints**

[int, default: *rcParams["legend.scatterpoints"]* (default: 1)] The number of marker points in the legend when creating a legend entry for a *PathCollection* (scatter plot).

**scatteryoffsets**

[iterable of floats, default: [0.375, 0.5, 0.3125]] The vertical offset (relative to the font size) for the markers created for a scatter plot legend entry. 0.0 is at the base the legend text, and 1.0 is at the top. To draw all markers at the same height, set to [0.5].

**markerscale**

[float, default: *rcParams["legend.markerscale"]* (default: 1.0)] The relative size of legend markers compared to the originally drawn ones.

**markerfirst**

[bool, default: True] If *True*, legend marker is placed to the left of the legend label. If *False*, legend marker is placed to the right of the legend label.

**reverse**

[bool, default: False] If *True*, the legend labels are displayed in reverse order from the input. If *False*, the legend labels are displayed in the same order as the input.

New in version 3.7.

**frameon**

[bool, default: *rcParams["legend.frameon"]* (default: True)] Whether the legend should be drawn on a patch (frame).

**fancybox**

[bool, default: *rcParams["legend.fancybox"]* (default: `True`)] Whether round edges should be enabled around the *FancyBboxPatch* which makes up the legend's background.

**shadow**

[None, bool or dict, default: *rcParams["legend.shadow"]* (default: `False`)] Whether to draw a shadow behind the legend. The shadow can be configured using *Patch* keywords. Customization via *rcParams["legend.shadow"]* (default: `False`) is currently not supported.

**framealpha**

[float, default: *rcParams["legend.framealpha"]* (default: `0.8`)] The alpha transparency of the legend's background. If *shadow* is activated and *framealpha* is `None`, the default value is ignored.

**facecolor**

["inherit" or color, default: *rcParams["legend.facecolor"]* (default: `'inherit'`)] The legend's background color. If `"inherit"`, use *rcParams["axes.facecolor"]* (default: `'white'`).

**edgecolor**

["inherit" or color, default: *rcParams["legend.edgecolor"]* (default: `'0.8'`)] The legend's background patch edge color. If `"inherit"`, use *rcParams["axes.edgecolor"]* (default: `'black'`).

**mode**

[{"expand", None}] If *mode* is set to `"expand"` the legend will be horizontally expanded to fill the Axes area (or *bbox_to_anchor* if defines the legend's size).

**bbox_transform**

[None or *Transform*] The transform for the bounding box (*bbox_to_anchor*). For a value of `None` (default) the Axes' `transAxes` transform will be used.

**title**

[str or None] The legend's title. Default is no title (`None`).

**title_fontproperties**

[None or *FontProperties* or dict] The font properties of the legend's title. If None (default), the *title_fontsize* argument will be used if present; if *title_fontsize* is also None, the current *rcParams["legend.title_fontsize"]* (default: `None`) will be used.

**title_fontsize**

[int or {'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large'}, default: *rcParams["legend.title_fontsize"]* (default: `None`)] The font size of the legend's title. Note: This cannot be combined with *title_fontproperties*. If

you want to set the fontsize alongside other font properties, use the *size* parameter in *title_fontproperties*.

**alignment**

[{'center', 'left', 'right'}, default: 'center'] The alignment of the legend title and the box of entries. The entries are aligned as a single block, so that markers always lined up.

**borderpad**

[float, default: *rcParams["legend.borderpad"]* (default: 0.4)] The fractional whitespace inside the legend border, in font-size units.

**labelspacing**

[float, default: *rcParams["legend.labelspacing"]* (default: 0.5)] The vertical space between the legend entries, in font-size units.

**handlelength**

[float, default: *rcParams["legend.handlelength"]* (default: 2.0)] The length of the legend handles, in font-size units.

**handleheight**

[float, default: *rcParams["legend.handleheight"]* (default: 0.7)] The height of the legend handles, in font-size units.

**handletextpad**

[float, default: *rcParams["legend.handletextpad"]* (default: 0.8)] The pad between the legend handle and text, in font-size units.

**borderaxespad**

[float, default: *rcParams["legend.borderaxespad"]* (default: 0.5)] The pad between the Axes and legend border, in font-size units.

**columnspacing**

[float, default: *rcParams["legend.columnspacing"]* (default: 2.0)] The spacing between columns, in font-size units.

**handler_map**

[dict or None] The custom dictionary mapping instances or types to a legend handler. This *handler_map* updates the default handler map found at *matplotlib.legend.Legend.get_legend_handler_map*.

**draggable**

[bool, default: False] Whether the legend can be dragged with the mouse.

**See also:**

***Figure.legend***

### Notes

**Note:** This is the *pyplot wrapper* for `axes.Axes.legend`.

Some artists are not supported by this function. See *Legend guide* for details.

### Examples

## Examples using `matplotlib.pyplot.legend`

- *Errorbar limit selection*
- *Discrete distribution as horizontal bar chart*
- *Plotting masked and NaN values*
- *Scatter plots with a legend*
- *Stairs Demo*
- *Step Demo*
- *Contourf Hatching*
- *Contourf and log color scale*
- *Tricontour Demo*
- *Labeling a pie and a donut*
- *Polar legend*
- *Legend using pre-defined labels*
- *Infinite lines*
- *Findobj Demo*
- *Zorder Demo*
- *The Sankey class*
- *SVG Histogram*
- *Quick start guide*

### Vector fields

| | |
|---|---|
| *barbs* | Plot a 2D field of wind barbs. |
| *quiver* | Plot a 2D field of arrows. |
| *quiverkey* | Add a key to a quiver plot. |
| *streamplot* | Draw streamlines of a vector flow. |

## matplotlib.pyplot.barbs

matplotlib.pyplot.**barbs**(**args*, *data=None*, ***kwargs*)

Plot a 2D field of wind barbs.

Call signature:

```
barbs([X, Y], U, V, [C], **kwargs)
```

Where *X*, *Y* define the barb locations, *U*, *V* define the barb directions, and *C* optionally sets the color.

All arguments may be 1D or 2D. *U*, *V*, *C* may be masked arrays, but masked *X*, *Y* are not supported at present.

Barbs are traditionally used in meteorology as a way to plot the speed and direction of wind observations, but can technically be used to plot any two dimensional vector quantity. As opposed to arrows, which give vector magnitude by the length of the arrow, the barbs give more quantitative information about the vector magnitude by putting slanted lines or a triangle for various increments in magnitude, as show schematically below:

```
:                     /\     \
:                    /  \     \
:                   /    \     \     \
:                  /      \     \     \
:                 ------------------------------
```

The largest increment is given by a triangle (or "flag"). After those come full lines (barbs). The smallest increment is a half line. There is only, of course, ever at most 1 half line. If the magnitude is small and only needs a single half-line and no full lines or triangles, the half-line is offset from the end of the barb so that it can be easily distinguished from barbs with a single full line. The magnitude for the barb shown above would nominally be 65, using the standard increments of 50, 10, and 5.

See also https://en.wikipedia.org/wiki/Wind_barb.

**Parameters**

**X, Y**

[1D or 2D array-like, optional] The x and y coordinates of the barb locations. See *pivot* for how the barbs are drawn to the x, y positions.

If not given, they will be generated as a uniform integer meshgrid based on the dimensions of *U* and *V*.

If *X* and *Y* are 1D but *U*, *V* are 2D, *X*, *Y* are expanded to 2D using `X, Y = np.meshgrid(X, Y)`. In this case `len(X)` and `len(Y)` must match the column and row dimensions of *U* and *V*.

**U, V**

[1D or 2D array-like] The x and y components of the barb shaft.

**C**

[1D or 2D array-like, optional] Numeric data that defines the barb colors by colormapping via *norm* and *cmap*.

This does not support explicit colors. If you want to set colors directly, use *barbcolor* instead.

**length**

[float, default: 7] Length of the barb in points; the other parts of the barb are scaled against this.

**pivot**

[{'tip', 'middle'} or float, default: 'tip'] The part of the arrow that is anchored to the *X*, *Y* grid. The barb rotates about this point. This can also be a number, which shifts the start of the barb that many points away from grid point.

**barbcolor**

[*color* or color sequence] The color of all parts of the barb except for the flags. This parameter is analogous to the *edgecolor* parameter for polygons, which can be used instead. However this parameter will override facecolor.

**flagcolor**

[*color* or color sequence] The color of any flags on the barb. This parameter is analogous to the *facecolor* parameter for polygons, which can be used instead. However, this parameter will override facecolor. If this is not set (and *C* has not either) then *flagcolor* will be set to match *barbcolor* so that the barb has a uniform color. If *C* has been set, *flagcolor* has no effect.

**sizes**

[dict, optional] A dictionary of coefficients specifying the ratio of a given feature to the length of the barb. Only those values one wishes to override need to be included. These features include:

- 'spacing' - space between features (flags, full/half barbs)
- 'height' - height (distance from shaft to top) of a flag or full barb
- 'width' - width of a flag, twice the width of a full barb
- 'emptybarb' - radius of the circle used for low magnitudes

**fill_empty**

[bool, default: False] Whether the empty barbs (circles) that are drawn should be filled with the flag color. If they are not filled, the center is transparent.

**rounding**

[bool, default: True] Whether the vector magnitude should be rounded when allocating barb components. If True, the magnitude is rounded to the nearest multiple of the half-barb increment. If False, the magnitude is simply truncated to the next lowest multiple.

**barb_increments**

[dict, optional] A dictionary of increments specifying values to associate with different parts of the barb. Only those values one wishes to override need to be included.

- 'half' - half barbs (Default is 5)
- 'full' - full barbs (Default is 10)
- 'flag' - flags (default is 50)

**flip_barb**

[bool or array-like of bool, default: False] Whether the lines and flags should point opposite to normal. Normal behavior is for the barbs and lines to point right (comes from wind barbs having these features point towards low pressure in the Northern Hemisphere).

A single value is applied to all barbs. Individual barbs can be flipped by passing a bool array of the same size as *U* and *V*.

**Returns**

**barbs**

[*Barbs*]

**Other Parameters**

**data**

[indexable object, optional] If given, all parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception).

****kwargs**

The barbs can further be customized using *PolyCollection* keyword arguments:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a d |
| *alpha* | array-like or scalar or None |
| *animated* | bool |
| *antialiased* or aa or antialiaseds | bool or list of bools |
| *array* | array-like or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clim* | (vmin: float, vmax: float) |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *cmap* | *Colormap* or str or None |
| *color* | *color* or list of RGBA tuples |
| *edgecolor* or ec or edgecolors | *color* or list of *color* or 'face' |
| *facecolor* or facecolors or fc | *color* or list of *color* |
| *figure* | *Figure* |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or dashes or linestyles or ls | str or tuple or list thereof |
| *linewidth* or linewidths or lw | float or list of floats |
| *mouseover* | bool |
| *norm* | *Normalize* or str or None |
| *offset_transform* or transOffset | *Transform* |
| *offsets* | (N, 2) or (2,) array-like |
| *path_effects* | list of *AbstractPathEffect* |
| *paths* | list of array-like |
| *picker* | None or bool or float or callable |
| *pickradius* | float |
| *rasterized* | bool |
| sizes | numpy.ndarray or None |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *urls* | list of str or None |
| *verts* | list of array-like |
| *verts_and_codes* | unknown |
| *visible* | bool |
| *zorder* | float |

### Notes

**Note:** This is the *pyplot wrapper* for `axes.Axes.barbs`.

## Examples using `matplotlib.pyplot.barbs`

- *Wind Barbs*

## matplotlib.pyplot.quiver

matplotlib.pyplot.**quiver**(**args*, *data=None*, ***kwargs*)

Plot a 2D field of arrows.

Call signature:

```
quiver([X, Y], U, V, [C], **kwargs)
```

*X*, *Y* define the arrow locations, *U*, *V* define the arrow directions, and *C* optionally sets the color.

**Arrow length**

The default settings auto-scales the length of the arrows to a reasonable size. To change this behavior see the *scale* and *scale_units* parameters.

**Arrow shape**

The arrow shape is determined by *width*, *headwidth*, *headlength* and *headaxislength*. See the notes below.

**Arrow styling**

Each arrow is internally represented by a filled polygon with a default edge linewidth of 0. As a result, an arrow is rather a filled area, not a line with a head, and `PolyCollection` properties like *linewidth*, *edgecolor*, *facecolor*, etc. act accordingly.

**Parameters**

**X, Y**

[1D or 2D array-like, optional] The x and y coordinates of the arrow locations.

If not given, they will be generated as a uniform integer meshgrid based on the dimensions of *U* and *V*.

If *X* and *Y* are 1D but *U*, *V* are 2D, *X*, *Y* are expanded to 2D using `X, Y = np.meshgrid(X, Y)`. In this case `len(X)` and `len(Y)` must match the column and row dimensions of *U* and *V*.

**U, V**

[1D or 2D array-like] The x and y direction components of the arrow vectors. The interpretation of these components (in data or in screen space) depends on *angles*.

*U* and *V* must have the same number of elements, matching the number of arrow locations in *X*, *Y*. *U* and *V* may be masked. Locations masked in any of *U*, *V*, and *C* will not be drawn.

**C**

[1D or 2D array-like, optional] Numeric data that defines the arrow colors by colormapping via *norm* and *cmap*.

This does not support explicit colors. If you want to set colors directly, use *color* instead. The size of *C* must match the number of arrow locations.

**angles**

[{'uv', 'xy'} or array-like, default: 'uv'] Method for determining the angle of the arrows.

- 'uv': Arrow direction in screen coordinates. Use this if the arrows symbolize a quantity that is not based on *X*, *Y* data coordinates.

  If $U == V$ the orientation of the arrow on the plot is 45 degrees counter-clockwise from the horizontal axis (positive to the right).

- 'xy': Arrow direction in data coordinates, i.e. the arrows point from (x, y) to (x+u, y+v). Use this e.g. for plotting a gradient field.

- Arbitrary angles may be specified explicitly as an array of values in degrees, counter-clockwise from the horizontal axis.

  In this case *U*, *V* is only used to determine the length of the arrows.

Note: inverting a data axis will correspondingly invert the arrows only with `angles='xy'`.

**pivot**

[{'tail', 'mid', 'middle', 'tip'}, default: 'tail'] The part of the arrow that is anchored to the *X*, *Y* grid. The arrow rotates about this point.

'mid' is a synonym for 'middle'.

**scale**

[float, optional] Scales the length of the arrow inversely.

Number of data units per arrow length unit, e.g., m/s per plot width; a smaller scale parameter makes the arrow longer. Default is *None*.

If *None*, a simple autoscaling algorithm is used, based on the average vector length and the number of vectors. The arrow length unit is given by the *scale_units* parameter.

**scale_units**

[{'width', 'height', 'dots', 'inches', 'x', 'y', 'xy'}, optional] If the *scale* kwarg is *None*, the arrow length unit. Default is *None*.

e.g. *scale_units* is 'inches', *scale* is 2.0, and `(u, v) = (1, 0)`, then the vector will be 0.5 inches long.

If *scale_units* is 'width' or 'height', then the vector will be half the width/height of the axes.

If *scale_units* is 'x' then the vector will be 0.5 x-axis units. To plot vectors in the x-y plane, with u and v having the same units as x and y, use `angles='xy', scale_units='xy', scale=1`.

**units**

[{'width', 'height', 'dots', 'inches', 'x', 'y', 'xy'}, default: 'width'] Affects the arrow size (except for the length). In particular, the shaft *width* is measured in multiples of this unit.

Supported values are:

- 'width', 'height': The width or height of the Axes.
- 'dots', 'inches': Pixels or inches based on the figure dpi.
- 'x', 'y', 'xy': $X$, $Y$ or $\sqrt{X^2 + Y^2}$ in data units.

The following table summarizes how these values affect the visible arrow size under zooming and figure size changes:

| units | zoom | figure size change |
|---|---|---|
| 'x', 'y', 'xy' | arrow size scales | — |
| 'width', 'height' | — | arrow size scales |
| 'dots', 'inches' | — | — |

**width**

[float, optional] Shaft width in arrow units. All head parameters are relative to *width*.

The default depends on choice of *units* above, and number of vectors; a typical starting value is about 0.005 times the width of the plot.

**headwidth**

[float, default: 3] Head width as multiple of shaft *width*. See the notes below.

**headlength**

[float, default: 5] Head length as multiple of shaft *width*. See the notes below.

**headaxislength**

[float, default: 4.5] Head length at shaft intersection as multiple of shaft *width*. See the notes below.

**minshaft**

[float, default: 1] Length below which arrow scales, in units of head length. Do not set this to less than 1, or small arrows will look terrible!

**minlength**

[float, default: 1] Minimum length as a multiple of shaft width; if an arrow length is less than this, plot a dot (hexagon) of this diameter instead.

**color**

[*color* or list *color*, optional] Explicit color(s) for the arrows. If *C* has been set, *color* has no effect.

This is a synonym for the `PolyCollection` *facecolor* parameter.

**Returns**

***Quiver***

**Other Parameters**

**data**

[indexable object, optional] If given, all parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception).

****kwargs**

[`PolyCollection` properties, optional] All other keyword arguments are passed on to `PolyCollection`:

| Property | Description |
|---|---|
| `agg_filter` | a filter function, which takes a (m, n, 3) float array and a d |
| `alpha` | array-like or scalar or None |
| `animated` | bool |
| `antialiased` or aa or antialiaseds | bool or list of bools |
| `array` | array-like or None |
| `capstyle` | `CapStyle` or {'butt', 'projecting', 'round'} |
| `clim` | (vmin: float, vmax: float) |
| `clip_box` | `BboxBase` or None |
| `clip_on` | bool |
| `clip_path` | Patch or (Path, Transform) or None |
| `cmap` | `Colormap` or str or None |
| `color` | *color* or list of RGBA tuples |
| `edgecolor` or ec or edgecolors | *color* or list of *color* or 'face' |
| `facecolor` or facecolors or fc | *color* or list of *color* |

Table 133 – continued fr

| Property | Description |
|---|---|
| figure | Figure |
| gid | str |
| hatch | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| in_layout | bool |
| joinstyle | JoinStyle or {'miter', 'round', 'bevel'} |
| label | object |
| linestyle or dashes or linestyles or ls | str or tuple or list thereof |
| linewidth or linewidths or lw | float or list of floats |
| mouseover | bool |
| norm | Normalize or str or None |
| offset_transform or transOffset | Transform |
| offsets | (N, 2) or (2,) array-like |
| path_effects | list of AbstractPathEffect |
| paths | list of array-like |
| picker | None or bool or float or callable |
| pickradius | float |
| rasterized | bool |
| sizes | numpy.ndarray or None |
| sketch_params | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| transform | Transform |
| url | str |
| urls | list of str or None |
| verts | list of array-like |
| verts_and_codes | unknown |
| visible | bool |
| zorder | float |

**See also:**

***Axes.quiverkey***

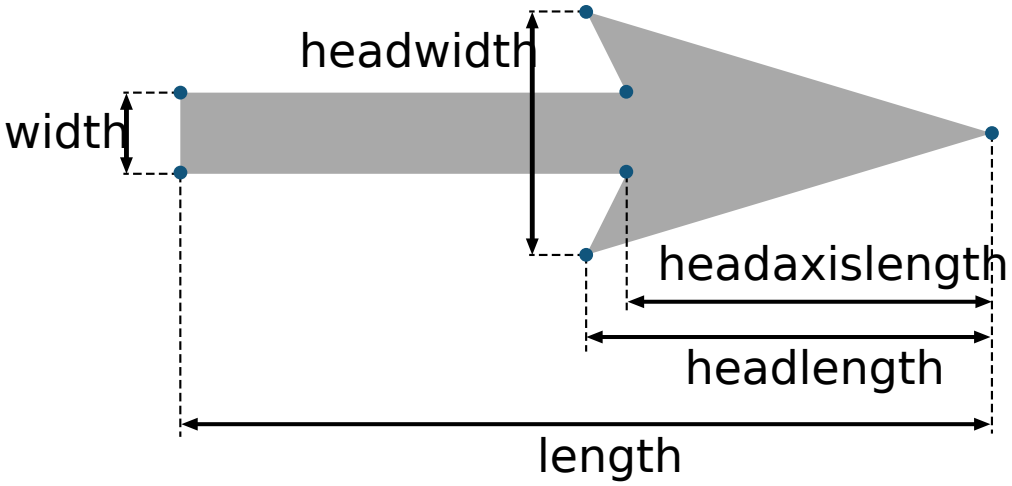
Add a key to a quiver plot.

## Notes

**Note:** This is the *pyplot wrapper* for axes.Axes.quiver.

### Arrow shape

The arrow is drawn as a polygon using the nodes as shown below. The values *headwidth*, *headlength*, and *headaxislength* are in units of *width*.




The defaults give a slightly swept-back arrow. Here are some guidelines how to get other head shapes:

- To make the head a triangle, make *headaxislength* the same as *headlength*.
- To make the arrow more pointed, reduce *headwidth* or increase *headlength* and *headaxislength*.
- To make the head smaller relative to the shaft, scale down all the head parameters proportionally.
- To remove the head completely, set all *head* parameters to 0.
- To get a diamond-shaped head, make *headaxislength* larger than *headlength*.
- Warning: For *headaxislength* < (*headlength* / *headwidth*), the "headaxis" nodes (i.e. the ones connecting the head with the shaft) will protrude out of the head in forward direction so that the arrow head looks broken.

### Examples using `matplotlib.pyplot.quiver`

- *Advanced quiver and quiverkey functions*
- *Quiver Simple Demo*
- *Trigradient Demo*

### matplotlib.pyplot.quiverkey

`matplotlib.pyplot.`**`quiverkey`**(*Q*, *X*, *Y*, *U*, *label*, ***kwargs*)

Add a key to a quiver plot.

The positioning of the key depends on *X*, *Y*, *coordinates*, and *labelpos*. If *labelpos* is 'N' or 'S', *X*, *Y* give the position of the middle of the key arrow. If *labelpos* is 'E', *X*, *Y* positions the head, and if *labelpos* is 'W', *X*, *Y* positions the tail; in either of these two cases, *X*, *Y* is somewhere in the middle of the arrow+label key object.

**Parameters**

**Q**

[*Quiver*] A *Quiver* object as returned by a call to *quiver()*.

**X, Y**

[float] The location of the key.

**U**

[float] The length of the key.

**label**

[str] The key label (e.g., length and units of the key).

**angle**

[float, default: 0] The angle of the key arrow, in degrees anti-clockwise from the horizontal axis.

**coordinates**

[{'axes', 'figure', 'data', 'inches'}, default: 'axes'] Coordinate system and units for *X*, *Y*: 'axes' and 'figure' are normalized coordinate systems with (0, 0) in the lower left and (1, 1) in the upper right; 'data' are the axes data coordinates (used for the locations of the vectors in the quiver plot itself); 'inches' is position in the figure in inches, with (0, 0) at the lower left corner.

**color**

[*color*] Overrides face and edge colors from *Q*.

**labelpos**

[{'N', 'S', 'E', 'W'}] Position the label above, below, to the right, to the left of the arrow, respectively.

**labelsep**

[float, default: 0.1] Distance in inches between the arrow and the label.

**labelcolor**

[*color*, default: *rcParams["text.color"]* (default: `'black'`)] Label color.

**fontproperties**

[dict, optional] A dictionary with keyword arguments accepted by the *FontProperties* initializer: *family*, *style*, *variant*, *size*, *weight*.

****kwargs**

Any additional keyword arguments are used to override vector properties taken from *Q*.

### Notes

**Note:** This is the *pyplot wrapper* for `axes.Axes.quiverkey`.

## Examples using `matplotlib.pyplot.quiverkey`

- *Advanced quiver and quiverkey functions*
- *Quiver Simple Demo*

## matplotlib.pyplot.streamplot

matplotlib.pyplot.**streamplot**(*x*, *y*, *u*, *v*, *density=1*, *linewidth=None*, *color=None*, *cmap=None*, *norm=None*, *arrowsize=1*, *arrowstyle='-|>'*, *minlength=0.1*, *transform=None*, *zorder=None*, *start_points=None*, *maxlength=4.0*, *integration_direction='both'*, *broken_streamlines=True*, ***, *data=None*)

Draw streamlines of a vector flow.

**Parameters**

**x, y**

[1D/2D arrays] Evenly spaced strictly increasing arrays to make a grid. If 2D, all rows of *x* must be equal and all columns of *y* must be equal; i.e., they must be as if generated by `np.meshgrid(x_1d, y_1d)`.

**u, v**

[2D arrays] *x* and *y*-velocities. The number of rows and columns must match the length of *y* and *x*, respectively.

**density**

[float or (float, float)] Controls the closeness of streamlines. When `density = 1`, the domain is divided into a 30x30 grid. *density* linearly scales this grid. Each cell in the grid can have, at most, one traversing streamline. For different densities in each direction, use a tuple (density_x, density_y).

**linewidth**

[float or 2D array] The width of the streamlines. With a 2D array the line width can be varied across the grid. The array must have the same shape as *u* and *v*.

**color**

[*color* or 2D array] The streamline color. If given an array, its values are converted to colors using *cmap* and *norm*. The array must have the same shape as *u* and *v*.

**cmap, norm**

Data normalization and colormapping parameters for *color*; only used if *color* is an array of floats. See *imshow* for a detailed description.

**arrowsize**

[float] Scaling factor for the arrow size.

**arrowstyle**

[str] Arrow style specification. See *FancyArrowPatch*.

**minlength**

[float] Minimum length of streamline in axes coordinates.

**start_points**

[(N, 2) array] Coordinates of starting points for the streamlines in data coordinates (the same coordinates as the *x* and *y* arrays).

**zorder**

[float] The zorder of the streamlines and arrows. Artists with lower zorder values are drawn first.

**maxlength**

[float] Maximum length of streamline in axes coordinates.

**integration_direction**

[{'forward', 'backward', 'both'}, default: 'both'] Integrate the streamline in forward, backward or both directions.

**data**

[indexable object, optional] If given, the following parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception):

*x*, *y*, *u*, *v*, *start_points*

**broken_streamlines**

[boolean, default: True] If False, forces streamlines to continue until they leave the plot domain. If True, they may be terminated if they come too close to another streamline.

**Returns**

**StreamplotSet**

Container object with attributes

- `lines`: *LineCollection* of streamlines
- `arrows`: *PatchCollection* containing *FancyArrowPatch* objects representing the arrows half-way along streamlines.

This container will probably change in the future to allow changes to the colormap, alpha, etc. for both lines and arrows, but these changes should be backward compatible.

### Notes

**Note:** This is the *pyplot wrapper* for `axes.Axes.streamplot`.

### Examples using `matplotlib.pyplot.streamplot`

- *Streamplot*

### Axis configuration

| | |
|---|---|
| `autoscale` | Autoscale the axis view to the data (toggle). |
| `axis` | Convenience method to get or set some axis properties. |
| `box` | Turn the Axes box on or off on the current Axes. |
| `grid` | Configure the grid lines. |
| `locator_params` | Control behavior of major tick locators. |
| `minorticks_off` | Remove minor ticks from the Axes. |
| `minorticks_on` | Display minor ticks on the Axes. |
| `rgrids` | Get or set the radial gridlines on the current polar plot. |
| `thetagrids` | Get or set the theta gridlines on the current polar plot. |
| `tick_params` | Change the appearance of ticks, tick labels, and gridlines. |
| `ticklabel_format` | Configure the `ScalarFormatter` used by default for linear Axes. |
| `xlabel` | Set the label for the x-axis. |
| `xlim` | Get or set the x limits of the current Axes. |
| `xscale` | Set the xaxis' scale. |
| `xticks` | Get or set the current tick locations and labels of the x-axis. |
| `ylabel` | Set the label for the y-axis. |
| `ylim` | Get or set the y-limits of the current Axes. |
| `yscale` | Set the yaxis' scale. |
| `yticks` | Get or set the current tick locations and labels of the y-axis. |
| `suptitle` | Add a centered suptitle to the figure. |
| `title` | Set a title for the Axes. |

### matplotlib.pyplot.autoscale

matplotlib.pyplot.**autoscale**(*enable=True*, *axis='both'*, *tight=None*)

Autoscale the axis view to the data (toggle).

Convenience method for simple axis view autoscaling. It turns autoscaling on or off, and then, if autoscaling for either axis is on, it performs the autoscaling on the specified axis or Axes.

**Parameters**

**enable**

[bool or None, default: True] True turns autoscaling on, False turns it off. None leaves the autoscaling state unchanged.

**axis**

[{'both', 'x', 'y'}, default: 'both'] The axis on which to operate. (For 3D Axes, *axis* can also be set to 'z', and 'both' refers to all three Axes.)

**tight**

[bool or None, default: None] If True, first set the margins to zero. Then, this argument is forwarded to `autoscale_view` (regardless of its value); see the description of its behavior there.

#### Notes

**Note:** This is the *pyplot wrapper* for `axes.Axes.autoscale`.

### matplotlib.pyplot.axis

matplotlib.pyplot.**axis**(*arg=None*, */*, ***, *emit=True*, ***kwargs*)

Convenience method to get or set some axis properties.

Call signatures:

```
xmin, xmax, ymin, ymax = axis()
xmin, xmax, ymin, ymax = axis([xmin, xmax, ymin, ymax])
xmin, xmax, ymin, ymax = axis(option)
xmin, xmax, ymin, ymax = axis(**kwargs)
```

**Parameters**

**xmin, xmax, ymin, ymax**

[float, optional] The axis limits to be set. This can also be achieved using

```
ax.set(xlim=(xmin, xmax), ylim=(ymin, ymax))
```

**option**

[bool or str] If a bool, turns axis lines and labels on or off. If a string, possible values are:

| Value | Description |
|---|---|
| 'off' or Fals | Hide all axis decorations, i.e. axis labels, spines, tick marks, tick labels, and grid lines. This is the same as *set_axis_off()*. |
| 'on' or True | Do not hide all axis decorations, i.e. axis labels, spines, tick marks, tick labels, and grid lines. This is the same as *set_axis_on()*. |
| 'equal | Set equal scaling (i.e., make circles circular) by changing the axis limits. This is the same as `ax.set_aspect('equal', adjustable='datalim')`. Explicit data limits may not be respected in this case. |
| 'scale | Set equal scaling (i.e., make circles circular) by changing dimensions of the plot box. This is the same as `ax.set_aspect('equal', adjustable='box', anchor='C')`. Additionally, further autoscaling will be disabled. |
| 'tight' | Set limits just large enough to show all data, then disable further autoscaling. |
| 'auto' | Automatic scaling (fill plot box with data). |
| 'image' | 'scaled' with axis limits equal to data limits. |
| 'squar | Square plot; similar to 'scaled', but initially forcing `xmax-xmin == ymax-ymin`. |

**emit**

[bool, default: True] Whether observers are notified of the axis limit change. This option is passed on to *set_xlim* and *set_ylim*.

**Returns**

**xmin, xmax, ymin, ymax**

[float] The axis limits.

**See also:**

***matplotlib.axes.Axes.set_xlim***
***matplotlib.axes.Axes.set_ylim***

### Notes

**Note:** This is the *pyplot wrapper* for `axes.Axes.axis`.

For 3D Axes, this method additionally takes *zmin*, *zmax* as parameters and likewise returns them.

### Examples using `matplotlib.pyplot.axis`

- *Filled polygon*
- *Auto-wrapping text*
- *Pyplot tutorial*

## matplotlib.pyplot.box

matplotlib.pyplot.**box**(*on=None*)

Turn the Axes box on or off on the current Axes.

**Parameters**

**on**

[bool or None] The new `Axes` box state. If `None`, toggle the state.

**See also:**

***matplotlib.axes.Axes.set_frame_on()***
***matplotlib.axes.Axes.get_frame_on()***

## matplotlib.pyplot.grid

matplotlib.pyplot.**grid**(*visible=None*, *which='major'*, *axis='both'*, ***kwargs*)

Configure the grid lines.

**Parameters**

**visible**

[bool or None, optional] Whether to show the grid lines. If any *kwargs* are supplied, it is assumed you want the grid on and *visible* will be set to True.

If *visible* is *None* and there are no *kwargs*, this toggles the visibility of the lines.

**which**

[{'major', 'minor', 'both'}, optional] The grid lines to apply the changes on.

**axis**

[{'both', 'x', 'y'}, optional] The axis to apply the changes on.

****kwargs**

[*Line2D* properties] Define the line properties of the grid, e.g.:

```
grid(color='r', linestyle='-', linewidth=2)
```

Valid keyword arguments are:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi va |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* or aa | bool |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* or c | *color* |
| *dash_capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *dash_joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *dashes* | sequence of floats (on/off ink in points) or (None, None) |
| *data* | (2, N) array or two 1D arrays |
| *drawstyle* or ds | {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: |
| *figure* | *Figure* |
| *fillstyle* | {'full', 'left', 'right', 'bottom', 'top', 'none'} |
| *gapcolor* | *color* or None |
| *gid* | str |
| *in_layout* | bool |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float |
| *marker* | marker style string, *Path* or *MarkerStyle* |
| *markeredgecolor* or mec | *color* |
| *markeredgewidth* or mew | float |
| *markerfacecolor* or mfc | *color* |
| *markerfacecoloralt* or mfcalt | *color* |
| *markersize* or ms | float |
| *markevery* | None or int or (int, int) or slice or list[int] or float or (float, floa |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | float or callable[[Artist, Event], tuple[bool, dict]] |
| *pickradius* | float |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |

Table 134 – continued from

| Property | Description |
|---|---|
| *solid_capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *solid_joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *transform* | unknown |
| *url* | str |
| *visible* | bool |
| *xdata* | 1D array |
| *ydata* | 1D array |
| *zorder* | float |

### Notes

**Note:** This is the *pyplot wrapper* for *axes.Axes.grid*.

The axis is drawn as a unit, so the effective zorder for drawing the grid is determined by the zorder of each axis, not by the zorder of the *Line2D* objects comprising the grid. Therefore, to set grid zorder, use *set_axisbelow* or, for more control, call the *set_zorder* method of each axis.

### Examples using `matplotlib.pyplot.grid`

- *Step Demo*
- *Geographic Projections*
- *Text and mathtext using pyplot*
- *Customize Rc*
- *Findobj Demo*
- *Custom scale*
- *SkewT-logP diagram: using transforms and custom projections*
- *Pyplot tutorial*

## matplotlib.pyplot.locator_params

matplotlib.pyplot.**locator_params**(*axis='both'*, *tight=None*, ***kwargs*)

Control behavior of major tick locators.

Because the locator is involved in autoscaling, *autoscale_view* is called automatically after the parameters are changed.

**Parameters**

**axis**

[{'both', 'x', 'y'}, default: 'both'] The axis on which to operate. (For 3D Axes, *axis* can also be set to 'z', and 'both' refers to all three axes.)

**tight**

[bool or None, optional] Parameter passed to *autoscale_view*. Default is None, for no change.

**Other Parameters**

****kwargs**

Remaining keyword arguments are passed to directly to the `set_params()` method of the locator. Supported keywords depend on the type of the locator. See for example *set_params* for the *ticker.MaxNLocator* used by default for linear.

### Notes

**Note:** This is the *pyplot wrapper* for *axes.Axes.locator_params*.

### Examples

When plotting small subplots, one might want to reduce the maximum number of ticks and use tight bounds, for example:

```
ax.locator_params(tight=True, nbins=4)
```

## matplotlib.pyplot.minorticks_off

matplotlib.pyplot.**minorticks_off**()

Remove minor ticks from the Axes.

### Notes

**Note:** This is the *pyplot wrapper* for *axes.Axes.minorticks_off*.

### matplotlib.pyplot.minorticks_on

matplotlib.pyplot.**minorticks_on**()

Display minor ticks on the Axes.

Displaying minor ticks may reduce performance; you may turn them off using *minorticks_off()* if drawing speed is a problem.

#### Notes

**Note:** This is the *pyplot wrapper* for *axes.Axes.minorticks_on*.

### matplotlib.pyplot.rgrids

matplotlib.pyplot.**rgrids**(*radii=None*, *labels=None*, *angle=None*, *fmt=None*, ***kwargs*)

Get or set the radial gridlines on the current polar plot.

Call signatures:

```
lines, labels = rgrids()
lines, labels = rgrids(radii, labels=None, angle=22.5, fmt=None,␣
 ↪**kwargs)
```

When called with no arguments, *rgrids* simply returns the tuple (*lines*, *labels*). When called with arguments, the labels will appear at the specified radial distances and angle.

**Parameters**

**radii**

[tuple with floats] The radii for the radial gridlines

**labels**

[tuple with strings or None] The labels to use at each radial gridline. The *matplotlib.ticker.ScalarFormatter* will be used if None.

**angle**

[float] The angular position of the radius labels in degrees.

**fmt**

[str or None] Format string used in *matplotlib.ticker.FormatStrFormatter*. For example '%f'.

**Returns**

**lines**

[list of *lines.Line2D*] The radial gridlines.

**labels**

[list of *text.Text*] The tick labels.

**Other Parameters**

****kwargs**

*kwargs* are optional *Text* properties for the labels.

**See also:**

***pyplot.thetagrids***
***projections.polar.PolarAxes.set_rgrids***
***Axis.get_gridlines***
***Axis.get_ticklabels***

### Examples

```
# set the locations of the radial gridlines
lines, labels = rgrids( (0.25, 0.5, 1.0) )

# set the locations and labels of the radial gridlines
lines, labels = rgrids( (0.25, 0.5, 1.0), ('Tom', 'Dick', 'Harry' ))
```

## matplotlib.pyplot.thetagrids

matplotlib.pyplot.**thetagrids**(*angles=None*, *labels=None*, *fmt=None*, ***kwargs*)

Get or set the theta gridlines on the current polar plot.

Call signatures:

```
lines, labels = thetagrids()
lines, labels = thetagrids(angles, labels=None, fmt=None, **kwargs)
```

When called with no arguments, *thetagrids* simply returns the tuple (*lines*, *labels*). When called with arguments, the labels will appear at the specified angles.

**Parameters**

**angles**

[tuple with floats, degrees] The angles of the theta gridlines.

**labels**

[tuple with strings or None] The labels to use at each radial gridline. The *projections.polar.ThetaFormatter* will be used if None.

**fmt**

[str or None] Format string used in *matplotlib.ticker.FormatStrFormatter*. For example '%f'. Note that the angle in radians will be used.

**Returns**

**lines**

[list of *lines.Line2D*] The theta gridlines.

**labels**

[list of *text.Text*] The tick labels.

**Other Parameters**

****kwargs**

*kwargs* are optional *Text* properties for the labels.

**See also:**

***pyplot.rgrids***
***projections.polar.PolarAxes.set_thetagrids***
***Axis.get_gridlines***
***Axis.get_ticklabels***

## Examples

```
# set the locations of the angular gridlines
lines, labels = thetagrids(range(45, 360, 90))

# set the locations and labels of the angular gridlines
lines, labels = thetagrids(range(45, 360, 90), ('NE', 'NW', 'SW', 'SE'))
```

### matplotlib.pyplot.tick_params

matplotlib.pyplot.**tick_params**(*axis='both'*, ***kwargs*)

Change the appearance of ticks, tick labels, and gridlines.

Tick properties that are not explicitly set using the keyword arguments remain unchanged unless *reset* is True. For the current style settings, see *Axis.get_tick_params*.

**Parameters**

**axis**

[{'x', 'y', 'both'}, default: 'both'] The axis to which the parameters are applied.

**which**

[{'major', 'minor', 'both'}, default: 'major'] The group of ticks to which the parameters are applied.

**reset**

[bool, default: False] Whether to reset the ticks to defaults before updating them.

**Other Parameters**

**direction**

[{'in', 'out', 'inout'}] Puts ticks inside the Axes, outside the Axes, or both.

**length**

[float] Tick length in points.

**width**

[float] Tick width in points.

**color**

[*color*] Tick color.

**pad**

[float] Distance in points between tick and label.

**labelsize**

[float or str] Tick label font size in points or as a string (e.g., 'large').

**labelcolor**

[*color*] Tick label color.

**labelfontfamily**

[str] Tick label font.

**colors**

[*color*] Tick color and label color.

**zorder**

[float] Tick and label zorder.

**bottom, top, left, right**

[bool] Whether to draw the respective ticks.

**labelbottom, labeltop, labelleft, labelright**

[bool] Whether to draw the respective tick labels.

**labelrotation**

[float] Tick label rotation

**grid_color**

[*color*] Gridline color.

**grid_alpha**

[float] Transparency of gridlines: 0 (transparent) to 1 (opaque).

**grid_linewidth**

[float] Width of gridlines in points.

**grid_linestyle**

[str] Any valid *Line2D* line style spec.

## Notes

**Note:** This is the *pyplot wrapper* for *axes.Axes.tick_params*.

## Examples

```
ax.tick_params(direction='out', length=6, width=2, colors='r',
               grid_color='r', grid_alpha=0.5)
```

This will make all major ticks be red, pointing out of the box, and with dimensions 6 points by 2 points. Tick labels will also be red. Gridlines will be red and translucent.

### Examples using `matplotlib.pyplot.tick_params`

- *Shared axis*
- *Plots with different scales*

## matplotlib.pyplot.ticklabel_format

matplotlib.pyplot.**ticklabel_format**(*, *axis='both'*, *style=None*, *scilimits=None*, *useOffset=None*, *useLocale=None*, *useMathText=None*)

Configure the *ScalarFormatter* used by default for linear Axes.

If a parameter is not set, the corresponding property of the formatter is left unchanged.

**Parameters**

**axis**

[{'x', 'y', 'both'}, default: 'both'] The axis to configure. Only major ticks are affected.

**style**

[{'sci', 'scientific', 'plain'}] Whether to use scientific notation. The formatter default is to use scientific notation. 'sci' is equivalent to 'scientific'.

**scilimits**

[pair of ints (m, n)] Scientific notation is used only for numbers outside the range $10^m$ to $10^n$ (and only if the formatter is configured to use scientific notation at all). Use (0, 0) to include all numbers. Use (m, m) where m != 0 to fix the order of magnitude to $10^m$. The formatter default is *rcParams["axes.formatter.limits"]* (default: `[-5, 6]`).

**useOffset**

[bool or float] If True, the offset is calculated as needed. If False, no offset is used. If a numeric value, it sets the offset. The formatter default is *rcParams["axes.formatter.useoffset"]* (default: `True`).

**useLocale**

[bool] Whether to format the number using the current locale or using the C (English) locale. This affects e.g. the decimal separator. The formatter default is *rcParams["axes.formatter.use_locale"]* (default: `False`).

**useMathText**

[bool] Render the offset and scientific notation in mathtext. The formatter default is *rcParams["axes.formatter.use_mathtext"]* (default: `False`).

**Raises**

**AttributeError**

If the current formatter is not a *ScalarFormatter*.

### Notes

**Note:** This is the *pyplot wrapper* for *axes.Axes.ticklabel_format*.

## matplotlib.pyplot.xlabel

matplotlib.pyplot.**xlabel**(*xlabel*, *fontdict=None*, *labelpad=None*, *, *loc=None*, ***kwargs*)

Set the label for the x-axis.

**Parameters**

**xlabel**

[str] The label text.

**labelpad**

[float, default: *rcParams["axes.labelpad"]* (default: 4.0)] Spacing in points from the Axes bounding box including ticks and tick labels. If None, the previous value is left as is.

**loc**

[{'left', 'center', 'right'}, default: *rcParams["xaxis.labellocation"]* (default: 'center')] The label position. This is a high-level alternative for passing parameters *x* and *horizontalalignment*.

**Other Parameters**

****kwargs**

[*Text* properties] *Text* properties control the appearance of the label.

**See also:**

***text***

Documents the properties supported by *Text*.

### Notes

**Note:** This is the *pyplot wrapper* for `axes.Axes.set_xlabel`.

## Examples using `matplotlib.pyplot.xlabel`

- *Multiple subplots*
- *Controlling style of text and labels using a dictionary*
- *Solarized Light stylesheet*
- *Infinite lines*
- *Text and mathtext using pyplot*
- *Findobj Demo*
- *Custom scale*
- *Pyplot tutorial*
- *Quick start guide*

## matplotlib.pyplot.xlim

matplotlib.pyplot.**xlim**(**args*, ***kwargs*)

Get or set the x limits of the current Axes.

Call signatures:

```
left, right = xlim()   # return the current xlim
xlim((left, right))    # set the xlim to left, right
xlim(left, right)      # set the xlim to left, right
```

If you do not specify args, you can pass *left* or *right* as kwargs, i.e.:

```
xlim(right=3)  # adjust the right leaving left unchanged
xlim(left=1)  # adjust the left leaving right unchanged
```

Setting limits turns autoscaling off for the x-axis.

**Returns**

**left, right**

A tuple of the new x-axis limits.

### Notes

Calling this function with no arguments (e.g. `xlim()`) is the pyplot equivalent of calling *get_xlim* on the current Axes. Calling this function with arguments is the pyplot equivalent of calling *set_xlim* on the current Axes. All arguments are passed though.

## Examples using `matplotlib.pyplot.xlim`

- *Shared axis*
- *Infinite lines*
- *Frame grabbing*
- *Interactive functions*

## matplotlib.pyplot.xscale

matplotlib.pyplot.**xscale**(*value*, ***kwargs*)

Set the xaxis' scale.

**Parameters**

**value**

[{"linear", "log", "symlog", "logit", ...} or *ScaleBase*] The axis scale type to apply.

****kwargs**

Different keyword arguments are accepted, depending on the scale. See the respective class keyword arguments:

- *matplotlib.scale.LinearScale*
- *matplotlib.scale.LogScale*
- *matplotlib.scale.SymmetricalLogScale*
- *matplotlib.scale.LogitScale*
- *matplotlib.scale.FuncScale*
- *matplotlib.scale.AsinhScale*

### Notes

**Note:** This is the *pyplot wrapper* for `axes.Axes.set_xscale`.

By default, Matplotlib supports the above-mentioned scales. Additionally, custom scales may be registered using `matplotlib.scale.register_scale`. These scales can then also be used here.

## matplotlib.pyplot.xticks

`matplotlib.pyplot.`**`xticks`**(*ticks=None*, *labels=None*, *, *minor=False*, ***kwargs*)

Get or set the current tick locations and labels of the x-axis.

Pass no arguments to return the current values without modifying them.

**Parameters**

**ticks**

[array-like, optional] The list of xtick locations. Passing an empty list removes all xticks.

**labels**

[array-like, optional] The labels to place at the given *ticks* locations. This argument can only be passed if *ticks* is passed as well.

**minor**

[bool, default: False] If `False`, get/set the major ticks/labels; if `True`, the minor ticks/labels.

****kwargs**

`Text` properties can be used to control the appearance of the labels.

**Returns**

**locs**

The list of xtick locations.

**labels**

The list of xlabel `Text` objects.

### Notes

Calling this function with no arguments (e.g. `xticks()`) is the pyplot equivalent of calling *get_xticks* and *get_xticklabels* on the current Axes. Calling this function with arguments is the pyplot equivalent of calling *set_xticks* and *set_xticklabels* on the current Axes.

### Examples

```
>>> locs, labels = xticks()  # Get the current locations and labels.
>>> xticks(np.arange(0, 1, step=0.2))  # Set label locations.
>>> xticks(np.arange(3), ['Tom', 'Dick', 'Sue'])  # Set text labels.
>>> xticks([0, 1, 2], ['January', 'February', 'March'],
...        rotation=20)  # Set text labels and properties.
>>> xticks([])  # Disable xticks.
```

## Examples using `matplotlib.pyplot.xticks`

- *Table Demo*

## matplotlib.pyplot.ylabel

matplotlib.pyplot.**ylabel**(*ylabel*, *fontdict=None*, *labelpad=None*, *, *loc=None*, ***kwargs*)

Set the label for the y-axis.

**Parameters**

**ylabel**

[str] The label text.

**labelpad**

[float, default: *rcParams["axes.labelpad"]* (default: `4.0`)] Spacing in points from the Axes bounding box including ticks and tick labels. If None, the previous value is left as is.

**loc**

[{'bottom', 'center', 'top'}, default: *rcParams["yaxis.labellocation"]* (default: `'center'`)] The label position. This is a high-level alternative for passing parameters *y* and *horizontalalignment*.

**Other Parameters**

****kwargs**

[*Text* properties] *Text* properties control the appearance of the label.

**See also:**

***text***

Documents the properties supported by *Text*.

### Notes

**Note:** This is the *pyplot wrapper* for *axes.Axes.set_ylabel*.

## Examples using `matplotlib.pyplot.ylabel`

- *Multiple subplots*
- *Controlling style of text and labels using a dictionary*
- *Solarized Light stylesheet*
- *Simple plot*
- *Text and mathtext using pyplot*
- *Findobj Demo*
- *Table Demo*
- *Custom scale*
- *Pyplot tutorial*
- *Quick start guide*

## matplotlib.pyplot.ylim

matplotlib.pyplot.**ylim**(**args*, ***kwargs*)

Get or set the y-limits of the current Axes.

Call signatures:

```
bottom, top = ylim()  # return the current ylim
ylim((bottom, top))   # set the ylim to bottom, top
ylim(bottom, top)     # set the ylim to bottom, top
```

If you do not specify args, you can alternatively pass *bottom* or *top* as kwargs, i.e.:

```
ylim(top=3)  # adjust the top leaving bottom unchanged
ylim(bottom=1)  # adjust the bottom leaving top unchanged
```

Setting limits turns autoscaling off for the y-axis.

**Returns**

**bottom, top**

A tuple of the new y-axis limits.

### Notes

Calling this function with no arguments (e.g. `ylim()`) is the pyplot equivalent of calling *get_ylim* on the current Axes. Calling this function with arguments is the pyplot equivalent of calling *set_ylim* on the current Axes. All arguments are passed though.

### Examples using `matplotlib.pyplot.ylim`

- *Infinite lines*
- *Frame grabbing*
- *Interactive functions*
- *Findobj Demo*
- *Pyplot tutorial*

## matplotlib.pyplot.yscale

matplotlib.pyplot.**yscale**(*value*, ***kwargs*)

Set the yaxis' scale.

**Parameters**

**value**

[{"linear", "log", "symlog", "logit", ...} or *ScaleBase*] The axis scale type to apply.

****kwargs**

Different keyword arguments are accepted, depending on the scale. See the respective class keyword arguments:

- *matplotlib.scale.LinearScale*
- *matplotlib.scale.LogScale*
- *matplotlib.scale.SymmetricalLogScale*
- *matplotlib.scale.LogitScale*
- *matplotlib.scale.FuncScale*
- *matplotlib.scale.AsinhScale*

### Notes

**Note:** This is the *pyplot wrapper* for `axes.Axes.set_yscale`.

By default, Matplotlib supports the above-mentioned scales. Additionally, custom scales may be registered using `matplotlib.scale.register_scale`. These scales can then also be used here.

### Examples using `matplotlib.pyplot.yscale`

- *Custom scale*
- *Pyplot tutorial*

## matplotlib.pyplot.yticks

matplotlib.pyplot.**yticks**(*ticks=None*, *labels=None*, *, *minor=False*, ***kwargs*)

Get or set the current tick locations and labels of the y-axis.

Pass no arguments to return the current values without modifying them.

**Parameters**

**ticks**

[array-like, optional] The list of ytick locations. Passing an empty list removes all yticks.

**labels**

[array-like, optional] The labels to place at the given *ticks* locations. This argument can only be passed if *ticks* is passed as well.

**minor**

[bool, default: False] If `False`, get/set the major ticks/labels; if `True`, the minor ticks/labels.

****kwargs**

`Text` properties can be used to control the appearance of the labels.

**Returns**

**locs**

The list of ytick locations.

**labels**

The list of ylabel `Text` objects.

### Notes

Calling this function with no arguments (e.g. `yticks()`) is the pyplot equivalent of calling *get_yticks* and *get_yticklabels* on the current Axes. Calling this function with arguments is the pyplot equivalent of calling *set_yticks* and *set_yticklabels* on the current Axes.

### Examples

```
>>> locs, labels = yticks()  # Get the current locations and labels.
>>> yticks(np.arange(0, 1, step=0.2))  # Set label locations.
>>> yticks(np.arange(3), ['Tom', 'Dick', 'Sue'])  # Set text labels.
>>> yticks([0, 1, 2], ['January', 'February', 'March'],
...        rotation=45)  # Set text labels and properties.
>>> yticks([])  # Disable yticks.
```

## Examples using `matplotlib.pyplot.yticks`

- *Table Demo*

## matplotlib.pyplot.suptitle

matplotlib.pyplot.**suptitle**(*t*, ***kwargs*)

Add a centered suptitle to the figure.

**Parameters**

**t**

[str] The suptitle text.

**x**

[float, default: 0.5] The x location of the text in figure coordinates.

**y**

[float, default: 0.98] The y location of the text in figure coordinates.

**horizontalalignment, ha**

[{'center', 'left', 'right'}, default: center] The horizontal alignment of the text relative to (*x*, *y*).

**verticalalignment, va**

[{'top', 'center', 'bottom', 'baseline'}, default: top] The vertical alignment of the text relative to (*x*, *y*).

**fontsize, size**

[default: *rcParams["figure.titlesize"]* (default: `'large'`)] The font size of the text. See *Text.set_size* for possible values.

**fontweight, weight**

[default: *rcParams["figure.titleweight"]* (default: `'normal'`)] The font weight of the text. See *Text.set_weight* for possible values.

**Returns**

**text**

The *Text* instance of the suptitle.

**Other Parameters**

**fontproperties**

[None or dict, optional] A dict of font properties. If *fontproperties* is given the default values for font size and weight are taken from the *FontProperties* defaults. *rcParams["figure.titlesize"]* (default: `'large'`) and *rcParams["figure.titleweight"]* (default: `'normal'`) are ignored in this case.

****kwargs**

Additional kwargs are *matplotlib.text.Text* properties.

## Notes

**Note:** This is the *pyplot wrapper* for *Figure.suptitle*.

## Examples using `matplotlib.pyplot.suptitle`

- *Nested Gridspecs*
- *Pyplot tutorial*
- *Constrained layout guide*

### matplotlib.pyplot.title

matplotlib.pyplot.**title**(*label*, *fontdict=None*, *loc=None*, *pad=None*, *, *y=None*, ***kwargs*)

Set a title for the Axes.

Set one of the three available Axes titles. The available titles are positioned above the Axes in the center, flush with the left edge, and flush with the right edge.

**Parameters**

**label**

[str] Text to use for the title

**fontdict**

[dict]

---

**Discouraged**

The use of *fontdict* is discouraged. Parameters should be passed as individual keyword arguments or using dictionary-unpacking `set_title(..., **fontdict)`.

---

A dictionary controlling the appearance of the title text, the default *fontdict* is:

```
{'fontsize': rcParams['axes.titlesize'],
 'fontweight': rcParams['axes.titleweight'],
 'color': rcParams['axes.titlecolor'],
 'verticalalignment': 'baseline',
 'horizontalalignment': loc}
```

**loc**

[{'center', 'left', 'right'}, default: *rcParams["axes.titlelocation"]* (default: `'center'`)] Which title to set.

**y**

[float, default: *rcParams["axes.titley"]* (default: `None`)] Vertical Axes location for the title (1.0 is the top). If None (the default) and *rcParams["axes.titley"]* (default: `None`) is also None, y is determined automatically to avoid decorators on the Axes.

**pad**

[float, default: *rcParams["axes.titlepad"]* (default: `6.0`)] The offset of the title from the top of the Axes, in points.

**Returns**

***Text***

The matplotlib text instance representing the title

**Other Parameters**

****kwargs**

[*Text* properties] Other keyword arguments are text properties, see *Text* for a list of valid text properties.

## Notes

**Note:** This is the *pyplot wrapper* for *axes.Axes.set_title*.

## Examples using `matplotlib.pyplot.title`

- *Plotting masked and NaN values*
- *Stairs Demo*
- *Step Demo*
- *Geographic Projections*
- *Multiple subplots*
- *Controlling style of text and labels using a dictionary*
- *Title positioning*
- *Solarized Light stylesheet*
- *Style sheets reference*
- *Text and mathtext using pyplot*
- *Interactive functions*
- *Findobj Demo*
- *Multipage PDF*
- *Set and get properties*
- *Table Demo*
- *Zorder Demo*
- *Rotating a 3D plot*
- *Custom scale*
- *The Sankey class*
- *SVG Histogram*

- *Pyplot tutorial*
- *Quick start guide*

### Layout

| | |
|---|---|
| *margins* | Set or retrieve autoscaling margins. |
| *subplots_adjust* | Adjust the subplot layout parameters. |
| *subplot_tool* | Launch a subplot tool window for a figure. |
| *tight_layout* | Adjust the padding between and around subplots. |

### matplotlib.pyplot.margins

matplotlib.pyplot.**margins**(**margins*, *x=None*, *y=None*, *tight=True*)

Set or retrieve autoscaling margins.

The padding added to each limit of the Axes is the *margin* times the data interval. All input parameters must be floats greater than -0.5. Passing both positional and keyword arguments is invalid and will raise a TypeError. If no arguments (positional or otherwise) are provided, the current margins will remain unchanged and simply be returned.

Specifying any margin changes only the autoscaling; for example, if *xmargin* is not None, then *xmargin* times the X data interval will be added to each end of that interval before it is used in autoscaling.

**Parameters**

***margins**

[float, optional] If a single positional argument is provided, it specifies both margins of the x-axis and y-axis limits. If two positional arguments are provided, they will be interpreted as *xmargin*, *ymargin*. If setting the margin on a single axis is desired, use the keyword arguments described below.

**x, y**

[float, optional] Specific margin values for the x-axis and y-axis, respectively. These cannot be used with positional arguments, but can be used individually to alter on e.g., only the y-axis.

**tight**

[bool or None, default: True] The *tight* parameter is passed to *autoscale_view*, which is executed after a margin is changed; the default here is *True*, on the assumption that when margins are specified, no additional padding to match tick marks is usually desired. Setting *tight* to *None* preserves the previous setting.

**Returns**

**xmargin, ymargin**

[float]

### Notes

**Note:** This is the *pyplot wrapper* for `axes.Axes.margins`.

If a previously used Axes method such as `pcolor()` has set `use_sticky_edges` to `True`, only the limits not set by the "sticky artists" will be modified. To force all of the margins to be set, set `use_sticky_edges` to `False` before calling `margins()`.

### Examples using `matplotlib.pyplot.margins`

- *Controlling view limits using margins and sticky_edges*

## matplotlib.pyplot.subplots_adjust

matplotlib.pyplot.**subplots_adjust**(*left=None*, *bottom=None*, *right=None*, *top=None*, *wspace=None*, *hspace=None*)

Adjust the subplot layout parameters.

Unset parameters are left unmodified; initial values are given by `rcParams["figure.subplot.[name]"]`.

**Parameters**

**left**

[float, optional] The position of the left edge of the subplots, as a fraction of the figure width.

**right**

[float, optional] The position of the right edge of the subplots, as a fraction of the figure width.

**bottom**

[float, optional] The position of the bottom edge of the subplots, as a fraction of the figure height.

**top**

[float, optional] The position of the top edge of the subplots, as a fraction of the figure height.

**wspace**

[float, optional] The width of the padding between subplots, as a fraction of the average Axes width.

**hspace**

[float, optional] The height of the padding between subplots, as a fraction of the average Axes height.

### Notes

**Note:** This is the *pyplot wrapper* for `Figure.subplots_adjust`.

### Examples using `matplotlib.pyplot.subplots_adjust`

- *Contour plot of irregularly spaced data*
- *Subplots spacings and margins*
- *Violin plot customization*
- *Controlling style of text and labels using a dictionary*
- *Parasite axis demo*
- *Table Demo*
- *Pyplot tutorial*

## matplotlib.pyplot.subplot_tool

matplotlib.pyplot.**subplot_tool**(*targetfig=None*)

Launch a subplot tool window for a figure.

**Returns**

***matplotlib.widgets.SubplotTool***

### matplotlib.pyplot.tight_layout

matplotlib.pyplot.**tight_layout**(*, *pad=1.08*, *h_pad=None*, *w_pad=None*, *rect=None*)

Adjust the padding between and around subplots.

To exclude an artist on the Axes from the bounding box calculation that determines the subplot parameters (i.e. legend, or annotation), set `a.set_in_layout(False)` for that artist.

**Parameters**

**pad**

[float, default: 1.08] Padding between the figure edge and the edges of subplots, as a fraction of the font size.

**h_pad, w_pad**

[float, default: *pad*] Padding (height/width) between edges of adjacent subplots, as a fraction of the font size.

**rect**

[tuple (left, bottom, right, top), default: (0, 0, 1, 1)] A rectangle in normalized figure coordinates into which the whole subplots area (including labels) will fit.

**See also:**

*Figure.set_layout_engine*
*pyplot.tight_layout*

#### Notes

**Note:** This is the *pyplot wrapper* for `Figure.tight_layout`.

### Examples using `matplotlib.pyplot.tight_layout`

- *Linestyles*
- *Creating annotated heatmaps*
- *Interpolations for imshow*
- *Streamplot*
- *Resizing Axes with tight layout*
- *Labeling ticks using engineering notation*
- *Figure legend demo*

- *Zorder Demo*
- *3D wireframe plots in one direction*
- *Tick locators*
- *Tight layout guide*

### Colormapping

| | |
|---|---|
| *clim* | Set the color limits of the current image. |
| *colorbar* | Add a colorbar to a plot. |
| *gci* | Get the current colorable artist. |
| *sci* | Set the current image. |
| *get_cmap* | Get a colormap instance, defaulting to rc values if *name* is None. |
| *set_cmap* | Set the default colormap, and applies it to the current image if any. |
| *imread* | Read an image from a file into an array. |
| *imsave* | Colormap and save an array as an image file. |

### matplotlib.pyplot.clim

matplotlib.pyplot.**clim**(*vmin=None*, *vmax=None*)

Set the color limits of the current image.

If either *vmin* or *vmax* is None, the image min/max respectively will be used for color scaling.

If you want to set the clim of multiple images, use *set_clim* on every image, for example:

```
for im in gca().get_images():
    im.set_clim(0, 0.5)
```

### matplotlib.pyplot.colorbar

matplotlib.pyplot.**colorbar**(*mappable=None*, *cax=None*, *ax=None*, ***kwargs*)

Add a colorbar to a plot.

**Parameters**

**mappable**

The *matplotlib.cm.ScalarMappable* (i.e., *AxesImage*, *ContourSet*, etc.) described by this colorbar. This argument is mandatory for the *Figure.colorbar* method but optional for the *pyplot.colorbar* function, which sets the default to the current image.

Note that one can create a *ScalarMappable* "on-the-fly" to generate colorbars not attached to a previously drawn artist, e.g.

```
fig.colorbar(cm.ScalarMappable(norm=norm, cmap=cmap),␣
↪ax=ax)
```

**cax**

[*Axes*, optional] Axes into which the colorbar will be drawn. If `None`, then a new Axes is created and the space for it will be stolen from the Axes(s) specified in *ax*.

**ax**

[*Axes* or iterable or `numpy.ndarray` of Axes, optional] The one or more parent Axes from which space for a new colorbar Axes will be stolen. This parameter is only used if *cax* is not set.

Defaults to the Axes that contains the mappable used to create the colorbar.

**use_gridspec**

[bool, optional] If *cax* is `None`, a new *cax* is created as an instance of Axes. If *ax* is positioned with a subplotspec and *use_gridspec* is `True`, then *cax* is also positioned with a subplotspec.

**Returns**

**colorbar**

[*Colorbar*]

**Other Parameters**

**location**

[None or {'left', 'right', 'top', 'bottom'}] The location, relative to the parent Axes, where the colorbar Axes is created. It also determines the *orientation* of the colorbar (colorbars on the left and right are vertical, colorbars at the top and bottom are horizontal). If None, the location will come from the *orientation* if it is set (vertical colorbars on the right, horizontal ones at the bottom), or default to 'right' if *orientation* is unset.

**orientation**

[None or {'vertical', 'horizontal'}] The orientation of the colorbar. It is preferable to set the *location* of the colorbar, as that also determines the *orientation*; passing incompatible values for *location* and *orientation* raises an exception.

**fraction**

[float, default: 0.15] Fraction of original Axes to use for colorbar.

**shrink**

[float, default: 1.0] Fraction by which to multiply the size of the colorbar.

**aspect**

[float, default: 20] Ratio of long to short dimensions.

**pad**

[float, default: 0.05 if vertical, 0.15 if horizontal] Fraction of original Axes between colorbar and new image Axes.

**anchor**

[(float, float), optional] The anchor point of the colorbar Axes. Defaults to (0.0, 0.5) if vertical; (0.5, 1.0) if horizontal.

**panchor**

[(float, float), or *False*, optional] The anchor point of the colorbar parent Axes. If *False*, the parent axes' anchor will be unchanged. Defaults to (1.0, 0.5) if vertical; (0.5, 0.0) if horizontal.

**extend**

[{'neither', 'both', 'min', 'max'}] Make pointed end(s) for out-of-range values (unless 'neither'). These are set for a given colormap using the colormap set_under and set_over methods.

**extendfrac**

[{*None*, 'auto', length, lengths}] If set to *None*, both the minimum and maximum triangular colorbar extensions will have a length of 5% of the interior colorbar length (this is the default setting).

If set to 'auto', makes the triangular colorbar extensions the same lengths as the interior boxes (when *spacing* is set to 'uniform') or the same lengths as the respective adjacent interior boxes (when *spacing* is set to 'proportional').

If a scalar, indicates the length of both the minimum and maximum triangular colorbar extensions as a fraction of the interior colorbar length. A two-element sequence of fractions may also be given, indicating the lengths of the minimum and maximum colorbar extensions respectively as a fraction of the interior colorbar length.

**extendrect**

[bool] If *False* the minimum and maximum colorbar extensions will be triangular (the default). If *True* the extensions will be rectangular.

**spacing**

[{'uniform', 'proportional'}] For discrete colorbars (`BoundaryNorm` or contours), 'uniform' gives each color the same space; 'proportional' makes the space proportional to the data interval.

**ticks**

[None or list of ticks or Locator] If None, ticks are determined automatically from the input.

**format**

[None or str or Formatter] If None, *ScalarFormatter* is used. Format strings, e.g., `"%4.2e"` or `"{x:.2e}"`, are supported. An alternative *Formatter* may be given instead.

**drawedges**

[bool] Whether to draw lines at color boundaries.

**label**

[str] The label on the colorbar's long axis.

**boundaries, values**

[None or a sequence] If unset, the colormap will be displayed on a 0-1 scale. If sequences, *values* must have a length 1 less than *boundaries*. For each region delimited by adjacent entries in *boundaries*, the color mapped to the corresponding value in values will be used. Normally only useful for indexed colors (i.e. `norm=NoNorm()`) or other unusual circumstances.

## Notes

---

**Note:** This is the *pyplot wrapper* for *Figure.colorbar*.

---

If *mappable* is a *ContourSet*, its *extend* kwarg is included automatically.

The *shrink* kwarg provides a simple way to scale the colorbar with respect to the Axes. Note that if *cax* is specified, it determines the size of the colorbar, and *shrink* and *aspect* are ignored.

For more precise control, you can manually specify the positions of the axes objects in which the mappable and the colorbar are drawn. In this case, do not use any of the Axes properties kwargs.

It is known that some vector graphics viewers (svg and pdf) render white gaps between segments of the colorbar. This is due to bugs in the viewers, not Matplotlib. As a workaround, the colorbar can be rendered with overlapping segments:

```
cbar = colorbar()
cbar.solids.set_edgecolor("face")
draw()
```

However, this has negative consequences in other circumstances, e.g. with semi-transparent images (alpha < 1) and colorbar extensions; therefore, this workaround is not used by default (see issue #1188).

### Examples using `matplotlib.pyplot.colorbar`

- *Contour Demo*
- *Contour Image*
- *Contourf demo*
- *Contourf Hatching*
- *Contourf and log color scale*
- *Creating annotated heatmaps*
- *Image Masked*
- *Multiple images*
- *pcolor images*
- *pcolormesh*
- *Tricontour Demo*
- *Subplots spacings and margins*
- *Colorbar*
- *Creating a colormap from a list of colors*
- *Ellipse Collection*
- *Plotting multiple lines with a LineCollection*
- *Axes divider*
- *Colorbar Tick Labelling*
- *Image tutorial*
- *Tight layout guide*

## matplotlib.pyplot.gci

matplotlib.pyplot.**gci**()

Get the current colorable artist.

Specifically, returns the current *ScalarMappable* instance (`Image` created by *imshow* or *figimage*, *Collection* created by *pcolor* or *scatter*, etc.), or *None* if no such instance has been defined.

The current image is an attribute of the current Axes, or the nearest earlier Axes in the current figure that contains an image.

### Notes

Historically, the only colorable artists were images; hence the name `gci` (get current image).

## matplotlib.pyplot.sci

`matplotlib.pyplot.`**`sci`**(*im*)

Set the current image.

This image will be the target of colormap functions like `pyplot.viridis`, and other functions such as *clim*. The current image is an attribute of the current Axes.

## matplotlib.pyplot.get_cmap

`matplotlib.pyplot.`**`get_cmap`**(*name=None*, *lut=None*)

Get a colormap instance, defaulting to rc values if *name* is None.

**Parameters**

**name**

[*Colormap* or str or None, default: None] If a *Colormap* instance, it will be returned. Otherwise, the name of a colormap known to Matplotlib, which will be resampled by *lut*. The default, None, means *rcParams["image.cmap"]* (default: `'viridis'`).

**lut**

[int or None, default: None] If *name* is not already a Colormap instance and *lut* is not None, the colormap will be resampled to have *lut* entries in the lookup table.

**Returns**

**Colormap**

## Examples using `matplotlib.pyplot.get_cmap`

- *pie(x)*

### matplotlib.pyplot.set_cmap

matplotlib.pyplot.**set_cmap**(*cmap*)

Set the default colormap, and applies it to the current image if any.

**Parameters**

**cmap**

[*Colormap* or str] A colormap instance or the name of a registered colormap.

**See also:**

***colormaps***
***get_cmap***

### matplotlib.pyplot.imread

matplotlib.pyplot.**imread**(*fname*, *format=None*)

Read an image from a file into an array.

---

**Note:** This function exists for historical reasons. It is recommended to use `PIL.Image.open` instead for loading images.

---

**Parameters**

**fname**

[str or file-like] The image file to read: a filename, a URL or a file-like object opened in read-binary mode.

Passing a URL is deprecated. Please open the URL for reading and pass the result to Pillow, e.g. with `np.array(PIL.Image.open(urllib.request.urlopen(url)))`.

**format**

[str, optional] The image file format assumed for reading the data. The image is loaded as a PNG file if *format* is set to "png", if *fname* is a path or opened file with a ".png" extension, or if it is a URL. In all other cases, *format* is ignored and the format is auto-detected by `PIL.Image.open`.

**Returns**

**numpy.array**

The image data. The returned array has shape

- (M, N) for grayscale images.
- (M, N, 3) for RGB images.
- (M, N, 4) for RGBA images.

PNG images are returned as float arrays (0-1). All other formats are returned as int arrays, with a bit depth determined by the file's contents.

### Notes

**Note:** This is equivalent to `matplotlib.image.imread`.

### Examples using `matplotlib.pyplot.imread`

- *Clipping images with patches*
- *Many ways to plot images*
- *Watermark image*
- *AnnotationBbox demo*
- *Using a text as a Path*
- *Convert texts to images*
- *Ribbon Box*
- *mplcvd -- an example of figure hook*

## matplotlib.pyplot.imsave

matplotlib.pyplot.**imsave**(*fname*, *arr*, ***kwargs*)

Colormap and save an array as an image file.

RGB(A) images are passed through. Single channel images will be colormapped according to *cmap* and *norm*.

**Note:** If you want to save a single channel image as gray scale please use an image I/O library (such as pillow, tifffile, or imageio) directly.

**Parameters**

**fname**

[str or path-like or file-like] A path or a file-like object to store the image in. If *format* is not set, then the output format is inferred from the extension of *fname*, if any, and from *rcParams["savefig.format"]* (default: 'png') otherwise. If *format* is set, it determines the output format.

**arr**

[array-like] The image data. The shape can be one of MxN (luminance), MxNx3 (RGB) or MxNx4 (RGBA).

**vmin, vmax**

[float, optional] *vmin* and *vmax* set the color scaling for the image by fixing the values that map to the colormap color limits. If either *vmin* or *vmax* is None, that limit is determined from the *arr* min/max value.

**cmap**

[str or *Colormap*, default: *rcParams["image.cmap"]* (default: 'viridis')] A Colormap instance or registered colormap name. The colormap maps scalar data to colors. It is ignored for RGB(A) data.

**format**

[str, optional] The file format, e.g. 'png', 'pdf', 'svg', ... The behavior when this is unset is documented under *fname*.

**origin**

[{'upper', 'lower'}, default: *rcParams["image.origin"]* (default: 'upper')] Indicates whether the `(0, 0)` index of the array is in the upper left or lower left corner of the Axes.

**dpi**

[float] The DPI to store in the metadata of the file. This does not affect the resolution of the output image. Depending on file format, this may be rounded to the nearest integer.

**metadata**

[dict, optional] Metadata in the image file. The supported keys depend on the output format, see the documentation of the respective backends for more information. Currently only supported for "png", "pdf", "ps", "eps", and "svg".

**pil_kwargs**

[dict, optional] Keyword arguments passed to `PIL.Image.Image.save`. If the 'pnginfo' key is present, it completely overrides *metadata*, including the default 'Software' key.

### Notes

**Note:** This is equivalent to *matplotlib.image.imsave*.

Colormaps are available via the colormap registry *matplotlib.colormaps*. For convenience this registry is available in `pyplot` as

`matplotlib.pyplot.`**`colormaps`**

Container for colormaps that are known to Matplotlib by name.

The universal registry instance is *matplotlib.colormaps*. There should be no need for users to instantiate *ColormapRegistry* themselves.

Read access uses a dict-like interface mapping names to *Colormap*s:

```
import matplotlib as mpl
cmap = mpl.colormaps['viridis']
```

Returned *Colormap*s are copies, so that their modification does not change the global definition of the colormap.

Additional colormaps can be added via *ColormapRegistry.register*:

```
mpl.colormaps.register(my_colormap)
```

To get a list of all registered colormaps, you can do:

```
from matplotlib import colormaps
list(colormaps)
```

Additionally, there are shortcut functions to set builtin colormaps; e.g. `plt.viridis()` is equivalent to `plt.set_cmap('viridis')`.

`matplotlib.pyplot.`**`color_sequences`**

Container for sequences of colors that are known to Matplotlib by name.

The universal registry instance is *matplotlib.color_sequences*. There should be no need for users to instantiate *ColorSequenceRegistry* themselves.

Read access uses a dict-like interface mapping names to lists of colors:

```
import matplotlib as mpl
cmap = mpl.color_sequences['tab10']
```

The returned lists are copies, so that their modification does not change the global definition of the color sequence.

Additional color sequences can be added via *ColorSequenceRegistry.register*:

```
mpl.color_sequences.register('rgb', ['r', 'g', 'b'])
```

### Configuration

| | |
|---|---|
| *rc* | Set the current *rcParams*. *group* is the grouping for the rc, e.g., for `lines.linewidth` the group is `lines`, for `axes.facecolor`, the group is `axes`, and so on. Group may also be a list or tuple of group names, e.g., (*xtick*, *ytick*). *kwargs* is a dictionary attribute name/value pairs, e.g.,::. |
| *rc_context* | Return a context manager for temporarily changing rcParams. |
| *rcdefaults* | Restore the *rcParams* from Matplotlib's internal default style. |

### matplotlib.pyplot.rc

`matplotlib.pyplot.`**`rc`**(*group*, ***kwargs*)

Set the current *rcParams*. *group* is the grouping for the rc, e.g., for `lines.linewidth` the group is `lines`, for `axes.facecolor`, the group is `axes`, and so on. Group may also be a list or tuple of group names, e.g., (*xtick*, *ytick*). *kwargs* is a dictionary attribute name/value pairs, e.g.,:

```
rc('lines', linewidth=2, color='r')
```

sets the current *rcParams* and is equivalent to:

```
rcParams['lines.linewidth'] = 2
rcParams['lines.color'] = 'r'
```

The following aliases are available to save typing for interactive users:

| Alias | Property |
|---|---|
| 'lw' | 'linewidth' |
| 'ls' | 'linestyle' |
| 'c' | 'color' |
| 'fc' | 'facecolor' |
| 'ec' | 'edgecolor' |
| 'mew' | 'markeredgewidth' |
| 'aa' | 'antialiased' |

Thus you could abbreviate the above call as:

```
rc('lines', lw=2, c='r')
```

Note you can use python's kwargs dictionary facility to store dictionaries of default parameters. e.g., you can customize the font rc as follows:

```
font = {'family' : 'monospace',
        'weight' : 'bold',
        'size'   : 'larger'}
rc('font', **font)  # pass in the font dict as kwargs
```

This enables you to easily switch between several configurations. Use `matplotlib.style.use('default')` or *rcdefaults()* to restore the default *rcParams* after changes.

### Notes

---

**Note:** This is equivalent to *matplotlib.rc*.

---

Similar functionality is available by using the normal dict interface, i.e. `rcParams.update({"lines.linewidth": 2, ...})` (but `rcParams.update` does not support abbreviations or grouping).

## Examples using `matplotlib.pyplot.rc`

- *Customizing dashed line styles*
- *Styling with cycler*

## matplotlib.pyplot.rc_context

`matplotlib.pyplot.`**`rc_context`**(*rc=None*, *fname=None*)

Return a context manager for temporarily changing rcParams.

The *rcParams["backend"]* will not be reset by the context manager.

rcParams changed both through the context manager invocation and in the body of the context will be reset on context exit.

**Parameters**

**rc**

[dict] The rcParams to temporarily set.

**fname**

[str or path-like] A file with Matplotlib rc settings. If both *fname* and *rc* are given, settings from *rc* take precedence.

**See also:**

*The matplotlibrc file*

### Notes

**Note:** This is equivalent to *matplotlib.rc_context*.

### Examples

Passing explicit values via a dict:

```
with mpl.rc_context({'interactive': False}):
    fig, ax = plt.subplots()
    ax.plot(range(3), range(3))
    fig.savefig('example.png')
    plt.close(fig)
```

Loading settings from a file:

```
with mpl.rc_context(fname='print.rc'):
    plt.plot(x, y)  # uses 'print.rc'
```

Setting in the context body:

```
with mpl.rc_context():
    # will be reset
    mpl.rcParams['lines.linewidth'] = 5
    plt.plot(x, y)
```

## Examples using `matplotlib.pyplot.rc_context`

- *Style sheets reference*
- *Matplotlib logo*

## matplotlib.pyplot.rcdefaults

matplotlib.pyplot.**rcdefaults**()

Restore the *rcParams* from Matplotlib's internal default style.

Style-blacklisted *rcParams* (defined in `matplotlib.style.core.STYLE_BLACKLIST`) are not updated.

**See also:**

***matplotlib.rc_file_defaults***

Restore the *rcParams* from the rc file originally loaded by Matplotlib.

### *matplotlib.style.use*

Use a specific style file. Call `style.use('default')` to restore the default style.

### Notes

**Note:** This is equivalent to *matplotlib.rcdefaults*.

## Examples using `matplotlib.pyplot.rcdefaults`

- *Customize Rc*

## Output

| | |
|---|---|
| *draw* | Redraw the current figure. |
| *draw_if_interactive* | Redraw the current figure if in interactive mode. |
| *ioff* | Disable interactive mode. |
| *ion* | Enable interactive mode. |
| *install_repl_displayhook* | Connect to the display hook of the current shell. |
| *isinteractive* | Return whether plots are updated after every plotting command. |
| *pause* | Run the GUI event loop for *interval* seconds. |
| *savefig* | Save the current figure as an image or vector graphic to a file. |
| *show* | Display all open figures. |
| *switch_backend* | Set the pyplot backend. |
| *uninstall_repl_displayhook* | Disconnect from the display hook of the current shell. |

## matplotlib.pyplot.draw

matplotlib.pyplot.**draw**()

Redraw the current figure.

This is used to update a figure that has been altered, but not automatically re-drawn. If interactive mode is on (via *ion()*), this should be only rarely needed, but there may be ways to modify the state of a figure without marking it as "stale". Please report these cases as bugs.

This is equivalent to calling `fig.canvas.draw_idle()`, where `fig` is the current figure.

**See also:**

*FigureCanvasBase.draw_idle*
*FigureCanvasBase.draw*

### Examples using `matplotlib.pyplot.draw`

- *Interactive functions*
- *Rotating a 3D plot*
- *Buttons*
- *Textbox*

## matplotlib.pyplot.draw_if_interactive

matplotlib.pyplot.**draw_if_interactive**()

Redraw the current figure if in interactive mode.

> **Warning:** End users will typically not have to call this function because the the interactive mode takes care of this.

## matplotlib.pyplot.ioff

matplotlib.pyplot.**ioff**()

Disable interactive mode.

See *pyplot.isinteractive* for more details.

**See also:**

*ion*

Enable interactive mode.

*isinteractive*

Whether interactive mode is enabled.

*show*

Show all figures (and maybe block).

*pause*

Show all figures, and block for a time.

### Notes

For a temporary change, this can be used as a context manager:

```
# if interactive mode is on
# then figures will be shown on creation
plt.ion()
# This figure will be shown immediately
fig = plt.figure()

with plt.ioff():
    # interactive mode will be off
    # figures will not automatically be shown
    fig2 = plt.figure()
    # ...
```

To enable optional usage as a context manager, this function returns a context manager object, which is not intended to be stored or accessed by the user.

## matplotlib.pyplot.ion

matplotlib.pyplot.**ion**()

Enable interactive mode.

See *pyplot.isinteractive* for more details.

**See also:**

***ioff***

Disable interactive mode.

***isinteractive***

Whether interactive mode is enabled.

***show***

Show all figures (and maybe block).

***pause***

Show all figures, and block for a time.

### Notes

For a temporary change, this can be used as a context manager:

```
# if interactive mode is off
# then figures will not be shown on creation
plt.ioff()
# This figure will not be shown immediately
fig = plt.figure()

with plt.ion():
    # interactive mode will be on
    # figures will automatically be shown
    fig2 = plt.figure()
    # ...
```

To enable optional usage as a context manager, this function returns a context manager object, which is not intended to be stored or accessed by the user.

## matplotlib.pyplot.install_repl_displayhook

matplotlib.pyplot.**install_repl_displayhook**()

Connect to the display hook of the current shell.

The display hook gets called when the read-evaluate-print-loop (REPL) of the shell has finished the execution of a command. We use this callback to be able to automatically update a figure in interactive mode.

This works both with IPython and with vanilla python shells.

## matplotlib.pyplot.isinteractive

matplotlib.pyplot.**isinteractive**()

Return whether plots are updated after every plotting command.

The interactive mode is mainly useful if you build plots from the command line and want to see the effect of each command while you are building the figure.

In interactive mode:

- newly created figures will be shown immediately;
- figures will automatically redraw on change;
- *pyplot.show* will not block by default.

In non-interactive mode:

- newly created figures and changes to figures will not be reflected until explicitly asked to be;
- *pyplot.show* will block by default.

**See also:**

*ion*

Enable interactive mode.

*ioff*

Disable interactive mode.

*show*

Show all figures (and maybe block).

*pause*

Show all figures, and block for a time.

### matplotlib.pyplot.pause

matplotlib.pyplot.**pause**(*interval*)

Run the GUI event loop for *interval* seconds.

If there is an active figure, it will be updated and displayed before the pause, and the GUI event loop (if any) will run during the pause.

This can be used for crude animation. For more complex animation use *matplotlib.animation*.

If there is no active figure, sleep for *interval* seconds instead.

**See also:**

*matplotlib.animation*

Proper animations

*show*

Show all figures and optional block until all figures are closed.

### Examples using `matplotlib.pyplot.pause`

- *pyplot animation*
- *Rotating a 3D plot*
- *Animate a 3D wireframe plot*
- *Faster rendering by using blitting*

### matplotlib.pyplot.savefig

matplotlib.pyplot.**savefig**(**args*, ***kwargs*)

Save the current figure as an image or vector graphic to a file.

Call signature:

```
savefig(fname, *, transparent=None, dpi='figure', format=None,
        metadata=None, bbox_inches=None, pad_inches=0.1,
        facecolor='auto', edgecolor='auto', backend=None,
        **kwargs
       )
```

The available output formats depend on the backend being used.

**Parameters**

**fname**

[str or path-like or binary file-like] A path, or a Python file-like object, or possibly some backend-dependent object such as *matplotlib.backends.backend_pdf.PdfPages*.

If *format* is set, it determines the output format, and the file is saved as *fname*. Note that *fname* is used verbatim, and there is no attempt to make the extension, if any, of *fname* match *format*, and no extension is appended.

If *format* is not set, then the format is inferred from the extension of *fname*, if there is one. If *format* is not set and *fname* has no extension, then the file is saved with *rcParams["savefig.format"]* (default: `'png'`) and the appropriate extension is appended to *fname*.

**Other Parameters**

**transparent**

[bool, default: *rcParams["savefig.transparent"]* (default: `False`)] If *True*, the Axes patches will all be transparent; the Figure patch will also be transparent unless *facecolor* and/or *edgecolor* are specified via kwargs.

If *False* has no effect and the color of the Axes and Figure patches are unchanged (unless the Figure patch is specified via the *facecolor* and/or *edgecolor* keyword arguments in which case those colors are used).

The transparency of these patches will be restored to their original values upon exit of this function.

This is useful, for example, for displaying a plot on top of a colored background on a web page.

**dpi**

[float or 'figure', default: *rcParams["savefig.dpi"]* (default: `'figure'`)] The resolution in dots per inch. If 'figure', use the figure's dpi value.

**format**

[str] The file format, e.g. 'png', 'pdf', 'svg', ... The behavior when this is unset is documented under *fname*.

**metadata**

[dict, optional] Key/value pairs to store in the image metadata. The supported keys and defaults depend on the image format and backend:

- 'png' with Agg backend: See the parameter `metadata` of *print_png*.
- 'pdf' with pdf backend: See the parameter `metadata` of *PdfPages*.
- 'svg' with svg backend: See the parameter `metadata` of *print_svg*.
- 'eps' and 'ps' with PS backend: Only 'Creator' is supported.

Not supported for 'pgf', 'raw', and 'rgba' as those formats do not support embedding metadata. Does not currently support 'jpg', 'tiff', or 'webp', but may include embedding EXIF metadata in the future.

**bbox_inches**

[str or *Bbox*, default: *rcParams["savefig.bbox"]* (default: `None`)] Bounding box in inches: only the given portion of the figure is saved. If 'tight', try to figure out the tight bbox of the figure.

**pad_inches**

[float or 'layout', default: *rcParams["savefig.pad_inches"]* (default: `0.1`)] Amount of padding in inches around the figure when bbox_inches is 'tight'. If 'layout' use the padding from the constrained or compressed layout engine; ignored if one of those engines is not in use.

**facecolor**

[*color* or 'auto', default: *rcParams["savefig.facecolor"]* (default: `'auto'`)] The facecolor of the figure. If 'auto', use the current figure facecolor.

**edgecolor**

[*color* or 'auto', default: *rcParams["savefig.edgecolor"]* (default: `'auto'`)] The edgecolor of the figure. If 'auto', use the current figure edgecolor.

**backend**

[str, optional] Use a non-default backend to render the file, e.g. to render a png file with the "cairo" backend rather than the default "agg", or a pdf file with the "pgf" backend rather than the default "pdf". Note that the default backend is normally sufficient. See *The builtin backends* for a list of valid backends for each file format. Custom backends can be referenced as "module://...".

**orientation**

[{'landscape', 'portrait'}] Currently only supported by the postscript backend.

**papertype**

[str] One of 'letter', 'legal', 'executive', 'ledger', 'a0' through 'a10', 'b0' through 'b10'. Only supported for postscript output.

**bbox_extra_artists**

[list of *Artist*, optional] A list of extra artists that will be considered when the tight bbox is calculated.

**pil_kwargs**

[dict, optional] Additional keyword arguments that are passed to `PIL.Image.Image.save` when saving the figure.

### Notes

**Note:** This is the *pyplot wrapper* for *Figure.savefig*.

### Examples using `matplotlib.pyplot.savefig`

- *Print Stdout*
- *Rasterization for vector graphics*
- *SVG Filter Line*
- *SVG filter pie*
- *CanvasAgg demo*
- *SVG Histogram*
- *SVG Tooltip*

## matplotlib.pyplot.show

`matplotlib.pyplot.`**`show`**`(*, block=None)`

Display all open figures.

**Parameters**

**block**

[bool, optional] Whether to wait for all figures to be closed before returning.

If `True` block and run the GUI main loop until all figure windows are closed.

If `False` ensure that all figure windows are displayed and return immediately. In this case, you are responsible for ensuring that the event loop is running to have responsive figures.

Defaults to True in non-interactive mode and to False in interactive mode (see *pyplot.isinteractive*).

**See also:**

***ion***

Enable interactive mode, which shows / updates the figure after every plotting command, so that calling `show()` is not necessary.

***ioff***

Disable interactive mode.

***savefig***

Save the figure to an image file instead of showing it on screen.

## Notes

**Saving figures to file and showing a window at the same time**

If you want an image file as well as a user interface window, use *pyplot.savefig* before *pyplot.show*. At the end of (a blocking) `show()` the figure is closed and thus unregistered from pyplot. Calling *pyplot.savefig* afterwards would save a new and thus empty figure. This limitation of command order does not apply if the show is non-blocking or if you keep a reference to the figure and use *Figure.savefig*.

**Auto-show in jupyter notebooks**

The jupyter backends (activated via `%matplotlib inline`, `%matplotlib notebook`, or `%matplotlib widget`), call `show()` at the end of every cell by default. Thus, you usually don't have to call it explicitly there.

## Examples using `matplotlib.pyplot.show`

- *Bar color demo*
- *Bar Label Demo*
- *Stacked bar chart*
- *Grouped bar chart with labels*
- *Horizontal bar chart*
- *Broken Barh*
- *CapStyle*

- *Plotting categorical variables*
- *Plotting the coherence of two signals*
- *Cross spectral density (CSD)*
- *Curve with error band*
- *Errorbar limit selection*
- *Errorbar subsampling*
- *EventCollection Demo*
- *Eventplot demo*
- *Filled polygon*
- *Fill Between and Alpha*
- *Fill Betweenx Demo*
- *Hatch-filled histograms*
- *Bar chart with gradients*
- *Hat graph*
- *Discrete distribution as horizontal bar chart*
- *JoinStyle*
- *Customizing dashed line styles*
- *Lines with a ticked patheffect*
- *Linestyles*
- *Marker reference*
- *Markevery Demo*
- *Plotting masked and NaN values*
- *Multicolored lines*
- *Mapping marker properties to multivariate data*
- *Power spectral density (PSD)*
- *Scatter Demo2*
- *Scatter plot with histograms*
- *Scatter Masked*
- *Marker examples*
- *Scatter plots with a legend*
- *Simple Plot*
- *Shade regions defined by a logical mask using fill_between*

- *Spectrum representations*
- *Stackplots and streamgraphs*
- *Stairs Demo*
- *Stem Plot*
- *Step Demo*
- *Creating a timeline with lines, dates, and text*
- *hlines and vlines*
- *Cross- and auto-correlation*
- *Affine transform of an image*
- *Wind Barbs*
- *Barcode*
- *Interactive Adjustment of Colormap Range*
- *Colormap normalizations*
- *Colormap normalizations SymLogNorm*
- *Contour Corner Mask*
- *Contour Demo*
- *Contour Image*
- *Contour Label Demo*
- *Contourf demo*
- *Contourf Hatching*
- *Contourf and log color scale*
- *Contouring the solution space of optimizations*
- *BboxImage Demo*
- *Figimage Demo*
- *Creating annotated heatmaps*
- *Image antialiasing*
- *Clipping images with patches*
- *Many ways to plot images*
- *Image Masked*
- *Image nonuniform*
- *Blend transparency with color in 2D images*
- *Modifying the coordinate formatter*

- *Interpolations for imshow*
- *Contour plot of irregularly spaced data*
- *Layer Images*
- *Visualize matrices with matshow*
- *Multiple images*
- *pcolor images*
- *pcolormesh grids and shading*
- *pcolormesh*
- *Streamplot*
- *QuadMesh Demo*
- *Advanced quiver and quiverkey functions*
- *Quiver Simple Demo*
- *Shading example*
- *Spectrogram*
- *Spy Demos*
- *Tricontour Demo*
- *Tricontour Smooth Delaunay*
- *Tricontour Smooth User*
- *Trigradient Demo*
- *Triinterp Demo*
- *Tripcolor Demo*
- *Triplot Demo*
- *Watermark image*
- *Aligning Labels and Titles*
- *Programmatically controlling subplot adjustment*
- *Axes box aspect*
- *Axes Demo*
- *Controlling view limits using margins and sticky_edges*
- *Axes Props*
- *Axes Zoom Effect*
- *axhspan Demo*
- *Equal axis aspect ratio*

- *Axis Label Position*
- *Broken Axis*
- *Resizing Axes with constrained layout*
- *Resizing Axes with tight layout*
- *Different scales on the same Axes*
- *Figure size in different units*
- *Figure labels: suptitle, supxlabel, supylabel*
- *Creating adjacent subplots*
- *Geographic Projections*
- *Combining two subplots using subplots and GridSpec*
- *Using Gridspec to make multi-column/row subplot layouts*
- *Nested Gridspecs*
- *Inverted axis*
- *Managing multiple figures in pyplot*
- *Secondary Axis*
- *Sharing axis limits and views*
- *Shared axis*
- *Figure subfigures*
- *Multiple subplots*
- *Subplots spacings and margins*
- *Creating multiple subplots using plt.subplots*
- *Plots with different scales*
- *Zoom region inset Axes*
- *Percentiles as horizontal bar chart*
- *Artist customization in box plots*
- *Box plots with custom fill colors*
- *Boxplots*
- *Box plot vs. violin plot comparison*
- *Separate calculation and plotting of boxplots*
- *Plot a confidence ellipse of a two-dimensional dataset*
- *Violin plot customization*
- *Errorbar function*

- *Different ways of specifying error bars*
- *Including upper and lower limits in error bars*
- *Creating boxes from error bars using PatchCollection*
- *Hexagonal binned plot*
- *Histograms*
- *Bihistogram*
- *Plotting cumulative distributions*
- *Demo of the histogram function's different histtype settings*
- *The histogram (hist) function with multiple data sets*
- *Histogram bins, density, and weight*
- *Producing multiple histograms side by side*
- *Time Series Histogram*
- *Violin plot basics*
- *Pie charts*
- *Bar of pie*
- *Nested pie charts*
- *Labeling a pie and a donut*
- *Bar chart on polar axis*
- *Polar plot*
- *Error bar rendering on polar axis*
- *Polar legend*
- *Scatter plot on polar axis*
- *Accented text*
- *Align y-labels*
- *Scale invariant angle label*
- *Angle annotations on bracket arrows*
- *Annotate Transform*
- *Annotating a plot*
- *Annotating Plots*
- *Annotation Polar*
- *Arrow Demo*
- *Auto-wrapping text*

- *Composing Custom Legends*
- *Date tick labels*
- *AnnotationBbox demo*
- *Using a text as a Path*
- *Text Rotation Mode*
- *The difference between \dfrac and \frac*
- *Labeling ticks using engineering notation*
- *Annotation arrow style reference*
- *Styling text boxes*
- *Figure legend demo*
- *Configuring the font family*
- *Using ttf font files*
- *Font table*
- *Fonts demo (object-oriented style)*
- *Fonts demo (keyword arguments)*
- *Labelling subplots*
- *Legend using pre-defined labels*
- *Legend Demo*
- *Artist within an artist*
- *Convert texts to images*
- *Mathtext*
- *Mathtext Examples*
- *Math fontfamily*
- *Multiline*
- *Placing text boxes*
- *Concatenating text objects with different properties*
- *STIX Fonts*
- *Rendering math equations using TeX*
- *Text alignment*
- *Text Commands*
- *Controlling style of text and labels using a dictionary*
- *Text Rotation Relative To Line*

- *Title positioning*
- *Unicode minus*
- *Usetex Baseline Test*
- *Usetex Fonteffects*
- *Text watermark*
- *Color Demo*
- *Color by y-value*
- *Colors in the default property cycle*
- *Colorbar*
- *Creating a colormap from a list of colors*
- *Selecting individual colors from a colormap*
- *List of named colors*
- *Ways to set a color's alpha value*
- *Arrow guide*
- *Reference for Matplotlib artists*
- *Line, Poly and RegularPoly Collection with autoscaling*
- *Compound path*
- *Dolphins*
- *Mmh Donuts!!!*
- *Ellipse with orientation arrow demo*
- *Ellipse Collection*
- *Ellipse Demo*
- *Drawing fancy boxes*
- *Hatch demo*
- *Plotting multiple lines with a LineCollection*
- *Circles, Wedges and Polygons*
- *PathPatch object*
- *Bezier Curve*
- *Scatter plot*
- *Bayesian Methods for Hackers style sheet*
- *Dark background style sheet*
- *FiveThirtyEight style sheet*

- *ggplot style sheet*
- *Grayscale style sheet*
- *Solarized Light stylesheet*
- *Style sheets reference*
- *Infinite lines*
- *Simple plot*
- *Text and mathtext using pyplot*
- *Multiple lines using pyplot*
- *Two subplots using pyplot*
- *Anchored Direction Arrow*
- *Axes divider*
- *Demo Axes Grid*
- *Axes Grid2*
- *HBoxDivider and VBoxDivider demo*
- *Showing RGB channels using RGBAxes*
- *Colorbar with AxesDivider*
- *Controlling the position and size of colorbars with Inset Axes*
- *Per-row or per-column colorbars*
- *Axes with a fixed physical size*
- *Setting a fixed aspect on ImageGrid cells*
- *Inset locator demo*
- *Inset locator demo 2*
- *Make room for ylabel using axes_grid*
- *Parasite Simple*
- *Parasite Simple2*
- *Scatter Histogram (Locatable Axes)*
- *Simple Anchored Artists*
- *Simple Axes Divider 1*
- *Simple axes divider 3*
- *Simple ImageGrid*
- *Simple ImageGrid 2*
- *Simple Axisline4*

- *Axis Direction*
- *axis_direction demo*
- *Axis line styles*
- *Curvilinear grid demo*
- *Demo CurveLinear Grid2*
- *floating_axes features*
- *floating_axis demo*
- *Parasite Axes demo*
- *Parasite axis demo*
- *Ticklabel alignment*
- *Ticklabel direction*
- *Simple axis direction*
- *Simple axis tick label and tick directions*
- *Simple Axis Pad*
- *Custom spines with axisartist*
- *Simple Axisline*
- *Simple Axisline3*
- *Anatomy of a figure*
- *Firefox*
- *Integral as the area under a curve*
- *Shaded & power normalized rendering*
- *Stock prices over 32 years*
- *XKCD*
- *Decay*
- *Animated histogram*
- *The Bayes update*
- *The double pendulum problem*
- *Animated image using a precomputed list of images*
- *Multiple Axes animation*
- *Pausing and Resuming an Animation*
- *Rain simulation*
- *Animated 3D random walk*

- *Animated line plot*
- *Animated scatter saved as GIF*
- *Oscilloscope*
- *MATPLOTLIB UNCHAINED*
- *Close Event*
- *Mouse move and click events*
- *Cross-hair cursor*
- *Data browser*
- *Figure/Axes enter and leave events*
- *Interactive functions*
- *Scroll event*
- *Keypress event*
- *Lasso Demo*
- *Legend picking*
- *Looking Glass*
- *Path editor*
- *Pick event demo*
- *Pick event demo 2*
- *Poly Editor*
- *Pong*
- *Resampling Data*
- *Timers*
- *Trifinder Event Demo*
- *Viewlims*
- *Zoom Window*
- *Anchored Artists*
- *Changing colors of lines intersecting a box*
- *Manual Contour*
- *Coords Report*
- *Custom projection*
- *Customize Rc*
- *AGG filter*

- *Ribbon Box*
- *Adding lines to figures*
- *Fill Spiral*
- *Findobj Demo*
- *Building histograms using Rectangles and PolyCollections*
- *Plotting with keywords*
- *Matplotlib logo*
- *Multiprocessing*
- *Packed-bubble chart*
- *Patheffect Demo*
- *Set and get properties*
- *Table Demo*
- *TickedStroke patheffect*
- *transforms.offset_copy*
- *Zorder Demo*
- *Plot 2D data on 3D plot*
- *Demo of 3D bar charts*
- *Create 2D bar graphs in different planes*
- *3D box surface plot*
- *Plot contour (level) curves in 3D*
- *Plot contour (level) curves in 3D using the extend3d option*
- *Project contour profiles onto a graph*
- *Filled contours*
- *Project filled contour onto a graph*
- *Custom hillshading in a 3D surface plot*
- *3D errorbars*
- *Create 3D histogram of 2D data*
- *Parametric curve*
- *Lorenz attractor*
- *2D and 3D Axes in same figure*
- *Automatic text offsetting*
- *Draw flat objects in 3D plot*

- *Generate polygons to fill under 3D line graph*
- *3D plot projection types*
- *3D quiver plot*
- *3D scatterplot*
- *3D stem*
- *3D plots as subplots*
- *3D surface (colormap)*
- *3D surface (solid color)*
- *3D surface (checkerboard)*
- *3D surface with polar coordinates*
- *Text annotations in 3D*
- *Triangular 3D contour plot*
- *Triangular 3D filled contour plot*
- *Triangular 3D surfaces*
- *More triangular 3D surfaces*
- *Primary 3D view planes*
- *3D voxel / volumetric plot*
- *3D voxel plot of the NumPy logo*
- *3D voxel / volumetric plot with RGB colors*
- *3D voxel / volumetric plot with cylindrical coordinates*
- *3D wireframe plot*
- *3D wireframe plots in one direction*
- *Asinh Demo*
- *Loglog Aspect*
- *Custom scale*
- *Log Bar*
- *Log Demo*
- *Logit Demo*
- *Exploring normalizations*
- *Scales*
- *Log Axis*
- *Symlog Demo*

- *Hillshading*
- *Anscombe's quartet*
- *Hinton diagrams*
- *Ishikawa Diagram*
- *Left ventricle bullseye*
- *MRI with EEG*
- *Radar chart (aka spider or star chart)*
- *The Sankey class*
- *Long chain of connections using Sankey*
- *Rankine power cycle*
- *SkewT-logP diagram: using transforms and custom projections*
- *Topographic hillshading*
- *Spines*
- *Spine placement*
- *Dropped spines*
- *Multiple y-axis with Spines*
- *Centered spines with arrows*
- *Automatically setting tick positions*
- *Centering labels between ticks*
- *Colorbar Tick Labelling*
- *Custom Ticker*
- *Formatting date ticks using ConciseDateFormatter*
- *Date Demo Convert*
- *Placing date ticks using recurrence rules*
- *Custom tick formatter for time series*
- *Date Precision and Epochs*
- *Dollar ticks*
- *Fig Axes Customize Simple*
- *Major and minor ticks*
- *Multilevel (nested) ticks*
- *The default tick formatter*
- *Tick formatters*

- *Tick locators*
- *Set default y-axis tick labels on the right*
- *Setting tick labels from a list of values*
- *Move x-axis tick labels to the top*
- *Rotating custom tick labels*
- *Fixing too many ticks*
- *Annotation with units*
- *Artist tests*
- *Bar demo with units*
- *Group barchart with units*
- *Ellipse with units*
- *Evans test*
- *Radian ticks*
- *Inches and Centimeters*
- *Unit handling*
- *mplcvd -- an example of figure hook*
- *pyplot with GTK3*
- *pyplot with GTK4*
- *Tool Manager*
- *Annotated cursor*
- *Buttons*
- *Check buttons*
- *Cursor*
- *Lasso Selector*
- *Menu*
- *Mouse Cursor*
- *Multicursor*
- *Select indices from a collection using polygon selector*
- *Radio Buttons*
- *Thresholding an Image with RangeSlider*
- *Rectangle and ellipse selectors*
- *Slider*

- *Snapping Sliders to Discrete Values*
- *Span Selector*
- *Textbox*
- *Annotate Explain*
- *Annotate Text Arrow*
- *Connection styles for annotations*
- *Custom box styles*
- *subplot2grid demo*
- *GridSpec demo*
- *Nested GridSpecs*
- *Simple Annotate01*
- *Simple Legend01*
- *Simple Legend02*
- *Pyplot tutorial*
- *The Lifecycle of a Plot*
- *Artist tutorial*
- *plot(x, y)*
- *scatter(x, y)*
- *bar(x, height)*
- *stem(x, y)*
- *fill_between(x, y1, y2)*
- *stackplot(x, y)*
- *stairs(values)*
- *hist(x)*
- *boxplot(X)*
- *errorbar(x, y, yerr, xerr)*
- *violinplot(D)*
- *eventplot(D)*
- *hist2d(x, y)*
- *hexbin(x, y, C)*
- *pie(x)*
- *ecdf(x)*

- *imshow(Z)*
- *pcolormesh(X, Y, Z)*
- *contour(X, Y, Z)*
- *contourf(X, Y, Z)*
- *barbs(X, Y, U, V)*
- *quiver(X, Y, U, V)*
- *streamplot(X, Y, U, V)*
- *tricontour(x, y, z)*
- *tricontourf(x, y, z)*
- *tripcolor(x, y, z)*
- *triplot(x, y)*
- *scatter(xs, ys, zs)*
- *plot_surface(X, Y, Z)*
- *plot_trisurf(x, y, z)*
- *voxels([x, y, z], filled)*
- *plot_wireframe(X, Y, Z)*
- *Quick start guide*
- *Customizing Matplotlib with style sheets and rcParams*
- *Animations using Matplotlib*
- *Faster rendering by using blitting*
- *Styling with cycler*
- *origin and extent in imshow*
- *Path effects guide*
- *Path Tutorial*
- *Transformations Tutorial*
- *Legend guide*
- *Constrained layout guide*
- *Tight layout guide*
- *Arranging multiple Axes in a Figure*
- *Axis ticks*
- *Specifying colors*
- *Customized Colorbars Tutorial*

- *Creating Colormaps in Matplotlib*
- *Colormap normalization*
- *Choosing Colormaps in Matplotlib*
- *Text in Matplotlib*
- *Text properties and layout*

### matplotlib.pyplot.switch_backend

matplotlib.pyplot.**switch_backend**(*newbackend*)

Set the pyplot backend.

Switching to an interactive backend is possible only if no event loop for another interactive backend has started. Switching to and from non-interactive backends is always possible.

If the new backend is different than the current backend then all open Figures will be closed via `plt.close('all')`.

**Parameters**

**newbackend**

[str] The case-insensitive name of the backend to use.

### matplotlib.pyplot.uninstall_repl_displayhook

matplotlib.pyplot.**uninstall_repl_displayhook**()

Disconnect from the display hook of the current shell.

### Other

| | |
|---|---|
| *connect* | Bind function *func* to event *s*. |
| *disconnect* | Disconnect the callback with id *cid*. |
| *findobj* | Find artist objects. |
| *get* | Return the value of an *Artist*'s *property*, or print all of them. |
| *getp* | Return the value of an *Artist*'s *property*, or print all of them. |
| *get_current_fig_manager* | Return the figure manager of the current figure. |
| *ginput* | Blocking call to interact with a figure. |
| *new_figure_manager* | Create a new figure manager instance. |
| *set_loglevel* | Configure Matplotlib's logging levels. |
| *setp* | Set one or more properties on an *Artist*, or list allowed values. |
| *waitforbuttonpress* | Blocking call to interact with the figure. |
| *xkcd* | Turn on xkcd sketch-style drawing mode. |

### matplotlib.pyplot.connect

matplotlib.pyplot.**connect**(*s*, *func*)

Bind function *func* to event *s*.

**Parameters**

**s**

[str] One of the following events ids:

- 'button_press_event'
- 'button_release_event'
- 'draw_event'
- 'key_press_event'
- 'key_release_event'
- 'motion_notify_event'
- 'pick_event'
- 'resize_event'
- 'scroll_event'
- 'figure_enter_event',
- 'figure_leave_event',
- 'axes_enter_event',

- 'axes_leave_event'
- 'close_event'.

**func**

[callable] The callback function to be executed, which must have the signature:

```
def func(event: Event) -> Any
```

For the location events (button and key press/release), if the mouse is over the Axes, the `inaxes` attribute of the event will be set to the *Axes* the event occurs is over, and additionally, the variables `xdata` and `ydata` attributes will be set to the mouse location in data coordinates. See *KeyEvent* and *MouseEvent* for more info.

---

**Note:** If func is a method, this only stores a weak reference to the method. Thus, the figure does not influence the lifetime of the associated object. Usually, you want to make sure that the object is kept alive throughout the lifetime of the figure by holding a reference to it.

---

**Returns**

**cid**

A connection id that can be used with *FigureCanvasBase.mpl_disconnect*.

## Notes

---

**Note:** This is the *pyplot wrapper* for *FigureCanvasBase.mpl_connect*.

---

## Examples

```
def on_press(event):
    print('you pressed', event.button, event.xdata, event.ydata)

cid = canvas.mpl_connect('button_press_event', on_press)
```

### Examples using `matplotlib.pyplot.connect`

- *Mouse move and click events*

## matplotlib.pyplot.disconnect

matplotlib.pyplot.**disconnect**(*cid*)

Disconnect the callback with id *cid*.

### Notes

**Note:** This is the *pyplot wrapper* for *FigureCanvasBase.mpl_disconnect*.

### Examples

```
cid = canvas.mpl_connect('button_press_event', on_press)
# ... later
canvas.mpl_disconnect(cid)
```

### Examples using `matplotlib.pyplot.disconnect`

- *Mouse move and click events*

## matplotlib.pyplot.findobj

matplotlib.pyplot.**findobj**(*o=None*, *match=None*, *include_self=True*)

Find artist objects.

Recursively find all *Artist* instances contained in the artist.

**Parameters**

**match**

A filter criterion for the matches. This can be

- *None*: Return all objects contained in artist.
- A function with signature `def match(artist: Artist) -> bool`. The result will only contain artists for which the function returns *True*.
- A class instance: e.g., *Line2D*. The result will only contain artists of this class or its subclasses (`isinstance` check).

**include_self**

[bool] Include *self* in the list to be checked for a match.

**Returns**

**list of *Artist***

## matplotlib.pyplot.get

matplotlib.pyplot.**get**(*obj*, **args*, ***kwargs*)

Return the value of an *Artist*'s *property*, or print all of them.

**Parameters**

**obj**

[*Artist*] The queried artist; e.g., a *Line2D*, a *Text*, or an *Axes*.

**property**

[str or None, default: None] If *property* is 'somename', this function returns `obj.get_somename()`.

If it's None (or unset), it *prints* all gettable properties from *obj*. Many properties have aliases for shorter typing, e.g. 'lw' is an alias for 'linewidth'. In the output, aliases and full property names will be listed as:

property or alias = value

e.g.:

linewidth or lw = 2

**See also:**

***setp***

### Notes

**Note:** This is equivalent to *matplotlib.artist.getp*.

### matplotlib.pyplot.getp

matplotlib.pyplot.**getp**(*obj*, **args*, ***kwargs*)

Return the value of an *Artist*'s *property*, or print all of them.

**Parameters**

**obj**

[*Artist*] The queried artist; e.g., a *Line2D*, a *Text*, or an *Axes*.

**property**

[str or None, default: None] If *property* is 'somename', this function returns `obj.get_somename()`.

If it's None (or unset), it *prints* all gettable properties from *obj*. Many properties have aliases for shorter typing, e.g. 'lw' is an alias for 'linewidth'. In the output, aliases and full property names will be listed as:

property or alias = value

e.g.:

linewidth or lw = 2

**See also:**

***setp***

#### Notes

**Note:** This is equivalent to *matplotlib.artist.getp*.

### Examples using `matplotlib.pyplot.getp`

- *Set and get properties*

### matplotlib.pyplot.get_current_fig_manager

matplotlib.pyplot.**get_current_fig_manager**()

Return the figure manager of the current figure.

The figure manager is a container for the actual backend-depended window that displays the figure on screen.

If no current figure exists, a new one is created, and its figure manager is returned.

**Returns**

*FigureManagerBase* **or backend-dependent subclass thereof**

### matplotlib.pyplot.ginput

matplotlib.pyplot.**ginput**(*n=1*, *timeout=30*, *show_clicks=True*, *mouse_add=MouseButton.LEFT*, *mouse_pop=MouseButton.RIGHT*, *mouse_stop=MouseButton.MIDDLE*)

Blocking call to interact with a figure.

Wait until the user clicks *n* times on the figure, and return the coordinates of each click in a list.

There are three possible interactions:

- Add a point.
- Remove the most recently added point.
- Stop the interaction and return the points added so far.

The actions are assigned to mouse buttons via the arguments *mouse_add*, *mouse_pop* and *mouse_stop*.

**Parameters**

**n**

[int, default: 1] Number of mouse clicks to accumulate. If negative, accumulate clicks until the input is terminated manually.

**timeout**

[float, default: 30 seconds] Number of seconds to wait before timing out. If zero or negative will never time out.

**show_clicks**

[bool, default: True] If True, show a red cross at the location of each click.

**mouse_add**

[*MouseButton* or None, default: *MouseButton.LEFT*] Mouse button used to add points.

**mouse_pop**

[*MouseButton* or None, default: *MouseButton.RIGHT*] Mouse button used to remove the most recently added point.

**mouse_stop**

[*MouseButton* or None, default: *MouseButton.MIDDLE*] Mouse button used to stop input.

**Returns**

**list of tuples**

A list of the clicked (x, y) coordinates.

### Notes

**Note:** This is the *pyplot wrapper* for `Figure.ginput`.

The keyboard can also be used to select points in case your mouse does not have one or more of the buttons. The delete and backspace keys act like right-clicking (i.e., remove last point), the enter key terminates input and any other key (not already used by the window manager) selects a point.

### Examples using `matplotlib.pyplot.ginput`

- *Interactive functions*

## matplotlib.pyplot.new_figure_manager

`matplotlib.pyplot.`**`new_figure_manager`**(*num*, **args*, ***kwargs*)

Create a new figure manager instance.

## matplotlib.pyplot.set_loglevel

`matplotlib.pyplot.`**`set_loglevel`**(**args*, ***kwargs*)

Configure Matplotlib's logging levels.

Matplotlib uses the standard library `logging` framework under the root logger 'matplotlib'. This is a helper function to:

- set Matplotlib's root logger level
- set the root logger handler's level, creating the handler if it does not exist yet

Typically, one should call `set_loglevel("info")` or `set_loglevel("debug")` to get additional debugging information.

Users or applications that are installing their own logging handlers may want to directly manipulate `logging.getLogger('matplotlib')` rather than use this function.

**Parameters**

**level**

[{"notset", "debug", "info", "warning", "error", "critical"}] The log level of the handler.

### Notes

**Note:** This is equivalent to *matplotlib.set_loglevel*.

The first time this function is called, an additional handler is attached to Matplotlib's root handler; this handler is reused every time and this function simply manipulates the logger and handler's level.

## matplotlib.pyplot.setp

matplotlib.pyplot.**setp**(*obj*, **args*, ***kwargs*)

Set one or more properties on an *Artist*, or list allowed values.

**Parameters**

**obj**

[*Artist* or list of *Artist*] The artist(s) whose properties are being set or queried. When setting properties, all artists are affected; when querying the allowed values, only the first instance in the sequence is queried.

For example, two lines can be made thicker and red with a single call:

```
>>> x = arange(0, 1, 0.01)
>>> lines = plot(x, sin(2*pi*x), x, sin(4*pi*x))
>>> setp(lines, linewidth=2, color='r')
```

**file**

[file-like, default: sys.stdout] Where *setp* writes its output when asked to list allowed values.

```
>>> with open('output.log') as file:
...     setp(line, file=file)
```

The default, None, means sys.stdout.

***args, **kwargs**

The properties to set. The following combinations are supported:

- Set the linestyle of a line to be dashed:

```
>>> line, = plot([1, 2, 3])
>>> setp(line, linestyle='--')
```

- Set multiple properties at once:

```
>>> setp(line, linewidth=2, color='r')
```

- List allowed values for a line's linestyle:

```
>>> setp(line, 'linestyle')
linestyle: {'-', '--', '-.', ':', '', (offset, on-off-
 ↪seq), ...}
```

- List all properties that can be set, and their allowed values:

```
>>> setp(line)
agg_filter: a filter function, ...
[long output listing omitted]
```

*setp* also supports MATLAB style string/value pairs. For example, the following are equivalent:

```
>>> setp(lines, 'linewidth', 2, 'color', 'r')  # MATLAB␣
 ↪style
>>> setp(lines, linewidth=2, color='r')        # Python␣
 ↪style
```

**See also:**

***getp***

### Notes

---

**Note:** This is equivalent to *matplotlib.artist.setp*.

---

### Examples using `matplotlib.pyplot.setp`

- *Creating annotated heatmaps*
- *Boxplots*
- *Labeling a pie and a donut*
- *Patheffect Demo*
- *Set and get properties*
- *Topographic hillshading*
- *Evans test*
- *The Lifecycle of a Plot*

### matplotlib.pyplot.waitforbuttonpress

matplotlib.pyplot.**waitforbuttonpress**(*timeout=-1*)

Blocking call to interact with the figure.

Wait for user input and return True if a key was pressed, False if a mouse button was pressed and None if no input was given within *timeout* seconds. Negative values deactivate *timeout*.

#### Notes

**Note:** This is the *pyplot wrapper* for `Figure.waitforbuttonpress`.

### Examples using `matplotlib.pyplot.waitforbuttonpress`

- *Interactive functions*

### matplotlib.pyplot.xkcd

matplotlib.pyplot.**xkcd**(*scale=1*, *length=100*, *randomness=2*)

Turn on xkcd sketch-style drawing mode.

This will only have an effect on things drawn after this function is called.

For best results, install the xkcd script font; xkcd fonts are not packaged with Matplotlib.

**Parameters**

**scale**

[float, optional] The amplitude of the wiggle perpendicular to the source line.

**length**

[float, optional] The length of the wiggle along the line.

**randomness**

[float, optional] The scale factor by which the length is shrunken or expanded.

### Notes

This function works by a number of rcParams, so it will probably override others you have set before.

If you want the effects of this function to be temporary, it can be used as a context manager, for example:

```
with plt.xkcd():
    # This figure will be in XKCD-style
    fig1 = plt.figure()
    # ...

# This figure will be in regular style
fig2 = plt.figure()
```

### Examples using `matplotlib.pyplot.xkcd`

- *XKCD*

## 7.2.41 `matplotlib.projections`

Non-separable transforms that map from data space to screen space.

Projections are defined as *Axes* subclasses. They include the following elements:

- A transformation from data coordinates into display coordinates.
- An inverse of that transformation. This is used, for example, to convert mouse positions from screen space back into data space.
- Transformations for the gridlines, ticks and ticklabels. Custom projections will often need to place these elements in special locations, and Matplotlib has a facility to help with doing so.
- Setting up default values (overriding *cla*), since the defaults for a rectilinear Axes may not be appropriate.
- Defining the shape of the Axes, for example, an elliptical Axes, that will be used to draw the background of the plot and for clipping any data elements.
- Defining custom locators and formatters for the projection. For example, in a geographic projection, it may be more convenient to display the grid in degrees, even if the data is in radians.
- Set up interactive panning and zooming. This is left as an "advanced" feature left to the reader, but there is an example of this for polar plots in *matplotlib.projections.polar*.
- Any additional methods for additional convenience or features.

Once the projection Axes is defined, it can be used in one of two ways:

- By defining the class attribute `name`, the projection Axes can be registered with *matplotlib.projections.register_projection* and subsequently simply invoked by name:

```
fig.add_subplot(projection="my_proj_name")
```

- For more complex, parameterisable projections, a generic "projection" object may be defined which includes the method `_as_mpl_axes`. `_as_mpl_axes` should take no arguments and return the projection's Axes subclass and a dictionary of additional arguments to pass to the subclass' `__init__` method. Subsequently a parameterised projection can be initialised with:

```
fig.add_subplot(projection=MyProjection(param1=param1_value))
```

  where MyProjection is an object which implements a `_as_mpl_axes` method.

A full-fledged and heavily annotated example is in *Custom projection*. The polar plot functionality in *matplotlib.projections.polar* may also be of interest.

**class** matplotlib.projections.**ProjectionRegistry**

Bases: object

A mapping of registered projection names to projection classes.

**get_projection_class**(*name*)

Get a projection class from its *name*.

**get_projection_names**()

Return the names of all projections currently registered.

**register**(**projections*)

Register a new set of projections.

matplotlib.projections.**get_projection_class**(*projection=None*)

Get a projection class from its name.

If *projection* is None, a standard rectilinear projection is returned.

matplotlib.projections.**get_projection_names**()

Return the names of all projections currently registered.

matplotlib.projections.**register_projection**(*cls*)

## Built-in projections

Matplotlib has built-in support for polar and some geographic projections. See the following pages for more information:

### matplotlib.projections.polar

**class** matplotlib.projections.polar.**InvertedPolarTransform**(*axis=None*, *use_rmin=True*, ***, *apply_theta_transforms=True*)

Bases: *Transform*

The inverse of the polar transform, mapping Cartesian coordinate space *x* and *y* back to *theta* and *r*.

**Parameters**

**axis**

[*Axis*, optional] Axis associated with this transform. This is used to get the minimum radial limit.

**use_rmin**

[bool, optional] If `True`, add the minimum radial axis limit after transforming from Cartesian coordinates. *axis* must also be specified for this to take effect.

**has_inverse = True**

True if this transform has a corresponding inverse transform.

**input_dims = 2**

The number of input dimensions of this transform. Must be overridden (with integers) in the subclass.

**inverted**()

Return the corresponding inverse transformation.

It holds `x == self.inverted().transform(self.transform(x))`.

The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

**output_dims = 2**

The number of output dimensions of this transform. Must be overridden (with integers) in the subclass.

**transform_non_affine**(*values*)

Apply only the non-affine part of this transformation.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

**Parameters**

**values**

[array] The input values as an array of length *input_dims* or shape (N, *input_dims*).

**Returns**

**array**

The output values as an array of length *output_dims* or shape (N, *output_dims*), depending on the input.

**class** matplotlib.projections.polar.**PolarAffine**(*scale_transform*, *limits*)

Bases: *Affine2DBase*

The affine part of the polar projection.

Scales the output so that maximum radius rests on the edge of the Axes circle and the origin is mapped to (0.5, 0.5). The transform applied is the same to x and y components and given by:

$$x_1 = 0.5 \left[ \frac{x_0}{(r_{\max} - r_{\min})} + 1 \right]$$

$r_{\min}, r_{\max}$ are the minimum and maximum radial limits after any scaling (e.g. log scaling) has been removed.

**Parameters**

**scale_transform**

[*Transform*] Scaling transform for the data. This is used to remove any scaling from the radial view limits.

**limits**

[*BboxBase*] View limits of the data. The only part of its bounds that is used is the y limits (for the radius limits).

**get_matrix**()

Get the matrix for the affine part of this transform.

**class** matplotlib.projections.polar.**PolarAxes**(**args*, *theta_offset=0*, *theta_direction=1*, *rlabel_position=22.5*, ***kwargs*)

Bases: *Axes*

A polar graph projection, where the input dimensions are *theta*, *r*.

Theta starts pointing east and goes anti-clockwise.

Build an Axes in a figure.

**Parameters**

**fig**

[*Figure*] The Axes is built in the *Figure* *fig*.

***args**

`*args` can be a single `(left, bottom, width, height)` rectangle or a single *Bbox*. This specifies the rectangle (in figure coordinates) where the Axes is positioned.

`*args` can also consist of three numbers or a single three-digit number; in the latter case, the digits are considered as independent numbers. The numbers are interpreted as `(nrows, ncols, index)`: `(nrows, ncols)` specifies the

size of an array of subplots, and `index` is the 1-based index of the subplot being created. Finally, `*args` can also directly be a *SubplotSpec* instance.

**sharex, sharey**

[*Axes*, optional] The x- or y-*axis* is shared with the x- or y-axis in the input *Axes*.

**frameon**

[bool, default: True] Whether the Axes frame is visible.

**box_aspect**

[float, optional] Set a fixed aspect for the Axes box, i.e. the ratio of height to width. See *set_box_aspect* for details.

**forward_navigation_events**

[bool or "auto", default: "auto"] Control whether pan/zoom events are passed through to Axes below this one. "auto" is *True* for axes with an invisible patch and *False* otherwise.

****kwargs**

Other optional keyword arguments:

| Property | Description |
|---|---|
| *adjustable* | {'box', 'datalim'} |
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi v |
| *alpha* | scalar or None |
| *anchor* | (float, float) or {'C', 'SW', 'S', 'SE', 'E', 'NE', ...} |
| *animated* | bool |
| *aspect* | {'auto', 'equal'} or float |
| *autoscale_on* | bool |
| *autoscalex_on* | unknown |
| *autoscaley_on* | unknown |
| *axes_locator* | Callable[[Axes, Renderer], Bbox] |
| *axisbelow* | bool or 'line' |
| *box_aspect* | float or None |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *facecolor* or fc | *color* |
| *figure* | *Figure* |
| *forward_navigation_events* | bool or "auto" |
| *frame_on* | bool |
| *gid* | str |
| *in_layout* | bool |
| *label* | object |
| *mouseover* | bool |

Table 135 – continued fro

| Property | Description |
|---|---|
| navigate | bool |
| navigate_mode | unknown |
| path_effects | list of AbstractPathEffect |
| picker | None or bool or float or callable |
| position | [left, bottom, width, height] or Bbox |
| prop_cycle | Cycler |
| rasterization_zorder | float or None |
| rasterized | bool |
| sketch_params | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| subplotspec | unknown |
| title | str |
| transform | Transform |
| url | str |
| visible | bool |
| xbound | (lower: float, upper: float) |
| xlabel | str |
| xlim | (left: float, right: float) |
| xmargin | float greater than -0.5 |
| xscale | unknown |
| xticklabels | unknown |
| xticks | unknown |
| ybound | (lower: float, upper: float) |
| ylabel | str |
| ylim | (bottom: float, top: float) |
| ymargin | float greater than -0.5 |
| yscale | unknown |
| yticklabels | unknown |
| yticks | unknown |
| zorder | float |

**Returns**

***Axes***

The new Axes object.

**class InvertedPolarTransform**(*axis=None*, *use_rmin=True*, ***, *apply_theta_transforms=True*)

Bases: Transform

The inverse of the polar transform, mapping Cartesian coordinate space *x* and *y* back to *theta* and *r*.

**Parameters**

**axis**

[*Axis*, optional] Axis associated with this transform. This is used to get the minimum radial limit.

**use_rmin**

[bool, optional] If `True`, add the minimum radial axis limit after transforming from Cartesian coordinates. *axis* must also be specified for this to take effect.

**has_inverse = True**

True if this transform has a corresponding inverse transform.

**input_dims = 2**

The number of input dimensions of this transform. Must be overridden (with integers) in the subclass.

**inverted**()

Return the corresponding inverse transformation.

It holds `x == self.inverted().transform(self.transform(x))`.

The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

**output_dims = 2**

The number of output dimensions of this transform. Must be overridden (with integers) in the subclass.

**transform_non_affine**(*values*)

Apply only the non-affine part of this transformation.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

**Parameters**

**values**

[array] The input values as an array of length *input_dims* or shape (N, *input_dims*).

**Returns**

**array**

The output values as an array of length *output_dims* or shape (N, *output_dims*), depending on the input.

**class PolarAffine**(*scale_transform*, *limits*)

Bases: *Affine2DBase*

The affine part of the polar projection.

Scales the output so that maximum radius rests on the edge of the Axes circle and the origin is mapped to (0.5, 0.5). The transform applied is the same to x and y components and given by:

$$x_1 = 0.5 \left[ \frac{x_0}{(r_{\max} - r_{\min})} + 1 \right]$$

$r_{\min}, r_{\max}$ are the minimum and maximum radial limits after any scaling (e.g. log scaling) has been removed.

**Parameters**

**scale_transform**

[*Transform*] Scaling transform for the data. This is used to remove any scaling from the radial view limits.

**limits**

[*BboxBase*] View limits of the data. The only part of its bounds that is used is the y limits (for the radius limits).

**get_matrix**()

Get the matrix for the affine part of this transform.

**class PolarTransform**(*axis=None*, *use_rmin=True*, *, *apply_theta_transforms=True*, *scale_transform=None*)

Bases: *Transform*

The base polar transform.

This transform maps polar coordinates $\theta, r$ into Cartesian coordinates $x, y = r\cos(\theta), r\sin(\theta)$ (but does not fully transform into Axes coordinates or handle positioning in screen space).

This transformation is designed to be applied to data after any scaling along the radial axis (e.g. log-scaling) has been applied to the input data.

Path segments at a fixed radius are automatically transformed to circular arcs as long as `path._interpolation_steps > 1`.

**Parameters**

**axis**

[*Axis*, optional] Axis associated with this transform. This is used to get the minimum radial limit.

**use_rmin**

[`bool`, optional] If `True`, subtract the minimum radial axis limit before transforming to Cartesian coordinates. *axis* must also be specified for this to take effect.

**has_inverse = True**

True if this transform has a corresponding inverse transform.

**input_dims = 2**

The number of input dimensions of this transform. Must be overridden (with integers) in the subclass.

**inverted**()

Return the corresponding inverse transformation.

It holds `x == self.inverted().transform(self.transform(x))`.

The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

**output_dims = 2**

The number of output dimensions of this transform. Must be overridden (with integers) in the subclass.

**transform_non_affine**(*values*)

Apply only the non-affine part of this transformation.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

**Parameters**

**values**

[array] The input values as an array of length *input_dims* or shape (N, *input_dims*).

**Returns**

**array**

The output values as an array of length *output_dims* or shape (N, *output_dims*), depending on the input.

**transform_path_non_affine**(*path*)

Apply the non-affine part of this transform to *Path* *path*, returning a new *Path*.

`transform_path(path)` is equivalent to `transform_path_affine(transform_path_non_affine(values))`.

**class RadialLocator**(*base*, *axes=None*)

Bases: *Locator*

Used to locate radius ticks.

Ensures that all ticks are strictly positive. For all other tasks, it delegates to the base *Locator* (which may be different depending on the scale of the *r*-axis).

**nonsingular**(*vmin*, *vmax*)

Adjust a range as needed to avoid singularities.

This method gets called during autoscaling, with `(v0, v1)` set to the data limits on the Axes if the Axes contains any data, or `(-inf, +inf)` if not.

- If `v0 == v1` (possibly up to some floating point slop), this method returns an expanded interval around this value.
- If `(v0, v1) == (-inf, +inf)`, this method returns appropriate default view limits.
- Otherwise, `(v0, v1)` is returned without modification.

**set_axis**(*axis*)

**view_limits**(*vmin*, *vmax*)

Select a scale for the range from vmin to vmax.

Subclasses should override this method to change locator behaviour.

**class ThetaFormatter**

Bases: *Formatter*

Used to format the *theta* tick labels. Converts the native unit of radians into degrees and adds a degree symbol.

**class ThetaLocator**(*base*)

Bases: *Locator*

Used to locate theta ticks.

This will work the same as the base locator except in the case that the view spans the entire circle. In such cases, the previously used default locations of every 45 degrees are returned.

**set_axis**(*axis*)

**view_limits**(*vmin*, *vmax*)

Select a scale for the range from vmin to vmax.

Subclasses should override this method to change locator behaviour.

**can_pan**()

Return whether this Axes supports the pan/zoom button functionality.

For a polar Axes, this is slightly misleading. Both panning and zooming are performed by the same button. Panning is performed in azimuth while zooming is done along the radial.

**can_zoom**()

Return whether this Axes supports the zoom box button functionality.

A polar Axes does not support zoom boxes.

**clear**()

Clear the Axes.

**drag_pan**(*button*, *key*, *x*, *y*)

Called when the mouse moves during a pan operation.

**Parameters**

**button**

[*MouseButton*] The pressed mouse button.

**key**

[str or None] The pressed key, if any.

**x, y**

[float] The mouse coordinates in display coords.

### Notes

This is intended to be overridden by new projection types.

**draw**(*renderer*)

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

**Parameters**

**renderer**

[*RendererBase* subclass.]

### Notes

This method is overridden in the Artist subclasses.

**end_pan**()

Called when a pan operation completes (when the mouse button is up.)

### Notes

This is intended to be overridden by new projection types.

**format_coord**(*theta*, *r*)

Return a format string formatting the *x*, *y* coordinates.

**get_data_ratio**()

Return the aspect ratio of the data itself. For a polar plot, this should always be 1.0

**get_rlabel_position**()

**Returns**

**float**

The theta position of the radius labels in degrees.

**get_rmax**()

**Returns**

**float**

Outer radial limit.

**get_rmin**()

**Returns**

**float**

The inner radial limit.

**get_rorigin**()

**Returns**

**float**

**get_rsign**()

**get_theta_direction**()

Get the direction in which theta increases.

**-1:**

Theta increases in the clockwise direction

**1:**

Theta increases in the counterclockwise direction

**get_theta_offset**()

Get the offset for the location of 0 in radians.

**get_thetamax**()

Return the maximum theta limit in degrees.

**get_thetamin**()

Get the minimum theta limit in degrees.

**get_xaxis_text1_transform**(*pad*)

**Returns**

**transform**

[Transform] The transform used for drawing x-axis labels, which will add *pad_points* of padding (in points) between the axis and the label. The x-direction is in data coordinates and the y-direction is in axis coordinates

**valign**

[{'center', 'top', 'bottom', 'baseline', 'center_baseline'}] The text vertical alignment.

**halign**

[{'center', 'left', 'right'}] The text horizontal alignment.

### Notes

This transformation is primarily used by the *Axis* class, and is meant to be overridden by new kinds of projections that may need to place axis elements in different locations.

**get_xaxis_text2_transform**(*pad*)

**Returns**

**transform**

[Transform] The transform used for drawing secondary x-axis labels, which will add *pad_points* of padding (in points) between the axis and the label. The x-direction is in data coordinates and the y-direction is in axis coordinates

**valign**

[{'center', 'top', 'bottom', 'baseline', 'center_baseline'}] The text vertical alignment.

**halign**

[{'center', 'left', 'right'}] The text horizontal alignment.

### Notes

This transformation is primarily used by the *Axis* class, and is meant to be overridden by new kinds of projections that may need to place axis elements in different locations.

**get_xaxis_transform**(*which='grid'*)

Get the transformation used for drawing x-axis labels, ticks and gridlines. The x-direction is in data coordinates and the y-direction is in axis coordinates.

---

**Note:** This transformation is primarily used by the *Axis* class, and is meant to be overridden by new kinds of projections that may need to place axis elements in different locations.

---

**Parameters**

**which**

[{'grid', 'tick1', 'tick2'}]

**get_yaxis_text1_transform**(*pad*)

**Returns**

**transform**

[Transform] The transform used for drawing y-axis labels, which will add *pad_points* of padding (in points) between the axis and the label. The x-direction is in axis coordinates and the y-direction is in data coordinates

**valign**

[{'center', 'top', 'bottom', 'baseline', 'center_baseline'}] The text vertical alignment.

**halign**

[{'center', 'left', 'right'}] The text horizontal alignment.

### Notes

This transformation is primarily used by the *Axis* class, and is meant to be overridden by new kinds of projections that may need to place axis elements in different locations.

**get_yaxis_text2_transform**(*pad*)

**Returns**

**transform**

[Transform] The transform used for drawing secondart y-axis labels, which will add *pad_points* of padding (in points) between the axis and the label. The x-direction is in axis coordinates and the y-direction is in data coordinates

**valign**

[{'center', 'top', 'bottom', 'baseline', 'center_baseline'}] The text vertical alignment.

**halign**

[{'center', 'left', 'right'}] The text horizontal alignment.

### Notes

This transformation is primarily used by the *Axis* class, and is meant to be overridden by new kinds of projections that may need to place axis elements in different locations.

**get_yaxis_transform**(*which='grid'*)

Get the transformation used for drawing y-axis labels, ticks and gridlines. The x-direction is in axis coordinates and the y-direction is in data coordinates.

**Note:** This transformation is primarily used by the *Axis* class, and is meant to be overridden by new kinds of projections that may need to place axis elements in different locations.

**Parameters**

**which**

[{'grid', 'tick1', 'tick2'}]

**name = 'polar'**

**set**(*, *adjustable=<UNSET>*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *anchor=<UNSET>*, *animated=<UNSET>*, *aspect=<UNSET>*, *autoscale_on=<UNSET>*, *autoscalex_on=<UNSET>*, *autoscaley_on=<UNSET>*, *axes_locator=<UNSET>*, *axisbelow=<UNSET>*, *box_aspect=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *facecolor=<UNSET>*, *forward_navigation_events=<UNSET>*, *frame_on=<UNSET>*, *gid=<UNSET>*, *in_layout=<UNSET>*, *label=<UNSET>*, *mouseover=<UNSET>*, *navigate=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *position=<UNSET>*, *prop_cycle=<UNSET>*, *rasterization_zorder=<UNSET>*, *rasterized=<UNSET>*, *rgrids=<UNSET>*, *rlabel_position=<UNSET>*, *rlim=<UNSET>*, *rmax=<UNSET>*, *rmin=<UNSET>*, *rorigin=<UNSET>*, *rscale=<UNSET>*, *rticks=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *subplotspec=<UNSET>*, *theta_direction=<UNSET>*, *theta_offset=<UNSET>*, *theta_zero_location=<UNSET>*, *thetagrids=<UNSET>*, *thetalim=<UNSET>*, *thetamax=<UNSET>*, *thetamin=<UNSET>*, *title=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *xbound=<UNSET>*, *xlabel=<UNSET>*, *xlim=<UNSET>*, *xmargin=<UNSET>*, *xscale=<UNSET>*, *xticklabels=<UNSET>*, *xticks=<UNSET>*, *ybound=<UNSET>*, *ylabel=<UNSET>*, *ylim=<UNSET>*, *ymargin=<UNSET>*, *yscale=<UNSET>*, *yticklabels=<UNSET>*, *yticks=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *adjustable* | {'box', 'datalim'} |
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value |
| *alpha* | scalar or None |
| *anchor* | (float, float) or {'C', 'SW', 'S', 'SE', 'E', 'NE', ...} |
| *animated* | bool |
| *aspect* | {'auto', 'equal'} or float |
| *autoscale_on* | bool |
| *autoscalex_on* | unknown |
| *autoscaley_on* | unknown |
| *axes_locator* | Callable[[Axes, Renderer], Bbox] |
| *axisbelow* | bool or 'line' |

Table 136 – continued from pr

| Property | Description |
|---|---|
| box_aspect | float or None |
| clip_box | BboxBase or None |
| clip_on | bool |
| clip_path | Patch or (Path, Transform) or None |
| facecolor or fc | color |
| figure | Figure |
| forward_navigation_events | bool or "auto" |
| frame_on | bool |
| gid | str |
| in_layout | bool |
| label | object |
| mouseover | bool |
| navigate | bool |
| navigate_mode | unknown |
| path_effects | list of AbstractPathEffect |
| picker | None or bool or float or callable |
| position | [left, bottom, width, height] or Bbox |
| prop_cycle | Cycler |
| rasterization_zorder | float or None |
| rasterized | bool |
| rgrids | tuple with floats |
| rlabel_position | number |
| rlim | unknown |
| rmax | float |
| rmin | float |
| rorigin | float |
| rscale | unknown |
| rticks | unknown |
| sketch_params | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| subplotspec | unknown |
| theta_direction | unknown |
| theta_offset | unknown |
| theta_zero_location | str |
| thetagrids | tuple with floats, degrees |
| thetalim | unknown |
| thetamax | unknown |
| thetamin | unknown |
| title | str |
| transform | Transform |
| url | str |
| visible | bool |
| xbound | (lower: float, upper: float) |
| xlabel | str |

Table 136 – continued from pr

| Property | Description |
|---|---|
| *xlim* | (left: float, right: float) |
| *xmargin* | float greater than -0.5 |
| *xscale* | unknown |
| *xticklabels* | unknown |
| *xticks* | unknown |
| *ybound* | (lower: float, upper: float) |
| *ylabel* | str |
| *ylim* | (bottom: float, top: float) |
| *ymargin* | float greater than -0.5 |
| *yscale* | unknown |
| *yticklabels* | unknown |
| *yticks* | unknown |
| *zorder* | float |

**set_rgrids**(*radii*, *labels=None*, *angle=None*, *fmt=None*, ***kwargs*)

Set the radial gridlines on a polar plot.

**Parameters**

**radii**

[tuple with floats] The radii for the radial gridlines

**labels**

[tuple with strings or None] The labels to use at each radial gridline. The *matplotlib.ticker.ScalarFormatter* will be used if None.

**angle**

[float] The angular position of the radius labels in degrees.

**fmt**

[str or None] Format string used in *matplotlib.ticker.FormatStrFormatter*. For example '%f'.

**Returns**

**lines**

[list of *lines.Line2D*] The radial gridlines.

**labels**

[list of *text.Text*] The tick labels.

**Other Parameters**

****kwargs**

*kwargs* are optional `Text` properties for the labels.

> **Warning:** This only sets the properties of the current ticks. Ticks are not guaranteed to be persistent. Various operations can create, delete and modify the Tick instances. There is an imminent risk that these settings can get lost if you work on the figure further (including also panning/zooming on a displayed figure).
>
> Use `set_tick_params` instead if possible.

**See also:**

***PolarAxes.set_thetagrids***
***Axis.get_gridlines***
***Axis.get_ticklabels***

**set_rlabel_position**(*value*)

Update the theta position of the radius labels.

**Parameters**

**value**

[number] The angular position of the radius labels in degrees.

**set_rlim**(*bottom=None*, *top=None*, ***, *emit=True*, *auto=False*, ***kwargs*)

Set the radial axis view limits.

This function behaves like `Axes.set_ylim`, but additionally supports *rmin* and *rmax* as aliases for *bottom* and *top*.

**See also:**

***Axes.set_ylim***

**set_rmax**(*rmax*)

Set the outer radial limit.

**Parameters**

**rmax**

[float]

**set_rmin**(*rmin*)

Set the inner radial limit.

**Parameters**

**rmin**

[float]

**set_rorigin**(*rorigin*)

Update the radial origin.

**Parameters**

**rorigin**

[float]

**set_rscale**(**args*, ***kwargs*)

**set_rticks**(**args*, ***kwargs*)

**set_theta_direction**(*direction*)

Set the direction in which theta increases.

**clockwise, -1:**

Theta increases in the clockwise direction

**counterclockwise, anticlockwise, 1:**

Theta increases in the counterclockwise direction

**set_theta_offset**(*offset*)

Set the offset for the location of 0 in radians.

**set_theta_zero_location**(*loc*, *offset=0.0*)

Set the location of theta's zero.

This simply calls *set_theta_offset* with the correct value in radians.

**Parameters**

**loc**

[str] May be one of "N", "NW", "W", "SW", "S", "SE", "E", or "NE".

**offset**

[float, default: 0] An offset in degrees to apply from the specified *loc*. **Note:** this offset is *always* applied counter-clockwise regardless of the direction setting.

**set_thetagrids**(*angles*, *labels=None*, *fmt=None*, ***kwargs*)

Set the theta gridlines in a polar plot.

**Parameters**

**angles**

[tuple with floats, degrees] The angles of the theta gridlines.

**labels**

[tuple with strings or None] The labels to use at each theta gridline. The *projections.polar.ThetaFormatter* will be used if None.

**fmt**

[str or None] Format string used in *matplotlib.ticker.FormatStrFormatter*. For example '%f'. Note that the angle that is used is in radians.

**Returns**

**lines**

[list of *lines.Line2D*] The theta gridlines.

**labels**

[list of *text.Text*] The tick labels.

**Other Parameters**

****kwargs**

*kwargs* are optional *Text* properties for the labels.

> **Warning:** This only sets the properties of the current ticks. Ticks are not guaranteed to be persistent. Various operations can create, delete and modify the Tick instances. There is an imminent risk that these settings can get lost if you work on the figure further (including also panning/zooming on a displayed figure).
>
> Use *set_tick_params* instead if possible.

**See also:**

***PolarAxes.set_rgrids***
***Axis.get_gridlines***
***Axis.get_ticklabels***

**set_thetalim**(**args*, ***kwargs*)

Set the minimum and maximum theta values.

Can take the following signatures:

- `set_thetalim(minval, maxval)`: Set the limits in radians.
- `set_thetalim(thetamin=minval, thetamax=maxval)`: Set the limits in degrees.

where minval and maxval are the minimum and maximum limits. Values are wrapped in to the range $[0, 2\pi]$ (in radians), so for example it is possible to do `set_thetalim(-np.pi / 2, np.pi / 2)` to have an axis symmetric around 0. A ValueError is raised if the absolute angle difference is larger than a full circle.

**set_thetamax**(*thetamax*)

Set the maximum theta limit in degrees.

**set_thetamin**(*thetamin*)

Set the minimum theta limit in degrees.

**set_yscale**(**args*, ***kwargs*)

Set the yaxis' scale.

**Parameters**

**value**

[{"linear", "log", "symlog", "logit", ...} or *ScaleBase*] The axis scale type to apply.

****kwargs**

Different keyword arguments are accepted, depending on the scale. See the respective class keyword arguments:

- *matplotlib.scale.LinearScale*
- *matplotlib.scale.LogScale*
- *matplotlib.scale.SymmetricalLogScale*
- *matplotlib.scale.LogitScale*
- *matplotlib.scale.FuncScale*
- *matplotlib.scale.AsinhScale*

## Notes

By default, Matplotlib supports the above-mentioned scales. Additionally, custom scales may be registered using *matplotlib.scale.register_scale*. These scales can then also be used here.

**start_pan**(*x*, *y*, *button*)

Called when a pan operation has started.

**Parameters**

**x, y**

[float] The mouse coordinates in display coords.

**button**

[*MouseButton*] The pressed mouse button.

### Notes

This is intended to be overridden by new projection types.

**class** matplotlib.projections.polar.**PolarTransform**(*axis=None*, *use_rmin=True*, *, *apply_theta_transforms=True*, *scale_transform=None*)

Bases: *Transform*

The base polar transform.

This transform maps polar coordinates $\theta, r$ into Cartesian coordinates $x, y = r\cos(\theta), r\sin(\theta)$ (but does not fully transform into Axes coordinates or handle positioning in screen space).

This transformation is designed to be applied to data after any scaling along the radial axis (e.g. log-scaling) has been applied to the input data.

Path segments at a fixed radius are automatically transformed to circular arcs as long as `path._interpolation_steps > 1`.

**Parameters**

**axis**

[*Axis*, optional] Axis associated with this transform. This is used to get the minimum radial limit.

**use_rmin**

[`bool`, optional] If `True`, subtract the minimum radial axis limit before transforming to Cartesian coordinates. *axis* must also be specified for this to take effect.

**has_inverse = True**

True if this transform has a corresponding inverse transform.

**input_dims = 2**

The number of input dimensions of this transform. Must be overridden (with integers) in the subclass.

**inverted**()

Return the corresponding inverse transformation.

It holds `x == self.inverted().transform(self.transform(x))`.

The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

**output_dims = 2**

The number of output dimensions of this transform. Must be overridden (with integers) in the subclass.

**transform_non_affine**(*values*)

Apply only the non-affine part of this transformation.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

**Parameters**

**values**

[array] The input values as an array of length *input_dims* or shape (N, *input_dims*).

**Returns**

**array**

The output values as an array of length *output_dims* or shape (N, *output_dims*), depending on the input.

**transform_path_non_affine**(*path*)

Apply the non-affine part of this transform to *Path* *path*, returning a new *Path*.

`transform_path(path)` is equivalent to `transform_path_affine(transform_path_non_affi`

**class** matplotlib.projections.polar.**RadialAxis**(**args*, ***kwargs*)

Bases: *YAxis*

A radial Axis.

This overrides certain properties of a *YAxis* to provide special-casing for a radial axis.

**Parameters**

**axes**

[*Axes*] The *Axes* to which the created Axis belongs.

**pickradius**

[float] The acceptance radius for containment tests. See also *Axis.contains*.

**clear**

[bool, default: True] Whether to clear the Axis on creation. This is not required, e.g., when creating an Axis as part of an Axes, as `Axes.clear` will call `Axis.clear`. .. versionadded:: 3.8

**axis_name = 'radius'**

Read-only name identifying the axis.

**clear**()

Clear the axis.

This resets axis properties to their default values:

- the label
- the scale
- locators, formatters and ticks
- major and minor grid
- units
- registered callbacks

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *data_interval=<UNSET>*, *gid=<UNSET>*, *in_layout=<UNSET>*, *inverted=<UNSET>*, *label=<UNSET>*, *label_coords=<UNSET>*, *label_position=<UNSET>*, *label_text=<UNSET>*, *major_formatter=<UNSET>*, *major_locator=<UNSET>*, *minor_formatter=<UNSET>*, *minor_locator=<UNSET>*, *mouseover=<UNSET>*, *offset_position=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *pickradius=<UNSET>*, *rasterized=<UNSET>*, *remove_overlapping_locs=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *tick_params=<UNSET>*, *ticklabels=<UNSET>*, *ticks=<UNSET>*, *ticks_position=<UNSET>*, *transform=<UNSET>*, *units=<UNSET>*, *url=<UNSET>*, *view_interval=<UNSET>*, *visible=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, a |
| *alpha* | scalar or None |
| *animated* | bool |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *data_interval* | unknown |
| *figure* | *Figure* |
| *gid* | str |
| *in_layout* | bool |
| *inverted* | unknown |
| *label* | object |
| *label_coords* | unknown |
| *label_position* | {'left', 'right'} |

Table 137 – continued from pre

| Property | Description |
|---|---|
| *label_text* | str |
| *major_formatter* | *Formatter*, `str`, or function |
| *major_locator* | *Locator* |
| *minor_formatter* | *Formatter*, `str`, or function |
| *minor_locator* | *Locator* |
| *mouseover* | bool |
| *offset_position* | {'left', 'right'} |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *pickradius* | float |
| *rasterized* | bool |
| *remove_overlapping_locs* | unknown |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *tick_params* | unknown |
| *ticklabels* | sequence of str or of *Text*s |
| *ticks* | 1D array-like |
| *ticks_position* | {'left', 'right', 'both', 'default', 'none'} |
| *transform* | *Transform* |
| *units* | units tag |
| *url* | str |
| *view_interval* | unknown |
| *visible* | bool |
| *zorder* | float |

**class** matplotlib.projections.polar.**RadialLocator**(*base*, *axes=None*)

Bases: *Locator*

Used to locate radius ticks.

Ensures that all ticks are strictly positive. For all other tasks, it delegates to the base *Locator* (which may be different depending on the scale of the *r*-axis).

**nonsingular**(*vmin*, *vmax*)

Adjust a range as needed to avoid singularities.

This method gets called during autoscaling, with `(v0, v1)` set to the data limits on the Axes if the Axes contains any data, or `(-inf, +inf)` if not.

- If `v0 == v1` (possibly up to some floating point slop), this method returns an expanded interval around this value.
- If `(v0, v1) == (-inf, +inf)`, this method returns appropriate default view limits.
- Otherwise, `(v0, v1)` is returned without modification.

**set_axis**(*axis*)

**view_limits**(*vmin*, *vmax*)

Select a scale for the range from vmin to vmax.

Subclasses should override this method to change locator behaviour.

**class** matplotlib.projections.polar.**RadialTick**(**args*, ***kwargs*)

Bases: *YTick*

A radial-axis tick.

This subclass of *YTick* provides radial ticks with some small modification to their re-positioning such that ticks are rotated based on axes limits. This results in ticks that are correctly perpendicular to the spine. Labels are also rotated to be perpendicular to the spine, when 'auto' rotation is enabled.

bbox is the Bound2D bounding box in display coords of the Axes loc is the tick location in data coords size is the tick size in points

**set**(***, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *gid=<UNSET>*, *in_layout=<UNSET>*, *label=<UNSET>*, *label1=<UNSET>*, *label2=<UNSET>*, *mouseover=<UNSET>*, *pad=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| agg_fil | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image |
| alpha | scalar or None |
| animated | bool |
| clip_bo | BboxBase or None |
| clip_on | bool |
| clip_pa | Patch or (Path, Transform) or None |
| figure | Figure |
| gid | str |
| in_layo | bool |
| label | unknown |
| label1 | unknown |
| label2 | unknown |
| mouseov | bool |
| pad | float |
| path_ef | list of AbstractPathEffect |
| picker | None or bool or float or callable |
| rasterized | bool |
| sketch_ | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| transform | Transform |
| url | str |
| visible | bool |
| zorder | float |

**update_position**(*loc*)

Set the location of tick in data coords with scalar *loc*.

**class** matplotlib.projections.polar.**ThetaAxis**(**args*, ***kwargs*)

Bases: XAxis

A theta Axis.

This overrides certain properties of an XAxis to provide special-casing for an angular axis.

**Parameters**

**axes**

[Axes] The Axes to which the created Axis belongs.

**pickradius**

[float] The acceptance radius for containment tests. See also *Axis.contains*.

**clear**

[bool, default: True] Whether to clear the Axis on creation. This is not required, e.g., when creating an Axis as part of an Axes, as `Axes.clear` will call `Axis.clear`. .. versionadded:: 3.8

**axis_name = 'theta'**

Read-only name identifying the axis.

**clear**()

Clear the axis.

This resets axis properties to their default values:

- the label
- the scale
- locators, formatters and ticks
- major and minor grid
- units
- registered callbacks

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *data_interval=<UNSET>*, *gid=<UNSET>*, *in_layout=<UNSET>*, *inverted=<UNSET>*, *label=<UNSET>*, *label_coords=<UNSET>*, *label_position=<UNSET>*, *label_text=<UNSET>*, *major_formatter=<UNSET>*, *major_locator=<UNSET>*, *minor_formatter=<UNSET>*, *minor_locator=<UNSET>*, *mouseover=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *pickradius=<UNSET>*, *rasterized=<UNSET>*, *remove_overlapping_locs=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *tick_params=<UNSET>*, *ticklabels=<UNSET>*, *ticks=<UNSET>*, *ticks_position=<UNSET>*, *transform=<UNSET>*, *units=<UNSET>*, *url=<UNSET>*, *view_interval=<UNSET>*, *visible=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, a |
| *alpha* | scalar or None |
| *animated* | bool |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |

Table 138 – continued from pre

| Property | Description |
|---|---|
| *data_interval* | unknown |
| *figure* | *Figure* |
| *gid* | str |
| *in_layout* | bool |
| *inverted* | unknown |
| *label* | object |
| *label_coords* | unknown |
| *label_position* | {'top', 'bottom'} |
| *label_text* | str |
| *major_formatter* | *Formatter*, `str`, or function |
| *major_locator* | *Locator* |
| *minor_formatter* | *Formatter*, `str`, or function |
| *minor_locator* | *Locator* |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *pickradius* | float |
| *rasterized* | bool |
| *remove_overlapping_locs* | unknown |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *tick_params* | unknown |
| *ticklabels* | sequence of str or of *Text*s |
| *ticks* | 1D array-like |
| *ticks_position* | {'top', 'bottom', 'both', 'default', 'none'} |
| *transform* | *Transform* |
| *units* | units tag |
| *url* | str |
| *view_interval* | unknown |
| *visible* | bool |
| *zorder* | float |

**class** matplotlib.projections.polar.**ThetaFormatter**

Bases: *Formatter*

Used to format the *theta* tick labels. Converts the native unit of radians into degrees and adds a degree symbol.

**class** matplotlib.projections.polar.**ThetaLocator**(*base*)

Bases: *Locator*

Used to locate theta ticks.

This will work the same as the base locator except in the case that the view spans the entire circle. In such cases, the previously used default locations of every 45 degrees are returned.

**set_axis**(*axis*)

**view_limits**(*vmin*, *vmax*)

Select a scale for the range from vmin to vmax.

Subclasses should override this method to change locator behaviour.

**class** matplotlib.projections.polar.**ThetaTick**(*axes*, **args*, ***kwargs*)

Bases: *XTick*

A theta-axis tick.

This subclass of *XTick* provides angular ticks with some small modification to their re-positioning such that ticks are rotated based on tick location. This results in ticks that are correctly perpendicular to the arc spine.

When 'auto' rotation is enabled, labels are also rotated to be parallel to the spine. The label padding is also applied here since it's not possible to use a generic axes transform to produce tick-specific padding.

bbox is the Bound2D bounding box in display coords of the Axes loc is the tick location in data coords size is the tick size in points

**set**(***, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *gid=<UNSET>*, *in_layout=<UNSET>*, *label=<UNSET>*, *label1=<UNSET>*, *label2=<UNSET>*, *mouseover=<UNSET>*, *pad=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| agg_fil | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image |
| alpha | scalar or None |
| animated | bool |
| clip_bo | BboxBase or None |
| clip_on | bool |
| clip_pa | Patch or (Path, Transform) or None |
| figure | Figure |
| gid | str |
| in_layo | bool |
| label | unknown |
| label1 | unknown |
| label2 | unknown |
| mouseov | bool |
| pad | float |
| path_ef | list of AbstractPathEffect |
| picker | None or bool or float or callable |
| rasterized | bool |
| sketch_ | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| transform | Transform |
| url | str |
| visible | bool |
| zorder | float |

**update_position**(*loc*)

Set the location of tick in data coords with scalar *loc*.

## matplotlib.projections.geo

**class** matplotlib.projections.geo.**AitoffAxes**(**args*, ***kwargs*)

Bases: GeoAxes

Build an Axes in a figure.

**Parameters**

**fig**

[*Figure*] The Axes is built in the *Figure* *fig*.

***args**

`*args` can be a single `(left, bottom, width, height)` rectangle or a single *Bbox*. This specifies the rectangle (in figure coordinates) where the Axes is positioned.

`*args` can also consist of three numbers or a single three-digit number; in the latter case, the digits are considered as independent numbers. The numbers are interpreted as `(nrows, ncols, index)`: `(nrows, ncols)` specifies the size of an array of subplots, and `index` is the 1-based index of the subplot being created. Finally, `*args` can also directly be a *SubplotSpec* instance.

**sharex, sharey**

[*Axes*, optional] The x- or y-*axis* is shared with the x- or y-axis in the input *Axes*.

**frameon**

[bool, default: True] Whether the Axes frame is visible.

**box_aspect**

[float, optional] Set a fixed aspect for the Axes box, i.e. the ratio of height to width. See *set_box_aspect* for details.

**forward_navigation_events**

[bool or "auto", default: "auto"] Control whether pan/zoom events are passed through to Axes below this one. "auto" is *True* for axes with an invisible patch and *False* otherwise.

****kwargs**

Other optional keyword arguments:

| Property | Description |
|---|---|
| *adjustable* | {'box', 'datalim'} |
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi v |
| *alpha* | scalar or None |
| *anchor* | (float, float) or {'C', 'SW', 'S', 'SE', 'E', 'NE', ...} |
| *animated* | bool |
| *aspect* | {'auto', 'equal'} or float |
| *autoscale_on* | bool |
| *autoscalex_on* | unknown |
| *autoscaley_on* | unknown |
| *axes_locator* | Callable[[Axes, Renderer], Bbox] |
| *axisbelow* | bool or 'line' |
| *box_aspect* | float or None |
| *clip_box* | *BboxBase* or None |

Table 139 – continued fro

| Property | Description |
|---|---|
| clip_on | bool |
| clip_path | Patch or (Path, Transform) or None |
| facecolor or fc | color |
| figure | Figure |
| forward_navigation_events | bool or "auto" |
| frame_on | bool |
| gid | str |
| in_layout | bool |
| label | object |
| mouseover | bool |
| navigate | bool |
| navigate_mode | unknown |
| path_effects | list of AbstractPathEffect |
| picker | None or bool or float or callable |
| position | [left, bottom, width, height] or Bbox |
| prop_cycle | Cycler |
| rasterization_zorder | float or None |
| rasterized | bool |
| sketch_params | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| subplotspec | unknown |
| title | str |
| transform | Transform |
| url | str |
| visible | bool |
| xbound | (lower: float, upper: float) |
| xlabel | str |
| xlim | (left: float, right: float) |
| xmargin | float greater than -0.5 |
| xscale | unknown |
| xticklabels | unknown |
| xticks | unknown |
| ybound | (lower: float, upper: float) |
| ylabel | str |
| ylim | (bottom: float, top: float) |
| ymargin | float greater than -0.5 |
| yscale | unknown |
| yticklabels | unknown |
| yticks | unknown |
| zorder | float |

**Returns**

***Axes***

The new *Axes* object.

**class AitoffTransform**(*resolution*)

Bases: _GeoTransform

The base Aitoff transform.

Create a new geographical transform.

Resolution is the number of steps to interpolate between each input line segment to approximate its path in curved space.

**has_inverse = True**

True if this transform has a corresponding inverse transform.

**inverted**()

Return the corresponding inverse transformation.

It holds `x == self.inverted().transform(self.transform(x))`.

The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

**transform_non_affine**(*values*)

Apply only the non-affine part of this transformation.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

**Parameters**

**values**

[array] The input values as an array of length `input_dims` or shape (N, `input_dims`).

**Returns**

**array**

The output values as an array of length `output_dims` or shape (N, `output_dims`), depending on the input.

**class InvertedAitoffTransform**(*resolution*)

Bases: _GeoTransform

Create a new geographical transform.

Resolution is the number of steps to interpolate between each input line segment to approximate its path in curved space.

**has_inverse = True**

True if this transform has a corresponding inverse transform.

**inverted**()

Return the corresponding inverse transformation.

It holds `x == self.inverted().transform(self.transform(x))`.

The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

**transform_non_affine**(*values*)

Apply only the non-affine part of this transformation.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

**Parameters**

**values**

[array] The input values as an array of length `input_dims` or shape (N, `input_dims`).

**Returns**

**array**

The output values as an array of length `output_dims` or shape (N, `output_dims`), depending on the input.

**name = 'aitoff'**

**set**(*, *adjustable=<UNSET>*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *anchor=<UNSET>*, *animated=<UNSET>*, *aspect=<UNSET>*, *autoscale_on=<UNSET>*, *autoscalex_on=<UNSET>*, *autoscaley_on=<UNSET>*, *axes_locator=<UNSET>*, *axisbelow=<UNSET>*, *box_aspect=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *facecolor=<UNSET>*, *forward_navigation_events=<UNSET>*, *frame_on=<UNSET>*, *gid=<UNSET>*, *in_layout=<UNSET>*, *label=<UNSET>*, *latitude_grid=<UNSET>*, *longitude_grid=<UNSET>*, *longitude_grid_ends=<UNSET>*, *mouseover=<UNSET>*, *navigate=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *position=<UNSET>*, *prop_cycle=<UNSET>*, *rasterization_zorder=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *subplotspec=<UNSET>*, *title=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *xbound=<UNSET>*, *xlabel=<UNSET>*, *xlim=<UNSET>*, *xmargin=<UNSET>*, *xscale=<UNSET>*, *xticklabels=<UNSET>*, *xticks=<UNSET>*, *ybound=<UNSET>*, *ylabel=<UNSET>*, *ylim=<UNSET>*, *ymargin=<UNSET>*, *yscale=<UNSET>*, *yticklabels=<UNSET>*, *yticks=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| adjustable | {'box', 'datalim'} |
| agg_filter | a filter function, which takes a (m, n, 3) float array and a dpi value |
| alpha | scalar or None |
| anchor | (float, float) or {'C', 'SW', 'S', 'SE', 'E', 'NE', ...} |
| animated | bool |
| aspect | {'auto', 'equal'} or float |
| autoscale_on | bool |
| autoscalex_on | unknown |
| autoscaley_on | unknown |
| axes_locator | Callable[[Axes, Renderer], Bbox] |
| axisbelow | bool or 'line' |
| box_aspect | float or None |
| clip_box | BboxBase or None |
| clip_on | bool |
| clip_path | Patch or (Path, Transform) or None |
| facecolor or fc | color |
| figure | Figure |
| forward_navigation_events | bool or "auto" |
| frame_on | bool |
| gid | str |
| in_layout | bool |
| label | object |
| latitude_grid | unknown |
| longitude_grid | unknown |
| longitude_grid_ends | unknown |
| mouseover | bool |
| navigate | bool |
| navigate_mode | unknown |
| path_effects | list of AbstractPathEffect |
| picker | None or bool or float or callable |
| position | [left, bottom, width, height] or Bbox |
| prop_cycle | Cycler |
| rasterization_zorder | float or None |
| rasterized | bool |
| sketch_params | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| subplotspec | unknown |
| title | str |
| transform | Transform |
| url | str |
| visible | bool |
| xbound | (lower: float, upper: float) |
| xlabel | str |
| xlim | unknown |
| xmargin | float greater than -0.5 |

Table 140 – continued from pr

| Property | Description |
|---|---|
| *xscale* | unknown |
| *xticklabels* | unknown |
| *xticks* | unknown |
| *ybound* | (lower: float, upper: float) |
| *ylabel* | str |
| *ylim* | unknown |
| *ymargin* | float greater than -0.5 |
| *yscale* | unknown |
| *yticklabels* | unknown |
| *yticks* | unknown |
| *zorder* | float |

**class** matplotlib.projections.geo.**GeoAxes**(*fig*, **args*, *facecolor=None*, *frameon=True*, *sharex=None*, *sharey=None*, *label=''*, *xscale=None*, *yscale=None*, *box_aspect=None*, *forward_navigation_events='auto'*, ***kwargs*)

Bases: *Axes*

An abstract base class for geographic projections.

Build an Axes in a figure.

**Parameters**

**fig**

[*Figure*] The Axes is built in the *Figure* *fig*.

***args**

`*args` can be a single `(left, bottom, width, height)` rectangle or a single *Bbox*. This specifies the rectangle (in figure coordinates) where the Axes is positioned.

`*args` can also consist of three numbers or a single three-digit number; in the latter case, the digits are considered as independent numbers. The numbers are interpreted as `(nrows, ncols, index)`: `(nrows, ncols)` specifies the size of an array of subplots, and `index` is the 1-based index of the subplot being created. Finally, `*args` can also directly be a *SubplotSpec* instance.

**sharex, sharey**

[*Axes*, optional] The x- or y-*axis* is shared with the x- or y-axis in the input *Axes*.

**frameon**

[bool, default: True] Whether the Axes frame is visible.

**box_aspect**

[float, optional] Set a fixed aspect for the Axes box, i.e. the ratio of height to width. See *set_box_aspect* for details.

**forward_navigation_events**

[bool or "auto", default: "auto"] Control whether pan/zoom events are passed through to Axes below this one. "auto" is *True* for axes with an invisible patch and *False* otherwise.

****kwargs**

Other optional keyword arguments:

| Property | Description |
|---|---|
| *adjustable* | {'box', 'datalim'} |
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi v |
| *alpha* | scalar or None |
| *anchor* | (float, float) or {'C', 'SW', 'S', 'SE', 'E', 'NE', ...} |
| *animated* | bool |
| *aspect* | {'auto', 'equal'} or float |
| *autoscale_on* | bool |
| *autoscalex_on* | unknown |
| *autoscaley_on* | unknown |
| *axes_locator* | Callable[[Axes, Renderer], Bbox] |
| *axisbelow* | bool or 'line' |
| *box_aspect* | float or None |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *facecolor* or fc | *color* |
| *figure* | *Figure* |
| *forward_navigation_events* | bool or "auto" |
| *frame_on* | bool |
| *gid* | str |
| *in_layout* | bool |
| *label* | object |
| *mouseover* | bool |
| *navigate* | bool |
| *navigate_mode* | unknown |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *position* | [left, bottom, width, height] or *Bbox* |
| *prop_cycle* | *Cycler* |

Table 141 – continued fro

| Property | Description |
|---|---|
| *rasterization_zorder* | float or None |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *subplotspec* | unknown |
| *title* | str |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *xbound* | (lower: float, upper: float) |
| *xlabel* | str |
| *xlim* | (left: float, right: float) |
| *xmargin* | float greater than -0.5 |
| *xscale* | unknown |
| *xticklabels* | unknown |
| *xticks* | unknown |
| *ybound* | (lower: float, upper: float) |
| *ylabel* | str |
| *ylim* | (bottom: float, top: float) |
| *ymargin* | float greater than -0.5 |
| *yscale* | unknown |
| *yticklabels* | unknown |
| *yticks* | unknown |
| *zorder* | float |

**Returns**

***Axes***

The new *Axes* object.

**RESOLUTION = 75**

**class ThetaFormatter**(*round_to=1.0*)

Bases: *Formatter*

Used to format the theta tick labels. Converts the native unit of radians into degrees and adds a degree symbol.

**can_pan**()

Return whether this Axes supports the pan/zoom button functionality.

This Axes object does not support interactive pan/zoom.

**can_zoom**()

Return whether this Axes supports the zoom box button functionality.

This Axes object does not support interactive zoom box.

**clear**()

Clear the Axes.

**drag_pan**(*button*, *key*, *x*, *y*)

Called when the mouse moves during a pan operation.

**Parameters**

**button**

[*MouseButton*] The pressed mouse button.

**key**

[str or None] The pressed key, if any.

**x, y**

[float] The mouse coordinates in display coords.

### Notes

This is intended to be overridden by new projection types.

**end_pan**()

Called when a pan operation completes (when the mouse button is up.)

### Notes

This is intended to be overridden by new projection types.

**format_coord**(*lon*, *lat*)

Return a format string formatting the coordinate.

**get_data_ratio**()

Return the aspect ratio of the data itself.

**get_xaxis_text1_transform**(*pad*)

**Returns**

**transform**

[Transform] The transform used for drawing x-axis labels, which will add *pad_points* of padding (in points) between the axis and the label. The x-direction is in data coordinates and the y-direction is in axis coordinates

**valign**

[{'center', 'top', 'bottom', 'baseline', 'center_baseline'}] The text vertical alignment.

**halign**

[{'center', 'left', 'right'}] The text horizontal alignment.

### Notes

This transformation is primarily used by the *Axis* class, and is meant to be overridden by new kinds of projections that may need to place axis elements in different locations.

**get_xaxis_text2_transform**(*pad*)

**Returns**

**transform**

[Transform] The transform used for drawing secondary x-axis labels, which will add *pad_points* of padding (in points) between the axis and the label. The x-direction is in data coordinates and the y-direction is in axis coordinates

**valign**

[{'center', 'top', 'bottom', 'baseline', 'center_baseline'}] The text vertical alignment.

**halign**

[{'center', 'left', 'right'}] The text horizontal alignment.

### Notes

This transformation is primarily used by the *Axis* class, and is meant to be overridden by new kinds of projections that may need to place axis elements in different locations.

**get_xaxis_transform**(*which='grid'*)

Get the transformation used for drawing x-axis labels, ticks and gridlines. The x-direction is in data coordinates and the y-direction is in axis coordinates.

---

**Note:** This transformation is primarily used by the *Axis* class, and is meant to be overridden by new kinds of projections that may need to place axis elements in different locations.

---

**Parameters**

**which**

[{'grid', 'tick1', 'tick2'}]

**get_yaxis_text1_transform**(*pad*)

**Returns**

**transform**

[Transform] The transform used for drawing y-axis labels, which will add *pad_points* of padding (in points) between the axis and the label. The x-direction is in axis coordinates and the y-direction is in data coordinates

**valign**

[{'center', 'top', 'bottom', 'baseline', 'center_baseline'}] The text vertical alignment.

**halign**

[{'center', 'left', 'right'}] The text horizontal alignment.

### Notes

This transformation is primarily used by the *Axis* class, and is meant to be overridden by new kinds of projections that may need to place axis elements in different locations.

**get_yaxis_text2_transform**(*pad*)

**Returns**

**transform**

[Transform] The transform used for drawing secondart y-axis labels, which will add *pad_points* of padding (in points) between the axis and the label. The x-direction is in axis coordinates and the y-direction is in data coordinates

**valign**

[{'center', 'top', 'bottom', 'baseline', 'center_baseline'}] The text vertical alignment.

**halign**

[{'center', 'left', 'right'}] The text horizontal alignment.

### Notes

This transformation is primarily used by the *Axis* class, and is meant to be overridden by new kinds of projections that may need to place axis elements in different locations.

**get_yaxis_transform**(*which='grid'*)

Get the transformation used for drawing y-axis labels, ticks and gridlines. The x-direction is in axis coordinates and the y-direction is in data coordinates.

**Note:** This transformation is primarily used by the *Axis* class, and is meant to be overridden by new kinds of projections that may need to place axis elements in different locations.

**Parameters**

**which**

[{'grid', 'tick1', 'tick2'}]

**set** (*, *adjustable=<UNSET>*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *anchor=<UNSET>*, *animated=<UNSET>*, *aspect=<UNSET>*, *autoscale_on=<UNSET>*, *autoscalex_on=<UNSET>*, *autoscaley_on=<UNSET>*, *axes_locator=<UNSET>*, *axisbelow=<UNSET>*, *box_aspect=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *facecolor=<UNSET>*, *forward_navigation_events=<UNSET>*, *frame_on=<UNSET>*, *gid=<UNSET>*, *in_layout=<UNSET>*, *label=<UNSET>*, *latitude_grid=<UNSET>*, *longitude_grid=<UNSET>*, *longitude_grid_ends=<UNSET>*, *mouseover=<UNSET>*, *navigate=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *position=<UNSET>*, *prop_cycle=<UNSET>*, *rasterization_zorder=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *subplotspec=<UNSET>*, *title=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *xbound=<UNSET>*, *xlabel=<UNSET>*, *xlim=<UNSET>*, *xmargin=<UNSET>*, *xscale=<UNSET>*, *xticklabels=<UNSET>*, *xticks=<UNSET>*, *ybound=<UNSET>*, *ylabel=<UNSET>*, *ylim=<UNSET>*, *ymargin=<UNSET>*, *yscale=<UNSET>*, *yticklabels=<UNSET>*, *yticks=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *adjustable* | {'box', 'datalim'} |
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value |
| *alpha* | scalar or None |
| *anchor* | (float, float) or {'C', 'SW', 'S', 'SE', 'E', 'NE', ...} |
| *animated* | bool |
| *aspect* | {'auto', 'equal'} or float |
| *autoscale_on* | bool |
| *autoscalex_on* | unknown |
| *autoscaley_on* | unknown |
| *axes_locator* | Callable[[Axes, Renderer], Bbox] |
| *axisbelow* | bool or 'line' |
| *box_aspect* | float or None |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |

Table 142 – continued from pr

| Property | Description |
|---|---|
| facecolor or fc | color |
| figure | Figure |
| forward_navigation_events | bool or "auto" |
| frame_on | bool |
| gid | str |
| in_layout | bool |
| label | object |
| latitude_grid | unknown |
| longitude_grid | unknown |
| longitude_grid_ends | unknown |
| mouseover | bool |
| navigate | bool |
| navigate_mode | unknown |
| path_effects | list of AbstractPathEffect |
| picker | None or bool or float or callable |
| position | [left, bottom, width, height] or Bbox |
| prop_cycle | Cycler |
| rasterization_zorder | float or None |
| rasterized | bool |
| sketch_params | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| subplotspec | unknown |
| title | str |
| transform | Transform |
| url | str |
| visible | bool |
| xbound | (lower: float, upper: float) |
| xlabel | str |
| xlim | unknown |
| xmargin | float greater than -0.5 |
| xscale | unknown |
| xticklabels | unknown |
| xticks | unknown |
| ybound | (lower: float, upper: float) |
| ylabel | str |
| ylim | unknown |
| ymargin | float greater than -0.5 |
| yscale | unknown |
| yticklabels | unknown |
| yticks | unknown |
| zorder | float |

**set_latitude_grid**(*degrees*)

Set the number of degrees between each latitude grid.

**set_longitude_grid**(*degrees*)

Set the number of degrees between each longitude grid.

**set_longitude_grid_ends**(*degrees*)

Set the latitude(s) at which to stop drawing the longitude grids.

**set_xlim**(**args*, ***kwargs*)

Not supported. Please consider using Cartopy.

**set_xscale**(**args*, ***kwargs*)

Set the xaxis' scale.

**Parameters**

**value**

[{"linear", "log", "symlog", "logit", ...} or *ScaleBase*] The axis scale type to apply.

****kwargs**

Different keyword arguments are accepted, depending on the scale. See the respective class keyword arguments:

- *matplotlib.scale.LinearScale*
- *matplotlib.scale.LogScale*
- *matplotlib.scale.SymmetricalLogScale*
- *matplotlib.scale.LogitScale*
- *matplotlib.scale.FuncScale*
- *matplotlib.scale.AsinhScale*

### Notes

By default, Matplotlib supports the above-mentioned scales. Additionally, custom scales may be registered using *matplotlib.scale.register_scale*. These scales can then also be used here.

**set_ylim**(**args*, ***kwargs*)

Not supported. Please consider using Cartopy.

**set_yscale**(**args*, ***kwargs*)

Set the yaxis' scale.

**Parameters**

**value**

[{"linear", "log", "symlog", "logit", ...} or *ScaleBase*] The axis scale type to apply.

****kwargs**

Different keyword arguments are accepted, depending on the scale. See the respective class keyword arguments:

- *matplotlib.scale.LinearScale*
- *matplotlib.scale.LogScale*
- *matplotlib.scale.SymmetricalLogScale*
- *matplotlib.scale.LogitScale*
- *matplotlib.scale.FuncScale*
- *matplotlib.scale.AsinhScale*

### Notes

By default, Matplotlib supports the above-mentioned scales. Additionally, custom scales may be registered using *matplotlib.scale.register_scale*. These scales can then also be used here.

**start_pan**(*x*, *y*, *button*)

Called when a pan operation has started.

**Parameters**

**x, y**

[float] The mouse coordinates in display coords.

**button**

[*MouseButton*] The pressed mouse button.

### Notes

This is intended to be overridden by new projection types.

**class** matplotlib.projections.geo.**HammerAxes**(**args*, ***kwargs*)

Bases: *GeoAxes*

Build an Axes in a figure.

**Parameters**

**fig**

[*Figure*] The Axes is built in the *Figure* *fig*.

***args**

`*args` can be a single `(left, bottom, width, height)` rectangle or a single *Bbox*. This specifies the rectangle (in figure coordinates) where the Axes is positioned.

`*args` can also consist of three numbers or a single three-digit number; in the latter case, the digits are considered as independent numbers. The numbers are interpreted as `(nrows, ncols, index)`: `(nrows, ncols)` specifies the size of an array of subplots, and `index` is the 1-based index of the subplot being created. Finally, `*args` can also directly be a *SubplotSpec* instance.

**sharex, sharey**

[*Axes*, optional] The x- or y-*axis* is shared with the x- or y-axis in the input *Axes*.

**frameon**

[bool, default: True] Whether the Axes frame is visible.

**box_aspect**

[float, optional] Set a fixed aspect for the Axes box, i.e. the ratio of height to width. See *set_box_aspect* for details.

**forward_navigation_events**

[bool or "auto", default: "auto"] Control whether pan/zoom events are passed through to Axes below this one. "auto" is *True* for axes with an invisible patch and *False* otherwise.

****kwargs**

Other optional keyword arguments:

| Property | Description |
|---|---|
| *adjustable* | {'box', 'datalim'} |
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi v |
| *alpha* | scalar or None |
| *anchor* | (float, float) or {'C', 'SW', 'S', 'SE', 'E', 'NE', ...} |
| *animated* | bool |
| *aspect* | {'auto', 'equal'} or float |
| *autoscale_on* | bool |
| *autoscalex_on* | unknown |
| *autoscaley_on* | unknown |
| *axes_locator* | Callable[[Axes, Renderer], Bbox] |
| *axisbelow* | bool or 'line' |
| *box_aspect* | float or None |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |

Table 143 – continued fro

| Property | Description |
|---|---|
| facecolor or fc | color |
| figure | Figure |
| forward_navigation_events | bool or "auto" |
| frame_on | bool |
| gid | str |
| in_layout | bool |
| label | object |
| mouseover | bool |
| navigate | bool |
| navigate_mode | unknown |
| path_effects | list of AbstractPathEffect |
| picker | None or bool or float or callable |
| position | [left, bottom, width, height] or Bbox |
| prop_cycle | Cycler |
| rasterization_zorder | float or None |
| rasterized | bool |
| sketch_params | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| subplotspec | unknown |
| title | str |
| transform | Transform |
| url | str |
| visible | bool |
| xbound | (lower: float, upper: float) |
| xlabel | str |
| xlim | (left: float, right: float) |
| xmargin | float greater than -0.5 |
| xscale | unknown |
| xticklabels | unknown |
| xticks | unknown |
| ybound | (lower: float, upper: float) |
| ylabel | str |
| ylim | (bottom: float, top: float) |
| ymargin | float greater than -0.5 |
| yscale | unknown |
| yticklabels | unknown |
| yticks | unknown |
| zorder | float |

**Returns**

**Axes**

The new Axes object.

**class HammerTransform**(*resolution*)

Bases: _GeoTransform

The base Hammer transform.

Create a new geographical transform.

Resolution is the number of steps to interpolate between each input line segment to approximate its path in curved space.

**has_inverse = True**

True if this transform has a corresponding inverse transform.

**inverted**()

Return the corresponding inverse transformation.

It holds `x == self.inverted().transform(self.transform(x))`.

The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

**transform_non_affine**(*values*)

Apply only the non-affine part of this transformation.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

**Parameters**

**values**

[array] The input values as an array of length `input_dims` or shape (N, `input_dims`).

**Returns**

**array**

The output values as an array of length `output_dims` or shape (N, `output_dims`), depending on the input.

**class InvertedHammerTransform**(*resolution*)

Bases: _GeoTransform

Create a new geographical transform.

Resolution is the number of steps to interpolate between each input line segment to approximate its path in curved space.

**has_inverse = True**

True if this transform has a corresponding inverse transform.

**inverted**()

Return the corresponding inverse transformation.

It holds `x == self.inverted().transform(self.transform(x))`.

The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

**transform_non_affine**(*values*)

Apply only the non-affine part of this transformation.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

**Parameters**

**values**

[array] The input values as an array of length `input_dims` or shape (N, `input_dims`).

**Returns**

**array**

The output values as an array of length `output_dims` or shape (N, `output_dims`), depending on the input.

**name = 'hammer'**

**set**(*, *adjustable=<UNSET>*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *anchor=<UNSET>*, *animated=<UNSET>*, *aspect=<UNSET>*, *autoscale_on=<UNSET>*, *autoscalex_on=<UNSET>*, *autoscaley_on=<UNSET>*, *axes_locator=<UNSET>*, *axisbelow=<UNSET>*, *box_aspect=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *facecolor=<UNSET>*, *forward_navigation_events=<UNSET>*, *frame_on=<UNSET>*, *gid=<UNSET>*, *in_layout=<UNSET>*, *label=<UNSET>*, *latitude_grid=<UNSET>*, *longitude_grid=<UNSET>*, *longitude_grid_ends=<UNSET>*, *mouseover=<UNSET>*, *navigate=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *position=<UNSET>*, *prop_cycle=<UNSET>*, *rasterization_zorder=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *subplotspec=<UNSET>*, *title=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *xbound=<UNSET>*, *xlabel=<UNSET>*, *xlim=<UNSET>*, *xmargin=<UNSET>*, *xscale=<UNSET>*, *xticklabels=<UNSET>*, *xticks=<UNSET>*, *ybound=<UNSET>*, *ylabel=<UNSET>*, *ylim=<UNSET>*, *ymargin=<UNSET>*, *yscale=<UNSET>*, *yticklabels=<UNSET>*, *yticks=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *adjustable* | {'box', 'datalim'} |
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value |
| *alpha* | scalar or None |
| *anchor* | (float, float) or {'C', 'SW', 'S', 'SE', 'E', 'NE', ...} |
| *animated* | bool |
| *aspect* | {'auto', 'equal'} or float |
| *autoscale_on* | bool |
| *autoscalex_on* | unknown |
| *autoscaley_on* | unknown |
| *axes_locator* | Callable[[Axes, Renderer], Bbox] |
| *axisbelow* | bool or 'line' |
| *box_aspect* | float or None |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *facecolor* or fc | *color* |
| *figure* | *Figure* |
| *forward_navigation_events* | bool or "auto" |
| *frame_on* | bool |
| *gid* | str |
| *in_layout* | bool |
| *label* | object |
| *latitude_grid* | unknown |
| *longitude_grid* | unknown |
| *longitude_grid_ends* | unknown |
| *mouseover* | bool |
| *navigate* | bool |
| *navigate_mode* | unknown |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *position* | [left, bottom, width, height] or *Bbox* |
| *prop_cycle* | *Cycler* |
| *rasterization_zorder* | float or None |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *subplotspec* | unknown |
| *title* | str |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *xbound* | (lower: float, upper: float) |
| *xlabel* | str |
| *xlim* | unknown |
| *xmargin* | float greater than -0.5 |

Table 144 – continued from pr

| Property | Description |
|---|---|
| *xscale* | unknown |
| *xticklabels* | unknown |
| *xticks* | unknown |
| *ybound* | (lower: float, upper: float) |
| *ylabel* | str |
| *ylim* | unknown |
| *ymargin* | float greater than -0.5 |
| *yscale* | unknown |
| *yticklabels* | unknown |
| *yticks* | unknown |
| *zorder* | float |

**class** matplotlib.projections.geo.**LambertAxes**(**args*, *center_longitude=0*, *center_latitude=0*, ***kwargs*)

Bases: *GeoAxes*

Build an Axes in a figure.

**Parameters**

**fig**

[*Figure*] The Axes is built in the *Figure* *fig*.

***args**

`*args` can be a single `(left, bottom, width, height)` rectangle or a single *Bbox*. This specifies the rectangle (in figure coordinates) where the Axes is positioned.

`*args` can also consist of three numbers or a single three-digit number; in the latter case, the digits are considered as independent numbers. The numbers are interpreted as `(nrows, ncols, index)`: `(nrows, ncols)` specifies the size of an array of subplots, and `index` is the 1-based index of the subplot being created. Finally, `*args` can also directly be a *SubplotSpec* instance.

**sharex, sharey**

[*Axes*, optional] The x- or y-*axis* is shared with the x- or y-axis in the input *Axes*.

**frameon**

[bool, default: True] Whether the Axes frame is visible.

**box_aspect**

[float, optional] Set a fixed aspect for the Axes box, i.e. the ratio of height to width. See *set_box_aspect* for details.

**forward_navigation_events**

[bool or "auto", default: "auto"] Control whether pan/zoom events are passed through to Axes below this one. "auto" is *True* for axes with an invisible patch and *False* otherwise.

****kwargs**

Other optional keyword arguments:

| Property | Description |
|---|---|
| *adjustable* | {'box', 'datalim'} |
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi v |
| *alpha* | scalar or None |
| *anchor* | (float, float) or {'C', 'SW', 'S', 'SE', 'E', 'NE', ...} |
| *animated* | bool |
| *aspect* | {'auto', 'equal'} or float |
| *autoscale_on* | bool |
| *autoscalex_on* | unknown |
| *autoscaley_on* | unknown |
| *axes_locator* | Callable[[Axes, Renderer], Bbox] |
| *axisbelow* | bool or 'line' |
| *box_aspect* | float or None |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *facecolor* or fc | *color* |
| *figure* | *Figure* |
| *forward_navigation_events* | bool or "auto" |
| *frame_on* | bool |
| *gid* | str |
| *in_layout* | bool |
| *label* | object |
| *mouseover* | bool |
| *navigate* | bool |
| *navigate_mode* | unknown |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *position* | [left, bottom, width, height] or *Bbox* |
| *prop_cycle* | *Cycler* |
| *rasterization_zorder* | float or None |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *subplotspec* | unknown |
| *title* | str |
| *transform* | *Transform* |

Table 145 – continued fro

| Property | Description |
|---|---|
| *url* | str |
| *visible* | bool |
| *xbound* | (lower: float, upper: float) |
| *xlabel* | str |
| *xlim* | (left: float, right: float) |
| *xmargin* | float greater than -0.5 |
| *xscale* | unknown |
| *xticklabels* | unknown |
| *xticks* | unknown |
| *ybound* | (lower: float, upper: float) |
| *ylabel* | str |
| *ylim* | (bottom: float, top: float) |
| *ymargin* | float greater than -0.5 |
| *yscale* | unknown |
| *yticklabels* | unknown |
| *yticks* | unknown |
| *zorder* | float |

**Returns**

***Axes***

The new *Axes* object.

**class InvertedLambertTransform**(*center_longitude*, *center_latitude*, *resolution*)

Bases: _GeoTransform

Create a new geographical transform.

Resolution is the number of steps to interpolate between each input line segment to approximate its path in curved space.

**has_inverse = True**

True if this transform has a corresponding inverse transform.

**inverted**()

Return the corresponding inverse transformation.

It holds `x == self.inverted().transform(self.transform(x))`.

The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

**transform_non_affine**(*values*)

Apply only the non-affine part of this transformation.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

**Parameters**

**values**

[array] The input values as an array of length `input_dims` or shape (N, `input_dims`).

**Returns**

**array**

The output values as an array of length `output_dims` or shape (N, `output_dims`), depending on the input.

**class LambertTransform**(*center_longitude*, *center_latitude*, *resolution*)

Bases: `_GeoTransform`

The base Lambert transform.

Create a new Lambert transform. Resolution is the number of steps to interpolate between each input line segment to approximate its path in curved Lambert space.

**has_inverse = True**

True if this transform has a corresponding inverse transform.

**inverted**()

Return the corresponding inverse transformation.

It holds `x == self.inverted().transform(self.transform(x))`.

The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

**transform_non_affine**(*values*)

Apply only the non-affine part of this transformation.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

**Parameters**

**values**

[array] The input values as an array of length `input_dims` or shape (N, `input_dims`).

**Returns**

**array**

The output values as an array of length `output_dims` or shape (N, `output_dims`), depending on the input.

**clear**()

Clear the Axes.

**name = 'lambert'**

**set**(*, *adjustable=<UNSET>*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *anchor=<UNSET>*, *animated=<UNSET>*, *aspect=<UNSET>*, *autoscale_on=<UNSET>*, *autoscalex_on=<UNSET>*, *autoscaley_on=<UNSET>*, *axes_locator=<UNSET>*, *axisbelow=<UNSET>*, *box_aspect=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *facecolor=<UNSET>*, *forward_navigation_events=<UNSET>*, *frame_on=<UNSET>*, *gid=<UNSET>*, *in_layout=<UNSET>*, *label=<UNSET>*, *latitude_grid=<UNSET>*, *longitude_grid=<UNSET>*, *longitude_grid_ends=<UNSET>*, *mouseover=<UNSET>*, *navigate=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *position=<UNSET>*, *prop_cycle=<UNSET>*, *rasterization_zorder=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *subplotspec=<UNSET>*, *title=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *xbound=<UNSET>*, *xlabel=<UNSET>*, *xlim=<UNSET>*, *xmargin=<UNSET>*, *xscale=<UNSET>*, *xticklabels=<UNSET>*, *xticks=<UNSET>*, *ybound=<UNSET>*, *ylabel=<UNSET>*, *ylim=<UNSET>*, *ymargin=<UNSET>*, *yscale=<UNSET>*, *yticklabels=<UNSET>*, *yticks=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *adjustable* | {'box', 'datalim'} |
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value |
| *alpha* | scalar or None |
| *anchor* | (float, float) or {'C', 'SW', 'S', 'SE', 'E', 'NE', ...} |
| *animated* | bool |
| *aspect* | {'auto', 'equal'} or float |
| *autoscale_on* | bool |
| *autoscalex_on* | unknown |
| *autoscaley_on* | unknown |
| *axes_locator* | Callable[[Axes, Renderer], Bbox] |
| *axisbelow* | bool or 'line' |
| *box_aspect* | float or None |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *facecolor* or fc | *color* |
| *figure* | *Figure* |
| *forward_navigation_events* | bool or "auto" |
| *frame_on* | bool |
| *gid* | str |
| *in_layout* | bool |

Table 146 – continued from pr

| Property | Description |
|---|---|
| label | object |
| latitude_grid | unknown |
| longitude_grid | unknown |
| longitude_grid_ends | unknown |
| mouseover | bool |
| navigate | bool |
| navigate_mode | unknown |
| path_effects | list of AbstractPathEffect |
| picker | None or bool or float or callable |
| position | [left, bottom, width, height] or Bbox |
| prop_cycle | Cycler |
| rasterization_zorder | float or None |
| rasterized | bool |
| sketch_params | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| subplotspec | unknown |
| title | str |
| transform | Transform |
| url | str |
| visible | bool |
| xbound | (lower: float, upper: float) |
| xlabel | str |
| xlim | unknown |
| xmargin | float greater than -0.5 |
| xscale | unknown |
| xticklabels | unknown |
| xticks | unknown |
| ybound | (lower: float, upper: float) |
| ylabel | str |
| ylim | unknown |
| ymargin | float greater than -0.5 |
| yscale | unknown |
| yticklabels | unknown |
| yticks | unknown |
| zorder | float |

**class** matplotlib.projections.geo.**MollweideAxes**(**args*, ***kwargs*)

Bases: GeoAxes

Build an Axes in a figure.

**Parameters**

**fig**

[*Figure*] The Axes is built in the *Figure* *fig*.

***args**

`*args` can be a single `(left, bottom, width, height)` rectangle or a single *Bbox*. This specifies the rectangle (in figure coordinates) where the Axes is positioned.

`*args` can also consist of three numbers or a single three-digit number; in the latter case, the digits are considered as independent numbers. The numbers are interpreted as `(nrows, ncols, index)`: `(nrows, ncols)` specifies the size of an array of subplots, and `index` is the 1-based index of the subplot being created. Finally, `*args` can also directly be a *SubplotSpec* instance.

**sharex, sharey**

[*Axes*, optional] The x- or y-*axis* is shared with the x- or y-axis in the input *Axes*.

**frameon**

[bool, default: True] Whether the Axes frame is visible.

**box_aspect**

[float, optional] Set a fixed aspect for the Axes box, i.e. the ratio of height to width. See *set_box_aspect* for details.

**forward_navigation_events**

[bool or "auto", default: "auto"] Control whether pan/zoom events are passed through to Axes below this one. "auto" is *True* for axes with an invisible patch and *False* otherwise.

****kwargs**

Other optional keyword arguments:

| Property | Description |
|---|---|
| *adjustable* | {'box', 'datalim'} |
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi v |
| *alpha* | scalar or None |
| *anchor* | (float, float) or {'C', 'SW', 'S', 'SE', 'E', 'NE', ...} |
| *animated* | bool |
| *aspect* | {'auto', 'equal'} or float |
| *autoscale_on* | bool |
| *autoscalex_on* | unknown |
| *autoscaley_on* | unknown |
| *axes_locator* | Callable[[Axes, Renderer], Bbox] |
| *axisbelow* | bool or 'line' |
| *box_aspect* | float or None |
| *clip_box* | *BboxBase* or None |

Table 147 – continued fro

| Property | Description |
|---|---|
| clip_on | bool |
| clip_path | Patch or (Path, Transform) or None |
| facecolor or fc | color |
| figure | Figure |
| forward_navigation_events | bool or "auto" |
| frame_on | bool |
| gid | str |
| in_layout | bool |
| label | object |
| mouseover | bool |
| navigate | bool |
| navigate_mode | unknown |
| path_effects | list of AbstractPathEffect |
| picker | None or bool or float or callable |
| position | [left, bottom, width, height] or Bbox |
| prop_cycle | Cycler |
| rasterization_zorder | float or None |
| rasterized | bool |
| sketch_params | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| subplotspec | unknown |
| title | str |
| transform | Transform |
| url | str |
| visible | bool |
| xbound | (lower: float, upper: float) |
| xlabel | str |
| xlim | (left: float, right: float) |
| xmargin | float greater than -0.5 |
| xscale | unknown |
| xticklabels | unknown |
| xticks | unknown |
| ybound | (lower: float, upper: float) |
| ylabel | str |
| ylim | (bottom: float, top: float) |
| ymargin | float greater than -0.5 |
| yscale | unknown |
| yticklabels | unknown |
| yticks | unknown |
| zorder | float |

**Returns**

*Axes*

The new *Axes* object.

**class InvertedMollweideTransform**(*resolution*)

Bases: _GeoTransform

Create a new geographical transform.

Resolution is the number of steps to interpolate between each input line segment to approximate its path in curved space.

**has_inverse = True**

True if this transform has a corresponding inverse transform.

**inverted**()

Return the corresponding inverse transformation.

It holds `x == self.inverted().transform(self.transform(x))`.

The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

**transform_non_affine**(*values*)

Apply only the non-affine part of this transformation.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

**Parameters**

**values**

[array] The input values as an array of length `input_dims` or shape (N, `input_dims`).

**Returns**

**array**

The output values as an array of length `output_dims` or shape (N, `output_dims`), depending on the input.

**class MollweideTransform**(*resolution*)

Bases: _GeoTransform

The base Mollweide transform.

Create a new geographical transform.

Resolution is the number of steps to interpolate between each input line segment to approximate its path in curved space.

**has_inverse = True**

True if this transform has a corresponding inverse transform.

**inverted**()

Return the corresponding inverse transformation.

It holds `x == self.inverted().transform(self.transform(x))`.

The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

**transform_non_affine**(*values*)

Apply only the non-affine part of this transformation.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

**Parameters**

**values**

[array] The input values as an array of length `input_dims` or shape (N, `input_dims`).

**Returns**

**array**

The output values as an array of length `output_dims` or shape (N, `output_dims`), depending on the input.

**name = 'mollweide'**

**set**(*, *adjustable=<UNSET>*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *anchor=<UNSET>*, *animated=<UNSET>*, *aspect=<UNSET>*, *autoscale_on=<UNSET>*, *autoscalex_on=<UNSET>*, *autoscaley_on=<UNSET>*, *axes_locator=<UNSET>*, *axisbelow=<UNSET>*, *box_aspect=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *facecolor=<UNSET>*, *forward_navigation_events=<UNSET>*, *frame_on=<UNSET>*, *gid=<UNSET>*, *in_layout=<UNSET>*, *label=<UNSET>*, *latitude_grid=<UNSET>*, *longitude_grid=<UNSET>*, *longitude_grid_ends=<UNSET>*, *mouseover=<UNSET>*, *navigate=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *position=<UNSET>*, *prop_cycle=<UNSET>*, *rasterization_zorder=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *subplotspec=<UNSET>*, *title=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *xbound=<UNSET>*, *xlabel=<UNSET>*, *xlim=<UNSET>*, *xmargin=<UNSET>*, *xscale=<UNSET>*, *xticklabels=<UNSET>*, *xticks=<UNSET>*, *ybound=<UNSET>*, *ylabel=<UNSET>*, *ylim=<UNSET>*, *ymargin=<UNSET>*, *yscale=<UNSET>*, *yticklabels=<UNSET>*, *yticks=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *adjustable* | {'box', 'datalim'} |
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value |
| *alpha* | scalar or None |
| *anchor* | (float, float) or {'C', 'SW', 'S', 'SE', 'E', 'NE', ...} |
| *animated* | bool |
| *aspect* | {'auto', 'equal'} or float |
| *autoscale_on* | bool |
| *autoscalex_on* | unknown |
| *autoscaley_on* | unknown |
| *axes_locator* | Callable[[Axes, Renderer], Bbox] |
| *axisbelow* | bool or 'line' |
| *box_aspect* | float or None |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *facecolor* or fc | *color* |
| *figure* | *Figure* |
| *forward_navigation_events* | bool or "auto" |
| *frame_on* | bool |
| *gid* | str |
| *in_layout* | bool |
| *label* | object |
| *latitude_grid* | unknown |
| *longitude_grid* | unknown |
| *longitude_grid_ends* | unknown |
| *mouseover* | bool |
| *navigate* | bool |
| *navigate_mode* | unknown |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *position* | [left, bottom, width, height] or *Bbox* |
| *prop_cycle* | *Cycler* |
| *rasterization_zorder* | float or None |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *subplotspec* | unknown |
| *title* | str |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *xbound* | (lower: float, upper: float) |
| *xlabel* | str |
| *xlim* | unknown |
| *xmargin* | float greater than -0.5 |

Table 148 – continued from pr

| Property | Description |
|---|---|
| *xscale* | unknown |
| *xticklabels* | unknown |
| *xticks* | unknown |
| *ybound* | (lower: float, upper: float) |
| *ylabel* | str |
| *ylim* | unknown |
| *ymargin* | float greater than -0.5 |
| *yscale* | unknown |
| *yticklabels* | unknown |
| *yticks* | unknown |
| *zorder* | float |

### 7.2.42 matplotlib.quiver

Support for plotting vector fields.

Presently this contains Quiver and Barb. Quiver plots an arrow in the direction of the vector, with the size of the arrow related to the magnitude of the vector.

Barbs are like quiver in that they point along a vector, but the magnitude of the vector is given schematically by the presence of barbs or flags on the barb.

This will also become a home for things such as standard deviation ellipses, which can and will be derived very easily from the Quiver code.

#### Classes

| | |
|---|---|
| *Quiver*(ax, *args[, scale, headwidth, ...]) | Specialized PolyCollection for arrows. |
| *QuiverKey*(Q, X, Y, U, label, *[, angle, ...]) | Labelled arrow for use as a quiver plot scale key. |
| *Barbs*(ax, *args[, pivot, length, barbcolor, ...]) | Specialized PolyCollection for barbs. |

#### matplotlib.quiver.Quiver

**class** matplotlib.quiver.**Quiver**(*ax*, **args*, *scale=None*, *headwidth=3*, *headlength=5*, *headaxislength=4.5*, *minshaft=1*, *minlength=1*, *units='width'*, *scale_units=None*, *angles='uv'*, *width=None*, *color='k'*, *pivot='tail'*, ***kwargs*)

Bases: *PolyCollection*

Specialized PolyCollection for arrows.

The only API method is set_UVC(), which can be used to change the size, orientation, and color of the arrows; their locations are fixed when the class is instantiated. Possibly this method will be useful in animations.

Much of the work in this class is done in the draw() method so that as much information as possible is available about the plot. In subsequent draw() calls, recalculation is limited to things that might have changed, so there should be no performance penalty from putting the calculations in the draw() method.

The constructor takes one required argument, an Axes instance, followed by the args and kwargs described by the following pyplot interface documentation:

Plot a 2D field of arrows.

Call signature:

```
quiver([X, Y], U, V, [C], **kwargs)
```

*X*, *Y* define the arrow locations, *U*, *V* define the arrow directions, and *C* optionally sets the color.

**Arrow length**

The default settings auto-scales the length of the arrows to a reasonable size. To change this behavior see the *scale* and *scale_units* parameters.

**Arrow shape**

The arrow shape is determined by *width*, *headwidth*, *headlength* and *headaxislength*. See the notes below.

**Arrow styling**

Each arrow is internally represented by a filled polygon with a default edge linewidth of 0. As a result, an arrow is rather a filled area, not a line with a head, and `PolyCollection` properties like *linewidth*, *edgecolor*, *facecolor*, etc. act accordingly.

**Parameters**

**X, Y**

[1D or 2D array-like, optional] The x and y coordinates of the arrow locations.

If not given, they will be generated as a uniform integer meshgrid based on the dimensions of *U* and *V*.

If *X* and *Y* are 1D but *U*, *V* are 2D, *X*, *Y* are expanded to 2D using `X, Y = np.meshgrid(X, Y)`. In this case `len(X)` and `len(Y)` must match the column and row dimensions of *U* and *V*.

**U, V**

[1D or 2D array-like] The x and y direction components of the arrow vectors. The interpretation of these components (in data or in screen space) depends on *angles*.

*U* and *V* must have the same number of elements, matching the number of arrow locations in *X*, *Y*. *U* and *V* may be masked. Locations masked in any of *U*, *V*, and

*C* will not be drawn.

**C**

[1D or 2D array-like, optional] Numeric data that defines the arrow colors by colormapping via *norm* and *cmap*.

This does not support explicit colors. If you want to set colors directly, use *color* instead. The size of *C* must match the number of arrow locations.

**angles**

[{'uv', 'xy'} or array-like, default: 'uv'] Method for determining the angle of the arrows.

- 'uv': Arrow direction in screen coordinates. Use this if the arrows symbolize a quantity that is not based on *X*, *Y* data coordinates.

  If *U* == *V* the orientation of the arrow on the plot is 45 degrees counter-clockwise from the horizontal axis (positive to the right).

- 'xy': Arrow direction in data coordinates, i.e. the arrows point from (x, y) to (x+u, y+v). Use this e.g. for plotting a gradient field.

- Arbitrary angles may be specified explicitly as an array of values in degrees, counter-clockwise from the horizontal axis.

  In this case *U*, *V* is only used to determine the length of the arrows.

Note: inverting a data axis will correspondingly invert the arrows only with `angles='xy'`.

**pivot**

[{'tail', 'mid', 'middle', 'tip'}, default: 'tail'] The part of the arrow that is anchored to the *X*, *Y* grid. The arrow rotates about this point.

'mid' is a synonym for 'middle'.

**scale**

[float, optional] Scales the length of the arrow inversely.

Number of data units per arrow length unit, e.g., m/s per plot width; a smaller scale parameter makes the arrow longer. Default is *None*.

If *None*, a simple autoscaling algorithm is used, based on the average vector length and the number of vectors. The arrow length unit is given by the *scale_units* parameter.

**scale_units**

[{'width', 'height', 'dots', 'inches', 'x', 'y', 'xy'}, optional] If the *scale* kwarg is *None*, the arrow length unit. Default is *None*.

e.g. *scale_units* is 'inches', *scale* is 2.0, and `(u, v) = (1, 0)`, then the vector will be 0.5 inches long.

If *scale_units* is 'width' or 'height', then the vector will be half the width/height of the axes.

If *scale_units* is 'x' then the vector will be 0.5 x-axis units. To plot vectors in the x-y plane, with u and v having the same units as x and y, use `angles='xy', scale_units='xy', scale=1`.

**units**

[{'width', 'height', 'dots', 'inches', 'x', 'y', 'xy'}, default: 'width'] Affects the arrow size (except for the length). In particular, the shaft *width* is measured in multiples of this unit.

Supported values are:

- 'width', 'height': The width or height of the Axes.
- 'dots', 'inches': Pixels or inches based on the figure dpi.
- 'x', 'y', 'xy': $X$, $Y$ or $\sqrt{X^2 + Y^2}$ in data units.

The following table summarizes how these values affect the visible arrow size under zooming and figure size changes:

| units | zoom | figure size change |
|---|---|---|
| 'x', 'y', 'xy' | arrow size scales | — |
| 'width', 'height' | — | arrow size scales |
| 'dots', 'inches' | — | — |

**width**

[float, optional] Shaft width in arrow units. All head parameters are relative to *width*.

The default depends on choice of *units* above, and number of vectors; a typical starting value is about 0.005 times the width of the plot.

**headwidth**

[float, default: 3] Head width as multiple of shaft *width*. See the notes below.

**headlength**

[float, default: 5] Head length as multiple of shaft *width*. See the notes below.

**headaxislength**

[float, default: 4.5] Head length at shaft intersection as multiple of shaft *width*. See the notes below.

**minshaft**

[float, default: 1] Length below which arrow scales, in units of head length. Do not set this to less than 1, or small arrows will look terrible!

**minlength**

[float, default: 1] Minimum length as a multiple of shaft width; if an arrow length is less than this, plot a dot (hexagon) of this diameter instead.

**color**

[*color* or list *color*, optional] Explicit color(s) for the arrows. If *C* has been set, *color* has no effect.

This is a synonym for the `PolyCollection` *facecolor* parameter.

**Returns**

***Quiver***

**Other Parameters**

**data**

[indexable object, optional] DATA_PARAMETER_PLACEHOLDER

****kwargs**

[`PolyCollection` properties, optional] All other keyword arguments are passed on to `PolyCollection`:

| Property | Description |
|---|---|
| `agg_filter` | a filter function, which takes a (m, n, 3) float array and a d |
| `alpha` | array-like or scalar or None |
| `animated` | bool |
| `antialiased` or aa or antialiaseds | bool or list of bools |
| `array` | array-like or None |
| `capstyle` | `CapStyle` or {'butt', 'projecting', 'round'} |
| `clim` | (vmin: float, vmax: float) |
| `clip_box` | `BboxBase` or None |
| `clip_on` | bool |
| `clip_path` | Patch or (Path, Transform) or None |
| `cmap` | `Colormap` or str or None |
| `color` | *color* or list of RGBA tuples |
| `edgecolor` or ec or edgecolors | *color* or list of *color* or 'face' |
| `facecolor` or facecolors or fc | *color* or list of *color* |
| `figure` | `Figure` |
| `gid` | str |
| `hatch` | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| `in_layout` | bool |
| `joinstyle` | `JoinStyle` or {'miter', 'round', 'bevel'} |
| `label` | object |
| `linestyle` or dashes or linestyles or ls | str or tuple or list thereof |

Table 149 – continued fr

| Property | Description |
|---|---|
| *linewidth* or linewidths or lw | float or list of floats |
| *mouseover* | bool |
| *norm* | *Normalize* or str or None |
| *offset_transform* or transOffset | *Transform* |
| *offsets* | (N, 2) or (2,) array-like |
| *path_effects* | list of *AbstractPathEffect* |
| *paths* | list of array-like |
| *picker* | None or bool or float or callable |
| *pickradius* | float |
| *rasterized* | bool |
| sizes | numpy.ndarray or None |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *urls* | list of str or None |
| *verts* | list of array-like |
| *verts_and_codes* | unknown |
| *visible* | bool |
| *zorder* | float |

**See also:**

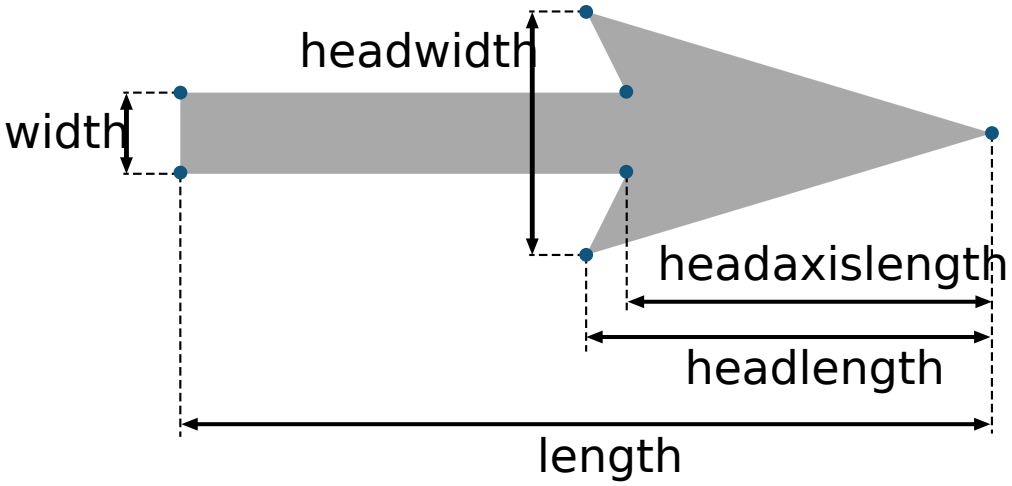
***Axes.quiverkey***

Add a key to a quiver plot.

## Notes

### Arrow shape

The arrow is drawn as a polygon using the nodes as shown below. The values *headwidth*, *headlength*, and *headaxislength* are in units of *width*.




The defaults give a slightly swept-back arrow. Here are some guidelines how to get other head shapes:

- To make the head a triangle, make *headaxislength* the same as *headlength*.
- To make the arrow more pointed, reduce *headwidth* or increase *headlength* and *headaxislength*.
- To make the head smaller relative to the shaft, scale down all the head parameters proportionally.
- To remove the head completely, set all *head* parameters to 0.
- To get a diamond-shaped head, make *headaxislength* larger than *headlength*.
- Warning: For *headaxislength* < (*headlength* / *headwidth*), the "headaxis" nodes (i.e. the ones connecting the head with the shaft) will protrude out of the head in forward direction so that the arrow head looks broken.

**draw**(*renderer*)

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

**Parameters**

**renderer**

[*RendererBase* subclass.]

### Notes

This method is overridden in the Artist subclasses.

**get_datalim**(*transData*)

**set**(*, *UVC=<UNSET>*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *antialiased=<UNSET>*, *array=<UNSET>*, *capstyle=<UNSET>*, *clim=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *cmap=<UNSET>*, *color=<UNSET>*, *edgecolor=<UNSET>*, *facecolor=<UNSET>*, *gid=<UNSET>*, *hatch=<UNSET>*, *in_layout=<UNSET>*, *joinstyle=<UNSET>*, *label=<UNSET>*, *linestyle=<UNSET>*, *linewidth=<UNSET>*, *mouseover=<UNSET>*, *norm=<UNSET>*, *offset_transform=<UNSET>*, *offsets=<UNSET>*, *path_effects=<UNSET>*, *paths=<UNSET>*, *picker=<UNSET>*, *pickradius=<UNSET>*, *rasterized=<UNSET>*, *sizes=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *urls=<UNSET>*, *verts=<UNSET>*, *verts_and_codes=<UNSET>*, *visible=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *UVC* | unknown |
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi v |
| *alpha* | array-like or scalar or None |
| *animated* | bool |
| *antialiased* or aa or antialiaseds | bool or list of bools |
| *array* | array-like or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clim* | (vmin: float, vmax: float) |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *cmap* | *Colormap* or str or None |
| *color* | *color* or list of RGBA tuples |
| *edgecolor* or ec or edgecolors | *color* or list of *color* or 'face' |
| *facecolor* or facecolors or fc | *color* or list of *color* |
| *figure* | *Figure* |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or dashes or linestyles or ls | str or tuple or list thereof |
| *linewidth* or linewidths or lw | float or list of floats |
| *mouseover* | bool |
| *norm* | *Normalize* or str or None |
| *offset_transform* or transOffset | *Transform* |
| *offsets* | (N, 2) or (2,) array-like |
| *path_effects* | list of *AbstractPathEffect* |
| *paths* | list of array-like |
| *picker* | None or bool or float or callable |

Table 150 – continued from

| Property | Description |
|---|---|
| *pickradius* | float |
| *rasterized* | bool |
| sizes | numpy.ndarray or None |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *urls* | list of str or None |
| *verts* | list of array-like |
| *verts_and_codes* | unknown |
| *visible* | bool |
| *zorder* | float |

**set_UVC**(*U*, *V*, *C=None*)

### Examples using matplotlib.quiver.Quiver

- *Advanced quiver and quiverkey functions*
- *Quiver Simple Demo*

## matplotlib.quiver.QuiverKey

**class** matplotlib.quiver.**QuiverKey**(*Q*, *X*, *Y*, *U*, *label*, ***, *angle=0*, *coordinates='axes'*, *color=None*, *labelsep=0.1*, *labelpos='N'*, *labelcolor=None*, *fontproperties=None*, ***kwargs*)

Bases: *Artist*

Labelled arrow for use as a quiver plot scale key.

Add a key to a quiver plot.

The positioning of the key depends on *X*, *Y*, *coordinates*, and *labelpos*. If *labelpos* is 'N' or 'S', *X*, *Y* give the position of the middle of the key arrow. If *labelpos* is 'E', *X*, *Y* positions the head, and if *labelpos* is 'W', *X*, *Y* positions the tail; in either of these two cases, *X*, *Y* is somewhere in the middle of the arrow+label key object.

**Parameters**

**Q**

[*Quiver*] A *Quiver* object as returned by a call to *quiver()*.

**X, Y**

[float] The location of the key.

**U**

[float] The length of the key.

**label**

[str] The key label (e.g., length and units of the key).

**angle**

[float, default: 0] The angle of the key arrow, in degrees anti-clockwise from the horizontal axis.

**coordinates**

[{'axes', 'figure', 'data', 'inches'}, default: 'axes'] Coordinate system and units for *X*, *Y*: 'axes' and 'figure' are normalized coordinate systems with (0, 0) in the lower left and (1, 1) in the upper right; 'data' are the axes data coordinates (used for the locations of the vectors in the quiver plot itself); 'inches' is position in the figure in inches, with (0, 0) at the lower left corner.

**color**

[*color*] Overrides face and edge colors from *Q*.

**labelpos**

[{'N', 'S', 'E', 'W'}] Position the label above, below, to the right, to the left of the arrow, respectively.

**labelsep**

[float, default: 0.1] Distance in inches between the arrow and the label.

**labelcolor**

[*color*, default: *rcParams["text.color"]* (default: `'black'`)] Label color.

**fontproperties**

[dict, optional] A dictionary with keyword arguments accepted by the *FontProperties* initializer: *family*, *style*, *variant*, *size*, *weight*.

****kwargs**

Any additional keyword arguments are used to override vector properties taken from *Q*.

**contains**(*mouseevent*)

Test whether the artist contains the mouse event.

**Parameters**

**mouseevent**

[*MouseEvent*]

**Returns**

**contains**

[bool] Whether any values are within the radius.

**details**

[dict] An artist-specific dictionary of details of the event context, such as which points are contained in the pick radius. See the individual Artist subclasses for details.

**draw**(*renderer*)

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

**Parameters**

**renderer**

[*RendererBase* subclass.]

### Notes

This method is overridden in the Artist subclasses.

**halign = {'E': 'left', 'N': 'center', 'S': 'center', 'W': 'right'}**

**property labelsep**

**pivot = {'E': 'tip', 'N': 'middle', 'S': 'middle', 'W': 'tail'}**

**set**(**, agg_filter=<UNSET>, alpha=<UNSET>, animated=<UNSET>, clip_box=<UNSET>, clip_on=<UNSET>, clip_path=<UNSET>, gid=<UNSET>, in_layout=<UNSET>, label=<UNSET>, mouseover=<UNSET>, path_effects=<UNSET>, picker=<UNSET>, rasterized=<UNSET>, sketch_params=<UNSET>, snap=<UNSET>, transform=<UNSET>, url=<UNSET>, visible=<UNSET>, zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| agg_fil | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image |
| alpha | scalar or None |
| animated | bool |
| clip_bo | BboxBase or None |
| clip_on | bool |
| clip_pa | Patch or (Path, Transform) or None |
| figure | unknown |
| gid | str |
| in_layo | bool |
| label | object |
| mouseov | bool |
| path_ef | list of AbstractPathEffect |
| picker | None or bool or float or callable |
| rasterized | bool |
| sketch_ | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| transform | Transform |
| url | str |
| visible | bool |
| zorder | float |

**set_figure**(*fig*)

Set the *Figure* instance the artist belongs to.

**Parameters**

**fig**

[*Figure*]

```
valign = {'E': 'center', 'N': 'bottom', 'S': 'top', 'W': 'center'}
```

### Examples using `matplotlib.quiver.QuiverKey`

- *Advanced quiver and quiverkey functions*

### matplotlib.quiver.Barbs

**class** matplotlib.quiver.**Barbs**(*ax*, **args*, *pivot='tip'*, *length=7*, *barbcolor=None*, *flagcolor=None*, *sizes=None*, *fill_empty=False*, *barb_increments=None*, *rounding=True*, *flip_barb=False*, ***kwargs*)

Bases: *PolyCollection*

Specialized PolyCollection for barbs.

The only API method is *set_UVC()*, which can be used to change the size, orientation, and color of the arrows. Locations are changed using the *set_offsets()* collection method. Possibly this method will be useful in animations.

There is one internal function `_find_tails()` which finds exactly what should be put on the barb given the vector magnitude. From there `_make_barbs()` is used to find the vertices of the polygon to represent the barb based on this information.

The constructor takes one required argument, an Axes instance, followed by the args and kwargs described by the following pyplot interface documentation:

Plot a 2D field of wind barbs.

Call signature:

```
barbs([X, Y], U, V, [C], **kwargs)
```

Where *X*, *Y* define the barb locations, *U*, *V* define the barb directions, and *C* optionally sets the color.

All arguments may be 1D or 2D. *U*, *V*, *C* may be masked arrays, but masked *X*, *Y* are not supported at present.

Barbs are traditionally used in meteorology as a way to plot the speed and direction of wind observations, but can technically be used to plot any two dimensional vector quantity. As opposed to arrows, which give vector magnitude by the length of the arrow, the barbs give more quantitative information about the vector magnitude by putting slanted lines or a triangle for various increments in magnitude, as show schematically below:

```
:                    /\     \
:                   /  \     \
:                  /    \     \     \
:                 /      \     \     \
:                ------------------------------
```

The largest increment is given by a triangle (or "flag"). After those come full lines (barbs). The smallest increment is a half line. There is only, of course, ever at most 1 half line. If the magnitude is small and only needs a single half-line and no full lines or triangles, the half-line is offset from the

end of the barb so that it can be easily distinguished from barbs with a single full line. The magnitude for the barb shown above would nominally be 65, using the standard increments of 50, 10, and 5.

See also https://en.wikipedia.org/wiki/Wind_barb.

**Parameters**

**X, Y**

[1D or 2D array-like, optional] The x and y coordinates of the barb locations. See *pivot* for how the barbs are drawn to the x, y positions.

If not given, they will be generated as a uniform integer meshgrid based on the dimensions of *U* and *V*.

If *X* and *Y* are 1D but *U*, *V* are 2D, *X*, *Y* are expanded to 2D using `X, Y = np.meshgrid(X, Y)`. In this case `len(X)` and `len(Y)` must match the column and row dimensions of *U* and *V*.

**U, V**

[1D or 2D array-like] The x and y components of the barb shaft.

**C**

[1D or 2D array-like, optional] Numeric data that defines the barb colors by colormapping via *norm* and *cmap*.

This does not support explicit colors. If you want to set colors directly, use *barbcolor* instead.

**length**

[float, default: 7] Length of the barb in points; the other parts of the barb are scaled against this.

**pivot**

[{'tip', 'middle'} or float, default: 'tip'] The part of the arrow that is anchored to the *X*, *Y* grid. The barb rotates about this point. This can also be a number, which shifts the start of the barb that many points away from grid point.

**barbcolor**

[*color* or color sequence] The color of all parts of the barb except for the flags. This parameter is analogous to the *edgecolor* parameter for polygons, which can be used instead. However this parameter will override facecolor.

**flagcolor**

[*color* or color sequence] The color of any flags on the barb. This parameter is analogous to the *facecolor* parameter for polygons, which can be used instead. However, this parameter will override facecolor. If this is not set (and *C* has not either) then *flagcolor* will be set to match *barbcolor* so that the barb has a uniform color. If *C* has been set, *flagcolor* has no effect.

**sizes**

[dict, optional] A dictionary of coefficients specifying the ratio of a given feature to the length of the barb. Only those values one wishes to override need to be included. These features include:

- 'spacing' - space between features (flags, full/half barbs)
- 'height' - height (distance from shaft to top) of a flag or full barb
- 'width' - width of a flag, twice the width of a full barb
- 'emptybarb' - radius of the circle used for low magnitudes

**fill_empty**

[bool, default: False] Whether the empty barbs (circles) that are drawn should be filled with the flag color. If they are not filled, the center is transparent.

**rounding**

[bool, default: True] Whether the vector magnitude should be rounded when allocating barb components. If True, the magnitude is rounded to the nearest multiple of the half-barb increment. If False, the magnitude is simply truncated to the next lowest multiple.

**barb_increments**

[dict, optional] A dictionary of increments specifying values to associate with different parts of the barb. Only those values one wishes to override need to be included.

- 'half' - half barbs (Default is 5)
- 'full' - full barbs (Default is 10)
- 'flag' - flags (default is 50)

**flip_barb**

[bool or array-like of bool, default: False] Whether the lines and flags should point opposite to normal. Normal behavior is for the barbs and lines to point right (comes from wind barbs having these features point towards low pressure in the Northern Hemisphere).

A single value is applied to all barbs. Individual barbs can be flipped by passing a bool array of the same size as *U* and *V*.

**Returns**

**barbs**

[*Barbs*]

**Other Parameters**

**data**

[indexable object, optional] DATA_PARAMETER_PLACEHOLDER

****kwargs**

The barbs can further be customized using *PolyCollection* keyword arguments:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a d |
| *alpha* | array-like or scalar or None |
| *animated* | bool |
| *antialiased* or aa or antialiaseds | bool or list of bools |
| *array* | array-like or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clim* | (vmin: float, vmax: float) |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *cmap* | *Colormap* or str or None |
| *color* | *color* or list of RGBA tuples |
| *edgecolor* or ec or edgecolors | *color* or list of *color* or 'face' |
| *facecolor* or facecolors or fc | *color* or list of *color* |
| *figure* | *Figure* |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or dashes or linestyles or ls | str or tuple or list thereof |
| *linewidth* or linewidths or lw | float or list of floats |
| *mouseover* | bool |
| *norm* | *Normalize* or str or None |
| *offset_transform* or transOffset | *Transform* |
| *offsets* | (N, 2) or (2,) array-like |
| *path_effects* | list of *AbstractPathEffect* |
| *paths* | list of array-like |
| *picker* | None or bool or float or callable |
| *pickradius* | float |
| *rasterized* | bool |
| sizes | numpy.ndarray or None |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *urls* | list of str or None |

Table 151 – continued fr

| Property | Description |
|---|---|
| *verts* | list of array-like |
| *verts_and_codes* | unknown |
| *visible* | bool |
| *zorder* | float |

**set** (*, *UVC=<UNSET>*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *antialiased=<UNSET>*, *array=<UNSET>*, *capstyle=<UNSET>*, *clim=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *cmap=<UNSET>*, *color=<UNSET>*, *edgecolor=<UNSET>*, *facecolor=<UNSET>*, *gid=<UNSET>*, *hatch=<UNSET>*, *in_layout=<UNSET>*, *joinstyle=<UNSET>*, *label=<UNSET>*, *linestyle=<UNSET>*, *linewidth=<UNSET>*, *mouseover=<UNSET>*, *norm=<UNSET>*, *offset_transform=<UNSET>*, *offsets=<UNSET>*, *path_effects=<UNSET>*, *paths=<UNSET>*, *picker=<UNSET>*, *pickradius=<UNSET>*, *rasterized=<UNSET>*, *sizes=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *urls=<UNSET>*, *verts=<UNSET>*, *verts_and_codes=<UNSET>*, *visible=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *UVC* | unknown |
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi v |
| *alpha* | array-like or scalar or None |
| *animated* | bool |
| *antialiased* or aa or antialiaseds | bool or list of bools |
| *array* | array-like or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clim* | (vmin: float, vmax: float) |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *cmap* | *Colormap* or str or None |
| *color* | *color* or list of RGBA tuples |
| *edgecolor* or ec or edgecolors | *color* or list of *color* or 'face' |
| *facecolor* or facecolors or fc | *color* or list of *color* |
| *figure* | *Figure* |
| *gid* | str |
| *hatch* | {'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or dashes or linestyles or ls | str or tuple or list thereof |

Table 152 – continued from

| Property | Description |
|---|---|
| *linewidth* or linewidths or lw | float or list of floats |
| *mouseover* | bool |
| *norm* | *Normalize* or str or None |
| *offset_transform* or transOffset | *Transform* |
| *offsets* | sequence of pairs of floats |
| *path_effects* | list of *AbstractPathEffect* |
| *paths* | list of array-like |
| *picker* | None or bool or float or callable |
| *pickradius* | float |
| *rasterized* | bool |
| sizes | numpy.ndarray or None |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *urls* | list of str or None |
| *verts* | list of array-like |
| *verts_and_codes* | unknown |
| *visible* | bool |
| *zorder* | float |

**set_UVC**(*U*, *V*, *C=None*)

**set_offsets**(*xy*)

Set the offsets for the barb polygons. This saves the offsets passed in and masks them as appropriate for the existing U/V data.

**Parameters**

**xy**

[sequence of pairs of floats]

### 7.2.43 matplotlib.rcsetup

The rcsetup module contains the validation code for customization using Matplotlib's rc settings.

Each rc setting is assigned a function used to validate any attempted changes to that setting. The validation functions are defined in the rcsetup module, and are used to construct the rcParams global object which stores the settings and is referenced throughout Matplotlib.

The default values of the rc settings are set in the default matplotlibrc file. Any additions or deletions to the parameter set listed here should also be propagated to the `lib/matplotlib/mpl-data/matplotlibrc` in Matplotlib's root source directory.

**class** matplotlib.rcsetup.**ValidateInStrings**(*key*, *valid*, *ignorecase=False*, *, *_deprecated_since=None*)

Bases: object

*valid* is a list of legal strings.

matplotlib.rcsetup.**cycler**(**args*, ***kwargs*)

Create a Cycler object much like cycler.cycler(), but includes input validation.

Call signatures:

```
cycler(cycler)
cycler(label=values[, label2=values2[, ...]])
cycler(label, values)
```

Form 1 copies a given Cycler object.

Form 2 creates a Cycler which cycles over one or more properties simultaneously. If multiple properties are given, their value lists must have the same length.

Form 3 creates a Cycler for a single property. This form exists for compatibility with the original cycler. Its use is discouraged in favor of the kwarg form, i.e. cycler(label=values).

**Parameters**

**cycler**

[Cycler] Copy constructor for Cycler.

**label**

[str] The property key. Must be a valid *Artist* property. For example, 'color' or 'linestyle'. Aliases are allowed, such as 'c' for 'color' and 'lw' for 'linewidth'.

**values**

[iterable] Finite-length iterable of the property values. These values are validated and will raise a ValueError if invalid.

**Returns**

**Cycler**

A new Cycler for the given properties.

### Examples

Creating a cycler for a single property:

```
>>> c = cycler(color=['red', 'green', 'blue'])
```

Creating a cycler for simultaneously cycling over multiple properties (e.g. red circle, green plus, blue cross):

```
>>> c = cycler(color=['red', 'green', 'blue'],
...            marker=['o', '+', 'x'])
```

matplotlib.rcsetup.**validate_any**(*s*)

matplotlib.rcsetup.**validate_anylist**(*s*)

matplotlib.rcsetup.**validate_aspect**(*s*)

matplotlib.rcsetup.**validate_axisbelow**(*s*)

matplotlib.rcsetup.**validate_backend**(*s*)

matplotlib.rcsetup.**validate_bbox**(*s*)

matplotlib.rcsetup.**validate_bool**(*b*)

Convert b to `bool` or raise.

matplotlib.rcsetup.**validate_color**(*s*)

Return a valid color arg.

matplotlib.rcsetup.**validate_color_for_prop_cycle**(*s*)

matplotlib.rcsetup.**validate_color_or_auto**(*s*)

matplotlib.rcsetup.**validate_color_or_inherit**(*s*)

Return a valid color arg.

matplotlib.rcsetup.**validate_colorlist**(*s*)

return a list of colorspecs

matplotlib.rcsetup.**validate_cycler**(*s*)

Return a Cycler object from a string repr or the object itself.

matplotlib.rcsetup.**validate_dashlist**(*s*)

return a list of floats

matplotlib.rcsetup.**validate_dpi**(*s*)

Confirm s is string 'figure' or convert s to float or raise.

matplotlib.rcsetup.**validate_fillstylelist**(*s*)

matplotlib.rcsetup.**validate_float**(*s*)

matplotlib.rcsetup.**validate_float_or_None**(*s*)

matplotlib.rcsetup.**validate_floatlist**(*s*)

return a list of floats

matplotlib.rcsetup.**validate_font_properties**(*s*)

matplotlib.rcsetup.**validate_fontsize**(*s*)

matplotlib.rcsetup.**validate_fontsize_None**(*s*)

matplotlib.rcsetup.**validate_fontsizelist**(*s*)

matplotlib.rcsetup.**validate_fontstretch**(*s*)

matplotlib.rcsetup.**validate_fonttype**(*s*)

Confirm that this is a Postscript or PDF font type that we know how to convert to.

matplotlib.rcsetup.**validate_fontweight**(*s*)

matplotlib.rcsetup.**validate_hatch**(*s*)

Validate a hatch pattern. A hatch pattern string can have any sequence of the following characters: `\ / | - + * . x o O`.

matplotlib.rcsetup.**validate_hatchlist**(*s*)

Validate a hatch pattern. A hatch pattern string can have any sequence of the following characters: `\ / | - + * . x o O`.

matplotlib.rcsetup.**validate_hist_bins**(*s*)

matplotlib.rcsetup.**validate_int**(*s*)

matplotlib.rcsetup.**validate_int_or_None**(*s*)

matplotlib.rcsetup.**validate_markevery**(*s*)

Validate the markevery property of a Line2D object.

**Parameters**

**s**

[None, int, (int, int), slice, float, (float, float), or list[int]]

**Returns**

**None, int, (int, int), slice, float, (float, float), or list[int]**

matplotlib.rcsetup.**validate_markeverylist**(*s*)

Validate the markevery property of a Line2D object.

**Parameters**

**s**

[None, int, (int, int), slice, float, (float, float), or list[int]]

**Returns**

**None, int, (int, int), slice, float, (float, float), or list[int]**

matplotlib.rcsetup.**validate_ps_distiller**(*s*)

matplotlib.rcsetup.**validate_sketch**(*s*)

matplotlib.rcsetup.**validate_string**(*s*)

matplotlib.rcsetup.**validate_string_or_None**(*s*)

matplotlib.rcsetup.**validate_stringlist**(*s*)

return a list of strings

matplotlib.rcsetup.**validate_whiskers**(*s*)

### 7.2.44 matplotlib.sankey

Module for creating Sankey diagrams using Matplotlib.

**class** matplotlib.sankey.**Sankey**(*ax=None*, *scale=1.0*, *unit=''*, *format='%G'*, *gap=0.25*, *radius=0.1*, *shoulder=0.03*, *offset=0.15*, *head_angle=100*, *margin=0.4*, *tolerance=1e-06*, ***kwargs*)

Bases: object

Sankey diagram.

Sankey diagrams are a specific type of flow diagram, in which the width of the arrows is shown proportionally to the flow quantity. They are typically used to visualize energy or material or cost transfers between processes. Wikipedia (6/1/2011)

Create a new Sankey instance.

The optional arguments listed below are applied to all subdiagrams so that there is consistent alignment and formatting.

In order to draw a complex Sankey diagram, create an instance of *Sankey* by calling it without any kwargs:

```
sankey = Sankey()
```

Then add simple Sankey sub-diagrams:

```
sankey.add() # 1
sankey.add() # 2
#...
sankey.add() # n
```

Finally, create the full diagram:

```
sankey.finish()
```

Or, instead, simply daisy-chain those calls:

```
Sankey().add().add...  .add().finish()
```

**Other Parameters**

**ax**

[*Axes*] Axes onto which the data should be plotted. If *ax* isn't provided, new Axes will be created.

**scale**

[float] Scaling factor for the flows. *scale* sizes the width of the paths in order to maintain proper layout. The same scale is applied to all subdiagrams. The value should be chosen such that the product of the scale and the sum of the inputs is approximately 1.0 (and the product of the scale and the sum of the outputs is approximately -1.0).

**unit**

[str] The physical unit associated with the flow quantities. If *unit* is None, then none of the quantities are labeled.

**format**

[str or callable] A Python number formatting string or callable used to label the flows with their quantities (i.e., a number times a unit, where the unit is given). If a format string is given, the label will be `format % quantity`. If a callable is given, it will be called with `quantity` as an argument.

**gap**

[float] Space between paths that break in/break away to/from the top or bottom.

**radius**

[float] Inner radius of the vertical paths.

**shoulder**

[float] Size of the shoulders of output arrows.

**offset**

[float] Text offset (from the dip or tip of the arrow).

**head_angle**

[float] Angle, in degrees, of the arrow heads (and negative of the angle of the tails).

**margin**

[float] Minimum space between Sankey outlines and the edge of the plot area.

**tolerance**

[float] Acceptable maximum of the magnitude of the sum of flows. The magnitude of the sum of connected flows cannot be greater than *tolerance*.

****kwargs**

Any additional keyword arguments will be passed to `add`, which will create the first subdiagram.

**See also:**

***Sankey.add***
***Sankey.finish***

## Examples

**add**(*patchlabel=''*, *flows=None*, *orientations=None*, *labels=''*, *trunklength=1.0*, *pathlengths=0.25*, *prior=None*, *connect=(0, 0)*, *rotation=0*, ***kwargs*)

Add a simple Sankey diagram with flows at the same hierarchical level.

**Parameters**

**patchlabel**

[str] Label to be placed at the center of the diagram. Note that *label* (not *patchlabel*) can be passed as keyword argument to create an entry in the legend.

**flows**

[list of float] Array of flow values. By convention, inputs are positive and outputs are negative.

Flows are placed along the top of the diagram from the inside out in order of their index within *flows*. They are placed along the sides of the diagram from the top down and along the bottom from the outside in.

If the sum of the inputs and outputs is nonzero, the discrepancy will appear as a cubic Bézier curve along the top and bottom edges of the trunk.

**orientations**

[list of {-1, 0, 1}] List of orientations of the flows (or a single orientation to be used for all flows). Valid values are 0 (inputs from the left, outputs to the right), 1 (from and to the top) or -1 (from and to the bottom).

**labels**

[list of (str or None)] List of labels for the flows (or a single label to be used for all flows). Each label may be *None* (no label), or a labeling string. If an entry is a (possibly empty) string, then the quantity for the corresponding flow will be shown below the string. However, if the *unit* of the main diagram is None, then quantities are never shown, regardless of the value of this argument.

**trunklength**

[float] Length between the bases of the input and output groups (in data-space units).

**pathlengths**

[list of float] List of lengths of the vertical arrows before break-in or after break-away. If a single value is given, then it will be applied to the first (inside) paths on the top and bottom, and the length of all other arrows will be justified accordingly. The *pathlengths* are not applied to the horizontal inputs and outputs.

**prior**

[int] Index of the prior diagram to which this diagram should be connected.

**connect**

[(int, int)] A (prior, this) tuple indexing the flow of the prior diagram and the flow of this diagram which should be connected. If this is the first diagram or *prior* is *None*, *connect* will be ignored.

**rotation**

[float] Angle of rotation of the diagram in degrees. The interpretation of the *orientations* argument will be rotated accordingly (e.g., if *rotation* == 90, an *orientations* entry of 1 means to/from the left). *rotation* is ignored if this diagram is connected to an existing one (using *prior* and *connect*).

**Returns**

**Sankey**

The current *Sankey* instance.

**Other Parameters**

****kwargs**

Additional keyword arguments set *matplotlib.patches.PathPatch* properties, listed below. For example, one may want to use `fill=False` or `label="A legend entry"`.

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and r |
| *alpha* | unknown |
| *animated* | bool |
| *antialiased* or aa | bool or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* | *color* |
| *edgecolor* or ec | *color* or None |
| *facecolor* or fc | *color* or None |
| *figure* | *Figure* |
| *fill* | bool |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float or None |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *zorder* | float |

**See also:**

***Sankey.finish***

**finish**()

Adjust the Axes and return a list of information about the Sankey subdiagram(s).

Returns a list of subdiagrams with the following fields:

| Field | Description |
|---|---|
| *patch* | Sankey outline (a `PathPatch`). |
| *flows* | Flow values (positive for input, negative for output). |
| *angles* | List of angles of the arrows [deg/90]. For example, if the diagram has not been rotated, an input to the top side has an angle of 3 (DOWN), and an output from the top side has an angle of 1 (UP). If a flow has been skipped (because its magnitude is less than *tolerance*), then its angle will be *None*. |
| *tips* | (N, 2)-array of the (x, y) positions of the tips (or "dips") of the flow paths. If the magnitude of a flow is less the *tolerance* of this `Sankey` instance, the flow is skipped and its tip will be at the center of the diagram. |
| *text* | `Text` instance for the diagram label. |
| *texts* | List of `Text` instances for the flow labels. |

**See also:**

***Sankey.add***

### 7.2.45 matplotlib.scale

Scales define the distribution of data values on an axis, e.g. a log scaling. They are defined as subclasses of `ScaleBase`.

See also `axes.Axes.set_xscale` and the scales examples in the documentation.

See *Custom scale* for a full example of defining a custom scale.

Matplotlib also supports non-separable transformations that operate on both `Axis` at the same time. They are known as projections, and defined in `matplotlib.projections`.

**class** matplotlib.scale.**AsinhScale**(*axis*, *, *linear_width=1.0*, *base=10*, *subs='auto'*, ***kwargs*)

Bases: `ScaleBase`

A quasi-logarithmic scale based on the inverse hyperbolic sine (asinh)

For values close to zero, this is essentially a linear scale, but for large magnitude values (either positive or negative) it is asymptotically logarithmic. The transition between these linear and logarithmic regimes is smooth, and has no discontinuities in the function gradient in contrast to the `SymmetricalLogScale` ("symlog") scale.

Specifically, the transformation of an axis coordinate $a$ is $a \to a_0 \sinh^{-1}(a/a_0)$ where $a_0$ is the effective width of the linear region of the transformation. In that region, the transformation is $a \to a + \mathcal{O}(a^3)$. For large values of $a$ the transformation behaves as $a \to a_0 \operatorname{sgn}(a) \ln|a| + \mathcal{O}(1)$.

**Note:** This API is provisional and may be revised in the future based on early user feedback.

**Parameters**

**linear_width**

[float, default: 1] The scale parameter (elsewhere referred to as $a_0$) defining the extent of the quasi-linear region, and the coordinate values beyond which the transformation becomes asymptotically logarithmic.

**base**

[int, default: 10] The number base used for rounding tick locations on a logarithmic scale. If this is less than one, then rounding is to the nearest integer multiple of powers of ten.

**subs**

[sequence of int] Multiples of the number base used for minor ticks. If set to 'auto', this will use built-in defaults, e.g. (2, 5) for base=10.

**auto_tick_multipliers = {3: (2,), 4: (2,), 5: (2,), 8: (2, 4), 10: (2, 5), 16: (2, 4, 8), 64: (4, 16), 1024: (256, 512)}**

**get_transform**()

Return the *Transform* object associated with this scale.

**property linear_width**

**name = 'asinh'**

**set_default_locators_and_formatters**(*axis*)

Set the locators and formatters of *axis* to instances suitable for this scale.

**class** matplotlib.scale.**AsinhTransform**(*linear_width*)

Bases: *Transform*

Inverse hyperbolic-sine transformation used by *AsinhScale*

**Parameters**

**shorthand_name**

[str] A string representing the "name" of the transform. The name carries no significance other than to improve the readability of `str(transform)` when DEBUG=True.

**has_inverse = True**

True if this transform has a corresponding inverse transform.

**input_dims = 1**

The number of input dimensions of this transform. Must be overridden (with integers) in the subclass.

**inverted**()

Return the corresponding inverse transformation.

It holds `x == self.inverted().transform(self.transform(x))`.

The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

**is_separable = True**

True if this transform is separable in the x- and y- dimensions.

**output_dims = 1**

The number of output dimensions of this transform. Must be overridden (with integers) in the subclass.

**transform_non_affine**(*values*)

Apply only the non-affine part of this transformation.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

**Parameters**

**values**

[array] The input values as an array of length *input_dims* or shape (N, *input_dims*).

**Returns**

**array**

The output values as an array of length *output_dims* or shape (N, *output_dims*), depending on the input.

**class** matplotlib.scale.**FuncScale**(*axis*, *functions*)

Bases: *ScaleBase*

Provide an arbitrary scale with user-supplied function for the axis.

**Parameters**

**axis**

[*Axis*] The axis for the scale.

**functions**

[(callable, callable)] two-tuple of the forward and inverse functions for the scale. The forward function must be monotonic.

Both functions must have the signature:

```
def forward(values: array-like) -> array-like
```

**get_transform**()

Return the *FuncTransform* associated with this scale.

**name = 'function'**

**set_default_locators_and_formatters**(*axis*)

Set the locators and formatters of *axis* to instances suitable for this scale.

**class** matplotlib.scale.**FuncScaleLog**(*axis*, *functions*, *base=10*)

Bases: *LogScale*

Provide an arbitrary scale with user-supplied function for the axis and then put on a logarithmic axes.

**Parameters**

**axis**

[*Axis*] The axis for the scale.

**functions**

[(callable, callable)] two-tuple of the forward and inverse functions for the scale. The forward function must be monotonic.

Both functions must have the signature:

```
def forward(values: array-like) -> array-like
```

**base**

[float, default: 10] Logarithmic base of the scale.

**property base**

**get_transform**()

Return the *Transform* associated with this scale.

**name = 'functionlog'**

**class** matplotlib.scale.**FuncTransform**(*forward*, *inverse*)

Bases: *Transform*

A simple transform that takes and arbitrary function for the forward and inverse transform.

**Parameters**

**forward**

[callable] The forward function for the transform. This function must have an inverse and, for best behavior, be monotonic. It must have the signature:

```
def forward(values: array-like) -> array-like
```

**inverse**

[callable] The inverse of the forward function. Signature as `forward`.

**has_inverse = True**

True if this transform has a corresponding inverse transform.

**input_dims = 1**

The number of input dimensions of this transform. Must be overridden (with integers) in the subclass.

**inverted**()

Return the corresponding inverse transformation.

It holds `x == self.inverted().transform(self.transform(x))`.

The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

**is_separable = True**

True if this transform is separable in the x- and y- dimensions.

**output_dims = 1**

The number of output dimensions of this transform. Must be overridden (with integers) in the subclass.

**transform_non_affine**(*values*)

Apply only the non-affine part of this transformation.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

**Parameters**

**values**

[array] The input values as an array of length *input_dims* or shape (N, *input_dims*).

**Returns**

**array**

The output values as an array of length *output_dims* or shape (N, *output_dims*), depending on the input.

**class** matplotlib.scale.**InvertedAsinhTransform**(*linear_width*)

Bases: *Transform*

Hyperbolic sine transformation used by *AsinhScale*

**Parameters**

**shorthand_name**

[str] A string representing the "name" of the transform. The name carries no significance other than to improve the readability of `str(transform)` when DEBUG=True.

**has_inverse = True**

True if this transform has a corresponding inverse transform.

**input_dims = 1**

The number of input dimensions of this transform. Must be overridden (with integers) in the subclass.

**inverted**()

Return the corresponding inverse transformation.

It holds `x == self.inverted().transform(self.transform(x))`.

The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

**is_separable = True**

True if this transform is separable in the x- and y- dimensions.

**output_dims = 1**

The number of output dimensions of this transform. Must be overridden (with integers) in the subclass.

**transform_non_affine**(*values*)

Apply only the non-affine part of this transformation.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

**Parameters**

**values**

[array] The input values as an array of length *input_dims* or shape (N, *input_dims*).

**Returns**

**array**

The output values as an array of length *output_dims* or shape (N, *output_dims*), depending on the input.

**class** matplotlib.scale.**InvertedLogTransform**(*base*)

Bases: *Transform*

**Parameters**

**shorthand_name**

[str] A string representing the "name" of the transform. The name carries no significance other than to improve the readability of `str(transform)` when DEBUG=True.

**has_inverse = True**

True if this transform has a corresponding inverse transform.

**input_dims = 1**

The number of input dimensions of this transform. Must be overridden (with integers) in the subclass.

**inverted**()

Return the corresponding inverse transformation.

It holds `x == self.inverted().transform(self.transform(x))`.

The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

**is_separable = True**

True if this transform is separable in the x- and y- dimensions.

**output_dims = 1**

The number of output dimensions of this transform. Must be overridden (with integers) in the subclass.

**transform_non_affine**(*values*)

Apply only the non-affine part of this transformation.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

**Parameters**

**values**

[array] The input values as an array of length *input_dims* or shape (N, *input_dims*).

**Returns**

**array**

The output values as an array of length *output_dims* or shape (N, *output_dims*), depending on the input.

**class** matplotlib.scale.**InvertedSymmetricalLogTransform**(*base*, *linthresh*, *linscale*)

Bases: *Transform*

**Parameters**

**shorthand_name**

[str] A string representing the "name" of the transform. The name carries no significance other than to improve the readability of `str(transform)` when DEBUG=True.

**has_inverse = True**

True if this transform has a corresponding inverse transform.

**input_dims = 1**

The number of input dimensions of this transform. Must be overridden (with integers) in the subclass.

**inverted**()

Return the corresponding inverse transformation.

It holds `x == self.inverted().transform(self.transform(x))`.

The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

**is_separable = True**

True if this transform is separable in the x- and y- dimensions.

**output_dims = 1**

The number of output dimensions of this transform. Must be overridden (with integers) in the subclass.

**transform_non_affine**(*values*)

Apply only the non-affine part of this transformation.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

**Parameters**

**values**

[array] The input values as an array of length *input_dims* or shape (N, *input_dims*).

**Returns**

**array**

The output values as an array of length *output_dims* or shape (N, *output_dims*), depending on the input.

**class** matplotlib.scale.**LinearScale**(*axis*)

Bases: *ScaleBase*

The default linear scale.

**get_transform**()

Return the transform for linear scaling, which is just the *IdentityTransform*.

**name = 'linear'**

**set_default_locators_and_formatters**(*axis*)

Set the locators and formatters of *axis* to instances suitable for this scale.

**class** matplotlib.scale.**LogScale**(*axis*, *, *base=10*, *subs=None*, *nonpositive='clip'*)

Bases: *ScaleBase*

A standard logarithmic scale. Care is taken to only plot positive values.

**Parameters**

**axis**

[*Axis*] The axis for the scale.

**base**

[float, default: 10] The base of the logarithm.

**nonpositive**

[{'clip', 'mask'}, default: 'clip'] Determines the behavior for non-positive values. They can either be masked as invalid, or clipped to a very small positive number.

**subs**

[sequence of int, default: None] Where to place the subticks between each major tick. For example, in a log10 scale, `[2, 3, 4, 5, 6, 7, 8, 9]` will place 8 logarithmically spaced minor ticks between each major tick.

**property base**

**get_transform**()

Return the *LogTransform* associated with this scale.

**limit_range_for_scale**(*vmin*, *vmax*, *minpos*)

Limit the domain to positive values.

**name = 'log'**

**set_default_locators_and_formatters**(*axis*)

Set the locators and formatters of *axis* to instances suitable for this scale.

**class** matplotlib.scale.**LogTransform**(*base*, *nonpositive='clip'*)

Bases: *Transform*

**Parameters**

**shorthand_name**

[str] A string representing the "name" of the transform. The name carries no significance other than to improve the readability of `str(transform)` when DEBUG=True.

**has_inverse = True**

True if this transform has a corresponding inverse transform.

**input_dims = 1**

The number of input dimensions of this transform. Must be overridden (with integers) in the subclass.

**inverted**()

Return the corresponding inverse transformation.

It holds `x == self.inverted().transform(self.transform(x))`.

The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

**is_separable = True**

True if this transform is separable in the x- and y- dimensions.

**output_dims = 1**

The number of output dimensions of this transform. Must be overridden (with integers) in the subclass.

**transform_non_affine**(*values*)

Apply only the non-affine part of this transformation.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

**Parameters**

**values**

[array] The input values as an array of length *input_dims* or shape (N, *input_dims*).

**Returns**

**array**

The output values as an array of length *output_dims* or shape (N, *output_dims*), depending on the input.

**class** matplotlib.scale.**LogisticTransform**(*nonpositive='mask'*)

Bases: *Transform*

**Parameters**

**shorthand_name**

[str] A string representing the "name" of the transform. The name carries no significance other than to improve the readability of `str(transform)` when DEBUG=True.

**has_inverse = True**

True if this transform has a corresponding inverse transform.

**input_dims = 1**

The number of input dimensions of this transform. Must be overridden (with integers) in the subclass.

**inverted**()

Return the corresponding inverse transformation.

It holds `x == self.inverted().transform(self.transform(x))`.

The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

**is_separable = True**

True if this transform is separable in the x- and y- dimensions.

**output_dims = 1**

The number of output dimensions of this transform. Must be overridden (with integers) in the subclass.

**transform_non_affine**(*values*)

logistic transform (base 10)

**class** matplotlib.scale.**LogitScale**(*axis*, *nonpositive='mask'*, **, one_half='\\frac{1}{2}'*, *use_overline=False*)

Bases: *ScaleBase*

Logit scale for data between zero and one, both excluded.

This scale is similar to a log scale close to zero and to one, and almost linear around 0.5. It maps the interval ]0, 1[ onto ]-infty, +infty[.

**Parameters**

**axis**

[*Axis*] Currently unused.

**nonpositive**

[{'mask', 'clip'}] Determines the behavior for values beyond the open interval ]0, 1[. They can either be masked as invalid, or clipped to a number very close to 0 or 1.

**use_overline**

[bool, default: False] Indicate the usage of survival notation (overline{x}) in place of standard notation (1-x) for probability close to one.

**one_half**

[str, default: r"frac{1}{2}"] The string used for ticks formatter to represent 1/2.

**get_transform**()

Return the *LogitTransform* associated with this scale.

**limit_range_for_scale**(*vmin*, *vmax*, *minpos*)

Limit the domain to values between 0 and 1 (excluded).

**name = 'logit'**

**set_default_locators_and_formatters**(*axis*)

Set the locators and formatters of *axis* to instances suitable for this scale.

**class** matplotlib.scale.**LogitTransform**(*nonpositive='mask'*)

Bases: *Transform*

**Parameters**

**shorthand_name**

[str] A string representing the "name" of the transform. The name carries no significance other than to improve the readability of `str(transform)` when DEBUG=True.

**has_inverse = True**

True if this transform has a corresponding inverse transform.

**input_dims = 1**

The number of input dimensions of this transform. Must be overridden (with integers) in the subclass.

**inverted**()

Return the corresponding inverse transformation.

It holds `x == self.inverted().transform(self.transform(x))`.

The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

**is_separable = True**

True if this transform is separable in the x- and y- dimensions.

**output_dims = 1**

The number of output dimensions of this transform. Must be overridden (with integers) in the subclass.

**transform_non_affine**(*values*)

logit transform (base 10), masked or clipped

**class** matplotlib.scale.**ScaleBase**(*axis*)

Bases: `object`

The base class for all scales.

Scales are separable transformations, working on a single dimension.

Subclasses should override

**name**

The scale's name.

***get_transform()***

A method returning a *Transform*, which converts data coordinates to scaled coordinates. This transform should be invertible, so that e.g. mouse positions can be converted back to data coordinates.

***set_default_locators_and_formatters()***

A method that sets default locators and formatters for an *Axis* that uses this scale.

***limit_range_for_scale()***

An optional method that "fixes" the axis range to acceptable values, e.g. restricting log-scaled axes to positive values.

Construct a new scale.

### Notes

The following note is for scale implementers.

For back-compatibility reasons, scales take an *Axis* object as first argument. However, this argument should not be used: a single scale object should be usable by multiple *Axis*es at the same time.

**get_transform**()

Return the *Transform* object associated with this scale.

**limit_range_for_scale**(*vmin*, *vmax*, *minpos*)

Return the range *vmin*, *vmax*, restricted to the domain supported by this scale (if any).

*minpos* should be the minimum positive value in the data. This is used by log scales to determine a minimum value.

**set_default_locators_and_formatters**(*axis*)

Set the locators and formatters of *axis* to instances suitable for this scale.

**class** matplotlib.scale.**SymmetricalLogScale**(*axis*, *, *base=10*, *linthresh=2*, *subs=None*, *linscale=1*)

Bases: *ScaleBase*

The symmetrical logarithmic scale is logarithmic in both the positive and negative directions from the origin.

Since the values close to zero tend toward infinity, there is a need to have a range around zero that is linear. The parameter *linthresh* allows the user to specify the size of this range (*-linthresh*, *linthresh*).

**Parameters**

**base**

[float, default: 10] The base of the logarithm.

**linthresh**

[float, default: 2] Defines the range `(-x, x)`, within which the plot is linear. This avoids having the plot go to infinity around zero.

**subs**

[sequence of int] Where to place the subticks between each major tick. For example, in a log10 scale: `[2, 3, 4, 5, 6, 7, 8, 9]` will place 8 logarithmically spaced minor ticks between each major tick.

**linscale**

[float, optional] This allows the linear range `(-linthresh, linthresh)` to be stretched relative to the logarithmic range. Its value is the number of decades to use for each half of the linear range. For example, when *linscale* == 1.0 (the default), the space used for the positive and negative halves of the linear range will be equal to one decade in the logarithmic range.

Construct a new scale.

### Notes

The following note is for scale implementers.

For back-compatibility reasons, scales take an *Axis* object as first argument. However, this argument should not be used: a single scale object should be usable by multiple *Axis*es at the same time.

**property base**

**get_transform**()

Return the *SymmetricalLogTransform* associated with this scale.

**property linscale**

**property linthresh**

**name = 'symlog'**

**set_default_locators_and_formatters**(*axis*)

Set the locators and formatters of *axis* to instances suitable for this scale.

**class** matplotlib.scale.**SymmetricalLogTransform**(*base*, *linthresh*, *linscale*)

Bases: *Transform*

**Parameters**

**shorthand_name**

[str] A string representing the "name" of the transform. The name carries no significance other than to improve the readability of `str(transform)` when DEBUG=True.

**has_inverse = True**

True if this transform has a corresponding inverse transform.

**input_dims = 1**

The number of input dimensions of this transform. Must be overridden (with integers) in the subclass.

**inverted**()

Return the corresponding inverse transformation.

It holds `x == self.inverted().transform(self.transform(x))`.

The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

**is_separable = True**

True if this transform is separable in the x- and y- dimensions.

**output_dims = 1**

The number of output dimensions of this transform. Must be overridden (with integers) in the subclass.

**transform_non_affine**(*values*)

Apply only the non-affine part of this transformation.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

**Parameters**

**values**

[array] The input values as an array of length *input_dims* or shape (N, *input_dims*).

**Returns**

**array**

The output values as an array of length *output_dims* or shape (N, *output_dims*), depending on the input.

matplotlib.scale.**get_scale_names**()

Return the names of the available scales.

matplotlib.scale.**register_scale**(*scale_class*)

Register a new kind of scale.

**Parameters**

**scale_class**

[subclass of *ScaleBase*] The scale to register.

matplotlib.scale.**scale_factory**(*scale*, *axis*, ***kwargs*)

Return a scale class by name.

**Parameters**

**scale**

[{'asinh', 'function', 'functionlog', 'linear', 'log', 'logit', 'symlog'}]

**axis**

[*Axis*]

### 7.2.46 matplotlib.sphinxext.mathmpl

#### A role and directive to display mathtext in Sphinx

The mathmpl Sphinx extension creates a mathtext image in Matplotlib and shows it in html output. Thus, it is a true and faithful representation of what you will see if you pass a given LaTeX string to Matplotlib (see *Writing mathematical expressions*).

**Warning:** In most cases, you will likely want to use one of Sphinx's builtin Math extensions instead of this one. The builtin Sphinx math directive uses MathJax to render mathematical expressions, and addresses accessibility concerns that mathmpl doesn't address.

Mathtext may be included in two ways:

1. Inline, using the role:

```
This text uses inline math: :mathmpl:`\alpha > \beta`.
```

   which produces:

   This text uses inline math: $\alpha > \beta$.

2. Standalone, using the directive:

```
Here is some standalone math:

.. mathmpl::

    \alpha > \beta
```

   which produces:

   Here is some standalone math:

$$\alpha > \beta \tag{7.1}$$

#### Options

The mathmpl role and directive both support the following options:

**fontset**

[str, default: 'cm'] The font set to use when displaying math. See *rcParams["mathtext.fontset"]* (default: 'dejavusans').

**fontsize**

[float] The font size, in points. Defaults to the value from the extension configuration option defined below.

### Configuration options

The mathtext extension has the following configuration options:

**mathmpl_fontsize**

[float, default: 10.0] Default font size, in points.

**mathmpl_srcset**

[list of str, default: []] Additional image sizes to generate when embedding in HTML, to support responsive resolution images. The list should contain additional x-descriptors (`'1.5x'`, `'2x'`, etc.) to generate (1x is the default and always included.)

**class** matplotlib.sphinxext.mathmpl.**MathDirective**(*name*, *arguments*, *options*, *content*, *lineno*, *content_offset*, *block_text*, *state*, *state_machine*)

The `.. mathmpl::` directive, as documented in the module's docstring.

**final_argument_whitespace = False**

May the final argument contain whitespace?

**has_content = True**

May the directive have content?

**option_spec = {'fontset': <function fontset_choice>, 'fontsize': <function _make_type_validator.<locals>.validate_float_or_None>}**

Mapping of option names to validator functions.

**optional_arguments = 0**

Number of optional arguments after the required arguments.

**required_arguments = 0**

Number of required directive arguments.

## 7.2.47 matplotlib.sphinxext.plot_directive

### A directive for including a Matplotlib plot in a Sphinx document

This is a Sphinx extension providing a reStructuredText directive `.. plot::` for including a plot in a Sphinx document.

In HTML output, `.. plot::` will include a .png file with a link to a high-res .png and .pdf. In LaTeX output, it will include a .pdf.

The plot content may be defined in one of three ways:

1. **A path to a source file** as the argument to the directive:

```
.. plot:: path/to/plot.py
```

When a path to a source file is given, the content of the directive may optionally contain a caption for the plot:

```
.. plot:: path/to/plot.py

   The plot caption.
```

Additionally, one may specify the name of a function to call (with no arguments) immediately after importing the module:

```
.. plot:: path/to/plot.py plot_function1
```

2. Included as **inline content** to the directive:

```
.. plot::

   import matplotlib.pyplot as plt
   plt.plot([1, 2, 3], [4, 5, 6])
   plt.title("A plotting exammple")
```

3. Using **doctest** syntax:

```
.. plot::

   A plotting example:
   >>> import matplotlib.pyplot as plt
   >>> plt.plot([1, 2, 3], [4, 5, 6])
```

### Options

The `.. plot::` directive supports the following options:

**`:format:`**

[{'python', 'doctest'}] The format of the input. If unset, the format is auto-detected.

**`:include-source:`**

[bool] Whether to display the source code. The default can be changed using the `plot_include_source` variable in `conf.py` (which itself defaults to False).

**`:show-source-link:`**

[bool] Whether to show a link to the source in HTML. The default can be changed using the `plot_html_show_source_link` variable in `conf.py` (which itself defaults to True).

**`:context:`**

[bool or str] If provided, the code will be run in the context of all previous plot directives for which the `:context:` option was specified. This only applies to inline code plot directives, not those run from files. If the `:context: reset` option is specified, the context is reset for this and future plots, and previous figures are closed prior to running the code. `:context: close-figs` keeps the context but closes previous figures before running the code.

**:nofigs:**

[bool] If specified, the code block will be run, but no figures will be inserted. This is usually useful with the `:context:` option.

**:caption:**

[str] If specified, the option's argument will be used as a caption for the figure. This overwrites the caption given in the content, when the plot is generated from a file.

Additionally, this directive supports all the options of the image directive, except for `:target:` (since plot will add its own target). These include `:alt:`, `:height:`, `:width:`, `:scale:`, `:align:` and `:class:`.

### Configuration options

The plot directive has the following configuration options:

**plot_include_source**

Default value for the include-source option (default: False).

**plot_html_show_source_link**

Whether to show a link to the source in HTML (default: True).

**plot_pre_code**

Code that should be executed before each plot. If None (the default), it will default to a string containing:

```
import numpy as np
from matplotlib import pyplot as plt
```

**plot_basedir**

Base directory, to which `plot::` file names are relative to. If None or empty (the default), file names are relative to the directory where the file containing the directive is.

**plot_formats**

File formats to generate (default: ['png', 'hires.png', 'pdf']). List of tuples or strings:

```
[(suffix, dpi), suffix, ...]
```

that determine the file format and the DPI. For entries whose DPI was omitted, sensible defaults are chosen. When passing from the command line through sphinx_build the list should be passed as suffix:dpi,suffix:dpi, ...

**plot_html_show_formats**

Whether to show links to the files in HTML (default: True).

**plot_rcparams**

A dictionary containing any non-standard rcParams that should be applied before each plot (default: {}).

**plot_apply_rcparams**

By default, rcParams are applied when `:context:` option is not used in a plot directive. If set, this configuration option overrides this behavior and applies rcParams before each plot.

**plot_working_directory**

By default, the working directory will be changed to the directory of the example, so the code can get at its data files, if any. Also its path will be added to `sys.path` so it can import any helper modules sitting beside it. This configuration option can be used to specify a central directory (also added to `sys.path`) where data files and helper modules for all code are located.

**plot_template**

Provide a customized template for preparing restructured text.

**plot_srcset**

Allow the srcset image option for responsive image resolutions. List of strings with the multiplicative factors followed by an "x". e.g. ["2.0x", "1.5x"]. "2.0x" will create a png with the default "png" resolution from plot_formats, multiplied by 2. If plot_srcset is specified, the plot directive uses the *matplotlib.sphinxext.figmpl_directive* (instead of the usual figure directive) in the intermediary rst file that is generated. The plot_srcset option is incompatible with *singlehtml* builds, and an error will be raised.

### Notes on how it works

The plot directive runs the code it is given, either in the source file or the code under the directive. The figure created (if any) is saved in the sphinx build directory under a subdirectory named `plot_directive`. It then creates an intermediate rst file that calls a `.. figure:` directive (or `.. figmpl::` directive if `plot_srcset` is being used) and has links to the `*.png` files in the `plot_directive` directory. These translations can be customized by changing the *plot_template*. See the source of *matplotlib.sphinxext.plot_directive* for the templates defined in *TEMPLATE* and *TEMPLATE_SRCSET*.

**class** matplotlib.sphinxext.plot_directive.**PlotDirective**(*name*, *arguments*, *options*, *content*, *lineno*, *content_offset*, *block_text*, *state*, *state_machine*)

The `.. plot::` directive, as documented in the module's docstring.

**final_argument_whitespace = False**

May the final argument contain whitespace?

**has_content = True**

May the directive have content?

**option_spec = {'align': <function Image.align>, 'alt': <function unchanged>, 'caption': <function unchanged>, 'class': <function class_option>, 'context': <function _option_context>, 'format': <function _option_format>, 'height': <function length_or_unitless>, 'include-source': <function _option_boolean>, 'nofigs': <function flag>, 'scale': <function nonnegative_int>, 'show-source-link': <function _option_boolean>, 'width': <function length_or_percentage_or_unitless>}**

Mapping of option names to validator functions.

**optional_arguments = 2**

Number of optional arguments after the required arguments.

**required_arguments = 0**

Number of required directive arguments.

**run**()

Run the plot directive.

**exception** matplotlib.sphinxext.plot_directive.**PlotError**

matplotlib.sphinxext.plot_directive.**mark_plot_labels**(*app*, *document*)

To make plots referenceable, we need to move the reference from the "htmlonly" (or "latexonly") node to the actual figure node itself.

matplotlib.sphinxext.plot_directive.**out_of_date**(*original*, *derived*, *includes=None*)

Return whether *derived* is out-of-date relative to *original* or any of the RST files included in it using the RST include directive (*includes*). *derived* and *original* are full paths, and *includes* is optionally a list of full paths which may have been included in the *original*.

matplotlib.sphinxext.plot_directive.**render_figures**(*code*, *code_path*, *output_dir*, *output_base*, *context*, *function_name*, *config*, *context_reset=False*, *close_figs=False*, *code_includes=None*)

Run a pyplot script and save the images in *output_dir*.

Save the images under *output_dir* with file names derived from *output_base*

### 7.2.48 matplotlib.sphinxext.figmpl_directive

Add a `figure-mpl` directive that is a responsive version of `figure`.

This implementation is very similar to `.. figure::`, except it also allows a `srcset=` argument to be passed to the image tag, hence allowing responsive resolution images.

There is no particular reason this could not be used standalone, but is meant to be used with *matplotlib.sphinxext.plot_directive*.

Note that the directory organization is a bit different than `.. figure::`. See the *FigureMpl* documentation below.

**class** matplotlib.sphinxext.figmpl_directive.**FigureMpl**(*name*, *arguments*, *options*, *content*, *lineno*, *content_offset*, *block_text*, *state*, *state_machine*)

Implements a directive to allow an optional hidpi image.

Meant to be used with the *plot_srcset* configuration option in conf.py, and gets set in the TEMPLATE of plot_directive.py

e.g.:

```
.. figure-mpl:: plot_directive/some_plots-1.png
    :alt: bar
    :srcset: plot_directive/some_plots-1.png,
             plot_directive/some_plots-1.2x.png 2.00x
    :class: plot-directive
```

The resulting html (at `some_plots.html`) is:

```
<img src="sphx_glr_bar_001_hidpi.png"
    srcset="_images/some_plot-1.png,
            _images/some_plots-1.2x.png 2.00x",
    alt="bar"
    class="plot_directive" />
```

Note that the handling of subdirectories is different than that used by the sphinx figure directive:

```
.. figure-mpl:: plot_directive/nestedpage/index-1.png
    :alt: bar
    :srcset: plot_directive/nestedpage/index-1.png
             plot_directive/nestedpage/index-1.2x.png 2.00x
    :class: plot_directive
```

The resulting html (at `nestedpage/index.html`):

```
<img src="../_images/nestedpage-index-1.png"
    srcset="../_images/nestedpage-index-1.png,
            ../_images/_images/nestedpage-index-1.2x.png 2.00x",
    alt="bar"
    class="sphx-glr-single-img" />
```

where the subdirectory is included in the image name for uniqueness.

**final_argument_whitespace = False**

May the final argument contain whitespace?

**has_content = False**

May the directive have content?

**option_spec = {'align': <function Image.align>, 'alt': <function unchanged>, 'caption': <function unchanged>, 'class': <function class_option>, 'height': <function length_or_unitless>, 'scale': <function nonnegative_int>, 'srcset': <function unchanged>, 'width': <function length_or_percentage_or_unitless>}**

Mapping of option names to validator functions.

**optional_arguments = 2**

Number of optional arguments after the required arguments.

**required_arguments = 1**

Number of required directive arguments.

**class** matplotlib.sphinxext.figmpl_directive.**figmplnode**(*rawsource=''*, **children*, ***attributes*)

### 7.2.49 matplotlib.spines

**class** matplotlib.spines.**Spine**(*axes*, *spine_type*, *path*, ***kwargs*)

Bases: *Patch*

An axis spine -- the line noting the data area boundaries.

Spines are the lines connecting the axis tick marks and noting the boundaries of the data area. They can be placed at arbitrary positions. See *set_position* for more information.

The default position is `('outward', 0)`.

Spines are subclasses of *Patch*, and inherit much of their behavior.

Spines draw a line, a circle, or an arc depending on if *set_patch_line*, *set_patch_circle*, or *set_patch_arc* has been called. Line-like is the default.

For examples see *Spines*.

**Parameters**

**axes**

[*Axes*] The *Axes* instance containing the spine.

**spine_type**

[str] The spine type.

**path**

[*Path*] The *Path* instance used to draw the spine.

**Other Parameters**

****kwargs**

Valid keyword arguments are:

| Property | Description |
|---|---|
| agg_filter | a filter function, which takes a (m, n, 3) float array and a dpi value, and return |
| alpha | unknown |
| animated | bool |
| antialiased or aa | bool or None |
| capstyle | CapStyle or {'butt', 'projecting', 'round'} |
| clip_box | BboxBase or None |
| clip_on | bool |
| clip_path | Patch or (Path, Transform) or None |
| color | color |
| edgecolor or ec | color or None |
| facecolor or fc | color or None |
| figure | Figure |
| fill | bool |
| gid | str |
| hatch | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| in_layout | bool |
| joinstyle | JoinStyle or {'miter', 'round', 'bevel'} |
| label | object |
| linestyle or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| linewidth or lw | float or None |
| mouseover | bool |
| path_effects | list of AbstractPathEffect |
| picker | None or bool or float or callable |
| rasterized | bool |
| sketch_params | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| transform | Transform |
| url | str |
| visible | bool |
| zorder | float |

**classmethod arc_spine**(*axes*, *spine_type*, *center*, *radius*, *theta1*, *theta2*, ***kwargs*)

Create and return an arc Spine.

**classmethod circular_spine**(*axes*, *center*, *radius*, ***kwargs*)

Create and return a circular Spine.

**clear**()

Clear the current spine.

**draw**(*renderer*)

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (Artist.get_visible returns False).

**Parameters**

**renderer**

[*RendererBase* subclass.]

### Notes

This method is overridden in the Artist subclasses.

**get_bounds**()

Get the bounds of the spine.

**get_patch_transform**()

Return the *Transform* instance mapping patch coordinates to data coordinates.

For example, one may define a patch of a circle which represents a radius of 5 by providing coordinates for a unit circle, and a transform which scales the coordinates (the patch coordinate) by 5.

**get_path**()

Return the path of this patch.

**get_position**()

Return the spine position.

**get_spine_transform**()

Return the spine transform.

**get_window_extent**(*renderer=None*)

Return the window extent of the spines in display space, including padding for ticks (but not their labels)

**See also:**

***matplotlib.axes.Axes.get_tightbbox***
***matplotlib.axes.Axes.get_window_extent***

**classmethod linear_spine**(*axes*, *spine_type*, ***kwargs*)

Create and return a linear *Spine*.

**register_axis**(*axis*)

Register an axis.

An axis should be registered with its corresponding spine from the Axes instance. This allows the spine to clear any axis properties when needed.

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *antialiased=<UNSET>*, *bounds=<UNSET>*, *capstyle=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *color=<UNSET>*, *edgecolor=<UNSET>*, *facecolor=<UNSET>*, *fill=<UNSET>*, *gid=<UNSET>*, *hatch=<UNSET>*, *in_layout=<UNSET>*, *joinstyle=<UNSET>*, *label=<UNSET>*, *linestyle=<UNSET>*, *linewidth=<UNSET>*, *mouseover=<UNSET>*, *patch_arc=<UNSET>*, *patch_circle=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *position=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* or aa | bool or None |
| *bounds* | (low: float, high: float) |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* | *color* |
| *edgecolor* or ec | *color* or None |
| *facecolor* or fc | *color* or None |
| *figure* | *Figure* |
| *fill* | bool |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float or None |
| *mouseover* | bool |
| *patch_arc* | unknown |
| *patch_circle* | unknown |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *position* | unknown |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |

Table 155 – continued from previous

| Property | Description |
|---|---|
| *visible* | bool |
| *zorder* | float |

**set_bounds**(*low=None*, *high=None*)

Set the spine bounds.

**Parameters**

**low**

[float or None, optional] The lower spine bound. Passing *None* leaves the limit unchanged.

The bounds may also be passed as the tuple (*low*, *high*) as the first positional argument.

**high**

[float or None, optional] The higher spine bound. Passing *None* leaves the limit unchanged.

**set_color**(*c*)

Set the edgecolor.

**Parameters**

**c**

[*color*]

### Notes

This method does not modify the facecolor (which defaults to "none"), unlike the *Patch.set_color* method defined in the parent class. Use *Patch.set_facecolor* to set the facecolor.

**set_patch_arc**(*center*, *radius*, *theta1*, *theta2*)

Set the spine to be arc-like.

**set_patch_circle**(*center*, *radius*)

Set the spine to be circular.

**set_patch_line**()

Set the spine to be linear.

**set_position**(*position*)

Set the position of the spine.

Spine position is specified by a 2 tuple of (position type, amount). The position types are:

- 'outward': place the spine out from the data area by the specified number of points. (Negative values place the spine inwards.)
- 'axes': place the spine at the specified Axes coordinate (0 to 1).
- 'data': place the spine at the specified data coordinate.

Additionally, shorthand notations define a special positions:

- 'center' -> `('axes', 0.5)`
- 'zero' -> `('data', 0.0)`

### Examples

*Spine placement*

**class** matplotlib.spines.**Spines**(***kwargs*)

Bases: `MutableMapping`

The container of all *Spine*s in an Axes.

The interface is dict-like mapping names (e.g. 'left') to *Spine* objects. Additionally, it implements some pandas.Series-like features like accessing elements by attribute:

```
spines['top'].set_visible(False)
spines.top.set_visible(False)
```

Multiple spines can be addressed simultaneously by passing a list:

```
spines[['top', 'right']].set_visible(False)
```

Use an open slice to address all spines:

```
spines[:].set_visible(False)
```

The latter two indexing methods will return a *SpinesProxy* that broadcasts all `set_*()` and `set()` calls to its members, but cannot be used for any other operation.

**classmethod from_dict**(*d*)

**class** matplotlib.spines.**SpinesProxy**(*spine_dict*)

Bases: `object`

A proxy to broadcast `set_*()` and `set()` method calls to contained *Spines*.

The proxy cannot be used for any other operations on its members.

The supported methods are determined dynamically based on the contained spines. If not all spines support a given method, it's executed only on the subset of spines that support it.

### 7.2.50 matplotlib.style

Styles are predefined sets of *rcParams* that define the visual appearance of a plot.

*Customizing Matplotlib with style sheets and rcParams* describes the mechanism and usage of styles.

The *Style sheets reference* gives an overview of the builtin styles.

matplotlib.style.**context**(*style*, *after_reset=False*)

Context manager for using style settings temporarily.

**Parameters**

**style**

[str, dict, Path or list] A style specification. Valid options are:

**str**

- One of the style names in *style.available* (a builtin style or a style installed in the user library path).
- A dotted name of the form "package.style_name"; in that case, "package" should be an importable Python package name, e.g. at `/path/to/package/__init__.py`; the loaded style file is `/path/to/package/style_name.mplstyle`. (Style files in subpackages are likewise supported.)
- The path or URL to a style file, which gets loaded by *rc_params_from_file*.

**dict**

A mapping of key/value pairs for *matplotlib.rcParams*.

**Path**

The path to a style file, which gets loaded by *rc_params_from_file*.

**list**

A list of style specifiers (str, Path or dict), which are applied from first to last in the list.

**after_reset**

[bool] If True, apply style after resetting settings to their defaults; otherwise, apply style on top of the current settings.

matplotlib.style.**reload_library**()

Reload the style library.

matplotlib.style.**use**(*style*)

Use Matplotlib style settings from a style specification.

The style name of 'default' is reserved for reverting back to the default style settings.

---

**Note:** This updates the *rcParams* with the settings from the style. *rcParams* not defined in the style are kept.

---

**Parameters**

**style**

[str, dict, Path or list] A style specification. Valid options are:

**str**

- One of the style names in *style.available* (a builtin style or a style installed in the user library path).
- A dotted name of the form "package.style_name"; in that case, "package" should be an importable Python package name, e.g. at `/path/to/package/__init__.py`; the loaded style file is `/path/to/package/style_name.mplstyle`. (Style files in subpackages are likewise supported.)
- The path or URL to a style file, which gets loaded by *rc_params_from_file*.

**dict**

A mapping of key/value pairs for *matplotlib.rcParams*.

**Path**

The path to a style file, which gets loaded by *rc_params_from_file*.

**list**

A list of style specifiers (str, Path or dict), which are applied from first to last in the list.

## Notes

The following *rcParams* are not related to style and will be ignored if found in a style specification:

- backend
- backend_fallback
- date.epoch
- docstring.hardcopy

- figure.max_open_warning
- figure.raise_window
- interactive
- savefig.directory
- timezone
- tk.window_focus
- toolbar
- webagg.address
- webagg.open_in_browser
- webagg.port
- webagg.port_retries

matplotlib.style.**library**

A dict mapping from style name to *rcParams* defining that style.

This is meant to be read-only. Use *reload_library* to update.

matplotlib.style.**available**

List of the names of the available styles.

This is meant to be read-only. Use *reload_library* to update.

### 7.2.51 matplotlib.table

Tables drawing.

---

**Note:** The table implementation in Matplotlib is lightly maintained. For a more featureful table implementation, you may wish to try blume.

---

Use the factory function *table* to create a ready-made table from texts. If you need more control, use the *Table* class and its methods.

The table consists of a grid of cells, which are indexed by (row, column). The cell (0, 0) is positioned at the top left.

Thanks to John Gill for providing the class and table.

**class** matplotlib.table.**Cell**(*xy*, *width*, *height*, ***, *edgecolor='k'*, *facecolor='w'*, *fill=True*, *text=''*, *loc='right'*, *fontproperties=None*, *visible_edges='closed'*)

Bases: *Rectangle*

A cell is a *Rectangle* with some associated *Text*.

As a user, you'll most likely not creates cells yourself. Instead, you should use either the *table* factory function or *Table.add_cell*.

**Parameters**

**xy**

[2-tuple] The position of the bottom left corner of the cell.

**width**

[float] The cell width.

**height**

[float] The cell height.

**edgecolor**

[*color*, default: 'k'] The color of the cell border.

**facecolor**

[*color*, default: 'w'] The cell facecolor.

**fill**

[bool, default: True] Whether the cell background is filled.

**text**

[str, optional] The cell text.

**loc**

[{'right', 'center', 'left'}] The alignment of the text within the cell.

**fontproperties**

[dict, optional] A dict defining the font properties of the text. Supported keys and values are the keyword arguments accepted by *FontProperties*.

**visible_edges**

[{'closed', 'open', 'horizontal', 'vertical'} or substring of 'BRTL'] The cell edges to be drawn with a line: a substring of 'BRTL' (bottom, right, top, left), or one of 'open' (no edges drawn), 'closed' (all edges drawn), 'horizontal' (bottom and top), 'vertical' (right and left).

**PAD = 0.1**

Padding between text and rectangle.

**auto_set_font_size**(*renderer*)

Shrink font size until the text fits into the cell width.

**draw**(*renderer*)

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

**Parameters**

**renderer**

[*RendererBase* subclass.]

### Notes

This method is overridden in the Artist subclasses.

**get_fontsize**()

Return the cell fontsize.

**get_path**()

Return a *Path* for the *visible_edges*.

**get_required_width**(*renderer*)

Return the minimal required width for the cell.

**get_text**()

Return the cell *Text* instance.

**get_text_bounds**(*renderer*)

Return the text bounds as *(x, y, width, height)* in table coordinates.

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *angle=<UNSET>*, *animated=<UNSET>*, *antialiased=<UNSET>*, *bounds=<UNSET>*, *capstyle=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *color=<UNSET>*, *edgecolor=<UNSET>*, *facecolor=<UNSET>*, *fill=<UNSET>*, *fontsize=<UNSET>*, *gid=<UNSET>*, *hatch=<UNSET>*, *height=<UNSET>*, *in_layout=<UNSET>*, *joinstyle=<UNSET>*, *label=<UNSET>*, *linestyle=<UNSET>*, *linewidth=<UNSET>*, *mouseover=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *text_props=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *width=<UNSET>*, *x=<UNSET>*, *xy=<UNSET>*, *y=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a |
| *alpha* | scalar or None |
| *angle* | unknown |

Table 156 – continued from previous

| Property | Description |
|---|---|
| *animated* | bool |
| *antialiased* or aa | bool or None |
| *bounds* | (left, bottom, width, height) |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* | *color* |
| *edgecolor* or ec | *color* or None |
| *facecolor* or fc | *color* or None |
| *figure* | unknown |
| *fill* | bool |
| *fontsize* | unknown |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *height* | unknown |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float or None |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *text_props* | unknown |
| *transform* | unknown |
| *url* | str |
| *visible* | bool |
| *width* | unknown |
| *x* | unknown |
| *xy* | (float, float) |
| *y* | unknown |
| *zorder* | float |

**set_figure**(*fig*)

Set the *Figure* instance the artist belongs to.

**Parameters**

**fig**

[*Figure*]

**set_fontsize**(*size*)

Set the text fontsize.

**set_text_props**(***kwargs*)

Update the text properties.

Valid keyword arguments are:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* | bool |
| *backgroundcolor* | *color* |
| *bbox* | dict with properties for *patches.FancyBboxPatch* |
| *clip_box* | unknown |
| *clip_on* | unknown |
| *clip_path* | unknown |
| *color* or c | *color* |
| *figure* | *Figure* |
| *fontfamily* or family or fontname | {FONTNAME, 'serif', 'sans-serif', 'cursive', 'fantasy', 'mo |
| *fontproperties* or font or font_properties | *font_manager.FontProperties* or *str* or *pat* |
| *fontsize* or size | float or {'xx-small', 'x-small', 'small', 'medium', 'large', 'x |
| *fontstretch* or stretch | {a numeric value in range 0-1000, 'ultra-condensed', 'ext |
| *fontstyle* or style | {'normal', 'italic', 'oblique'} |
| *fontvariant* or variant | {'normal', 'small-caps'} |
| *fontweight* or weight | {a numeric value in range 0-1000, 'ultralight', 'light', 'nor |
| *gid* | str |
| *horizontalalignment* or ha | {'left', 'center', 'right'} |
| *in_layout* | bool |
| *label* | object |
| *linespacing* | float (multiple of font size) |
| *math_fontfamily* | str |
| *mouseover* | bool |
| *multialignment* or ma | {'left', 'right', 'center'} |
| *parse_math* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *position* | (float, float) |
| *rasterized* | bool |
| *rotation* | float or {'vertical', 'horizontal'} |
| *rotation_mode* | {None, 'default', 'anchor'} |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |

Table 157 – co

| Property | Description |
|---|---|
| *text* | object |
| *transform* | *Transform* |
| *transform_rotates_text* | bool |
| *url* | str |
| *usetex* | bool, default: *rcParams["text.usetex"]* (defaul |
| *verticalalignment* or va | {'baseline', 'bottom', 'center', 'center_baseline', 'top'} |
| *visible* | bool |
| *wrap* | bool |
| *x* | float |
| *y* | float |
| *zorder* | float |

**set_transform**(*t*)

Set the artist transform.

**Parameters**

**t**

[*Transform*]

**property visible_edges**

The cell edges to be drawn with a line.

Reading this property returns a substring of 'BRTL' (bottom, right, top, left').

When setting this property, you can use a substring of 'BRTL' or one of {'open', 'closed', 'horizontal', 'vertical'}.

matplotlib.table.**CustomCell**

alias of *Cell*

**class** matplotlib.table.**Table**(*ax*, *loc=None*, *bbox=None*, ***kwargs*)

Bases: *Artist*

A table of cells.

The table consists of a grid of cells, which are indexed by (row, column).

For a simple table, you'll have a full grid of cells with indices from (0, 0) to (num_rows-1, num_cols-1), in which the cell (0, 0) is positioned at the top left. However, you can also add cells with negative indices. You don't have to add a cell to every grid position, so you can create tables that have holes.

*Note*: You'll usually not create an empty table from scratch. Instead use *table* to create a table from data.

**Parameters**

**ax**

[*Axes*] The *Axes* to plot the table into.

**loc**

[str, optional] The position of the cell with respect to *ax*. This must be one of the *codes*.

**bbox**

[*Bbox* or [xmin, ymin, width, height], optional] A bounding box to draw the table into. If this is not *None*, this overrides *loc*.

**Other Parameters**

****kwargs**

*Artist* properties.

**AXESPAD = 0.02**

The border between the Axes and the table edge in Axes units.

**FONTSIZE = 10**

**add_cell**(*row*, *col*, **args*, ***kwargs*)

Create a cell and add it to the table.

**Parameters**

**row**

[int] Row index.

**col**

[int] Column index.

***args, ****kwargs**

All other parameters are passed on to *Cell*.

**Returns**

***Cell***

The created cell.

**auto_set_column_width**(*col*)

Automatically set the widths of given columns to optimal sizes.

**Parameters**

**col**

[int or sequence of ints] The indices of the columns to auto-scale.

**auto_set_font_size**(*value=True*)

Automatically set font size.

**codes = {'best': 0, 'bottom': 17, 'bottom left': 12, 'bottom right': 13, 'center': 9, 'center left': 5, 'center right': 6, 'left': 15, 'lower center': 7, 'lower left': 3, 'lower right': 4, 'right': 14, 'top': 16, 'top left': 11, 'top right': 10, 'upper center': 8, 'upper left': 2, 'upper right': 1}**

Possible values where to place the table relative to the Axes.

**contains**(*mouseevent*)

Test whether the artist contains the mouse event.

**Parameters**

**mouseevent**

[*MouseEvent*]

**Returns**

**contains**

[bool] Whether any values are within the radius.

**details**

[dict] An artist-specific dictionary of details of the event context, such as which points are contained in the pick radius. See the individual Artist subclasses for details.

**draw**(*renderer*)

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

**Parameters**

**renderer**

[*RendererBase* subclass.]

### Notes

This method is overridden in the Artist subclasses.

**property edges**

The default value of *visible_edges* for newly added cells using *add_cell*.

### Notes

This setting does currently only affect newly created cells using *add_cell*.

To change existing cells, you have to set their edges explicitly:

```
for c in tab.get_celld().values():
    c.visible_edges = 'horizontal'
```

**get_celld**()

Return a dict of cells in the table mapping *(row, column)* to *Cell*s.

### Notes

You can also directly index into the Table object to access individual cells:

```
cell = table[row, col]
```

**get_children**()

Return the Artists contained by the table.

**get_window_extent**(*renderer=None*)

Get the artist's bounding box in display space.

The bounding box' width and height are nonnegative.

Subclasses should override for inclusion in the bounding box "tight" calculation. Default is to return an empty bounding box at 0, 0.

Be careful when using this function, the results will not update if the artist window extent of the artist changes. The extent can change due to any changes in the transform stack, such as changing the Axes limits, the figure size, or the canvas used (as is done when saving a figure). This can lead to unexpected behavior where interactive figures will look fine on the screen, but will save incorrectly.

**scale**(*xscale*, *yscale*)

Scale column widths by *xscale* and row heights by *yscale*.

**set**(**, agg_filter=<UNSET>, alpha=<UNSET>, animated=<UNSET>, clip_box=<UNSET>, clip_on=<UNSET>, clip_path=<UNSET>, fontsize=<UNSET>, gid=<UNSET>, in_layout=<UNSET>, label=<UNSET>, mouseover=<UNSET>, path_effects=<UNSET>, picker=<UNSET>, rasterized=<UNSET>, sketch_params=<UNSET>, snap=<UNSET>, transform=<UNSET>, url=<UNSET>, visible=<UNSET>, zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| agg_fil | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image |
| alpha | scalar or None |
| animated | bool |
| clip_bo | BboxBase or None |
| clip_on | bool |
| clip_pa | Patch or (Path, Transform) or None |
| figure | Figure |
| fontsize | float |
| gid | str |
| in_layo | bool |
| label | object |
| mouseov | bool |
| path_ef | list of AbstractPathEffect |
| picker | None or bool or float or callable |
| rasterized | bool |
| sketch_ | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| transform | Transform |
| url | str |
| visible | bool |
| zorder | float |

**set_fontsize**(*size*)

Set the font size, in points, of the cell text.

**Parameters**

**size**

[float]

### Notes

As long as auto font size has not been disabled, the value will be clipped such that the text fits horizontally into the cell.

You can disable this behavior using *auto_set_font_size*.

```
>>> the_table.auto_set_font_size(False)
>>> the_table.set_fontsize(20)
```

However, there is no automatic scaling of the row height so that the text may exceed the cell boundary.

matplotlib.table.**table**(*ax*, *cellText=None*, *cellColours=None*, *cellLoc='right'*, *colWidths=None*, *rowLabels=None*, *rowColours=None*, *rowLoc='left'*, *colLabels=None*, *colColours=None*, *colLoc='center'*, *loc='bottom'*, *bbox=None*, *edges='closed'*, ***kwargs*)

Add a table to an *Axes*.

At least one of *cellText* or *cellColours* must be specified. These parameters must be 2D lists, in which the outer lists define the rows and the inner list define the column values per row. Each row must have the same number of elements.

The table can optionally have row and column headers, which are configured using *rowLabels*, *rowColours*, *rowLoc* and *colLabels*, *colColours*, *colLoc* respectively.

For finer grained control over tables, use the *Table* class and add it to the Axes with *Axes.add_table*.

**Parameters**

**cellText**

[2D list of str, optional] The texts to place into the table cells.

*Note*: Line breaks in the strings are currently not accounted for and will result in the text exceeding the cell boundaries.

**cellColours**

[2D list of *color*, optional] The background colors of the cells.

**cellLoc**

[{'right', 'center', 'left'}] The alignment of the text within the cells.

**colWidths**

[list of float, optional] The column widths in units of the axes. If not given, all columns will have a width of *1 / ncols*.

**rowLabels**

[list of str, optional] The text of the row header cells.

**rowColours**

[list of *color*, optional] The colors of the row header cells.

**rowLoc**

[{'left', 'center', 'right'}] The text alignment of the row header cells.

**colLabels**

[list of str, optional] The text of the column header cells.

**colColours**

[list of *color*, optional] The colors of the column header cells.

**colLoc**

[{'center', 'left', 'right'}] The text alignment of the column header cells.

**loc**

[str, default: 'bottom'] The position of the cell with respect to *ax*. This must be one of the `codes`.

**bbox**

[`Bbox` or [xmin, ymin, width, height], optional] A bounding box to draw the table into. If this is not *None*, this overrides *loc*.

**edges**

[{'closed', 'open', 'horizontal', 'vertical'} or substring of 'BRTL'] The cell edges to be drawn with a line. See also `visible_edges`.

### Returns

***Table***

The created table.

### Other Parameters

****kwargs**

`Table` properties.

| Property | Description |
|---|---|
| *agg_fil* | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image |
| *alpha* | scalar or None |
| *animated* | bool |
| *clip_bo* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_pa* | Patch or (Path, Transform) or None |
| *figure* | *Figure* |
| *fontsize* | float |
| *gid* | str |
| *in_layo* | bool |
| *label* | object |
| *mouseov* | bool |
| *path_ef* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *rasterized* | bool |
| *sketch_* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *zorder* | float |

### 7.2.52 matplotlib.testing

#### matplotlib.testing

Helper functions for testing.

matplotlib.testing.**ipython_in_subprocess**(*requested_backend_or_gui_framework*, *all_expected_backends*)

matplotlib.testing.**set_font_settings_for_testing**()

matplotlib.testing.**set_reproducibility_for_testing**()

matplotlib.testing.**setup**()

matplotlib.testing.**subprocess_run_for_testing**(*command*, *env=None*, *timeout=60*, *stdout=None*, *stderr=None*, *check=False*, *text=True*, *capture_output=False*)

Create and run a subprocess.

Thin wrapper around `subprocess.run`, intended for testing. Will mark fork() failures on Cygwin as expected failures: not a success, but not indicating a problem with the code either.

**Parameters**

**args**

[list of str]

**env**

[dict[str, str]]

**timeout**

[float]

**stdout, stderr**
**check**

[bool]

**text**

[bool] Also called `universal_newlines` in subprocess. I chose this name since the main effect is returning bytes (`False`) vs. str (`True`), though it also tries to normalize newlines across platforms.

**capture_output**

[bool] Set stdout and stderr to subprocess.PIPE

**Returns**

**proc**

[subprocess.Popen]

**Raises**

**pytest.xfail**

If platform is Cygwin and subprocess reports a fork() failure.

**See also:**

**`subprocess.run`**

matplotlib.testing.**subprocess_run_helper**(*func*, **args*, *timeout*, *extra_env=None*)

Run a function in a sub-process.

**Parameters**

**func**

[function] The function to be run. It must be in a module that is importable.

***args**

[str] Any additional command line arguments to be passed in the first argument to `subprocess.run`.

**extra_env**

[dict[str, str]] Any additional environment variables to be set for the subprocess.

### matplotlib.testing.compare

Utilities for comparing image results.

matplotlib.testing.compare.**calculate_rms**(*expected_image*, *actual_image*)

Calculate the per-pixel errors, then compute the root mean square error.

matplotlib.testing.compare.**comparable_formats**()

Return the list of file formats that *compare_images* can compare on this system.

**Returns**

**list of str**

E.g. `['png', 'pdf', 'svg', 'eps']`.

matplotlib.testing.compare.**compare_images**(*expected*, *actual*, *tol*, *in_decorator=False*)

Compare two "image" files checking differences within a tolerance.

The two given filenames may point to files which are convertible to PNG via the `converter` dictionary. The underlying RMS is calculated with the *calculate_rms* function.

**Parameters**

**expected**

[str] The filename of the expected image.

**actual**

[str] The filename of the actual image.

**tol**

[float] The tolerance (a color value difference, where 255 is the maximal difference). The test fails if the average pixel difference is greater than this value.

**in_decorator**

[bool] Determines the output format. If called from image_comparison decorator, this should be True. (default=False)

**Returns**

**None or dict or str**

Return *None* if the images are equal within the given tolerance.

If the images differ, the return value depends on *in_decorator*. If *in_decorator* is true, a dict with the following entries is returned:

- *rms*: The RMS of the image difference.
- *expected*: The filename of the expected image.
- *actual*: The filename of the actual image.
- *diff_image*: The filename of the difference image.
- *tol*: The comparison tolerance.

Otherwise, a human-readable multi-line string representation of this information is returned.

### Examples

```
img1 = "./baseline/plot.png"
img2 = "./output/plot.png"
compare_images(img1, img2, 0.001)
```

## matplotlib.testing.decorators

matplotlib.testing.decorators.**check_figures_equal**(*, *extensions=('png', 'pdf', 'svg')*, *tol=0*)

Decorator for test cases that generate and compare two figures.

The decorated function must take two keyword arguments, *fig_test* and *fig_ref*, and draw the test and reference images on them. After the function returns, the figures are saved and compared.

This decorator should be preferred over *image_comparison* when possible in order to keep the size of the test suite from ballooning.

**Parameters**

**extensions**

[list, default: ["png", "pdf", "svg"]] The extensions to test.

**tol**

[float] The RMS threshold above which the test is considered failed.

**Raises**

**RuntimeError**

If any new figures are created (and not subsequently closed) inside the test function.

### Examples

Check that calling *Axes.plot* with a single argument plots it against `[0, 1, 2, ...]`:

```
@check_figures_equal()
def test_plot(fig_test, fig_ref):
    fig_test.subplots().plot([1, 3, 5])
    fig_ref.subplots().plot([0, 1, 2], [1, 3, 5])
```

matplotlib.testing.decorators.**image_comparison**(*baseline_images*, *extensions=None*, *tol=0*, *freetype_version=None*, *remove_text=False*, *savefig_kwarg=None*, *style=('classic', '_classic_test_patch')*)

Compare images generated by the test with those specified in *baseline_images*, which must correspond, else an `ImageComparisonFailure` exception will be raised.

**Parameters**

**baseline_images**

[list or None] A list of strings specifying the names of the images generated by calls to *Figure.savefig*.

If *None*, the test function must use the `baseline_images` fixture, either as a parameter or with `pytest.mark.usefixtures`. This value is only allowed when using pytest.

**extensions**

[None or list of str] The list of extensions to test, e.g. `['png', 'pdf']`.

If *None*, defaults to all supported extensions: png, pdf, and svg.

When testing a single extension, it can be directly included in the names passed to *baseline_images*. In that case, *extensions* must not be set.

In order to keep the size of the test suite from ballooning, we only include the `svg` or `pdf` outputs if the test is explicitly exercising a feature dependent on that backend (see also the *check_figures_equal* decorator for that purpose).

**tol**

[float, default: 0] The RMS threshold above which the test is considered failed.

Due to expected small differences in floating-point calculations, on 32-bit systems an additional 0.06 is added to this threshold.

**freetype_version**

[str or tuple] The expected freetype version or range of versions for this test to pass.

**remove_text**

[bool] Remove the title and tick text from the figure before comparison. This is useful to make the baseline images independent of variations in text rendering between different versions of FreeType.

This does not remove other, more deliberate, text, such as legends and annotations.

**savefig_kwarg**

[dict] Optional arguments that are passed to the savefig method.

**style**

[str, dict, or list] The optional style(s) to apply to the image test. The test itself can also apply additional styles if desired. Defaults to `["classic", "_classic_test_patch"]`.

matplotlib.testing.decorators.**remove_ticks_and_titles**(*figure*)

### matplotlib.testing.exceptions

**exception** matplotlib.testing.exceptions.**ImageComparisonFailure**

Bases: AssertionError

Raise this exception to mark a test as a comparison between two images.

## 7.2.53 matplotlib.text

Classes for including text in a figure.

**class** matplotlib.text.**Text**(*x=0*, *y=0*, *text=''*, ***, *color=None*, *verticalalignment='baseline'*, *horizontalalignment='left'*, *multialignment=None*, *fontproperties=None*, *rotation=None*, *linespacing=None*, *rotation_mode=None*, *usetex=None*, *wrap=False*, *transform_rotates_text=False*, *parse_math=None*, *antialiased=None*, ***kwargs*)

Bases: *Artist*

Handle storing and drawing of text in window or data coordinates.

Create a *Text* instance at *x*, *y* with string *text*.

The text is aligned relative to the anchor point (*x*, *y*) according to `horizontalalignment` (default: 'left') and `verticalalignment` (default: 'baseline'). See also *Text alignment*.

While Text accepts the 'label' keyword argument, by default it is not added to the handles of a legend.

Valid keyword arguments are:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* | bool |
| *backgroundcolor* | *color* |
| *bbox* | dict with properties for *patches.FancyBboxPatch* |
| *clip_box* | unknown |
| *clip_on* | unknown |
| *clip_path* | unknown |
| *color* or c | *color* |
| *figure* | *Figure* |
| *fontfamily* or family or fontname | {FONTNAME, 'serif', 'sans-serif', 'cursive', 'fantasy', 'monosp |
| *fontproperties* or font or font_properties | *font_manager.FontProperties* or `str` or `pathli` |
| *fontsize* or size | float or {'xx-small', 'x-small', 'small', 'medium', 'large', 'x-larg |
| *fontstretch* or stretch | {a numeric value in range 0-1000, 'ultra-condensed', 'extra-co |
| *fontstyle* or style | {'normal', 'italic', 'oblique'} |
| *fontvariant* or variant | {'normal', 'small-caps'} |
| *fontweight* or weight | {a numeric value in range 0-1000, 'ultralight', 'light', 'normal' |
| *gid* | str |
| *horizontalalignment* or ha | {'left', 'center', 'right'} |
| *in_layout* | bool |
| *label* | object |
| *linespacing* | float (multiple of font size) |
| *math_fontfamily* | str |
| *mouseover* | bool |
| *multialignment* or ma | {'left', 'right', 'center'} |
| *parse_math* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *position* | (float, float) |
| *rasterized* | bool |
| *rotation* | float or {'vertical', 'horizontal'} |
| *rotation_mode* | {None, 'default', 'anchor'} |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *text* | object |
| *transform* | *Transform* |

Table 158 – continu

| Property | Description |
|---|---|
| *transform_rotates_text* | bool |
| *url* | str |
| *usetex* | bool, default: *rcParams["text.usetex"]* (default: Fa |
| *verticalalignment* or va | {'baseline', 'bottom', 'center', 'center_baseline', 'top'} |
| *visible* | bool |
| *wrap* | bool |
| *x* | float |
| *y* | float |
| *zorder* | float |

**contains**(*mouseevent*)

Return whether the mouse event occurred inside the axis-aligned bounding-box of the text.

**draw**(*renderer*)

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

**Parameters**

**renderer**

[*RendererBase* subclass.]

### Notes

This method is overridden in the Artist subclasses.

**get_antialiased**()

Return whether antialiased rendering is used.

**get_bbox_patch**()

Return the bbox Patch, or None if the *patches.FancyBboxPatch* is not made.

**get_c**()

Alias for *get_color*.

**get_color**()

Return the color of the text.

**get_family**()

Alias for *get_fontfamily*.

**get_font**()

Alias for *get_fontproperties*.

**get_font_properties**()

Alias for *get_fontproperties*.

**get_fontfamily**()

Return the list of font families used for font lookup.

**See also:**

***font_manager.FontProperties.get_family***

**get_fontname**()

Return the font name as a string.

**See also:**

***font_manager.FontProperties.get_name***

**get_fontproperties**()

Return the *font_manager.FontProperties*.

**get_fontsize**()

Return the font size as an integer.

**See also:**

***font_manager.FontProperties.get_size_in_points***

**get_fontstyle**()

Return the font style as a string.

**See also:**

***font_manager.FontProperties.get_style***

**get_fontvariant**()

Return the font variant as a string.

**See also:**

***font_manager.FontProperties.get_variant***

**get_fontweight**()

Return the font weight as a string or a number.

**See also:**

***font_manager.FontProperties.get_weight***

**get_ha**()

Alias for *get_horizontalalignment*.

**get_horizontalalignment**()

Return the horizontal alignment as a string. Will be one of 'left', 'center' or 'right'.

**get_math_fontfamily**()

Return the font family name for math text rendered by Matplotlib.

The default value is *rcParams["mathtext.fontset"]* (default: 'dejavusans').

**See also:**

***set_math_fontfamily***

**get_name**()

Alias for *get_fontname*.

**get_parse_math**()

Return whether mathtext parsing is considered for this *Text*.

**get_position**()

Return the (x, y) position of the text.

**get_rotation**()

Return the text angle in degrees between 0 and 360.

**get_rotation_mode**()

Return the text rotation mode.

**get_size**()

Alias for *get_fontsize*.

**get_stretch**()

Return the font stretch as a string or a number.

**See also:**

***font_manager.FontProperties.get_stretch***

**get_style**()

Alias for *get_fontstyle*.

**get_text**()

Return the text string.

**get_transform_rotates_text**()

Return whether rotations of the transform affect the text direction.

**get_unitless_position**()

Return the (x, y) unitless position of the text.

**get_usetex**()

Return whether this *Text* object uses TeX for rendering.

**get_va**()

Alias for *get_verticalalignment*.

**get_variant**()

Alias for *get_fontvariant*.

**get_verticalalignment**()

Return the vertical alignment as a string. Will be one of 'top', 'center', 'bottom', 'baseline' or 'center_baseline'.

**get_weight**()

Alias for *get_fontweight*.

**get_window_extent**(*renderer=None*, *dpi=None*)

Return the *Bbox* bounding the text, in display units.

In addition to being used internally, this is useful for specifying clickable regions in a png file on a web page.

**Parameters**

**renderer**

[Renderer, optional] A renderer is needed to compute the bounding box. If the artist has already been drawn, the renderer is cached; thus, it is only necessary to pass this argument when calling *get_window_extent* before the first draw. In practice, it is usually easier to trigger a draw first, e.g. by calling *draw_without_rendering* or `plt.show()`.

**dpi**

[float, optional] The dpi value for computing the bbox, defaults to `self.figure.dpi` (*not* the renderer dpi); should be set e.g. if to match regions with a figure saved with a custom dpi value.

**get_wrap**()

Return whether the text can be wrapped.

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *antialiased=<UNSET>*, *backgroundcolor=<UNSET>*, *bbox=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *color=<UNSET>*, *fontfamily=<UNSET>*, *fontproperties=<UNSET>*, *fontsize=<UNSET>*, *fontstretch=<UNSET>*, *fontstyle=<UNSET>*, *fontvariant=<UNSET>*, *fontweight=<UNSET>*, *gid=<UNSET>*, *horizontalalignment=<UNSET>*, *in_layout=<UNSET>*, *label=<UNSET>*, *linespacing=<UNSET>*, *math_fontfamily=<UNSET>*, *mouseover=<UNSET>*, *multialignment=<UNSET>*, *parse_math=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *position=<UNSET>*, *rasterized=<UNSET>*, *rotation=<UNSET>*, *rotation_mode=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *text=<UNSET>*, *transform=<UNSET>*, *transform_rotates_text=<UNSET>*, *url=<UNSET>*, *usetex=<UNSET>*, *verticalalignment=<UNSET>*, *visible=<UNSET>*, *wrap=<UNSET>*, *x=<UNSET>*, *y=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* | bool |
| *backgroundcolor* | *color* |
| *bbox* | dict with properties for *patches.FancyBboxPatch* |
| *clip_box* | unknown |
| *clip_on* | unknown |
| *clip_path* | unknown |
| *color* | *color* |
| *figure* | *Figure* |
| *fontfamily* or fontname | {FONTNAME, 'serif', 'sans-serif', 'cursive', 'fantasy', 'monospace'} |
| *fontproperties* | *font_manager.FontProperties* or *str* or *pathlib.Path* |
| *fontsize* | float or {'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-lar |
| *fontstretch* | {a numeric value in range 0-1000, 'ultra-condensed', 'extra-condensed' |
| *fontstyle* | {'normal', 'italic', 'oblique'} |
| *fontvariant* | {'normal', 'small-caps'} |
| *fontweight* | {a numeric value in range 0-1000, 'ultralight', 'light', 'normal', 'regular' |
| *gid* | str |
| *horizontalalignment* | {'left', 'center', 'right'} |
| *in_layout* | bool |
| *label* | object |
| *linespacing* | float (multiple of font size) |
| *math_fontfamily* | str |
| *mouseover* | bool |
| *multialignment* | {'left', 'right', 'center'} |
| *parse_math* | bool |
| *path_effects* | list of *AbstractPathEffect* |

Table 159 – continued

| Property | Description |
|---|---|
| *picker* | None or bool or float or callable |
| *position* | (float, float) |
| *rasterized* | bool |
| *rotation* | float or {'vertical', 'horizontal'} |
| *rotation_mode* | {None, 'default', 'anchor'} |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *text* | object |
| *transform* | *Transform* |
| *transform_rotates_text* | bool |
| *url* | str |
| *usetex* | bool, default: *rcParams["text.usetex"]* (default: `False`) |
| *verticalalignment* | {'baseline', 'bottom', 'center', 'center_baseline', 'top'} |
| *visible* | bool |
| *wrap* | bool |
| *x* | float |
| *y* | float |
| *zorder* | float |

**set_antialiased**(*antialiased*)

Set whether to use antialiased rendering.

**Parameters**

**antialiased**

[bool]

### Notes

Antialiasing will be determined by *rcParams["text.antialiased"]* (default: `True`) and the parameter *antialiased* will have no effect if the text contains math expressions.

**set_backgroundcolor**(*color*)

Set the background color of the text by updating the bbox.

**Parameters**

**color**

[*color*]

**See also:**

*set_bbox*

To change the position of the bounding box

**set_bbox**(*rectprops*)

Draw a bounding box around self.

**Parameters**

**rectprops**

[dict with properties for *patches.FancyBboxPatch*] The default boxstyle is 'square'. The mutation scale of the *patches.FancyBboxPatch* is set to the fontsize.

### Examples

```
t.set_bbox(dict(facecolor='red', alpha=0.5))
```

**set_c**(*color*)

Alias for *set_color*.

**set_clip_box**(*clipbox*)

Set the artist's clip *Bbox*.

**Parameters**

**clipbox**

[*BboxBase* or None] Will typically be created from a *TransformedBbox*. For instance, `TransformedBbox(Bbox([[0, 0], [1, 1]]), ax.transAxes)` is the default clipping for an artist added to an Axes.

**set_clip_on**(*b*)

Set whether the artist uses clipping.

When False, artists will be visible outside the Axes which can lead to unexpected results.

**Parameters**

**b**

[bool]

**set_clip_path**(*path*, *transform=None*)

Set the artist's clip path.

**Parameters**

**path**

[*Patch* or *Path* or *TransformedPath* or None] The clip path. If given a *Path*, *transform* must be provided as well. If *None*, a previously set clip path is removed.

**transform**

[*Transform*, optional] Only used if *path* is a *Path*, in which case the given *Path* is converted to a *TransformedPath* using *transform*.

### Notes

For efficiency, if *path* is a *Rectangle* this method will set the clipping box to the corresponding rectangle and set the clipping path to `None`.

For technical reasons (support of *set*), a tuple (*path*, *transform*) is also accepted as a single positional parameter.

**set_color**(*color*)

Set the foreground color of the text

**Parameters**

**color**

[*color*]

**set_family**(*fontname*)

Alias for *set_fontfamily*.

**set_font**(*fp*)

Alias for *set_fontproperties*.

**set_font_properties**(*fp*)

Alias for *set_fontproperties*.

**set_fontfamily**(*fontname*)

Set the font family. Can be either a single string, or a list of strings in decreasing priority. Each string may be either a real font name or a generic font class name. If the latter, the specific font names will be looked up in the corresponding rcParams.

If a *Text* instance is constructed with `fontfamily=None`, then the font is set to *rcParams["font.family"]* (default: `['sans-serif']`), and the same is done when *set_fontfamily()* is called on an existing *Text* instance.

**Parameters**

**fontname**

[{FONTNAME, 'serif', 'sans-serif', 'cursive', 'fantasy', 'monospace'}]

**See also:**

*font_manager.FontProperties.set_family*

**set_fontname**(*fontname*)

Alias for *set_fontfamily*.

One-way alias only: the getter differs.

**Parameters**

**fontname**

[{FONTNAME, 'serif', 'sans-serif', 'cursive', 'fantasy', 'monospace'}]

**See also:**

*font_manager.FontProperties.set_family*

**set_fontproperties**(*fp*)

Set the font properties that control the text.

**Parameters**

**fp**

[*font_manager.FontProperties* or `str` or `pathlib.Path`] If a `str`, it is interpreted as a fontconfig pattern parsed by *FontProperties*. If a `pathlib.Path`, it is interpreted as the absolute path to a font file.

**set_fontsize**(*fontsize*)

Set the font size.

**Parameters**

**fontsize**

[float or {'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large'}] If a float, the fontsize in points. The string values denote sizes relative to the default font size.

**See also:**

*font_manager.FontProperties.set_size*

**set_fontstretch**(*stretch*)

Set the font stretch (horizontal condensation or expansion).

**Parameters**

**stretch**

[{a numeric value in range 0-1000, 'ultra-condensed', 'extra-condensed', 'condensed', 'semi-condensed', 'normal', 'semi-expanded', 'expanded', 'extra-expanded', 'ultra-expanded'}]

**See also:**

*font_manager.FontProperties.set_stretch*

**set_fontstyle**(*fontstyle*)

Set the font style.

**Parameters**

**fontstyle**

[{'normal', 'italic', 'oblique'}]

**See also:**

*font_manager.FontProperties.set_style*

**set_fontvariant**(*variant*)

Set the font variant.

**Parameters**

**variant**

[{'normal', 'small-caps'}]

**See also:**

*font_manager.FontProperties.set_variant*

**set_fontweight**(*weight*)

Set the font weight.

**Parameters**

**weight**

[{a numeric value in range 0-1000, 'ultralight', 'light', 'normal', 'regular', 'book', 'medium', 'roman', 'semibold', 'demibold', 'demi', 'bold', 'heavy', 'extra bold', 'black'}]

**See also:**

*font_manager.FontProperties.set_weight*

**set_ha**(*align*)

Alias for *set_horizontalalignment*.

**set_horizontalalignment**(*align*)

Set the horizontal alignment relative to the anchor point.

See also *Text alignment*.

**Parameters**

**align**

[{'left', 'center', 'right'}]

**set_linespacing**(*spacing*)

Set the line spacing as a multiple of the font size.

The default line spacing is 1.2.

**Parameters**

**spacing**

[float (multiple of font size)]

**set_ma**(*align*)

Alias for *set_multialignment*.

**set_math_fontfamily**(*fontfamily*)

Set the font family for math text rendered by Matplotlib.

This does only affect Matplotlib's own math renderer. It has no effect when rendering with TeX (`usetex=True`).

**Parameters**

**fontfamily**

[str] The name of the font family.

Available font families are defined in the *default matplotlibrc file*.

**See also:**

***get_math_fontfamily***

**set_multialignment**(*align*)

Set the text alignment for multiline texts.

The layout of the bounding box of all the lines is determined by the horizontalalignment and verticalalignment properties. This property controls the alignment of the text lines within that box.

**Parameters**

**align**

[{'left', 'right', 'center'}]

**set_name**(*fontname*)

Alias for *set_fontname*.

**set_parse_math**(*parse_math*)

Override switch to disable any mathtext parsing for this *Text*.

**Parameters**

**parse_math**

[bool] If False, this *Text* will never use mathtext. If True, mathtext will be used if there is an even number of unescaped dollar signs.

**set_position**(*xy*)

Set the (*x*, *y*) position of the text.

**Parameters**

**xy**

[(float, float)]

**set_rotation**(*s*)

Set the rotation of the text.

**Parameters**

**s**

[float or {'vertical', 'horizontal'}] The rotation angle in degrees in mathematically positive direction (counterclockwise). 'horizontal' equals 0, 'vertical' equals 90.

**set_rotation_mode**(*m*)

Set text rotation mode.

**Parameters**

**m**

[{None, 'default', 'anchor'}] If `"default"`, the text will be first rotated, then aligned according to their horizontal and vertical alignments. If `"anchor"`, then alignment occurs before rotation. Passing `None` will set the rotation mode to `"default"`.

**set_size**(*fontsize*)

Alias for *set_fontsize*.

**set_stretch**(*stretch*)

Alias for *set_fontstretch*.

**set_style**(*fontstyle*)

Alias for *set_fontstyle*.

**set_text**(*s*)

Set the text string *s*.

It may contain newlines (`\n`) or math in LaTeX syntax.

**Parameters**

**s**

[object] Any object gets converted to its `str` representation, except for `None` which is converted to an empty string.

**set_transform_rotates_text**(*t*)

Whether rotations of the transform affect the text direction.

**Parameters**

**t**

[bool]

**set_usetex**(*usetex*)

**Parameters**

**usetex**

[bool or None] Whether to render using TeX, `None` means to use *rcParams["text.usetex"]* (default: `False`).

**set_va**(*align*)

Alias for *set_verticalalignment*.

**set_variant**(*variant*)

Alias for *set_fontvariant*.

**set_verticalalignment**(*align*)

Set the vertical alignment relative to the anchor point.

See also *Text alignment*.

**Parameters**

**align**

[{'baseline', 'bottom', 'center', 'center_baseline', 'top'}]

**set_weight**(*weight*)

Alias for *set_fontweight*.

**set_wrap**(*wrap*)

Set whether the text can be wrapped.

**Parameters**

**wrap**

[bool]

### Notes

Wrapping does not work together with `savefig(..., bbox_inches='tight')` (which is also used internally by `%matplotlib inline` in IPython/Jupyter). The 'tight' setting rescales the canvas to accommodate all content and happens before wrapping.

**set_x**(*x*)

Set the *x* position of the text.

**Parameters**

**x**

[float]

**set_y**(*y*)

Set the *y* position of the text.

**Parameters**

**y**

[float]

**update**(*kwargs*)

Update this artist's properties from the dict *props*.

**Parameters**

**props**

[dict]

**update_bbox_position_size**(*renderer*)

Update the location and the size of the bbox.

This method should be used when the position and size of the bbox needs to be updated before actually drawing the bbox.

**update_from**(*other*)

Copy properties from *other* to *self*.

**zorder = 3**

**class** matplotlib.text.**Annotation**(*text*, *xy*, *xytext=None*, *xycoords='data'*, *textcoords=None*, *arrowprops=None*, *annotation_clip=None*, ***kwargs*)

Bases: *Text*, _AnnotationBase

An *Annotation* is a *Text* that can refer to a specific position *xy*. Optionally an arrow pointing from the text to *xy* can be drawn.

**Attributes**

**xy**

The annotated position.

**xycoords**

The coordinate system for *xy*.

**arrow_patch**

A *FancyArrowPatch* to point from *xytext* to *xy*.

Annotate the point *xy* with text *text*.

In the simplest form, the text is placed at *xy*.

Optionally, the text can be displayed in another position *xytext*. An arrow pointing from the text to the annotated point *xy* can then be added by defining *arrowprops*.

**Parameters**

**text**

[str] The text of the annotation.

**xy**

[(float, float)] The point *(x, y)* to annotate. The coordinate system is determined by *xycoords*.

**xytext**

[(float, float), default: *xy*] The position *(x, y)* to place the text at. The coordinate system is determined by *textcoords*.

**xycoords**

[single or two-tuple of str or *Artist* or *Transform* or callable, default: 'data'] The coordinate system that *xy* is given in. The following types of values are supported:

- One of the following strings:

| Value | Description |
|---|---|
| 'figure points' | Points from the lower left of the figure |
| 'figure pixels' | Pixels from the lower left of the figure |
| 'figure fraction' | Fraction of figure from lower left |
| 'subfigure points' | Points from the lower left of the subfigure |
| 'subfigure pixels' | Pixels from the lower left of the subfigure |
| 'subfigure fraction' | Fraction of subfigure from lower left |
| 'axes points' | Points from lower left corner of the Axes |
| 'axes pixels' | Pixels from lower left corner of the Axes |
| 'axes fraction' | Fraction of Axes from lower left |
| 'data' | Use the coordinate system of the object being annotated (default) |
| 'polar' | *(theta, r)* if not native 'data' coordinates |

  Note that 'subfigure pixels' and 'figure pixels' are the same for the parent figure, so users who want code that is usable in a subfigure can use 'subfigure pixels'.

- An *Artist*: *xy* is interpreted as a fraction of the artist's *Bbox*. E.g. *(0, 0)* would be the lower left corner of the bounding box and *(0.5, 1)* would be the center top of the bounding box.

- A *Transform* to transform *xy* to screen coordinates.

- A function with one of the following signatures:

```
def transform(renderer) -> Bbox
def transform(renderer) -> Transform
```

  where *renderer* is a *RendererBase* subclass.

  The result of the function is interpreted like the *Artist* and *Transform* cases above.

- A tuple *(xcoords, ycoords)* specifying separate coordinate systems for *x* and *y*. *xcoords* and *ycoords* must each be of one of the above described types.

See *Advanced annotation* for more details.

**textcoords**

[single or two-tuple of str or *Artist* or *Transform* or callable, default: value of *xycoords*] The coordinate system that *xytext* is given in.

All *xycoords* values are valid as well as the following strings:

| Value | Description |
|---|---|
| 'offset points' | Offset, in points, from the *xy* value |
| 'offset pixels' | Offset, in pixels, from the *xy* value |
| 'offset fontsize' | Offset, relative to fontsize, from the *xy* value |

**arrowprops**

[dict, optional] The properties used to draw a *FancyArrowPatch* arrow between the positions *xy* and *xytext*. Defaults to None, i.e. no arrow is drawn.

For historical reasons there are two different ways to specify arrows, "simple" and "fancy":

**Simple arrow:**

If *arrowprops* does not contain the key 'arrowstyle' the allowed keys are:

| Key | Description |
|---|---|
| width | The width of the arrow in points |
| headwidth | The width of the base of the arrow head in points |
| headlength | The length of the arrow head in points |
| shrink | Fraction of total length to shrink from both ends |
| ? | Any *FancyArrowPatch* property |

The arrow is attached to the edge of the text box, the exact position (corners or centers) depending on where it's pointing to.

**Fancy arrow:**

This is used if 'arrowstyle' is provided in the *arrowprops*.

Valid keys are the following *FancyArrowPatch* parameters:

| Key | Description |
|---|---|
| arrowstyle | The arrow style |
| connectionstyle | The connection style |
| relpos | See below; default is (0.5, 0.5) |
| patchA | Default is bounding box of the text |
| patchB | Default is None |
| shrinkA | In points. Default is 2 points |
| shrinkB | In points. Default is 2 points |
| mutation_scale | Default is text size (in points) |
| mutation_aspect | Default is 1 |
| ? | Any `FancyArrowPatch` property |

The exact starting point position of the arrow is defined by *relpos*. It's a tuple of relative coordinates of the text box, where (0, 0) is the lower left corner and (1, 1) is the upper right corner. Values <0 and >1 are supported and specify points outside the text box. By default (0.5, 0.5), so the starting point is centered in the text box.

**annotation_clip**

[bool or None, default: None] Whether to clip (i.e. not draw) the annotation when the annotation point *xy* is outside the Axes area.

- If *True*, the annotation will be clipped when *xy* is outside the Axes.
- If *False*, the annotation will always be drawn.
- If *None*, the annotation will be clipped when *xy* is outside the Axes and *xycoords* is 'data'.

****kwargs**

Additional kwargs are passed to `Text`.

**Returns**

***Annotation***

**See also:**

*Annotations*

**property anncoords**

The coordinate system to use for `Annotation.xyann`.

**contains**(*mouseevent*)

Return whether the mouse event occurred inside the axis-aligned bounding-box of the text.

**draw**(*renderer*)

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

**Parameters**

**renderer**

[*RendererBase* subclass.]

### Notes

This method is overridden in the Artist subclasses.

**get_anncoords**()

Return the coordinate system to use for *Annotation.xyann*.

See also *xycoords* in *Annotation*.

**get_tightbbox**(*renderer=None*)

Like *Artist.get_window_extent*, but includes any clipping.

**Parameters**

**renderer**

[*RendererBase* subclass, optional] renderer that will be used to draw the figures (i.e. `fig.canvas.get_renderer()`)

**Returns**

***Bbox* or None**

The enclosing bounding box (in figure pixel coordinates). Returns None if clipping results in no intersection.

**get_window_extent**(*renderer=None*)

Return the *Bbox* bounding the text, in display units.

In addition to being used internally, this is useful for specifying clickable regions in a png file on a web page.

**Parameters**

**renderer**

[Renderer, optional] A renderer is needed to compute the bounding box. If the artist has already been drawn, the renderer is cached; thus, it is only necessary to pass this argument when calling *get_window_extent* before the first draw. In practice, it is usually easier to trigger a draw first, e.g. by calling *draw_without_rendering* or `plt.show()`.

**dpi**

[float, optional] The dpi value for computing the bbox, defaults to `self.figure.dpi` (*not* the renderer dpi); should be set e.g. if to match regions with a figure saved with a custom dpi value.

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *anncoords=<UNSET>*, *annotation_clip=<UNSET>*, *antialiased=<UNSET>*, *backgroundcolor=<UNSET>*, *bbox=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *color=<UNSET>*, *fontfamily=<UNSET>*, *fontproperties=<UNSET>*, *fontsize=<UNSET>*, *fontstretch=<UNSET>*, *fontstyle=<UNSET>*, *fontvariant=<UNSET>*, *fontweight=<UNSET>*, *gid=<UNSET>*, *horizontalalignment=<UNSET>*, *in_layout=<UNSET>*, *label=<UNSET>*, *linespacing=<UNSET>*, *math_fontfamily=<UNSET>*, *mouseover=<UNSET>*, *multialignment=<UNSET>*, *parse_math=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *position=<UNSET>*, *rasterized=<UNSET>*, *rotation=<UNSET>*, *rotation_mode=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *text=<UNSET>*, *transform=<UNSET>*, *transform_rotates_text=<UNSET>*, *url=<UNSET>*, *usetex=<UNSET>*, *verticalalignment=<UNSET>*, *visible=<UNSET>*, *wrap=<UNSET>*, *x=<UNSET>*, *y=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a |
| *alpha* | scalar or None |
| *animated* | bool |
| *anncoords* | unknown |
| annotation_clip | bool or None |
| *antialiased* | bool |
| *backgroundcolor* | *color* |
| *bbox* | dict with properties for *patches.FancyBboxPatch* |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* or c | *color* |
| *figure* | unknown |
| *fontfamily* or family or fontname | {FONTNAME, 'serif', 'sans-serif', 'cursive', 'fantasy', 'mo |
| *fontproperties* or font or font_properties | *font_manager.FontProperties* or *str* or *pat* |
| *fontsize* or size | float or {'xx-small', 'x-small', 'small', 'medium', 'large', 'x |
| *fontstretch* or stretch | {a numeric value in range 0-1000, 'ultra-condensed', 'ext |
| *fontstyle* or style | {'normal', 'italic', 'oblique'} |
| *fontvariant* or variant | {'normal', 'small-caps'} |
| *fontweight* or weight | {a numeric value in range 0-1000, 'ultralight', 'light', 'nor |
| *gid* | str |

Table 160 – co

| Property | Description |
|---|---|
| *horizontalalignment* or ha | {'left', 'center', 'right'} |
| *in_layout* | bool |
| *label* | object |
| *linespacing* | float (multiple of font size) |
| *math_fontfamily* | str |
| *mouseover* | bool |
| *multialignment* or ma | {'left', 'right', 'center'} |
| *parse_math* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *position* | (float, float) |
| *rasterized* | bool |
| *rotation* | float or {'vertical', 'horizontal'} |
| *rotation_mode* | {None, 'default', 'anchor'} |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *text* | object |
| *transform* | *Transform* |
| *transform_rotates_text* | bool |
| *url* | str |
| *usetex* | bool, default: *rcParams["text.usetex"]* (defaul |
| *verticalalignment* or va | {'baseline', 'bottom', 'center', 'center_baseline', 'top'} |
| *visible* | bool |
| *wrap* | bool |
| *x* | float |
| *y* | float |
| *zorder* | float |

**set_anncoords**(*coords*)

Set the coordinate system to use for *Annotation.xyann*.

See also *xycoords* in *Annotation*.

**set_figure**(*fig*)

Set the *Figure* instance the artist belongs to.

**Parameters**

**fig**

[*Figure*]

**update_positions**(*renderer*)

Update the pixel positions of the annotation text and the arrow patch.

**property xyann**

The text position.

See also *xytext* in *Annotation*.

**property xycoords**

**class** matplotlib.text.**OffsetFrom**(*artist*, *ref_coord*, *unit='points'*)

Bases: object

Callable helper class for working with *Annotation*.

**Parameters**

**artist**

[*Artist* or *BboxBase* or *Transform*] The object to compute the offset from.

**ref_coord**

[(float, float)] If *artist* is an *Artist* or *BboxBase*, this values is the location to of the offset origin in fractions of the *artist* bounding box.

If *artist* is a transform, the offset origin is the transform applied to this value.

**unit**

[{'points, 'pixels'}, default: 'points'] The screen units to use (pixels or points) for the offset input.

**get_unit**()

Return the unit for input to the transform used by __call__.

**set_unit**(*unit*)

Set the unit for input to the transform used by __call__.

**Parameters**

**unit**

[{'points', 'pixels'}]

**class** matplotlib.text.**TextPath**(*xy*, *s*, *size=None*, *prop=None*, *_interpolation_steps=1*, *usetex=False*)

Bases: *Path*

Create a path from the text.

Create a path from the text. Note that it simply is a path, not an artist. You need to use the *PathPatch* (or other artists) to draw this path onto the canvas.

**Parameters**

**xy**

[tuple or array of two float values] Position of the text. For no offset, use `xy=(0, 0)`.

**s**

[str] The text to convert to a path.

**size**

[float, optional] Font size in points. Defaults to the size specified via the font properties *prop*.

**prop**

[*FontProperties*, optional] Font property. If not provided, will use a default *FontProperties* with parameters from the *rcParams*.

**_interpolation_steps**

[int, optional] (Currently ignored)

**usetex**

[bool, default: False] Whether to use tex rendering.

## Examples

The following creates a path from the string "ABC" with Helvetica font face; and another path from the latex fraction 1/2:

```
from matplotlib.text import TextPath
from matplotlib.font_manager import FontProperties

fp = FontProperties(family="Helvetica", style="italic")
path1 = TextPath((12, 12), "ABC", size=12, prop=fp)
path2 = TextPath((0, 0), r"$\frac{1}{2}$", size=12, usetex=True)
```

Also see *Using a text as a Path*.

**property codes**

Return the codes

**get_size**()

Get the text size.

**set_size**(*size*)

Set the text size.

**property vertices**

Return the cached path after updating it if necessary.

**class** matplotlib.text.**TextToPath**

Bases: object

A class that converts strings to paths.

**DPI = 72**

**FONT_SCALE = 100.0**

**get_glyphs_mathtext**(*prop*, *s*, *glyph_map=None*, *return_new_glyphs_only=False*)

Parse mathtext string *s* and convert it to a (vertices, codes) pair.

**get_glyphs_tex**(*prop*, *s*, *glyph_map=None*, *return_new_glyphs_only=False*)

Convert the string *s* to vertices and codes using usetex mode.

**get_glyphs_with_font**(*font*, *s*, *glyph_map=None*, *return_new_glyphs_only=False*)

Convert string *s* to vertices and codes using the provided ttf font.

**get_text_path**(*prop*, *s*, *ismath=False*)

Convert text *s* to path (a tuple of vertices and codes for matplotlib.path.Path).

**Parameters**

**prop**

[*FontProperties*] The font properties for the text.

**s**

[str] The text to be converted.

**ismath**

[{False, True, "TeX"}] If True, use mathtext parser. If "TeX", use tex for rendering.

**Returns**

**verts**

[list] A list of arrays containing the (x, y) coordinates of the vertices.

**codes**

[list] A list of path codes.

### Examples

Create a list of vertices and codes from a text, and create a *Path* from those:

```
from matplotlib.path import Path
from matplotlib.text import TextToPath
from matplotlib.font_manager import FontProperties

fp = FontProperties(family="Comic Neue", style="italic")
verts, codes = TextToPath().get_text_path(fp, "ABC")
path = Path(verts, codes, closed=False)
```

Also see *TextPath* for a more direct way to create a path from a text.

**get_text_width_height_descent**(*s*, *prop*, *ismath*)

## 7.2.54 matplotlib.texmanager

Support for embedded TeX expressions in Matplotlib.

Requirements:

- LaTeX.
- *Agg backends: dvipng>=1.6.
- PS backend: PSfrag, dvips, and Ghostscript>=9.0.
- PDF and SVG backends: if LuaTeX is present, it will be used to speed up some post-processing steps, but note that it is not used to parse the TeX string itself (only LaTeX is supported).

To enable TeX rendering of all text in your Matplotlib figure, set *rcParams["text.usetex"]* (default: `False`) to True.

TeX and dvipng/dvips processing results are cached in ~/.matplotlib/tex.cache for reuse between sessions.

*TexManager.get_rgba* can also be used to directly obtain raster output as RGBA NumPy arrays.

**class** matplotlib.texmanager.**TexManager**

Bases: `object`

Convert strings to dvi files using TeX, caching the results to a directory.

The cache directory is called `tex.cache` and is located in the directory returned by *get_cachedir*.

Repeated calls to this constructor always return the same instance.

**classmethod get_basefile**(*tex*, *fontsize*, *dpi=None*)

Return a filename based on a hash of the string, fontsize, and dpi.

**classmethod get_custom_preamble**()

Return a string containing user additions to the tex preamble.

**classmethod get_font_preamble**()

Return a string containing font configuration for the tex preamble.

**classmethod get_grey**(*tex*, *fontsize=None*, *dpi=None*)

Return the alpha channel.

**classmethod get_rgba**(*tex*, *fontsize=None*, *dpi=None*, *rgb=(0, 0, 0)*)

Return latex's rendering of the tex string as an RGBA array.

#### Examples

```
>>> texmanager = TexManager()
>>> s = r"\TeX\ is $\displaystyle\sum_n\frac{-e^{i\pi}}{2^n}$!"
>>> Z = texmanager.get_rgba(s, fontsize=12, dpi=80, rgb=(1, 0, 0))
```

**classmethod get_text_width_height_descent**(*tex*, *fontsize*, *renderer=None*)

Return width, height and descent of the text.

**classmethod make_dvi**(*tex*, *fontsize*)

Generate a dvi file containing latex's layout of tex string.

Return the file name.

**classmethod make_png**(*tex*, *fontsize*, *dpi*)

Generate a png file containing latex's rendering of tex string.

Return the file name.

**classmethod make_tex**(*tex*, *fontsize*)

Generate a tex file to render the tex string at a specific font size.

Return the file name.

**property texcache**

[*Deprecated*]

#### Notes

Deprecated since version 3.8:

### 7.2.55 matplotlib.ticker

#### Tick locating and formatting

This module contains classes for configuring tick locating and formatting. Generic tick locators and formatters are provided, as well as domain specific custom ones.

Although the locators know nothing about major or minor ticks, they are used by the Axis class to support major and minor tick locating and formatting.

### Tick locating

The Locator class is the base class for all tick locators. The locators handle autoscaling of the view limits based on the data limits, and the choosing of tick locations. A useful semi-automatic tick locator is *MultipleLocator*. It is initialized with a base, e.g., 10, and it picks axis limits and ticks that are multiples of that base.

The Locator subclasses defined here are:

| | |
|---|---|
| *AutoLocator* | *MaxNLocator* with simple defaults. This is the default tick locator for most plotting. |
| *MaxNLocator* | Finds up to a max number of intervals with ticks at nice locations. |
| *LinearLocator* | Space ticks evenly from min to max. |
| *LogLocator* | Space ticks logarithmically from min to max. |
| *MultipleLocator* | Ticks and range are a multiple of base; either integer or float. |
| *FixedLocator* | Tick locations are fixed. |
| *IndexLocator* | Locator for index plots (e.g., where `x = range(len(y))`). |
| *NullLocator* | No ticks. |
| *SymmetricalLogLocator* | Locator for use with the symlog norm; works like *LogLocator* for the part outside of the threshold and adds 0 if inside the limits. |
| *AsinhLocator* | Locator for use with the asinh norm, attempting to space ticks approximately uniformly. |
| *LogitLocator* | Locator for logit scaling. |
| *AutoMinorLocator* | Locator for minor ticks when the axis is linear and the major ticks are uniformly spaced. Subdivides the major tick interval into a specified number of minor intervals, defaulting to 4 or 5 depending on the major interval. |

There are a number of locators specialized for date locations - see the *dates* module.

You can define your own locator by deriving from Locator. You must override the `__call__` method, which returns a sequence of locations, and you will probably want to override the autoscale method to set the view limits from the data limits.

If you want to override the default locator, use one of the above or a custom locator and pass it to the x- or y-axis instance. The relevant methods are:

```
ax.xaxis.set_major_locator(xmajor_locator)
ax.xaxis.set_minor_locator(xminor_locator)
ax.yaxis.set_major_locator(ymajor_locator)
ax.yaxis.set_minor_locator(yminor_locator)
```

The default minor locator is *NullLocator*, i.e., no minor ticks on by default.

---

**Note:** *Locator* instances should not be used with more than one *Axis* or *Axes*. So instead of:

```
locator = MultipleLocator(5)
ax.xaxis.set_major_locator(locator)
ax2.xaxis.set_major_locator(locator)
```

do the following instead:

```
ax.xaxis.set_major_locator(MultipleLocator(5))
ax2.xaxis.set_major_locator(MultipleLocator(5))
```

---

### Tick formatting

Tick formatting is controlled by classes derived from Formatter. The formatter operates on a single tick value and returns a string to the axis.

| | |
|---|---|
| *NullFormatter* | No labels on the ticks. |
| *FixedFormatter* | Set the strings manually for the labels. |
| *FuncFormatter* | User defined function sets the labels. |
| *StrMethodFormatter* | Use string `format` method. |
| *FormatStrFormatter* | Use an old-style sprintf format string. |
| *ScalarFormatter* | Default formatter for scalars: autopick the format string. |
| *LogFormatter* | Formatter for log axes. |
| *LogFormatterExponent* | Format values for log axis using `exponent = log_base(value)`. |
| *LogFormatterMathtext* | Format values for log axis using `exponent = log_base(value)` using Math text. |
| *LogFormatterSciNotation* | Format values for log axis using scientific notation. |
| *LogitFormatter* | Probability formatter. |
| *EngFormatter* | Format labels in engineering notation. |
| *PercentFormatter* | Format labels as a percentage. |

You can derive your own formatter from the Formatter base class by simply overriding the `__call__` method. The formatter class has access to the axis view and data limits.

To control the major and minor tick label formats, use one of the following methods:

```
ax.xaxis.set_major_formatter(xmajor_formatter)
ax.xaxis.set_minor_formatter(xminor_formatter)
ax.yaxis.set_major_formatter(ymajor_formatter)
ax.yaxis.set_minor_formatter(yminor_formatter)
```

In addition to a *Formatter* instance, *set_major_formatter* and *set_minor_formatter* also accept a `str` or function. `str` input will be internally replaced with an autogenerated *StrMethodFormatter* with the input `str`. For function input, a *FuncFormatter* with the input function will be generated and used.

See *Major and minor ticks* for an example of setting major and minor ticks. See the *matplotlib.dates* module for more information and examples of using date locators and formatters.

**class** matplotlib.ticker.**AsinhLocator**(*linear_width*, *numticks=11*, *symthresh=0.2*, *base=10*, *subs=None*)

Bases: *Locator*

Place ticks spaced evenly on an inverse-sinh scale.

Generally used with the *AsinhScale* class.

---

**Note:** This API is provisional and may be revised in the future based on early user feedback.

---

**Parameters**

**linear_width**

[float] The scale parameter defining the extent of the quasi-linear region.

**numticks**

[int, default: 11] The approximate number of major ticks that will fit along the entire axis

**symthresh**

[float, default: 0.2] The fractional threshold beneath which data which covers a range that is approximately symmetric about zero will have ticks that are exactly symmetric.

**base**

[int, default: 10] The number base used for rounding tick locations on a logarithmic scale. If this is less than one, then rounding is to the nearest integer multiple of powers of ten.

**subs**

[tuple, default: None] Multiples of the number base, typically used for the minor ticks, e.g. (2, 5) when base=10.

**set_params**(*numticks=None*, *symthresh=None*, *base=None*, *subs=None*)

Set parameters within this locator.

**tick_values**(*vmin*, *vmax*)

Return the values of the located ticks given **vmin** and **vmax**.

---

**Note:** To get tick locations with the vmin and vmax values defined automatically for the associated `axis` simply call the Locator instance:

```
>>> print(type(loc))
<type 'Locator'>
>>> print(loc())
[1, 2, 3, 4]
```

---

**class** matplotlib.ticker.**AutoLocator**

Bases: *MaxNLocator*

Place evenly spaced ticks, with the step size and maximum number of ticks chosen automatically.

This is a subclass of *MaxNLocator*, with parameters *nbins = 'auto'* and *steps = [1, 2, 2.5, 5, 10]*.

To know the values of the non-public parameters, please have a look to the defaults of *MaxNLocator*.

**class** matplotlib.ticker.**AutoMinorLocator**(*n=None*)

Bases: *Locator*

Place evenly spaced minor ticks, with the step size and maximum number of ticks chosen automatically.

The Axis scale must be linear with evenly spaced major ticks .

*n* is the number of subdivisions of the interval between major ticks; e.g., n=2 will place a single minor tick midway between major ticks.

If *n* is omitted or None, the value stored in rcParams will be used. In case *n* is set to 'auto', it will be set to 4 or 5. If the distance between the major ticks equals 1, 2.5, 5 or 10 it can be perfectly divided in 5 equidistant sub-intervals with a length multiple of 0.05. Otherwise it is divided in 4 sub-intervals.

**tick_values**(*vmin*, *vmax*)

Return the values of the located ticks given **vmin** and **vmax**.

---

**Note:** To get tick locations with the vmin and vmax values defined automatically for the associated `axis` simply call the Locator instance:

```
>>> print(type(loc))
<type 'Locator'>
>>> print(loc())
[1, 2, 3, 4]
```

---

**class** matplotlib.ticker.**EngFormatter**(*unit=''*, *places=None*, *sep=' '*, ***, *usetex=None*, *useMathText=None*)

Bases: *Formatter*

Format axis values using engineering prefixes to represent powers of 1000, plus a specified unit, e.g., 10 MHz instead of 1e7.

**Parameters**

**unit**

[str, default: ""] Unit symbol to use, suitable for use with single-letter representations of powers of 1000. For example, 'Hz' or 'm'.

**places**

[int, default: None] Precision with which to display the number, specified in digits after the decimal point (there will be between one and three digits before the decimal point). If it is None, the formatting falls back to the floating point format '%g', which displays up to 6 *significant* digits, i.e. the equivalent value for *places* varies between 0 and 5 (inclusive).

**sep**

[str, default: " "] Separator used between the value and the prefix/unit. For example, one get '3.14 mV' if `sep` is " " (default) and '3.14mV' if `sep` is "". Besides the default behavior, some other useful options may be:

- `sep=""` to append directly the prefix/unit to the value;
- `sep="\N{THIN SPACE}"` (`U+2009`);
- `sep="\N{NARROW NO-BREAK SPACE}"` (`U+202F`);
- `sep="\N{NO-BREAK SPACE}"` (`U+00A0`).

**usetex**

[bool, default: *rcParams["text.usetex"]* (default: `False`)] To enable/disable the use of TeX's math mode for rendering the numbers in the formatter.

**useMathText**

[bool, default: *rcParams["axes.formatter.use_mathtext"]* (default: `False`)] To enable/disable the use mathtext for rendering the numbers in the formatter.

**ENG_PREFIXES = {-30: 'q', -27: 'r', -24: 'y', -21: 'z', -18: 'a', -15: 'f', -12: 'p', -9: 'n', -6: 'µ', -3: 'm', 0: '', 3: 'k', 6: 'M', 9: 'G', 12: 'T', 15: 'P', 18: 'E', 21: 'Z', 24: 'Y', 27: 'R', 30: 'Q'}**

**format_eng**(*num*)

Format a number in engineering notation, appending a letter representing the power of 1000 of the original number. Some examples:

```
>>> format_eng(0)        # for self.places = 0
'0'
```

```
>>> format_eng(1000000)  # for self.places = 1
'1.0 M'
```

```
>>> format_eng(-1e-6)   # for self.places = 2
'-1.00 µ'
```

**get_useMathText**()

**get_usetex**()

**set_useMathText**(*val*)

**set_usetex**(*val*)

**property useMathText**

**property usetex**

**class** matplotlib.ticker.**FixedFormatter**(*seq*)

Bases: *Formatter*

Return fixed strings for tick labels based only on position, not value.

---

**Note:** *FixedFormatter* should only be used together with *FixedLocator*. Otherwise, the labels may end up in unexpected positions.

---

Set the sequence *seq* of strings that will be used for labels.

**get_offset**()

**set_offset_string**(*ofs*)

**class** matplotlib.ticker.**FixedLocator**(*locs*, *nbins=None*)

Bases: *Locator*

Place ticks at a set of fixed values.

If *nbins* is None ticks are placed at all values. Otherwise, the *locs* array of possible positions will be subsampled to keep the number of ticks <= *nbins* * +1. The subsampling will be done to include the smallest absolute value; for example, if zero is included in the array of possibilities, then it of the chosen ticks.

**set_params**(*nbins=None*)

Set parameters within this locator.

**tick_values**(*vmin*, *vmax*)

Return the locations of the ticks.

**Note:** Because the values are fixed, vmin and vmax are not used in this method.

**class** matplotlib.ticker.**FormatStrFormatter**(*fmt*)

Bases: *Formatter*

Use an old-style ('%' operator) format string to format the tick.

The format string should have a single variable format (%) in it. It will be applied to the value (not the position) of the tick.

Negative numeric values (e.g., -1) will use a dash, not a Unicode minus; use mathtext to get a Unicode minus by wrapping the format specifier with $ (e.g. "$%g$").

**class** matplotlib.ticker.**Formatter**

Bases: *TickHelper*

Create a string based on a tick value and location.

**static fix_minus**(*s*)

Some classes may want to replace a hyphen for minus with the proper Unicode symbol (U+2212) for typographical correctness. This is a helper method to perform such a replacement when it is enabled via *rcParams["axes.unicode_minus"]* (default: True).

**format_data**(*value*)

Return the full string representation of the value with the position unspecified.

**format_data_short**(*value*)

Return a short string version of the tick value.

Defaults to the position-independent long value.

**format_ticks**(*values*)

Return the tick labels for all the ticks at once.

**get_offset**()

**locs = []**

**set_locs**(*locs*)

Set the locations of the ticks.

This method is called before computing the tick labels because some formatters need to know all tick locations to do so.

**class** matplotlib.ticker.**FuncFormatter**(*func*)

Bases: *Formatter*

Use a user-defined function for formatting.

The function should take in two inputs (a tick value `x` and a position `pos`), and return a string containing the corresponding tick label.

**get_offset**()

**set_offset_string**(*ofs*)

**class** matplotlib.ticker.**IndexLocator**(*base*, *offset*)

Bases: *Locator*

Place ticks at every nth point plotted.

IndexLocator assumes index plotting; i.e., that the ticks are placed at integer values in the range between 0 and len(data) inclusive.

Place ticks every *base* data point, starting at *offset*.

**set_params**(*base=None*, *offset=None*)

Set parameters within this locator

**tick_values**(*vmin*, *vmax*)

Return the values of the located ticks given **vmin** and **vmax**.

---

**Note:** To get tick locations with the vmin and vmax values defined automatically for the associated `axis` simply call the Locator instance:

```
>>> print(type(loc))
<type 'Locator'>
>>> print(loc())
[1, 2, 3, 4]
```

---

**class** matplotlib.ticker.**LinearLocator**(*numticks=None*, *presets=None*)

Bases: *Locator*

Place ticks at evenly spaced values.

The first time this function is called it will try to set the number of ticks to make a nice tick partitioning. Thereafter, the number of ticks will be fixed so that interactive navigation will be nice

**Parameters**

**numticks**

[int or None, default None] Number of ticks. If None, *numticks* = 11.

**presets**

[dict or None, default: None] Dictionary mapping `(vmin, vmax)` to an array of locations. Overrides *numticks* if there is an entry for the current `(vmin, vmax)`.

**property numticks**

**set_params**(*numticks=None*, *presets=None*)

Set parameters within this locator.

**tick_values**(*vmin*, *vmax*)

Return the values of the located ticks given **vmin** and **vmax**.

---

**Note:** To get tick locations with the vmin and vmax values defined automatically for the associated `axis` simply call the Locator instance:

```
>>> print(type(loc))
<type 'Locator'>
>>> print(loc())
[1, 2, 3, 4]
```

---

**view_limits**(*vmin*, *vmax*)

Try to choose the view limits intelligently.

**class** matplotlib.ticker.**Locator**

Bases: *TickHelper*

Determine tick locations.

Note that the same locator should not be used across multiple *Axis* because the locator stores references to the Axis data and view limits.

**MAXTICKS = 1000**

**nonsingular**(*v0*, *v1*)

Adjust a range as needed to avoid singularities.

This method gets called during autoscaling, with `(v0, v1)` set to the data limits on the Axes if the Axes contains any data, or `(-inf, +inf)` if not.

- If `v0 == v1` (possibly up to some floating point slop), this method returns an expanded interval around this value.
- If `(v0, v1) == (-inf, +inf)`, this method returns appropriate default view limits.
- Otherwise, `(v0, v1)` is returned without modification.

**raise_if_exceeds**(*locs*)

Log at WARNING level if *locs* is longer than *Locator.MAXTICKS*.

This is intended to be called immediately before returning *locs* from `__call__` to inform users in case their Locator returns a huge number of ticks, causing Matplotlib to run out of memory.

The "strange" name of this method dates back to when it would raise an exception instead of emitting a log.

**set_params**(***kwargs*)

Do nothing, and raise a warning. Any locator class not supporting the set_params() function will call this.

**tick_values**(*vmin*, *vmax*)

Return the values of the located ticks given **vmin** and **vmax**.

---

**Note:** To get tick locations with the vmin and vmax values defined automatically for the associated `axis` simply call the Locator instance:

```
>>> print(type(loc))
<type 'Locator'>
>>> print(loc())
[1, 2, 3, 4]
```

---

**view_limits**(*vmin*, *vmax*)

Select a scale for the range from vmin to vmax.

Subclasses should override this method to change locator behaviour.

**class** matplotlib.ticker.**LogFormatter**(*base=10.0*, *labelOnlyBase=False*, *minor_thresholds=None*, *linthresh=None*)

Bases: *Formatter*

Base class for formatting ticks on a log or symlog scale.

It may be instantiated directly, or subclassed.

**Parameters**

**base**

[float, default: 10.] Base of the logarithm used in all calculations.

**labelOnlyBase**

[bool, default: False] If True, label ticks only at integer powers of base. This is normally True for major ticks and False for minor ticks.

**minor_thresholds**

[(subset, all), default: (1, 0.4)] If labelOnlyBase is False, these two numbers control the labeling of ticks that are not at integer powers of base; normally these are the minor ticks. The controlling parameter is the log of the axis data range. In the typical case where base is 10 it is the number of decades spanned by the axis, so we can call it 'numdec'. If `numdec <= all`, all minor ticks will be labeled. If `all < numdec <= subset`, then only a subset of minor ticks will be labeled, so as to avoid crowding. If `numdec > subset` then no minor ticks will be labeled.

**linthresh**

[None or float, default: None] If a symmetric log scale is in use, its `linthresh` parameter must be supplied here.

### Notes

The *set_locs* method must be called to enable the subsetting logic controlled by the `minor_thresholds` parameter.

In some cases such as the colorbar, there is no distinction between major and minor ticks; the tick locations might be set manually, or by a locator that puts ticks at integer powers of base and at intermediate locations. For this situation, disable the minor_thresholds logic by using `minor_thresholds=(np.inf, np.inf)`, so that all ticks will be labeled.

To disable labeling of minor ticks when 'labelOnlyBase' is False, use `minor_thresholds=(0, 0)`. This is the default for the "classic" style.

### Examples

To label a subset of minor ticks when the view limits span up to 2 decades, and all of the ticks when zoomed in to 0.5 decades or less, use `minor_thresholds=(2, 0.5)`.

To label all minor ticks when the view limits span up to 1.5 decades, use `minor_thresholds=(1.5, 1.5)`.

**format_data**(*value*)

Return the full string representation of the value with the position unspecified.

**format_data_short**(*value*)

Return a short string version of the tick value.

Defaults to the position-independent long value.

**set_base**(*base*)

Change the *base* for labeling.

> **Warning:** Should always match the base used for *LogLocator*

**set_label_minor**(*labelOnlyBase*)

Switch minor tick labeling on or off.

**Parameters**

**labelOnlyBase**

[bool] If True, label ticks only at integer powers of base.

**set_locs**(*locs=None*)

Use axis view limits to control which ticks are labeled.

The *locs* parameter is ignored in the present algorithm.

**class** matplotlib.ticker.**LogFormatterExponent**(*base=10.0*, *labelOnlyBase=False*, *minor_thresholds=None*, *linthresh=None*)

Bases: *LogFormatter*

Format values for log axis using `exponent = log_base(value)`.

**class** matplotlib.ticker.**LogFormatterMathtext**(*base=10.0*, *labelOnlyBase=False*, *minor_thresholds=None*, *linthresh=None*)

Bases: *LogFormatter*

Format values for log axis using `exponent = log_base(value)`.

**class** matplotlib.ticker.**LogFormatterSciNotation**(*base=10.0*, *labelOnlyBase=False*, *minor_thresholds=None*, *linthresh=None*)

Bases: *LogFormatterMathtext*

Format values following scientific notation in a logarithmic axis.

**class** matplotlib.ticker.**LogLocator**(*base=10.0*, *subs=(1.0, )*, *numdecs=<deprecated parameter>*, *numticks=None*)

Bases: *Locator*

Place logarithmically spaced ticks.

Places ticks at the values `subs[j] * base**i`.

**Parameters**

**base**

[float, default: 10.0] The base of the log used, so major ticks are placed at `base**n`, where `n` is an integer.

**subs**

[None or {'auto', 'all'} or sequence of float, default: (1.0,)] Gives the multiples of integer powers of the base at which to place ticks. The default of `(1.0, )` places ticks only at integer powers of the base. Permitted string values are `'auto'` and `'all'`. Both of these use an algorithm based on the axis view limits to determine whether and how to put ticks between integer powers of the base: - `'auto'`: Ticks are placed only between integer powers. - `'all'`: Ticks are placed between *and* at integer powers. - `None`: Equivalent to `'auto'`.

**numticks**

[None or int, default: None] The maximum number of ticks to allow on a given axis. The default of `None` will try to choose intelligently as long as this Locator has already been assigned to an axis using *get_tick_space*, but otherwise falls back to 9.

**nonsingular**(*vmin*, *vmax*)

Adjust a range as needed to avoid singularities.

This method gets called during autoscaling, with `(v0, v1)` set to the data limits on the Axes if the Axes contains any data, or `(-inf, +inf)` if not.

- If `v0 == v1` (possibly up to some floating point slop), this method returns an expanded interval around this value.
- If `(v0, v1) == (-inf, +inf)`, this method returns appropriate default view limits.
- Otherwise, `(v0, v1)` is returned without modification.

**property numdecs**

[*Deprecated*]

### Notes

Deprecated since version 3.8: This attribute has no effect.

**set_params**(*base=None*, *subs=None*, *numdecs=<deprecated parameter>*, *numticks=None*)

Set parameters within this locator.

**tick_values**(*vmin*, *vmax*)

Return the values of the located ticks given **vmin** and **vmax**.

---

**Note:** To get tick locations with the vmin and vmax values defined automatically for the associated `axis` simply call the Locator instance:

```
>>> print(type(loc))
<type 'Locator'>
>>> print(loc())
[1, 2, 3, 4]
```

---

**view_limits**(*vmin*, *vmax*)

Try to choose the view limits intelligently.

**class** matplotlib.ticker.**LogitFormatter**(*, *use_overline=False*, *one_half='\\frac{1}{2}'*, *minor=False*, *minor_threshold=25*, *minor_number=6*)

Bases: *Formatter*

Probability formatter (using Math text).

**Parameters**

**use_overline**

[bool, default: False] If x > 1/2, with x = 1-v, indicate if x should be displayed as $overline{v}$. The default is to display $1-v$.

**one_half**

[str, default: r"frac{1}{2}"] The string used to represent 1/2.

**minor**

[bool, default: False] Indicate if the formatter is formatting minor ticks or not. Basically minor ticks are not labelled, except when only few ticks are provided, ticks with most space with neighbor ticks are labelled. See other parameters to change the default behavior.

**minor_threshold**

[int, default: 25] Maximum number of locs for labelling some minor ticks. This parameter have no effect if minor is False.

**minor_number**

[int, default: 6] Number of ticks which are labelled when the number of ticks is below the threshold.

**format_data_short**(*value*)

Return a short string version of the tick value.

Defaults to the position-independent long value.

**set_locs**(*locs*)

Set the locations of the ticks.

This method is called before computing the tick labels because some formatters need to know all tick locations to do so.

**set_minor_number**(*minor_number*)

Set the number of minor ticks to label when some minor ticks are labelled.

**Parameters**

**minor_number**

[int] Number of ticks which are labelled when the number of ticks is below the threshold.

**set_minor_threshold**(*minor_threshold*)

Set the threshold for labelling minors ticks.

**Parameters**

**minor_threshold**

[int] Maximum number of locations for labelling some minor ticks. This parameter have no effect if minor is False.

**set_one_half**(*one_half*)

Set the way one half is displayed.

**one_half**

[str, default: r"frac{1}{2}"] The string used to represent 1/2.

**use_overline**(*use_overline*)

Switch display mode with overline for labelling p>1/2.

**Parameters**

**use_overline**

[bool, default: False] If x > 1/2, with x = 1-v, indicate if x should be displayed as $overline{v}$. The default is to display $1-v$.

**class** matplotlib.ticker.**LogitLocator**(*minor=False*, *, *nbins='auto'*)

Bases: *MaxNLocator*

Place ticks spaced evenly on a logit scale.

**Parameters**

**nbins**

[int or 'auto', optional] Number of ticks. Only used if minor is False.

**minor**

[bool, default: False] Indicate if this locator is for minor ticks or not.

**property minor**

**nonsingular**(*vmin*, *vmax*)

Adjust a range as needed to avoid singularities.

This method gets called during autoscaling, with `(v0, v1)` set to the data limits on the Axes if the Axes contains any data, or `(-inf, +inf)` if not.

- If `v0 == v1` (possibly up to some floating point slop), this method returns an expanded interval around this value.
- If `(v0, v1) == (-inf, +inf)`, this method returns appropriate default view limits.
- Otherwise, `(v0, v1)` is returned without modification.

**set_params**(*minor=None*, ***kwargs*)

Set parameters within this locator.

**tick_values**(*vmin*, *vmax*)

Return the values of the located ticks given **vmin** and **vmax**.

**Note:** To get tick locations with the vmin and vmax values defined automatically for the associated `axis` simply call the Locator instance:

```
>>> print(type(loc))
<type 'Locator'>
>>> print(loc())
[1, 2, 3, 4]
```

**class** matplotlib.ticker.**MaxNLocator**(*nbins=None*, ***kwargs*)

Bases: *Locator*

Place evenly spaced ticks, with a cap on the total number of ticks.

Finds nice tick locations with no more than *nbins* + 1 ticks being within the view limits. Locations beyond the limits are added to support autoscaling.

**Parameters**

**nbins**

[int or 'auto', default: 10] Maximum number of intervals; one less than max number of ticks. If the string 'auto', the number of bins will be automatically determined based on the length of the axis.

**steps**

[array-like, optional] Sequence of acceptable tick multiples, starting with 1 and ending with 10. For example, if `steps=[1, 2, 4, 5, 10]`, `20, 40, 60` or `0.4, 0.6, 0.8` would be possible sets of ticks because they are multiples of 2. `30, 60, 90` would not be generated because 3 does not appear in this example list of steps.

**integer**

[bool, default: False] If True, ticks will take only integer values, provided at least *min_n_ticks* integers are found within the view limits.

**symmetric**

[bool, default: False] If True, autoscaling will result in a range symmetric about zero.

**prune**

[{'lower', 'upper', 'both', None}, default: None] Remove the 'lower' tick, the 'upper' tick, or ticks on 'both' sides *if they fall exactly on an axis' edge* (this typically occurs when *rcParams["axes.autolimit_mode"]* (default: `'data'`) is 'round_numbers'). Removing such ticks is mostly useful for stacked or ganged plots, where the upper tick of an Axes overlaps with the lower tick of the axes above it.

**min_n_ticks**

[int, default: 2] Relax *nbins* and *integer* constraints if necessary to obtain this minimum number of ticks.

**default_params = {'integer': False, 'min_n_ticks': 2, 'nbins': 10, 'prune': None, 'steps': None, 'symmetric': False}**

**set_params**(***kwargs*)

Set parameters for this locator.

**Parameters**

**nbins**

[int or 'auto', optional] see *MaxNLocator*

**steps**

[array-like, optional] see *MaxNLocator*

**integer**

[bool, optional] see *MaxNLocator*

**symmetric**

[bool, optional] see *MaxNLocator*

**prune**

[{'lower', 'upper', 'both', None}, optional] see *MaxNLocator*

**min_n_ticks**

[int, optional] see *MaxNLocator*

**tick_values**(*vmin*, *vmax*)

Return the values of the located ticks given **vmin** and **vmax**.

---

**Note:** To get tick locations with the vmin and vmax values defined automatically for the associated `axis` simply call the Locator instance:

```
>>> print(type(loc))
<type 'Locator'>
>>> print(loc())
[1, 2, 3, 4]
```

---

**view_limits**(*dmin*, *dmax*)

Select a scale for the range from vmin to vmax.

Subclasses should override this method to change locator behaviour.

**class** matplotlib.ticker.**MultipleLocator**(*base=1.0*, *offset=0.0*)

Bases: *Locator*

Place ticks at every integer multiple of a base plus an offset.

**Parameters**

**base**

[float > 0] Interval between ticks.

**offset**

[float] Value added to each multiple of *base*.

New in version 3.8.

**set_params**(*base=None*, *offset=None*)

Set parameters within this locator.

**Parameters**

**base**

[float > 0] Interval between ticks.

**offset**

[float] Value added to each multiple of *base*.

New in version 3.8.

**tick_values**(*vmin*, *vmax*)

Return the values of the located ticks given **vmin** and **vmax**.

---

**Note:** To get tick locations with the vmin and vmax values defined automatically for the associated `axis` simply call the Locator instance:

```
>>> print(type(loc))
<type 'Locator'>
>>> print(loc())
[1, 2, 3, 4]
```

---

**view_limits**(*dmin*, *dmax*)

Set the view limits to the nearest tick values that contain the data.

**class** matplotlib.ticker.**NullFormatter**

Bases: *Formatter*

Always return the empty string.

**class** matplotlib.ticker.**NullLocator**

Bases: *Locator*

No ticks

**tick_values**(*vmin*, *vmax*)

Return the locations of the ticks.

---

**Note:** Because the values are Null, vmin and vmax are not used in this method.

---

**class** matplotlib.ticker.**PercentFormatter**(*xmax=100*, *decimals=None*, *symbol='%'*, *is_latex=False*)

Bases: *Formatter*

Format numbers as a percentage.

**Parameters**

**xmax**

[float] Determines how the number is converted into a percentage. *xmax* is the data value that corresponds to 100%. Percentages are computed as `x / xmax * 100`. So if the data is already scaled to be percentages, *xmax* will be 100. Another common situation is where *xmax* is 1.0.

**decimals**

[None or int] The number of decimal places to place after the point. If *None* (the default), the number will be computed automatically.

**symbol**

[str or None] A string that will be appended to the label. It may be *None* or empty to indicate that no symbol should be used. LaTeX special characters are escaped in *symbol* whenever latex mode is enabled, unless *is_latex* is *True*.

**is_latex**

[bool] If *False*, reserved LaTeX characters in *symbol* will be escaped.

**convert_to_pct**(*x*)

**format_pct**(*x*, *display_range*)

Format the number as a percentage number with the correct number of decimals and adds the percent symbol, if any.

If `self.decimals` is `None`, the number of digits after the decimal point is set based on the *display_range* of the axis as follows:

| display_range | decimals | sample |
|---|---|---|
| >50 | 0 | `x = 34.5` => 35% |
| >5 | 1 | `x = 34.5` => 34.5% |
| >0.5 | 2 | `x = 34.5` => 34.50% |
| ... | ... | ... |

This method will not be very good for tiny axis ranges or extremely large ones. It assumes that the values on the chart are percentages displayed on a reasonable scale.

**property symbol**

The configured percent symbol as a string.

If LaTeX is enabled via *rcParams["text.usetex"]* (default: `False`), the special characters `{'#', '$', '%', '&', '~', '_', '^', '\', '{', '}'}` are automatically escaped in the string.

**class** matplotlib.ticker.**ScalarFormatter**(*useOffset=None*, *useMathText=None*, *useLocale=None*)

Bases: *Formatter*

Format tick values as a number.

**Parameters**

**useOffset**

[bool or float, default: *rcParams["axes.formatter.useoffset"]* (default: `True`)] Whether to use offset notation. See *set_useOffset*.

**useMathText**

[bool, default: *rcParams["axes.formatter.use_mathtext"]* (default: `False`)] Whether to use fancy math formatting. See *set_useMathText*.

**useLocale**

[bool, default: *rcParams["axes.formatter.use_locale"]* (default: `False`).] Whether to use locale settings for decimal sign and positive sign. See *set_useLocale*.

## Notes

In addition to the parameters above, the formatting of scientific vs. floating point representation can be configured via *set_scientific* and *set_powerlimits*).

**Offset notation and scientific notation**

Offset notation and scientific notation look quite similar at first sight. Both split some information from the formatted tick values and display it at the end of the axis.

- The scientific notation splits up the order of magnitude, i.e. a multiplicative scaling factor, e.g. `1e6`.
- The offset notation separates an additive constant, e.g. `+1e6`. The offset notation label is always prefixed with a + or − sign and is thus distinguishable from the order of magnitude label.

The following plot with x limits `1_000_000` to `1_000_010` illustrates the different formatting. Note the labels at the right edge of the x axis.

**format_data**(*value*)

Return the full string representation of the value with the position unspecified.

**format_data_short**(*value*)

Return a short string version of the tick value.

Defaults to the position-independent long value.

**get_offset**()

Return scientific notation, plus offset.

**get_useLocale**()

Return whether locale settings are used for formatting.

**See also:**

*ScalarFormatter.set_useLocale*

**get_useMathText**()

Return whether to use fancy math formatting.

**See also:**

*ScalarFormatter.set_useMathText*

**get_useOffset**()

Return whether automatic mode for offset notation is active.

This returns True if `set_useOffset(True)`; it returns False if an explicit offset was set, e.g. `set_useOffset(1000)`.

**See also:**

*ScalarFormatter.set_useOffset*

**set_locs**(*locs*)

Set the locations of the ticks.

This method is called before computing the tick labels because some formatters need to know all tick locations to do so.

**set_powerlimits**(*lims*)

Set size thresholds for scientific notation.

**Parameters**

**lims**

[(int, int)] A tuple *(min_exp, max_exp)* containing the powers of 10 that determine the switchover threshold. For a number representable as $a \times 10^{\text{exp}}$ with $1 <= |a| < 10$, scientific notation will be used if `exp <= min_exp` or `exp >= max_exp`.

The default limits are controlled by *`rcParams["axes.formatter.limits"]`* (default: `[-5, 6]`).

In particular numbers with *exp* equal to the thresholds are written in scientific notation.

Typically, *min_exp* will be negative and *max_exp* will be positive.

For example, `formatter.set_powerlimits((-3, 4))` will provide the following formatting: $1 \times 10^{-3}, 9.9 \times 10^{-3}, 0.01, 9999, 1 \times 10^{4}$.

**See also:**

***ScalarFormatter.set_scientific***

**set_scientific**(*b*)

Turn scientific notation on or off.

**See also:**

***ScalarFormatter.set_powerlimits***

**set_useLocale**(*val*)

Set whether to use locale settings for decimal sign and positive sign.

**Parameters**

**val**

[bool or None] *None* resets to *`rcParams["axes.formatter.use_locale"]`* (default: `False`).

**set_useMathText**(*val*)

Set whether to use fancy math formatting.

If active, scientific notation is formatted as $1.2 \times 10^{3}$.

**Parameters**

**val**

[bool or None] *None* resets to *rcParams["axes.formatter.use_mathtext"]* (default: `False`).

**set_useOffset**(*val*)

Set whether to use offset notation.

When formatting a set numbers whose value is large compared to their range, the formatter can separate an additive constant. This can shorten the formatted numbers so that they are less likely to overlap when drawn on an axis.

**Parameters**

**val**

[bool or float]

- If False, do not use offset notation.
- If True (=automatic mode), use offset notation if it can make the residual numbers significantly shorter. The exact behavior is controlled by *rcParams["axes.formatter.offset_threshold"]* (default: `4`).
- If a number, force an offset of the given value.

### Examples

With active offset notation, the values

```
100_000, 100_002, 100_004, 100_006, 100_008
```

will be formatted as `0, 2, 4, 6, 8` plus an offset `+1e5`, which is written to the edge of the axis.

**property useLocale**

Return whether locale settings are used for formatting.

**See also:**

*ScalarFormatter.set_useLocale*

**property useMathText**

Return whether to use fancy math formatting.

**See also:**

*ScalarFormatter.set_useMathText*

**property useOffset**

Return whether automatic mode for offset notation is active.

This returns True if `set_useOffset(True)`; it returns False if an explicit offset was set, e.g. `set_useOffset(1000)`.

**See also:**

*ScalarFormatter.set_useOffset*

**class** matplotlib.ticker.**StrMethodFormatter**(*fmt*)

Bases: *Formatter*

Use a new-style format string (as used by `str.format`) to format the tick.

The field used for the tick value must be labeled *x* and the field used for the tick position must be labeled *pos*.

The formatter will respect *rcParams["axes.unicode_minus"]* (default: `True`) when formatting negative numeric values.

It is typically unnecessary to explicitly construct *StrMethodFormatter* objects, as *set_major_formatter* directly accepts the format string itself.

**class** matplotlib.ticker.**SymmetricalLogLocator**(*transform=None*, *subs=None*, *linthresh=None*, *base=None*)

Bases: *Locator*

Place ticks spaced linearly near zero and spaced logarithmically beyond a threshold.

**Parameters**

**transform**

[*SymmetricalLogTransform*, optional] If set, defines the *base* and *linthresh* of the symlog transform.

**base, linthresh**

[float, optional] The *base* and *linthresh* of the symlog transform, as documented for *SymmetricalLogScale*. These parameters are only used if *transform* is not set.

**subs**

[sequence of float, default: [1]] The multiples of integer powers of the base where ticks are placed, i.e., ticks are placed at `[sub * base**i for i in ... for sub in subs]`.

### Notes

Either *transform*, or both *base* and *linthresh*, must be given.

**set_params**(*subs=None*, *numticks=None*)

Set parameters within this locator.

**tick_values**(*vmin*, *vmax*)

Return the values of the located ticks given **vmin** and **vmax**.

---

**Note:** To get tick locations with the vmin and vmax values defined automatically for the associated `axis` simply call the Locator instance:

```
>>> print(type(loc))
<type 'Locator'>
>>> print(loc())
[1, 2, 3, 4]
```

---

**view_limits**(*vmin*, *vmax*)

Try to choose the view limits intelligently.

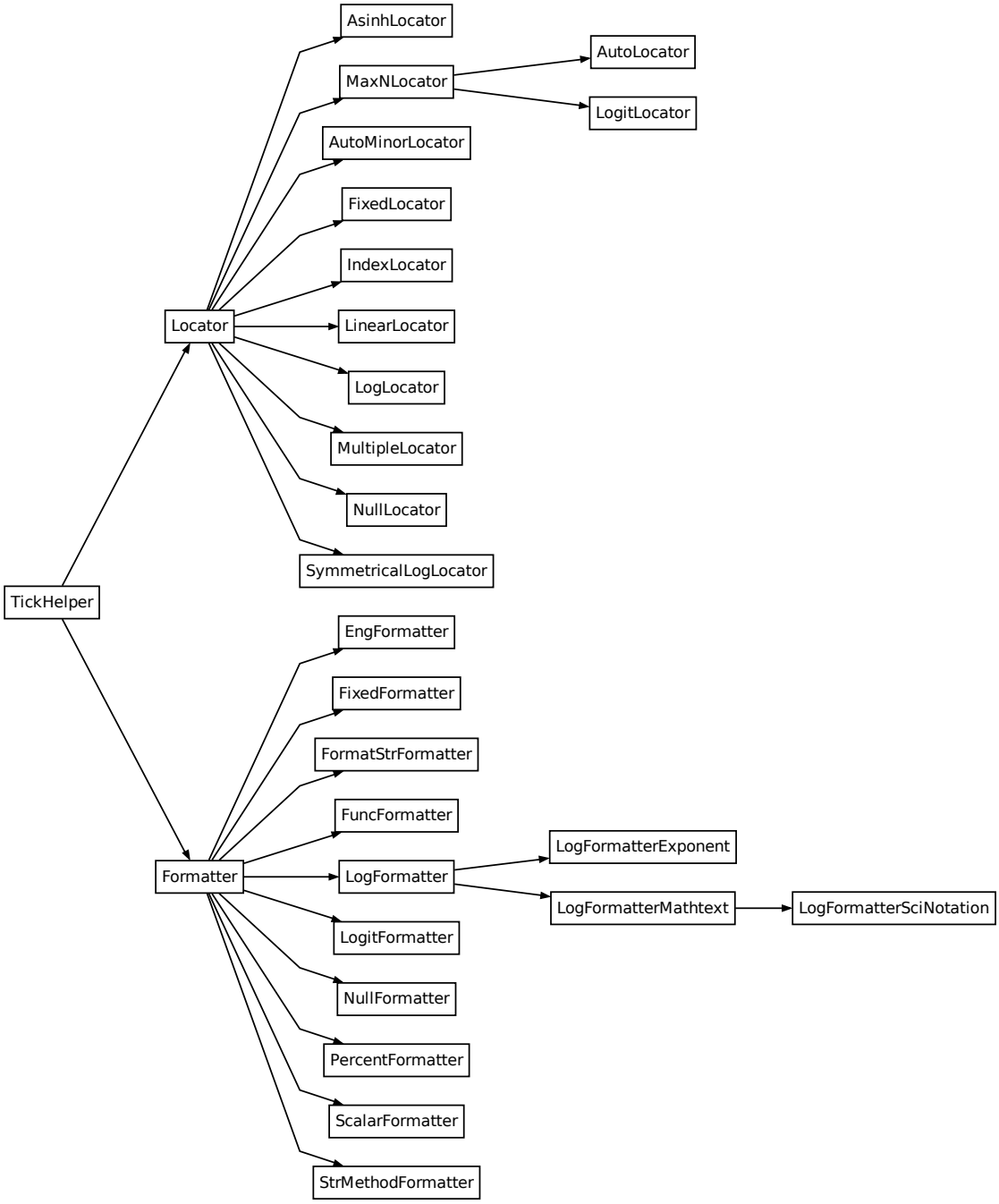
**class** matplotlib.ticker.**TickHelper**

Bases: `object`

**axis = None**

**create_dummy_axis**(***kwargs*)

**set_axis**(*axis*)


AsinhLocator
AutoLocator
MaxNLocator
LogitLocator
AutoMinorLocator
FixedLocator
IndexLocator
Locator
LinearLocator
LogLocator
MultipleLocator
NullLocator
SymmetricalLogLocator
TickHelper
EngFormatter
FixedFormatter
FormatStrFormatter
FuncFormatter
LogFormatterExponent
Formatter
LogFormatter
LogFormatterMathtext
LogFormatterSciNotation
LogitFormatter
NullFormatter
PercentFormatter
ScalarFormatter
StrMethodFormatter

### 7.2.56 matplotlib.tight_bbox

**Attention:** This module is considered internal.

Its use is deprecated and it will be removed in a future version.

Helper module for the *bbox_inches* parameter in `Figure.savefig`.

matplotlib._tight_bbox.**adjust_bbox**(*fig*, *bbox_inches*, *fixed_dpi=None*)

Temporarily adjust the figure so that only the specified area (bbox_inches) is saved.

It modifies fig.bbox, fig.bbox_inches, fig.transFigure._boxout, and fig.patch. While the figure size changes, the scale of the original figure is conserved. A function which restores the original values are returned.

matplotlib._tight_bbox.**process_figure_for_rasterizing**(*fig*, *bbox_inches_restore*, *fixed_dpi=None*)

A function that needs to be called when figure dpi changes during the drawing (e.g., rasterizing). It recovers the bbox and re-adjust it with the new dpi.

### 7.2.57 matplotlib.tight_layout

**Attention:** This module is considered internal.

Its use is deprecated and it will be removed in a future version.

Routines to adjust subplot params so that subplots are nicely fit in the figure. In doing so, only axis labels, tick labels, Axes titles and offsetboxes that are anchored to Axes are currently considered.

Internally, this module assumes that the margins (left margin, etc.) which are differences between `Axes.get_tightbbox` and `Axes.bbox` are independent of Axes position. This may fail if `Axes.adjustable` is `datalim` as well as such cases as when left or right margin are affected by xlabel.

matplotlib._tight_layout.**get_subplotspec_list**(*axes_list*, *grid_spec=None*)

Return a list of subplotspec from the given list of Axes.

For an instance of Axes that does not support subplotspec, None is inserted in the list.

If grid_spec is given, None is inserted for those not from the given grid_spec.

matplotlib._tight_layout.**get_tight_layout_figure**(*fig*, *axes_list*, *subplotspec_list*, *renderer*, *pad=1.08*, *h_pad=None*, *w_pad=None*, *rect=None*)

Return subplot parameters for tight-layouted-figure with specified padding.

**Parameters**

**fig**

[Figure]

**axes_list**

[list of Axes]

**subplotspec_list**

[list of *SubplotSpec*] The subplotspecs of each Axes.

**renderer**

[renderer]

**pad**

[float] Padding between the figure edge and the edges of subplots, as a fraction of the font size.

**h_pad, w_pad**

[float] Padding (height/width) between edges of adjacent subplots. Defaults to *pad*.

**rect**

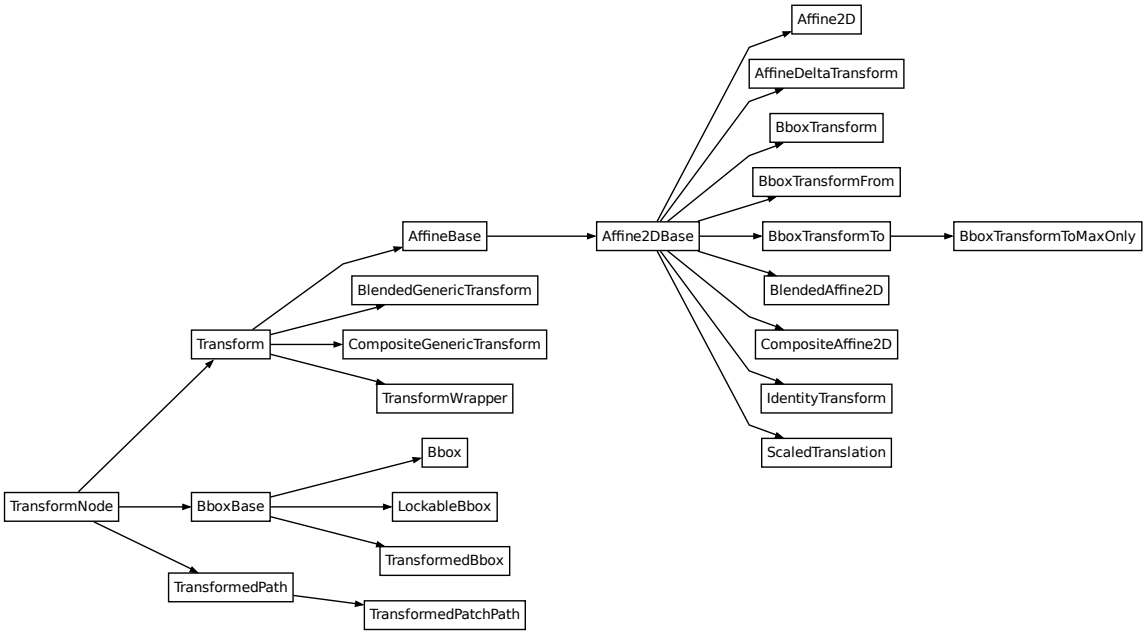
[tuple (left, bottom, right, top), default: None.] rectangle in normalized figure coordinates that the whole subplots area (including labels) will fit into. Defaults to using the entire figure.

**Returns**

**subplotspec or None**

subplotspec kwargs to be passed to *Figure.subplots_adjust* or None if tight_layout could not be accomplished.

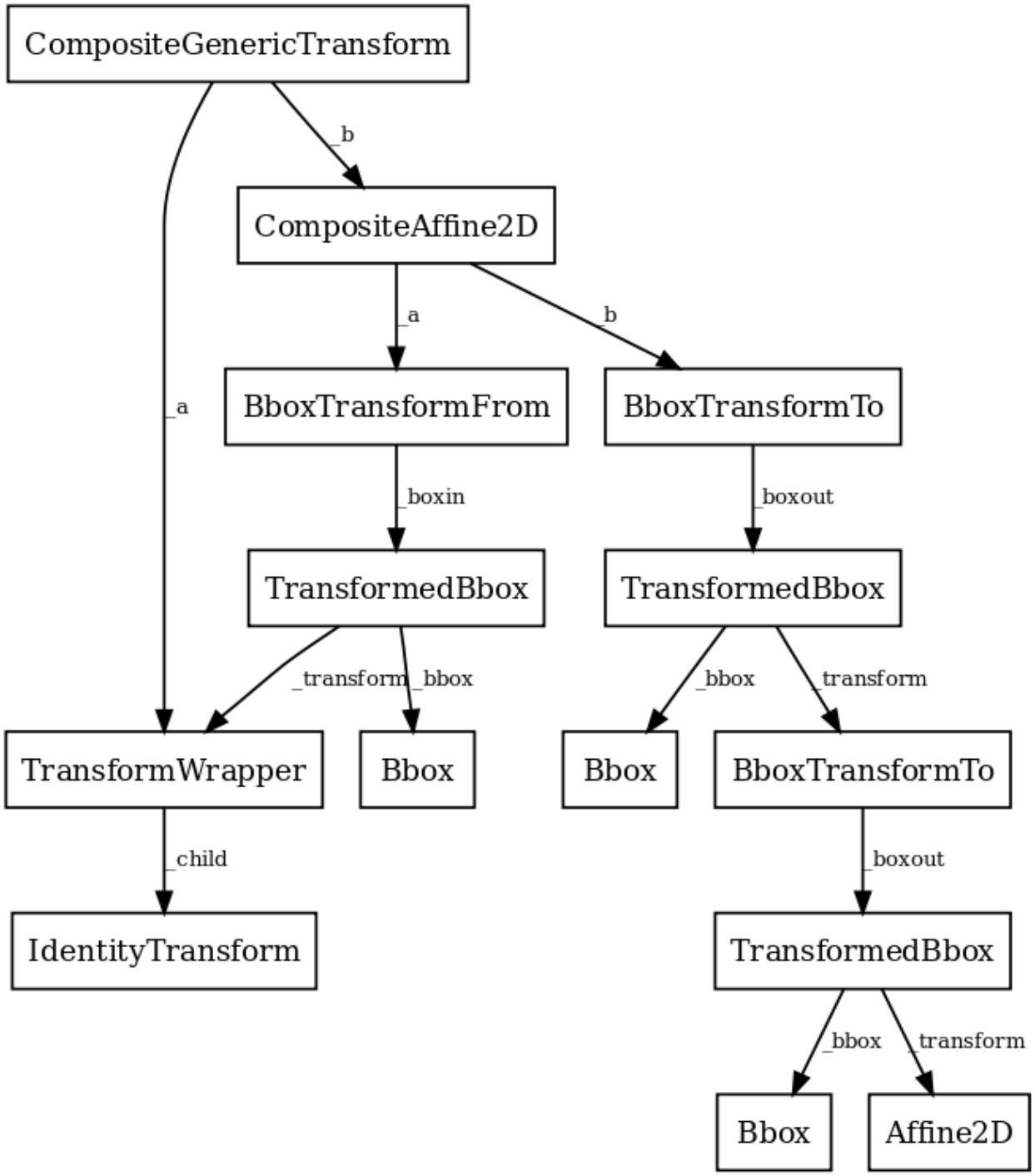
### 7.2.58 matplotlib.transforms




Matplotlib includes a framework for arbitrary geometric transformations that is used determine the final position of all elements drawn on the canvas.

Transforms are composed into trees of *TransformNode* objects whose actual value depends on their children. When the contents of children change, their parents are automatically invalidated. The next time an invalidated transform is accessed, it is recomputed to reflect those changes. This invalidation/caching approach prevents unnecessary recomputations of transforms, and contributes to better interactive performance.

For example, here is a graph of the transform tree used to plot data to the graph:




The framework can be used for both affine and non-affine transformations. However, for speed, we want to use the backend renderers to perform affine transformations whenever possible. Therefore, it is possible to perform just the affine or non-affine part of a transformation on a set of data. The affine is always assumed to occur after the non-affine. For any transform:

```
full transform == non-affine part + affine part
```

The backends are not expected to handle non-affine transformations themselves.

See the tutorial *Transformations Tutorial* for examples of how to use transforms.

**class** matplotlib.transforms.**Affine2D**(*matrix=None*, ***kwargs*)

Bases: *Affine2DBase*

A mutable 2D affine transformation.

Initialize an Affine transform from a 3x3 numpy float array:

```
a c e
b d f
0 0 1
```

If *matrix* is None, initialize with the identity transform.

**clear**()

Reset the underlying matrix to the identity transform.

**static from_values**(*a*, *b*, *c*, *d*, *e*, *f*)

Create a new Affine2D instance from the given values:

```
a c e
b d f
0 0 1
```

.

**get_matrix**()

Get the underlying transformation matrix as a 3x3 array:

```
a c e
b d f
0 0 1
```

.

**rotate**(*theta*)

Add a rotation (in radians) to this transform in place.

Returns *self*, so this method can easily be chained with more calls to *rotate()*, *rotate_deg()*, *translate()* and *scale()*.

**rotate_around**(*x*, *y*, *theta*)

Add a rotation (in radians) around the point (x, y) in place.

Returns *self*, so this method can easily be chained with more calls to *rotate()*, *rotate_deg()*, *translate()* and *scale()*.

**rotate_deg**(*degrees*)

Add a rotation (in degrees) to this transform in place.

Returns *self*, so this method can easily be chained with more calls to *rotate()*, *rotate_deg()*, *translate()* and *scale()*.

**rotate_deg_around**(*x*, *y*, *degrees*)

Add a rotation (in degrees) around the point (x, y) in place.

Returns *self*, so this method can easily be chained with more calls to *rotate()*, *rotate_deg()*, *translate()* and *scale()*.

**scale**(*sx*, *sy=None*)

Add a scale in place.

If *sy* is None, the same scale is applied in both the *x*- and *y*-directions.

Returns *self*, so this method can easily be chained with more calls to *rotate()*, *rotate_deg()*, *translate()* and *scale()*.

**set**(*other*)

Set this transformation from the frozen copy of another *Affine2DBase* object.

**set_matrix**(*mtx*)

Set the underlying transformation matrix from a 3x3 array:

```
a c e
b d f
0 0 1
```

.

**skew**(*xShear*, *yShear*)

Add a skew in place.

*xShear* and *yShear* are the shear angles along the *x*- and *y*-axes, respectively, in radians.

Returns *self*, so this method can easily be chained with more calls to *rotate()*, *rotate_deg()*, *translate()* and *scale()*.

**skew_deg**(*xShear*, *yShear*)

Add a skew in place.

*xShear* and *yShear* are the shear angles along the *x*- and *y*-axes, respectively, in degrees.

Returns *self*, so this method can easily be chained with more calls to *rotate()*, *rotate_deg()*, *translate()* and *scale()*.

**translate**(*tx*, *ty*)

Add a translation in place.

Returns *self*, so this method can easily be chained with more calls to *rotate()*, *rotate_deg()*, *translate()* and *scale()*.

**class** matplotlib.transforms.**Affine2DBase**(**args*, ***kwargs*)

Bases: *AffineBase*

The base class of all 2D affine transformations.

2D affine transformations are performed using a 3x3 numpy array:

```
a c e
b d f
0 0 1
```

This class provides the read-only interface. For a mutable 2D affine transformation, use *Affine2D*.

Subclasses of this class will generally only need to override a constructor and *get_matrix* that generates a custom 3x3 matrix.

**Parameters**

**shorthand_name**

[str] A string representing the "name" of the transform. The name carries no significance other than to improve the readability of `str(transform)` when DEBUG=True.

**frozen**()

Return a frozen copy of this transform node. The frozen copy will not be updated when its children change. Useful for storing a previously known state of a transform where `copy.deepcopy()` might normally be used.

**has_inverse = True**

True if this transform has a corresponding inverse transform.

**input_dims = 2**

The number of input dimensions of this transform. Must be overridden (with integers) in the subclass.

**inverted**()

Return the corresponding inverse transformation.

It holds `x == self.inverted().transform(self.transform(x))`.

The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

**property is_separable**

bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

**output_dims = 2**

The number of output dimensions of this transform. Must be overridden (with integers) in the subclass.

**to_values**()

Return the values of the matrix as an `(a, b, c, d, e, f)` tuple.

**transform_affine**(*values*)

Apply only the affine part of this transformation on the given array of values.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally a no-op. In affine transformations, this is equivalent to `transform(values)`.

**Parameters**

**values**

[array] The input values as an array of length *input_dims* or shape (N, *input_dims*).

**Returns**

**array**

The output values as an array of length *output_dims* or shape (N, *output_dims*), depending on the input.

**class** matplotlib.transforms.**AffineBase**(**args*, ***kwargs*)

Bases: *Transform*

The base class of all affine transformations of any number of dimensions.

**Parameters**

**shorthand_name**

[str] A string representing the "name" of the transform. The name carries no significance other than to improve the readability of `str(transform)` when DEBUG=True.

**get_affine**()

Get the affine part of this transform.

**is_affine = True**

**transform**(*values*)

Apply this transformation on the given array of *values*.

**Parameters**

**values**

[array-like] The input values as an array of length `input_dims` or shape (N, `input_dims`).

**Returns**

**array**

The output values as an array of length `output_dims` or shape (N, `output_dims`), depending on the input.

**`transform_affine`**(*values*)

Apply only the affine part of this transformation on the given array of values.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally a no-op. In affine transformations, this is equivalent to `transform(values)`.

**Parameters**

**values**

[array] The input values as an array of length `input_dims` or shape (N, `input_dims`).

**Returns**

**array**

The output values as an array of length `output_dims` or shape (N, `output_dims`), depending on the input.

**`transform_non_affine`**(*values*)

Apply only the non-affine part of this transformation.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

**Parameters**

**values**

[array] The input values as an array of length `input_dims` or shape (N, `input_dims`).

**Returns**

**array**

The output values as an array of length `output_dims` or shape (N, `output_dims`), depending on the input.

**transform_path**(*path*)

Apply the transform to *Path* *path*, returning a new *Path*.

In some cases, this transform may insert curves into the path that began as line segments.

**transform_path_affine**(*path*)

Apply the affine part of this transform to *Path* *path*, returning a new *Path*.

`transform_path(path)` is equivalent to `transform_path_affine(transform_path_non_affi`

**transform_path_non_affine**(*path*)

Apply the non-affine part of this transform to *Path* *path*, returning a new *Path*.

`transform_path(path)` is equivalent to `transform_path_affine(transform_path_non_affi`

**class** matplotlib.transforms.**AffineDeltaTransform**(*transform*, ***kwargs*)

Bases: *Affine2DBase*

A transform wrapper for transforming displacements between pairs of points.

This class is intended to be used to transform displacements ("position deltas") between pairs of points (e.g., as the `offset_transform` of *Collection*s): given a transform `t` such that `t = AffineDeltaTransform(t) + offset`, `AffineDeltaTransform` satisfies `AffineDeltaTransform(a - b) == AffineDeltaTransform(a) - AffineDeltaTransform(b)`.

This is implemented by forcing the offset components of the transform matrix to zero.

This class is experimental as of 3.3, and the API may change.

**Parameters**

**shorthand_name**

[str] A string representing the "name" of the transform. The name carries no significance other than to improve the readability of `str(transform)` when DEBUG=True.

**get_matrix**()

Get the matrix for the affine part of this transform.

**class** matplotlib.transforms.**Bbox**(*points*, ***kwargs*)

Bases: *BboxBase*

A mutable bounding box.

### Examples

#### Create from known bounds

The default constructor takes the boundary "points" `[[xmin, ymin], [xmax, ymax]]`.

```
>>> Bbox([[1, 1], [3, 7]])
Bbox([[1.0, 1.0], [3.0, 7.0]])
```

Alternatively, a Bbox can be created from the flattened points array, the so-called "extents" `(xmin, ymin, xmax, ymax)`

```
>>> Bbox.from_extents(1, 1, 3, 7)
Bbox([[1.0, 1.0], [3.0, 7.0]])
```

or from the "bounds" `(xmin, ymin, width, height)`.

```
>>> Bbox.from_bounds(1, 1, 2, 6)
Bbox([[1.0, 1.0], [3.0, 7.0]])
```

#### Create from collections of points

The "empty" object for accumulating Bboxs is the null bbox, which is a stand-in for the empty set.

```
>>> Bbox.null()
Bbox([[inf, inf], [-inf, -inf]])
```

Adding points to the null bbox will give you the bbox of those points.

```
>>> box = Bbox.null()
>>> box.update_from_data_xy([[1, 1]])
>>> box
Bbox([[1.0, 1.0], [1.0, 1.0]])
>>> box.update_from_data_xy([[2, 3], [3, 2]], ignore=False)
>>> box
Bbox([[1.0, 1.0], [3.0, 3.0]])
```

Setting `ignore=True` is equivalent to starting over from a null bbox.

```
>>> box.update_from_data_xy([[1, 1]], ignore=True)
>>> box
Bbox([[1.0, 1.0], [1.0, 1.0]])
```

**Warning:** It is recommended to always specify `ignore` explicitly. If not, the default value of `ignore` can be changed at any time by code with access to your Bbox, for example using the method *`ignore`*.

#### Properties of the ``null`` bbox

**Note:** The current behavior of *`Bbox.null()`* may be surprising as it does not have all of the

properties of the "empty set", and as such does not behave like a "zero" object in the mathematical sense. We may change that in the future (with a deprecation period).

---

The null bbox is the identity for intersections

```
>>> Bbox.intersection(Bbox([[1, 1], [3, 7]]), Bbox.null())
Bbox([[1.0, 1.0], [3.0, 7.0]])
```

except with itself, where it returns the full space.

```
>>> Bbox.intersection(Bbox.null(), Bbox.null())
Bbox([[-inf, -inf], [inf, inf]])
```

A union containing null will always return the full space (not the other set!)

```
>>> Bbox.union([Bbox([[0, 0], [0, 0]]), Bbox.null()])
Bbox([[-inf, -inf], [inf, inf]])
```

**Parameters**

**points**

[ndarray] A (2, 2) array of the form `[[x0, y0], [x1, y1]]`.

**property bounds**

Return (*x0*, *y0*, `width`, `height`).

**static from_bounds**(*x0*, *y0*, *width*, *height*)

Create a new *Bbox* from *x0*, *y0*, *width* and *height*.

*width* and *height* may be negative.

**static from_extents**(**args*, *minpos=None*)

Create a new Bbox from *left*, *bottom*, *right* and *top*.

The *y*-axis increases upwards.

**Parameters**

**left, bottom, right, top**

[float] The four extents of the bounding box.

**minpos**

[float or None] If this is supplied, the Bbox will have a minimum positive value set. This is useful when dealing with logarithmic scales and other scales where negative bounds result in floating point errors.

**frozen**()

The base class for anything that participates in the transform tree and needs to invalidate its parents or be invalidated. This includes classes that are not really transforms, such as bounding boxes, since some transforms depend on bounding boxes to compute their values.

**get_points**()

Get the points of the bounding box as an array of the form `[[x0, y0], [x1, y1]]`.

**ignore**(*value*)

Set whether the existing bounds of the box should be ignored by subsequent calls to *update_from_data_xy()*.

**value**

[bool]

- When `True`, subsequent calls to *update_from_data_xy* will ignore the existing bounds of the *Bbox*.
- When `False`, subsequent calls to *update_from_data_xy* will include the existing bounds of the *Bbox*.

**property intervalx**

The pair of *x* coordinates that define the bounding box.

This is not guaranteed to be sorted from left to right.

**property intervaly**

The pair of *y* coordinates that define the bounding box.

This is not guaranteed to be sorted from bottom to top.

**property minpos**

The minimum positive value in both directions within the Bbox.

This is useful when dealing with logarithmic scales and other scales where negative bounds result in floating point errors, and will be used as the minimum extent instead of *p0*.

**property minposx**

The minimum positive value in the *x*-direction within the Bbox.

This is useful when dealing with logarithmic scales and other scales where negative bounds result in floating point errors, and will be used as the minimum *x*-extent instead of *x0*.

**property minposy**

The minimum positive value in the *y*-direction within the Bbox.

This is useful when dealing with logarithmic scales and other scales where negative bounds result in floating point errors, and will be used as the minimum *y*-extent instead of *y0*.

**mutated**()

Return whether the bbox has changed since init.

**mutatedx**()

Return whether the x-limits have changed since init.

**mutatedy**()

Return whether the y-limits have changed since init.

**static null**()

Create a new null *Bbox* from (inf, inf) to (-inf, -inf).

**property p0**

The first pair of (*x*, *y*) coordinates that define the bounding box.

This is not guaranteed to be the bottom-left corner (for that, use `min`).

**property p1**

The second pair of (*x*, *y*) coordinates that define the bounding box.

This is not guaranteed to be the top-right corner (for that, use `max`).

**set**(*other*)

Set this bounding box from the "frozen" bounds of another *Bbox*.

**set_points**(*points*)

Set the points of the bounding box directly from an array of the form `[[x0, y0], [x1, y1]]`. No error checking is performed, as this method is mainly for internal use.

**static unit**()

Create a new unit *Bbox* from (0, 0) to (1, 1).

**update_from_data_x**(*x*, *ignore=None*)

Update the x-bounds of the *Bbox* based on the passed in data. After updating, the bounds will have positive *width*, and *x0* will be the minimal value.

**Parameters**

**x**

[`ndarray`] Array of x-values.

**ignore**

[bool, optional]

- When `True`, ignore the existing bounds of the *Bbox*.
- When `False`, include the existing bounds of the *Bbox*.
- When `None`, use the last value passed to *ignore()*.

**update_from_data_xy**(*xy*, *ignore=None*, *updatex=True*, *updatey=True*)

Update the *Bbox* bounds based on the passed in *xy* coordinates.

After updating, the bounds will have positive *width* and *height*; *x0* and *y0* will be the minimal values.

**Parameters**

**xy**

[(N, 2) array-like] The (x, y) coordinates.

**ignore**

[bool, optional]

- When `True`, ignore the existing bounds of the *Bbox*.
- When `False`, include the existing bounds of the *Bbox*.
- When `None`, use the last value passed to *ignore()*.

**updatex, updatey**

[bool, default: True] When `True`, update the x/y values.

**update_from_data_y**(*y*, *ignore=None*)

Update the y-bounds of the *Bbox* based on the passed in data. After updating, the bounds will have positive *height*, and *y0* will be the minimal value.

**Parameters**

**y**

[`ndarray`] Array of y-values.

**ignore**

[bool, optional]

- When `True`, ignore the existing bounds of the *Bbox*.
- When `False`, include the existing bounds of the *Bbox*.
- When `None`, use the last value passed to *ignore()*.

**update_from_path**(*path*, *ignore=None*, *updatex=True*, *updatey=True*)

Update the bounds of the *Bbox* to contain the vertices of the provided path. After updating, the bounds will have positive *width* and *height*; *x0* and *y0* will be the minimal values.

**Parameters**

**path**

[*Path*]

**ignore**

[bool, optional]

- When `True`, ignore the existing bounds of the *Bbox*.
- When `False`, include the existing bounds of the *Bbox*.

- When None, use the last value passed to *ignore()*.

**updatex, updatey**

[bool, default: True] When True, update the x/y values.

**property x0**

The first of the pair of *x* coordinates that define the bounding box.

This is not guaranteed to be less than *x1* (for that, use xmin).

**property x1**

The second of the pair of *x* coordinates that define the bounding box.

This is not guaranteed to be greater than *x0* (for that, use xmax).

**property y0**

The first of the pair of *y* coordinates that define the bounding box.

This is not guaranteed to be less than *y1* (for that, use ymin).

**property y1**

The second of the pair of *y* coordinates that define the bounding box.

This is not guaranteed to be greater than *y0* (for that, use ymax).

**class** matplotlib.transforms.**BboxBase**(*shorthand_name=None*)

Bases: *TransformNode*

The base class of all bounding boxes.

This class is immutable; *Bbox* is a mutable subclass.

The canonical representation is as two points, with no restrictions on their ordering. Convenience properties are provided to get the left, bottom, right and top edges and width and height, but these are not stored explicitly.

**Parameters**

**shorthand_name**

[str] A string representing the "name" of the transform. The name carries no significance other than to improve the readability of str(transform) when DEBUG=True.

**anchored**(*c*, *container=None*)

Return a copy of the *Bbox* anchored to *c* within *container*.

**Parameters**

**c**

[(float, float) or {'C', 'SW', 'S', 'SE', 'E', 'NE', ...}] Either an (*x*, *y*) pair of relative coordinates (0 is left or bottom, 1 is right or top), 'C' (center), or a cardinal direction ('SW', southwest, is bottom left, etc.).

**container**

[*Bbox*, optional] The box within which the *Bbox* is positioned.

**See also:**

***Axes.set_anchor***

**property bounds**

Return (*x0*, *y0*, *width*, *height*).

**coefs = {'C': (0.5, 0.5), 'E': (1.0, 0.5), 'N': (0.5, 1.0), 'NE': (1.0, 1.0), 'NW': (0, 1.0), 'S': (0.5, 0), 'SE': (1.0, 0), 'SW': (0, 0), 'W': (0, 0.5)}**

**contains**(*x*, *y*)

Return whether `(x, y)` is in the bounding box or on its edge.

**containsx**(*x*)

Return whether *x* is in the closed (*x0*, *x1*) interval.

**containsy**(*y*)

Return whether *y* is in the closed (*y0*, *y1*) interval.

**corners**()

Return the corners of this rectangle as an array of points.

Specifically, this returns the array `[[x0, y0], [x0, y1], [x1, y0], [x1, y1]]`.

**count_contains**(*vertices*)

Count the number of vertices contained in the *Bbox*. Any vertices with a non-finite x or y value are ignored.

**Parameters**

**vertices**

[(N, 2) array]

**count_overlaps**(*bboxes*)

Count the number of bounding boxes that overlap this one.

**Parameters**

**bboxes**

[sequence of *BboxBase*]

**expanded**(*sw*, *sh*)

Construct a *Bbox* by expanding this one around its center by the factors *sw* and *sh*.

**property extents**

Return (*x0*, *y0*, *x1*, *y1*).

**frozen**()

The base class for anything that participates in the transform tree and needs to invalidate its parents or be invalidated. This includes classes that are not really transforms, such as bounding boxes, since some transforms depend on bounding boxes to compute their values.

**fully_contains**(*x*, *y*)

Return whether `x, y` is in the bounding box, but not on its edge.

**fully_containsx**(*x*)

Return whether *x* is in the open (*x0*, *x1*) interval.

**fully_containsy**(*y*)

Return whether *y* is in the open (*y0*, *y1*) interval.

**fully_overlaps**(*other*)

Return whether this bounding box overlaps with the other bounding box, not including the edges.

**Parameters**

**other**

[*BboxBase*]

**get_points**()

**property height**

The (signed) height of the bounding box.

**static intersection**(*bbox1*, *bbox2*)

Return the intersection of *bbox1* and *bbox2* if they intersect, or None if they don't.

**property intervalx**

The pair of *x* coordinates that define the bounding box.

This is not guaranteed to be sorted from left to right.

**property intervaly**

The pair of *y* coordinates that define the bounding box.

This is not guaranteed to be sorted from bottom to top.

**is_affine = True**

**is_bbox = True**

**property max**

The top-right corner of the bounding box.

**property min**

The bottom-left corner of the bounding box.

**overlaps**(*other*)

Return whether this bounding box overlaps with the other bounding box.

**Parameters**

**other**

[*BboxBase*]

**property p0**

The first pair of (*x*, *y*) coordinates that define the bounding box.

This is not guaranteed to be the bottom-left corner (for that, use *min*).

**property p1**

The second pair of (*x*, *y*) coordinates that define the bounding box.

This is not guaranteed to be the top-right corner (for that, use *max*).

**padded**(*w_pad*, *h_pad=None*)

Construct a *Bbox* by padding this one on all four sides.

**Parameters**

**w_pad**

[float] Width pad

**h_pad**

[float, optional] Height pad. Defaults to *w_pad*.

**rotated**(*radians*)

Return the axes-aligned bounding box that bounds the result of rotating this *Bbox* by an angle of *radians*.

**shrunk**(*mx*, *my*)

Return a copy of the *Bbox*, shrunk by the factor *mx* in the *x* direction and the factor *my* in the *y* direction. The lower left corner of the box remains unchanged. Normally *mx* and *my* will be less than 1, but this is not enforced.

**shrunk_to_aspect**(*box_aspect*, *container=None*, *fig_aspect=1.0*)

Return a copy of the *Bbox*, shrunk so that it is as large as it can be while having the desired aspect ratio, *box_aspect*. If the box coordinates are relative (i.e. fractions of a larger box such as a figure) then the physical aspect ratio of that figure is specified with *fig_aspect*, so that *box_aspect* can also be given as a ratio of the absolute dimensions, not the relative dimensions.

**property size**

The (signed) width and height of the bounding box.

**splitx**(**args*)

Return a list of new *Bbox* objects formed by splitting the original one with vertical lines at fractional positions given by *args*.

**splity**(**args*)

Return a list of new *Bbox* objects formed by splitting the original one with horizontal lines at fractional positions given by *args*.

**transformed**(*transform*)

Construct a *Bbox* by statically transforming this one by *transform*.

**translated**(*tx*, *ty*)

Construct a *Bbox* by translating this one by *tx* and *ty*.

**static union**(*bboxes*)

Return a *Bbox* that contains all of the given *bboxes*.

**property width**

The (signed) width of the bounding box.

**property x0**

The first of the pair of *x* coordinates that define the bounding box.

This is not guaranteed to be less than *x1* (for that, use *xmin*).

**property x1**

The second of the pair of *x* coordinates that define the bounding box.

This is not guaranteed to be greater than *x0* (for that, use *xmax*).

**property xmax**

The right edge of the bounding box.

**property xmin**

The left edge of the bounding box.

**property y0**

The first of the pair of *y* coordinates that define the bounding box.

This is not guaranteed to be less than *y1* (for that, use *ymin*).

**property y1**

The second of the pair of *y* coordinates that define the bounding box.

This is not guaranteed to be greater than *y0* (for that, use *ymax*).

**property ymax**

The top edge of the bounding box.

**property ymin**

The bottom edge of the bounding box.

**class** matplotlib.transforms.**BboxTransform**(*boxin*, *boxout*, ***kwargs*)

Bases: *Affine2DBase*

*BboxTransform* linearly transforms points from one *Bbox* to another.

Create a new *BboxTransform* that linearly transforms points from *boxin* to *boxout*.

**get_matrix**()

Get the matrix for the affine part of this transform.

**is_separable = True**

True if this transform is separable in the x- and y- dimensions.

**class** matplotlib.transforms.**BboxTransformFrom**(*boxin*, ***kwargs*)

Bases: *Affine2DBase*

*BboxTransformFrom* linearly transforms points from a given *Bbox* to the unit bounding box.

**Parameters**

**shorthand_name**

[str] A string representing the "name" of the transform. The name carries no significance other than to improve the readability of `str(transform)` when DEBUG=True.

**get_matrix**()

Get the matrix for the affine part of this transform.

**is_separable = True**

True if this transform is separable in the x- and y- dimensions.

**class** matplotlib.transforms.**BboxTransformTo**(*boxout*, ***kwargs*)

Bases: *Affine2DBase*

*BboxTransformTo* is a transformation that linearly transforms points from the unit bounding box to a given *Bbox*.

Create a new *BboxTransformTo* that linearly transforms points from the unit bounding box to *boxout*.

**get_matrix**()

Get the matrix for the affine part of this transform.

**is_separable = True**

True if this transform is separable in the x- and y- dimensions.

**class** matplotlib.transforms.**BboxTransformToMaxOnly**(*boxout*, ***kwargs*)

Bases: *BboxTransformTo*

[*Deprecated*] *BboxTransformToMaxOnly* is a transformation that linearly transforms points from the unit bounding box to a given *Bbox* with a fixed upper left of (0, 0).

### Notes

Deprecated since version 3.9.

Create a new *BboxTransformTo* that linearly transforms points from the unit bounding box to *boxout*.

**get_matrix**()

Get the matrix for the affine part of this transform.

**class** matplotlib.transforms.**BlendedAffine2D**(*x_transform*, *y_transform*, ***kwargs*)

Bases: _BlendedMixin, *Affine2DBase*

A "blended" transform uses one transform for the *x*-direction, and another transform for the *y*-direction.

This version is an optimization for the case where both child transforms are of type *Affine2DBase*.

Create a new "blended" transform using *x_transform* to transform the *x*-axis and *y_transform* to transform the *y*-axis.

Both *x_transform* and *y_transform* must be 2D affine transforms.

You will generally not call this constructor directly but use the *blended_transform_factory* function instead, which can determine automatically which kind of blended transform to create.

**get_matrix**()

Get the matrix for the affine part of this transform.

**is_separable = True**

True if this transform is separable in the x- and y- dimensions.

**class** matplotlib.transforms.**BlendedGenericTransform**(*x_transform*, *y_transform*, ***kwargs*)

Bases: _BlendedMixin, *Transform*

A "blended" transform uses one transform for the *x*-direction, and another transform for the *y*-direction.

This "generic" version can handle any given child transform in the *x*- and *y*-directions.

Create a new "blended" transform using *x_transform* to transform the *x*-axis and *y_transform* to transform the *y*-axis.

You will generally not call this constructor directly but use the *blended_transform_factory* function instead, which can determine automatically which kind of blended transform to create.

**contains_branch**(*other*)

Return whether the given transform is a sub-tree of this transform.

This routine uses transform equality to identify sub-trees, therefore in many situations it is object id which will be used.

For the case where the given transform represents the whole of this transform, returns True.

**property depth**

Return the number of transforms which have been chained together to form this Transform instance.

---

**Note:** For the special case of a Composite transform, the maximum depth of the two is returned.

---

**frozen**()

Return a frozen copy of this transform node. The frozen copy will not be updated when its children change. Useful for storing a previously known state of a transform where `copy.deepcopy()` might normally be used.

**get_affine**()

Get the affine part of this transform.

**property has_inverse**

bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

**input_dims = 2**

The number of input dimensions of this transform. Must be overridden (with integers) in the subclass.

**inverted**()

Return the corresponding inverse transformation.

It holds `x == self.inverted().transform(self.transform(x))`.

The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

**property is_affine**

bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

**is_separable = True**

True if this transform is separable in the x- and y- dimensions.

**output_dims = 2**

The number of output dimensions of this transform. Must be overridden (with integers) in the subclass.

**pass_through = True**

If pass_through is True, all ancestors will always be invalidated, even if 'self' is already invalid.

**transform_non_affine**(*values*)

Apply only the non-affine part of this transformation.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

**Parameters**

**values**

[array] The input values as an array of length *input_dims* or shape (N, *input_dims*).

**Returns**

**array**

The output values as an array of length *output_dims* or shape (N, *output_dims*), depending on the input.

**class** matplotlib.transforms.**CompositeAffine2D**(*a*, *b*, ***kwargs*)

Bases: *Affine2DBase*

A composite transform formed by applying transform *a* then transform *b*.

This version is an optimization that handles the case where both *a* and *b* are 2D affines.

Create a new composite transform that is the result of applying *Affine2DBase* *a* then *Affine2DBase* *b*.

You will generally not call this constructor directly but write `a + b` instead, which will automatically choose the best kind of composite transform instance to create.

**property depth**

Return the number of transforms which have been chained together to form this Transform instance.

**Note:** For the special case of a Composite transform, the maximum depth of the two is returned.

**get_matrix**()

Get the matrix for the affine part of this transform.

**class** matplotlib.transforms.**CompositeGenericTransform**(*a*, *b*, ***kwargs*)

Bases: *Transform*

A composite transform formed by applying transform *a* then transform *b*.

This "generic" version can handle any two arbitrary transformations.

Create a new composite transform that is the result of applying transform *a* then transform *b*.

You will generally not call this constructor directly but write `a + b` instead, which will automatically choose the best kind of composite transform instance to create.

**property depth**

Return the number of transforms which have been chained together to form this Transform instance.

---

**Note:** For the special case of a Composite transform, the maximum depth of the two is returned.

---

**frozen**()

Return a frozen copy of this transform node. The frozen copy will not be updated when its children change. Useful for storing a previously known state of a transform where `copy.deepcopy()` might normally be used.

**get_affine**()

Get the affine part of this transform.

**property has_inverse**

bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

**inverted**()

Return the corresponding inverse transformation.

It holds `x == self.inverted().transform(self.transform(x))`.

The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

**property is_affine**

bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

**property is_separable**

bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

**pass_through = True**

If pass_through is True, all ancestors will always be invalidated, even if 'self' is already invalid.

**transform_affine**(*values*)

Apply only the affine part of this transformation on the given array of values.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally a no-op. In affine transformations, this is equivalent to `transform(values)`.

**Parameters**

**values**

[array] The input values as an array of length `input_dims` or shape (N, `input_dims`).

**Returns**

**array**

The output values as an array of length `output_dims` or shape (N, `output_dims`), depending on the input.

**transform_non_affine**(*values*)

Apply only the non-affine part of this transformation.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

**Parameters**

**values**

[array] The input values as an array of length `input_dims` or shape (N, `input_dims`).

**Returns**

**array**

The output values as an array of length `output_dims` or shape (N, `output_dims`), depending on the input.

**transform_path_non_affine**(*path*)

Apply the non-affine part of this transform to *Path* *path*, returning a new *Path*.

`transform_path(path)` is equivalent to `transform_path_affine(transform_path_non_aff`

**class** matplotlib.transforms.**IdentityTransform**(**args*, ***kwargs*)

Bases: *Affine2DBase*

A special class that does one thing, the identity transform, in a fast way.

**Parameters**

**shorthand_name**

[str] A string representing the "name" of the transform. The name carries no significance other than to improve the readability of `str(transform)` when DEBUG=True.

**frozen**()

Return a frozen copy of this transform node. The frozen copy will not be updated when its children change. Useful for storing a previously known state of a transform where `copy.deepcopy()` might normally be used.

**get_affine**()

Get the affine part of this transform.

**get_matrix**()

Get the matrix for the affine part of this transform.

**inverted**()

Return the corresponding inverse transformation.

It holds `x == self.inverted().transform(self.transform(x))`.

The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

**transform**(*values*)

Apply this transformation on the given array of *values*.

**Parameters**

**values**

[array-like] The input values as an array of length `input_dims` or shape (N, `input_dims`).

**Returns**

**array**

The output values as an array of length `output_dims` or shape (N, `output_dims`), depending on the input.

**transform_affine**(*values*)

Apply only the affine part of this transformation on the given array of values.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally a no-op. In affine transformations, this is equivalent to `transform(values)`.

**Parameters**

**values**

[array] The input values as an array of length `input_dims` or shape (N, `input_dims`).

**Returns**

**array**

The output values as an array of length `output_dims` or shape (N, `output_dims`), depending on the input.

**transform_non_affine**(*values*)

Apply only the non-affine part of this transformation.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

**Parameters**

**values**

[array] The input values as an array of length `input_dims` or shape (N, `input_dims`).

**Returns**

**array**

The output values as an array of length `output_dims` or shape (N, `output_dims`), depending on the input.

**transform_path**(*path*)

Apply the transform to *Path* *path*, returning a new *Path*.

In some cases, this transform may insert curves into the path that began as line segments.

**transform_path_affine**(*path*)

Apply the affine part of this transform to *Path* *path*, returning a new *Path*.

`transform_path(path)` is equivalent to `transform_path_affine(transform_path_non_affi`

**transform_path_non_affine**(*path*)

Apply the non-affine part of this transform to *Path* *path*, returning a new *Path*.

`transform_path(path)` is equivalent to `transform_path_affine(transform_path_non_affi`

**class** matplotlib.transforms.**LockableBbox**(*bbox*, *x0=None*, *y0=None*, *x1=None*, *y1=None*, ***kwargs*)

Bases: *BboxBase*

A *Bbox* where some elements may be locked at certain values.

When the child bounding box changes, the bounds of this bbox will update accordingly with the exception of the locked elements.

**Parameters**

**bbox**

[*Bbox*] The child bounding box to wrap.

**x0**

[float or None] The locked value for x0, or None to leave unlocked.

**y0**

[float or None] The locked value for y0, or None to leave unlocked.

**x1**

[float or None] The locked value for x1, or None to leave unlocked.

**y1**

[float or None] The locked value for y1, or None to leave unlocked.

**get_points**()

**property locked_x0**

float or None: The value used for the locked x0.

**property locked_x1**

float or None: The value used for the locked x1.

**property locked_y0**

float or None: The value used for the locked y0.

**property locked_y1**

float or None: The value used for the locked y1.

**class** matplotlib.transforms.**ScaledTranslation**(*xt*, *yt*, *scale_trans*, ***kwargs*)

Bases: *Affine2DBase*

A transformation that translates by *xt* and *yt*, after *xt* and *yt* have been transformed by *scale_trans*.

**Parameters**

**shorthand_name**

[str] A string representing the "name" of the transform. The name carries no significance other than to improve the readability of `str(transform)` when DEBUG=True.

**get_matrix**()

Get the matrix for the affine part of this transform.

**class** matplotlib.transforms.**Transform**(*shorthand_name=None*)

Bases: *TransformNode*

The base class of all *TransformNode* instances that actually perform a transformation.

All non-affine transformations should be subclasses of this class. New affine transformations should be subclasses of *Affine2D*.

Subclasses of this class should override the following members (at minimum):

- *input_dims*
- *output_dims*
- *transform()*
- *inverted()* (if an inverse exists)

The following attributes may be overridden if the default is unsuitable:

- *is_separable* (defaults to True for 1D -> 1D transforms, False otherwise)
- *has_inverse* (defaults to True if *inverted()* is overridden, False otherwise)

If the transform needs to do something non-standard with *matplotlib.path.Path* objects, such as adding curves where there were once line segments, it should override:

- *transform_path()*

**Parameters**

**shorthand_name**

[str] A string representing the "name" of the transform. The name carries no significance other than to improve the readability of `str(transform)` when DEBUG=True.

**__add__**(*other*)

Compose two transforms together so that *self* is followed by *other*.

`A + B` returns a transform `C` so that `C.transform(x) == B.transform(A.transform(x))`.

**__sub__**(*other*)

Compose *self* with the inverse of *other*, cancelling identical terms if any:

```
# In general:
A - B == A + B.inverted()
# (but see note regarding frozen transforms below).

# If A "ends with" B (i.e. A == A' + B for some A') we can cancel
# out B:
(A' + B) - B == A'

# Likewise, if B "starts with" A (B = A + B'), we can cancel out A:
A - (A + B') == B'.inverted() == B'^-1
```

Cancellation (rather than naively returning `A + B.inverted()`) is important for multiple reasons:

- It avoids floating-point inaccuracies when computing the inverse of B: `B - B` is guaranteed to cancel out exactly (resulting in the identity transform), whereas `B + B.inverted()` may differ by a small epsilon.
- `B.inverted()` always returns a frozen transform: if one computes `A + B + B.inverted()` and later mutates `B`, then `B.inverted()` won't be updated and the last two terms won't cancel out anymore; on the other hand, `A + B - B` will always be equal to `A` even if `B` is mutated.

**contains_branch**(*other*)

Return whether the given transform is a sub-tree of this transform.

This routine uses transform equality to identify sub-trees, therefore in many situations it is object id which will be used.

For the case where the given transform represents the whole of this transform, returns True.

**contains_branch_seperately**(*other_transform*)

Return whether the given branch is a sub-tree of this transform on each separate dimension.

A common use for this method is to identify if a transform is a blended transform containing an Axes' data transform. e.g.:

```
x_isdata, y_isdata = trans.contains_branch_seperately(ax.transData)
```

**property depth**

Return the number of transforms which have been chained together to form this Transform instance.

**Note:** For the special case of a Composite transform, the maximum depth of the two is returned.

**get_affine**()

Get the affine part of this transform.

**get_matrix**()

Get the matrix for the affine part of this transform.

**has_inverse = False**

True if this transform has a corresponding inverse transform.

**input_dims = None**

The number of input dimensions of this transform. Must be overridden (with integers) in the subclass.

**inverted**()

Return the corresponding inverse transformation.

It holds `x == self.inverted().transform(self.transform(x))`.

The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

**is_separable = False**

True if this transform is separable in the x- and y- dimensions.

**output_dims = None**

The number of output dimensions of this transform. Must be overridden (with integers) in the subclass.

**transform**(*values*)

Apply this transformation on the given array of *values*.

**Parameters**

**values**

[array-like] The input values as an array of length *input_dims* or shape (N, *input_dims*).

**Returns**

**array**

The output values as an array of length *output_dims* or shape (N, *output_dims*), depending on the input.

**transform_affine**(*values*)

Apply only the affine part of this transformation on the given array of values.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally a no-op. In affine transformations, this is equivalent to `transform(values)`.

**Parameters**

**values**

[array] The input values as an array of length *input_dims* or shape (N, *input_dims*).

**Returns**

**array**

The output values as an array of length *output_dims* or shape (N, *output_dims*), depending on the input.

**transform_angles**(*angles*, *pts*, *radians=False*, *pushoff=1e-05*)

Transform a set of angles anchored at specific locations.

**Parameters**

**angles**

[(N,) array-like] The angles to transform.

**pts**

[(N, 2) array-like] The points where the angles are anchored.

**radians**

[bool, default: False] Whether *angles* are radians or degrees.

**pushoff**

[float] For each point in *pts* and angle in *angles*, the transformed angle is computed by transforming a segment of length *pushoff* starting at that point and making that angle relative to the horizontal axis, and measuring the angle between the horizontal axis and the transformed segment.

**Returns**

**(N,) array**

**transform_bbox**(*bbox*)

Transform the given bounding box.

For smarter transforms including caching (a common requirement in Matplotlib), see *TransformedBbox*.

**transform_non_affine**(*values*)

Apply only the non-affine part of this transformation.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

**Parameters**

**values**

[array] The input values as an array of length *input_dims* or shape (N, *input_dims*).

**Returns**

**array**

The output values as an array of length *output_dims* or shape (N, *output_dims*), depending on the input.

**transform_path**(*path*)

Apply the transform to *Path* *path*, returning a new *Path*.

In some cases, this transform may insert curves into the path that began as line segments.

**transform_path_affine**(*path*)

Apply the affine part of this transform to *Path* *path*, returning a new *Path*.

`transform_path(path)` is equivalent to `transform_path_affine(transform_path_non_affi`

**transform_path_non_affine**(*path*)

Apply the non-affine part of this transform to *Path* *path*, returning a new *Path*.

`transform_path(path)` is equivalent to `transform_path_affine(transform_path_non_affi`

**transform_point**(*point*)

Return a transformed point.

This function is only kept for backcompatibility; the more general *transform* method is capable of transforming both a list of points and a single point.

The point is given as a sequence of length *input_dims*. The transformed point is returned as a sequence of length *output_dims*.

**class** matplotlib.transforms.**TransformNode**(*shorthand_name=None*)

Bases: object

The base class for anything that participates in the transform tree and needs to invalidate its parents or be invalidated. This includes classes that are not really transforms, such as bounding boxes, since some transforms depend on bounding boxes to compute their values.

**Parameters**

**shorthand_name**

[str] A string representing the "name" of the transform. The name carries no significance other than to improve the readability of `str(transform)` when DEBUG=True.

**INVALID = 3**

**INVALID_AFFINE = 2**

**INVALID_NON_AFFINE = 1**

**frozen**()

Return a frozen copy of this transform node. The frozen copy will not be updated when its children change. Useful for storing a previously known state of a transform where `copy.deepcopy()` might normally be used.

**invalidate**()

Invalidate this *TransformNode* and triggers an invalidation of its ancestors. Should be called any time the transform changes.

**is_affine = False**

**is_bbox = False**

**pass_through = False**

If pass_through is True, all ancestors will always be invalidated, even if 'self' is already invalid.

**set_children**(**children*)

Set the children of the transform, to let the invalidation system know which transforms can invalidate this transform. Should be called from the constructor of any transforms that depend on other transforms.

**class** matplotlib.transforms.**TransformWrapper**(*child*)

Bases: *Transform*

A helper class that holds a single child transform and acts equivalently to it.

This is useful if a node of the transform tree must be replaced at run time with a transform of a different type. This class allows that replacement to correctly trigger invalidation.

*TransformWrapper* instances must have the same input and output dimensions during their entire lifetime, so the child transform may only be replaced with another child transform of the same dimensions.

*child*: A *Transform* instance. This child may later be replaced with *set()*.

**frozen**()

Return a frozen copy of this transform node. The frozen copy will not be updated when its children change. Useful for storing a previously known state of a transform where `copy.deepcopy()` might normally be used.

**property has_inverse**

bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

**property input_dims**

**property is_affine**

bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

**property is_separable**

bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

**property output_dims**

**pass_through = True**

If pass_through is True, all ancestors will always be invalidated, even if 'self' is already invalid.

**set**(*child*)

Replace the current child of this transform with another one.

The new child must have the same number of input and output dimensions as the current child.

**class** matplotlib.transforms.**TransformedBbox**(*bbox*, *transform*, ***kwargs*)

Bases: *BboxBase*

A *Bbox* that is automatically transformed by a given transform. When either the child bounding box or transform changes, the bounds of this bbox will update accordingly.

**Parameters**

**bbox**

[*Bbox*]

**transform**

[*Transform*]

**contains**(*x*, *y*)

Return whether `(x, y)` is in the bounding box or on its edge.

**fully_contains**(*x*, *y*)

Return whether `x, y` is in the bounding box, but not on its edge.

**get_points**()

**class** matplotlib.transforms.**TransformedPatchPath**(*patch*)

Bases: *TransformedPath*

A *TransformedPatchPath* caches a non-affine transformed copy of the *Patch*. This cached copy is automatically updated when the non-affine part of the transform or the patch changes.

**Parameters**

**patch**

[*Patch*]

**class** matplotlib.transforms.**TransformedPath**(*path*, *transform*)

Bases: *TransformNode*

A *TransformedPath* caches a non-affine transformed copy of the *Path*. This cached copy is automatically updated when the non-affine part of the transform changes.

---

**Note:** Paths are considered immutable by this class. Any update to the path's vertices/codes will not trigger a transform recomputation.

---

**Parameters**

**path**

[*Path*]

**transform**

[*Transform*]

**get_affine**()

**get_fully_transformed_path**()

Return a fully-transformed copy of the child path.

**get_transformed_path_and_affine**()

Return a copy of the child path, with the non-affine part of the transform already applied, along with the affine part of the path necessary to complete the transformation.

**get_transformed_points_and_affine**()

Return a copy of the child path, with the non-affine part of the transform already applied, along with the affine part of the path necessary to complete the transformation. Unlike *get_transformed_path_and_affine()*, no interpolation will be performed.

matplotlib.transforms.**blended_transform_factory**(*x_transform*, *y_transform*)

Create a new "blended" transform using *x_transform* to transform the *x*-axis and *y_transform* to transform the *y*-axis.

A faster version of the blended transform is returned for the case where both child transforms are affine.

matplotlib.transforms.**composite_transform_factory**(*a*, *b*)

Create a new composite transform that is the result of applying transform a then transform b.

Shortcut versions of the blended transform are provided for the case where both child transforms are affine, or one or the other is the identity transform.

Composite transforms may also be created using the '+' operator, e.g.:

```
c = a + b
```

matplotlib.transforms.**interval_contains**(*interval*, *val*)

Check, inclusively, whether an interval includes a given value.

**Parameters**

**interval**

[(float, float)] The endpoints of the interval.

**val**

[float] Value to check is within interval.

**Returns**

**bool**

Whether *val* is within the *interval*.

matplotlib.transforms.**interval_contains_open**(*interval*, *val*)

Check, excluding endpoints, whether an interval includes a given value.

**Parameters**

**interval**

[(float, float)] The endpoints of the interval.

**val**

[float] Value to check is within interval.

**Returns**

**bool**

Whether *val* is within the *interval*.

matplotlib.transforms.**nonsingular**(*vmin*, *vmax*, *expander=0.001*, *tiny=1e-15*, *increasing=True*)

Modify the endpoints of a range as needed to avoid singularities.

**Parameters**

**vmin, vmax**

[float] The initial endpoints.

**expander**

[float, default: 0.001] Fractional amount by which *vmin* and *vmax* are expanded if the original interval is too small, based on *tiny*.

**tiny**

[float, default: 1e-15] Threshold for the ratio of the interval to the maximum absolute value of its endpoints. If the interval is smaller than this, it will be expanded. This value should be around 1e-15 or larger; otherwise the interval will be approaching the double precision resolution limit.

**increasing**

[bool, default: True] If True, swap *vmin*, *vmax* if *vmin* > *vmax*.

**Returns**

**vmin, vmax**

[float] Endpoints, expanded and/or swapped if necessary. If either input is inf or NaN, or if both inputs are 0 or very close to zero, it returns -*expander*, *expander*.

matplotlib.transforms.**offset_copy**(*trans*, *fig=None*, *x=0.0*, *y=0.0*, *units='inches'*)

Return a new transform with an added offset.

**Parameters**

**trans**

[*Transform* subclass] Any transform, to which offset will be applied.

**fig**

[*Figure*, default: None] Current figure. It can be None if *units* are 'dots'.

**x, y**

[float, default: 0.0] The offset to apply.

**units**

[{'inches', 'points', 'dots'}, default: 'inches'] Units of the offset.

**Returns**

***Transform* subclass**

Transform with applied offset.

### 7.2.59 matplotlib.tri

Unstructured triangular grid functions.

**class** matplotlib.tri.**Triangulation**(*x*, *y*, *triangles=None*, *mask=None*)

An unstructured triangular grid consisting of npoints points and ntri triangles. The triangles can either be specified by the user or automatically generated using a Delaunay triangulation.

**Parameters**

**x, y**

[(npoints,) array-like] Coordinates of grid points.

**triangles**

[(ntri, 3) array-like of int, optional] For each triangle, the indices of the three points that make up the triangle, ordered in an anticlockwise manner. If not specified, the Delaunay triangulation is calculated.

**mask**

[(ntri,) array-like of bool, optional] Which triangles are masked out.

#### Notes

For a Triangulation to be valid it must not have duplicate points, triangles formed from colinear points, or overlapping triangles.

**Attributes**

**triangles**

[(ntri, 3) array of int] For each triangle, the indices of the three points that make up the triangle, ordered in an anticlockwise manner. If you want to take the *mask* into account, use *get_masked_triangles* instead.

**mask**

[(ntri, 3) array of bool or None] Masked out triangles.

**is_delaunay**

[bool] Whether the Triangulation is a calculated Delaunay triangulation (where *triangles* was not specified) or not.

**calculate_plane_coefficients**(*z*)

Calculate plane equation coefficients for all unmasked triangles from the point (x, y) coordinates and specified z-array of shape (npoints). The returned array has shape (npoints, 3) and allows z-value at (x, y) position in triangle tri to be calculated using `z = array[tri, 0] * x + array[tri, 1] * y + array[tri, 2]`.

**property edges**

Return integer array of shape (nedges, 2) containing all edges of non-masked triangles.

Each row defines an edge by its start point index and end point index. Each edge appears only once, i.e. for an edge between points *i* and *j*, there will only be either *(i, j)* or *(j, i)*.

**get_cpp_triangulation**()

Return the underlying C++ Triangulation object, creating it if necessary.

**static get_from_args_and_kwargs**(**args*, ***kwargs*)

Return a Triangulation object from the args and kwargs, and the remaining args and kwargs with the consumed values removed.

There are two alternatives: either the first argument is a Triangulation object, in which case it is returned, or the args and kwargs are sufficient to create a new Triangulation to return. In the latter case, see Triangulation.__init__ for the possible args and kwargs.

**get_masked_triangles**()

Return an array of triangles taking the mask into account.

**get_trifinder**()

Return the default *matplotlib.tri.TriFinder* of this triangulation, creating it if necessary. This allows the same TriFinder object to be easily shared.

**property neighbors**

Return integer array of shape (ntri, 3) containing neighbor triangles.

For each triangle, the indices of the three triangles that share the same edges, or -1 if there is no such neighboring triangle. `neighbors[i, j]` is the triangle that is the neighbor to the edge from point index `triangles[i, j]` to point index `triangles[i, (j+1)%3]`.

**set_mask**(*mask*)

Set or clear the mask array.

**Parameters**

**mask**

[None or bool array of length ntri]

**class** matplotlib.tri.**TriContourSet**(*ax*, **args*, ***kwargs*)

Bases: *ContourSet*

Create and store a set of contour lines or filled regions for a triangular grid.

This class is typically not instantiated directly by the user but by *tricontour* and *tricontourf*.

**Attributes**

**ax**

[*Axes*] The Axes object in which the contours are drawn.

**collections**

[*silent_list* of *PathCollection*s] [*Deprecated*]

**levels**

[array] The values of the contour levels.

**layers**

[array] Same as levels for line contours; half-way between levels for filled contours. See `ContourSet._process_colors`.

Draw triangular grid contour lines or filled regions, depending on whether keyword arg *filled* is False (default) or True.

The first argument of the initializer must be an *Axes* object. The remaining arguments and keyword arguments are described in the docstring of *tricontour*.

**set**(***, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *antialiased=<UNSET>*, *array=<UNSET>*, *capstyle=<UNSET>*, *clim=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *cmap=<UNSET>*, *color=<UNSET>*, *edgecolor=<UNSET>*, *facecolor=<UNSET>*, *gid=<UNSET>*, *hatch=<UNSET>*, *in_layout=<UNSET>*, *joinstyle=<UNSET>*, *label=<UNSET>*, *linestyle=<UNSET>*, *linewidth=<UNSET>*, *mouseover=<UNSET>*, *norm=<UNSET>*, *offset_transform=<UNSET>*, *offsets=<UNSET>*, *path_effects=<UNSET>*, *paths=<UNSET>*, *picker=<UNSET>*, *pickradius=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *urls=<UNSET>*, *visible=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi v |
| *alpha* | array-like or scalar or None |
| *animated* | bool |
| *antialiased* or aa or antialiaseds | bool or list of bools |
| *array* | array-like or None |

Table 161 – continued from

| Property | Description |
|---|---|
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clim* | (vmin: float, vmax: float) |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *cmap* | *Colormap* or str or None |
| *color* | *color* or list of RGBA tuples |
| *edgecolor* or ec or edgecolors | *color* or list of *color* or 'face' |
| *facecolor* or facecolors or fc | *color* or list of *color* |
| *figure* | *Figure* |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or dashes or linestyles or ls | str or tuple or list thereof |
| *linewidth* or linewidths or lw | float or list of floats |
| *mouseover* | bool |
| *norm* | *Normalize* or str or None |
| *offset_transform* or transOffset | *Transform* |
| *offsets* | (N, 2) or (2,) array-like |
| *path_effects* | list of *AbstractPathEffect* |
| *paths* | unknown |
| *picker* | None or bool or float or callable |
| *pickradius* | float |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *urls* | list of str or None |
| *visible* | bool |
| *zorder* | float |

**class** matplotlib.tri.**TriFinder**(*triangulation*)

Abstract base class for classes used to find the triangles of a Triangulation in which (x, y) points lie.

Rather than instantiate an object of a class derived from TriFinder, it is usually better to use the function *Triangulation.get_trifinder*.

Derived classes implement __call__(x, y) where x and y are array-like point coordinates of the same shape.

**class** matplotlib.tri.**TrapezoidMapTriFinder**(*triangulation*)

Bases: *TriFinder*

*TriFinder* class implemented using the trapezoid map algorithm from the book "Computational Geometry, Algorithms and Applications", second edition, by M. de Berg, M. van Kreveld, M. Overmars and O. Schwarzkopf.

The triangulation must be valid, i.e. it must not have duplicate points, triangles formed from colinear points, or overlapping triangles. The algorithm has some tolerance to triangles formed from colinear points, but this should not be relied upon.

**class** matplotlib.tri.**TriInterpolator**(*triangulation*, *z*, *trifinder=None*)

Abstract base class for classes used to interpolate on a triangular grid.

Derived classes implement the following methods:

- `__call__(x, y)`, where x, y are array-like point coordinates of the same shape, and that returns a masked array of the same shape containing the interpolated z-values.
- `gradient(x, y)`, where x, y are array-like point coordinates of the same shape, and that returns a list of 2 masked arrays of the same shape containing the 2 derivatives of the interpolator (derivatives of interpolated z values with respect to x and y).

**class** matplotlib.tri.**LinearTriInterpolator**(*triangulation*, *z*, *trifinder=None*)

Bases: *TriInterpolator*

Linear interpolator on a triangular grid.

Each triangle is represented by a plane so that an interpolated value at point (x, y) lies on the plane of the triangle containing (x, y). Interpolated values are therefore continuous across the triangulation, but their first derivatives are discontinuous at edges between triangles.

**Parameters**

**triangulation**

[*Triangulation*] The triangulation to interpolate over.

**z**

[(npoints,) array-like] Array of values, defined at grid points, to interpolate between.

**trifinder**

[*TriFinder*, optional] If this is not specified, the Triangulation's default TriFinder will be used by calling *Triangulation.get_trifinder*.

### Methods

| | |
|---|---|
| **`__call__` (x, y)** | (Returns interpolated values at (x, y) points.) |
| **`gradient` (x, y)** | (Returns interpolated derivatives at (x, y) points.) |

**gradient**(*x*, *y*)

Returns a list of 2 masked arrays containing interpolated derivatives at the specified (x, y) points.

**Parameters**

**x, y**

[array-like] x and y coordinates of the same shape and any number of dimensions.

**Returns**

**dzdx, dzdy**

[np.ma.array] 2 masked arrays of the same shape as *x* and *y*; values corresponding to (x, y) points outside of the triangulation are masked out. The first returned array contains the values of $\frac{\partial z}{\partial x}$ and the second those of $\frac{\partial z}{\partial y}$.

**class** matplotlib.tri.**CubicTriInterpolator**(*triangulation*, *z*, *kind='min_E'*, *trifinder=None*, *dz=None*)

Bases: *TriInterpolator*

Cubic interpolator on a triangular grid.

In one-dimension - on a segment - a cubic interpolating function is defined by the values of the function and its derivative at both ends. This is almost the same in 2D inside a triangle, except that the values of the function and its 2 derivatives have to be defined at each triangle node.

The CubicTriInterpolator takes the value of the function at each node - provided by the user - and internally computes the value of the derivatives, resulting in a smooth interpolation. (As a special feature, the user can also impose the value of the derivatives at each node, but this is not supposed to be the common usage.)

**Parameters**

**triangulation**

[*Triangulation*] The triangulation to interpolate over.

**z**

[(npoints,) array-like] Array of values, defined at grid points, to interpolate between.

**kind**

[{'min_E', 'geom', 'user'}, optional] Choice of the smoothing algorithm, in order to compute the interpolant derivatives (defaults to 'min_E'):

- if 'min_E': (default) The derivatives at each node is computed to minimize a bending energy.
- if 'geom': The derivatives at each node is computed as a weighted average of relevant triangle normals. To be used for speed optimization (large grids).
- if 'user': The user provides the argument *dz*, no computation is hence needed.

**trifinder**

[*TriFinder*, optional] If not specified, the Triangulation's default TriFinder will be used by calling *Triangulation.get_trifinder*.

**dz**

[tuple of array-likes (dzdx, dzdy), optional] Used only if *kind* ='user'. In this case *dz* must be provided as (dzdx, dzdy) where dzdx, dzdy are arrays of the same shape as *z* and are the interpolant first derivatives at the *triangulation* points.

## Notes

This note is a bit technical and details how the cubic interpolation is computed.

The interpolation is based on a Clough-Tocher subdivision scheme of the *triangulation* mesh (to make it clearer, each triangle of the grid will be divided in 3 child-triangles, and on each child triangle the interpolated function is a cubic polynomial of the 2 coordinates). This technique originates from FEM (Finite Element Method) analysis; the element used is a reduced Hsieh-Clough-Tocher (HCT) element. Its shape functions are described in [1]. The assembled function is guaranteed to be C1-smooth, i.e. it is continuous and its first derivatives are also continuous (this is easy to show inside the triangles but is also true when crossing the edges).

In the default case (*kind* ='min_E'), the interpolant minimizes a curvature energy on the functional space generated by the HCT element shape functions - with imposed values but arbitrary derivatives at each node. The minimized functional is the integral of the so-called total curvature (implementation based on an algorithm from [2] - PCG sparse solver):

$$E(z) = \frac{1}{2} \int_{\Omega} \left( \left( \frac{\partial^2 z}{\partial x^2} \right)^2 + \left( \frac{\partial^2 z}{\partial y^2} \right)^2 + 2 \left( \frac{\partial^2 z}{\partial y \partial x} \right)^2 \right) dx\, dy$$

If the case *kind* ='geom' is chosen by the user, a simple geometric approximation is used (weighted average of the triangle normal vectors), which could improve speed on very large grids.

## References

[1], [2]

### Methods

| | |
|---|---|
| **`__call__` (x, y)** | (Returns interpolated values at (x, y) points.) |
| **`gradient` (x, y)** | (Returns interpolated derivatives at (x, y) points.) |

**gradient**(*x*, *y*)

Returns a list of 2 masked arrays containing interpolated derivatives at the specified (x, y) points.

**Parameters**

**x, y**

[array-like] x and y coordinates of the same shape and any number of dimensions.

**Returns**

**dzdx, dzdy**

[np.ma.array] 2 masked arrays of the same shape as *x* and *y*; values corresponding to (x, y) points outside of the triangulation are masked out. The first returned array contains the values of $\frac{\partial z}{\partial x}$ and the second those of $\frac{\partial z}{\partial y}$.

**class** matplotlib.tri.**TriRefiner**(*triangulation*)

Abstract base class for classes implementing mesh refinement.

A TriRefiner encapsulates a Triangulation object and provides tools for mesh refinement and interpolation.

Derived classes must implement:

- `refine_triangulation(return_tri_index=False, **kwargs)` , where the optional keyword arguments *kwargs* are defined in each TriRefiner concrete implementation, and which returns:
  - a refined triangulation,
  - optionally (depending on *return_tri_index*), for each point of the refined triangulation: the index of the initial triangulation triangle to which it belongs.
- `refine_field(z, triinterpolator=None, **kwargs)`, where:
  - *z* array of field values (to refine) defined at the base triangulation nodes,
  - *triinterpolator* is an optional `TriInterpolator`,
  - the other optional keyword arguments *kwargs* are defined in each TriRefiner concrete implementation;

  and which returns (as a tuple) a refined triangular mesh and the interpolated values of the field at the refined triangulation nodes.

**class** matplotlib.tri.**UniformTriRefiner**(*triangulation*)

Bases: *TriRefiner*

Uniform mesh refinement by recursive subdivisions.

**Parameters**

**triangulation**

[*Triangulation*] The encapsulated triangulation (to be refined)

**refine_field**(*z*, *triinterpolator=None*, *subdiv=3*)

Refine a field defined on the encapsulated triangulation.

**Parameters**

**z**

[(npoints,) array-like] Values of the field to refine, defined at the nodes of the encapsulated triangulation. (`n_points` is the number of points in the initial triangulation)

**triinterpolator**

[*TriInterpolator*, optional] Interpolator used for field interpolation. If not specified, a *CubicTriInterpolator* will be used.

**subdiv**

[int, default: 3] Recursion level for the subdivision. Each triangle is divided into `4**subdiv` child triangles.

**Returns**

**refi_tri**

[*Triangulation*] The returned refined triangulation.

**refi_z**

[1D array of length: *refi_tri* node count.] The returned interpolated field (at *refi_tri* nodes).

**refine_triangulation**(*return_tri_index=False*, *subdiv=3*)

Compute a uniformly refined triangulation *refi_triangulation* of the encapsulated `triangulation`.

This function refines the encapsulated triangulation by splitting each father triangle into 4 child sub-triangles built on the edges midside nodes, recursing *subdiv* times. In the end, each triangle is hence divided into `4**subdiv` child triangles.

**Parameters**

**return_tri_index**

[bool, default: False] Whether an index table indicating the father triangle index of each point is returned.

**subdiv**

[int, default: 3] Recursion level for the subdivision. Each triangle is divided into `4**subdiv` child triangles; hence, the default results in 64 refined subtriangles for each triangle of the initial triangulation.

**Returns**

**refi_triangulation**

[*Triangulation*] The refined triangulation.

**found_index**

[int array] Index of the initial triangulation containing triangle, for each point of *refi_triangulation*. Returned only if *return_tri_index* is set to True.

**class** matplotlib.tri.**TriAnalyzer**(*triangulation*)

Define basic tools for triangular mesh analysis and improvement.

A TriAnalyzer encapsulates a *Triangulation* object and provides basic tools for mesh analysis and mesh improvement.

**Parameters**

**triangulation**

[*Triangulation*] The encapsulated triangulation to analyze.

**Attributes**

***scale_factors***

Factors to rescale the triangulation into a unit square.

**circle_ratios**(*rescale=True*)

Return a measure of the triangulation triangles flatness.

The ratio of the incircle radius over the circumcircle radius is a widely used indicator of a triangle flatness. It is always `<= 0.5` and `== 0.5` only for equilateral triangles. Circle ratios below 0.01 denote very flat triangles.

To avoid unduly low values due to a difference of scale between the 2 axis, the triangular mesh can first be rescaled to fit inside a unit square with *scale_factors* (Only if *rescale* is True, which is its default value).

**Parameters**

**rescale**

[bool, default: True] If True, internally rescale (based on *scale_factors*), so that the (unmasked) triangles fit exactly inside a unit square mesh.

**Returns**

**masked array**

Ratio of the incircle radius over the circumcircle radius, for each 'rescaled' triangle of the encapsulated triangulation. Values corresponding to masked triangles are masked out.

**get_flat_tri_mask**(*min_circle_ratio=0.01*, *rescale=True*)

Eliminate excessively flat border triangles from the triangulation.

Returns a mask *new_mask* which allows to clean the encapsulated triangulation from its border-located flat triangles (according to their *circle_ratios()*). This mask is meant to be subsequently applied to the triangulation using *Triangulation.set_mask*. *new_mask* is an extension of the initial triangulation mask in the sense that an initially masked triangle will remain masked.

The *new_mask* array is computed recursively; at each step flat triangles are removed only if they share a side with the current mesh border. Thus, no new holes in the triangulated domain will be created.

**Parameters**

**min_circle_ratio**

[float, default: 0.01] Border triangles with incircle/circumcircle radii ratio r/R will be removed if r/R < *min_circle_ratio*.

**rescale**

[bool, default: True] If True, first, internally rescale (based on *scale_factors*) so that the (unmasked) triangles fit exactly inside a unit square mesh. This rescaling accounts for the difference of scale which might exist between the 2 axis.

**Returns**

**array of bool**

Mask to apply to encapsulated triangulation. All the initially masked triangles remain masked in the *new_mask*.

#### Notes

The rationale behind this function is that a Delaunay triangulation - of an unstructured set of points - sometimes contains almost flat triangles at its border, leading to artifacts in plots (especially for high-resolution contouring). Masked with computed *new_mask*, the encapsulated triangulation would contain no more unmasked border triangles with a circle ratio below *min_circle_ratio*, thus improving the mesh quality for subsequent plots or interpolation.

**property scale_factors**

Factors to rescale the triangulation into a unit square.

**Returns**

**(float, float)**

Scaling factors (kx, ky) so that the triangulation `[triangulation.x * kx, triangulation.y * ky]` fits exactly inside a unit square.

### 7.2.60 matplotlib.type1font

**Attention:** This module is considered internal.

Its use is deprecated and it will be removed in a future version.

A class representing a Type 1 font.

This version reads pfa and pfb files and splits them for embedding in pdf files. It also supports SlantFont and ExtendFont transformations, similarly to pdfTeX and friends. There is no support yet for subsetting.

Usage:

```
font = Type1Font(filename)
clear_part, encrypted_part, finale = font.parts
slanted_font = font.transform({'slant': 0.167})
extended_font = font.transform({'extend': 1.2})
```

Sources:

- Adobe Technical Note #5040, Supporting Downloadable PostScript Language Fonts.
- Adobe Type 1 Font Format, Adobe Systems Incorporated, third printing, v1.1, 1993. ISBN 0-201-57044-0.

**class** matplotlib._type1font.**Type1Font**(*input*)

Bases: object

A class representing a Type-1 font, for use by backends.

**Attributes**

**parts**

[tuple] A 3-tuple of the cleartext part, the encrypted part, and the finale of zeros.

**decrypted**

[bytes] The decrypted form of `parts[1]`.

**prop**

[dict[str, Any]] A dictionary of font properties. Noteworthy keys include:

- FontName: PostScript name of the font
- Encoding: dict from numeric codes to glyph names
- FontMatrix: bytes object encoding a matrix
- UniqueID: optional font identifier, dropped when modifying the font
- CharStrings: dict from glyph names to byte code
- Subrs: array of byte code subroutines
- OtherSubrs: bytes object encoding some PostScript code

Initialize a Type-1 font.

**Parameters**

**input**

[str or 3-tuple] Either a pfb file name, or a 3-tuple of already-decoded Type-1 font *parts*.

**decrypted**

**parts**

**prop**

**transform**(*effects*)

Return a new font that is slanted and/or extended.

**Parameters**

**effects**

[dict] A dict with optional entries:

- **'slant'**

  [float, default: 0] Tangent of the angle that the font is to be slanted to the right. Negative values slant to the left.

- **'extend'**

  [float, default: 1] Scaling factor for the font width. Values less than 1 condense the glyphs.

**Returns**

*Type1Font*

### 7.2.61 matplotlib.typing

matplotlib.typing.**RGBColorType**
alias of Union[tuple[float, float, float], str]

matplotlib.typing.**RGBColourType**
alias of Union[tuple[float, float, float], str]

matplotlib.typing.**RGBAColorType**
alias of Union[str, tuple[float, float, float, float], tuple[Union[tuple[float, float, float], str], float], tuple[tuple[float, float, float, float], float]]

matplotlib.typing.**RGBAColourType**
alias of Union[str, tuple[float, float, float, float], tuple[Union[tuple[float, float, float], str], float], tuple[tuple[float, float, float, float], float]]

matplotlib.typing.**ColorType**
alias of Union[tuple[float, float, float], str, tuple[float, float, float, float], tuple[Union[tuple[float, float, float], str], float], tuple[tuple[float, float, float, float], float]]

matplotlib.typing.**ColourType**
alias of Union[tuple[float, float, float], str, tuple[float, float, float, float], tuple[Union[tuple[float, float, float], str], float], tuple[tuple[float, float, float, float], float]]

matplotlib.typing.**LineStyleType**
alias of Union[str, tuple[float, Sequence[float]]]

matplotlib.typing.**DrawStyleType**
alias of Literal['default', 'steps', 'steps-pre', 'steps-mid', 'steps-post']

matplotlib.typing.**MarkEveryType**
alias of Union[None, int, tuple[int, int], slice, list[int], float, tuple[float, float], list[bool]]

matplotlib.typing.**FillStyleType**
alias of Literal['full', 'left', 'right', 'bottom', 'top', 'none']

matplotlib.typing.**CapStyleType**
alias of Union[*CapStyle*, Literal['butt', 'projecting', 'round']]

matplotlib.typing.**JoinStyleType**

alias of Union[*JoinStyle*, Literal['miter', 'round', 'bevel']]

matplotlib.typing.**RcStyleType**

alias of Union[str, dict[str, Any], Path, Sequence[Union[str, Path, dict[str, Any]]]]

matplotlib.typing.**HashableList**(*iterable=()*, */*)

A nested list of Hashable values.

alias of list[Union[_HT, HashableList[_HT]]]

### 7.2.62 matplotlib.units

The classes here provide support for using custom classes with Matplotlib, e.g., those that do not expose the array interface but know how to convert themselves to arrays. It also supports classes with units and units conversion. Use cases include converters for custom objects, e.g., a list of datetime objects, as well as for objects that are unit aware. We don't assume any particular units implementation; rather a units implementation must register with the Registry converter dictionary and provide a *ConversionInterface*. For example, here is a complete implementation which supports plotting with native datetime objects:

```
import matplotlib.units as units
import matplotlib.dates as dates
import matplotlib.ticker as ticker
import datetime

class DateConverter(units.ConversionInterface):

    @staticmethod
    def convert(value, unit, axis):
        "Convert a datetime value to a scalar or array."
        return dates.date2num(value)

    @staticmethod
    def axisinfo(unit, axis):
        "Return major and minor tick locators and formatters."
        if unit != 'date':
            return None
        majloc = dates.AutoDateLocator()
        majfmt = dates.AutoDateFormatter(majloc)
        return units.AxisInfo(majloc=majloc, majfmt=majfmt, label='date')

    @staticmethod
    def default_units(x, axis):
        "Return the default unit for x or None."
        return 'date'

# Finally we register our object type with the Matplotlib units registry.
units.registry[datetime.date] = DateConverter()
```

**class** matplotlib.units.**AxisInfo**(*majloc=None*, *minloc=None*, *majfmt=None*, *minfmt=None*, *label=None*, *default_limits=None*)

Bases: object

Information to support default axis labeling, tick labeling, and limits.

An instance of this class must be returned by *ConversionInterface.axisinfo*.

**Parameters**

**majloc, minloc**

[Locator, optional] Tick locators for the major and minor ticks.

**majfmt, minfmt**

[Formatter, optional] Tick formatters for the major and minor ticks.

**label**

[str, optional] The default axis label.

**default_limits**

[optional] The default min and max limits of the axis if no data has been plotted.

### Notes

If any of the above are None, the axis will simply use the default value.

**exception** matplotlib.units.**ConversionError**

Bases: TypeError

**class** matplotlib.units.**ConversionInterface**

Bases: object

The minimal interface for a converter to take custom data types (or sequences) and convert them to values Matplotlib can use.

**static axisinfo**(*unit*, *axis*)

Return an *AxisInfo* for the axis with the specified units.

**static convert**(*obj*, *unit*, *axis*)

Convert *obj* using *unit* for the specified *axis*.

If *obj* is a sequence, return the converted sequence. The output must be a sequence of scalars that can be used by the numpy array layer.

**static default_units**(*x*, *axis*)

Return the default unit for *x* or None for the given axis.

**class** matplotlib.units.**DecimalConverter**

Bases: *ConversionInterface*

Converter for decimal.Decimal data to float.

**static convert**(*value*, *unit*, *axis*)

Convert Decimals to floats.

The *unit* and *axis* arguments are not used.

**Parameters**

**value**

[decimal.Decimal or iterable] Decimal or list of Decimal need to be converted

**class** matplotlib.units.**Registry**

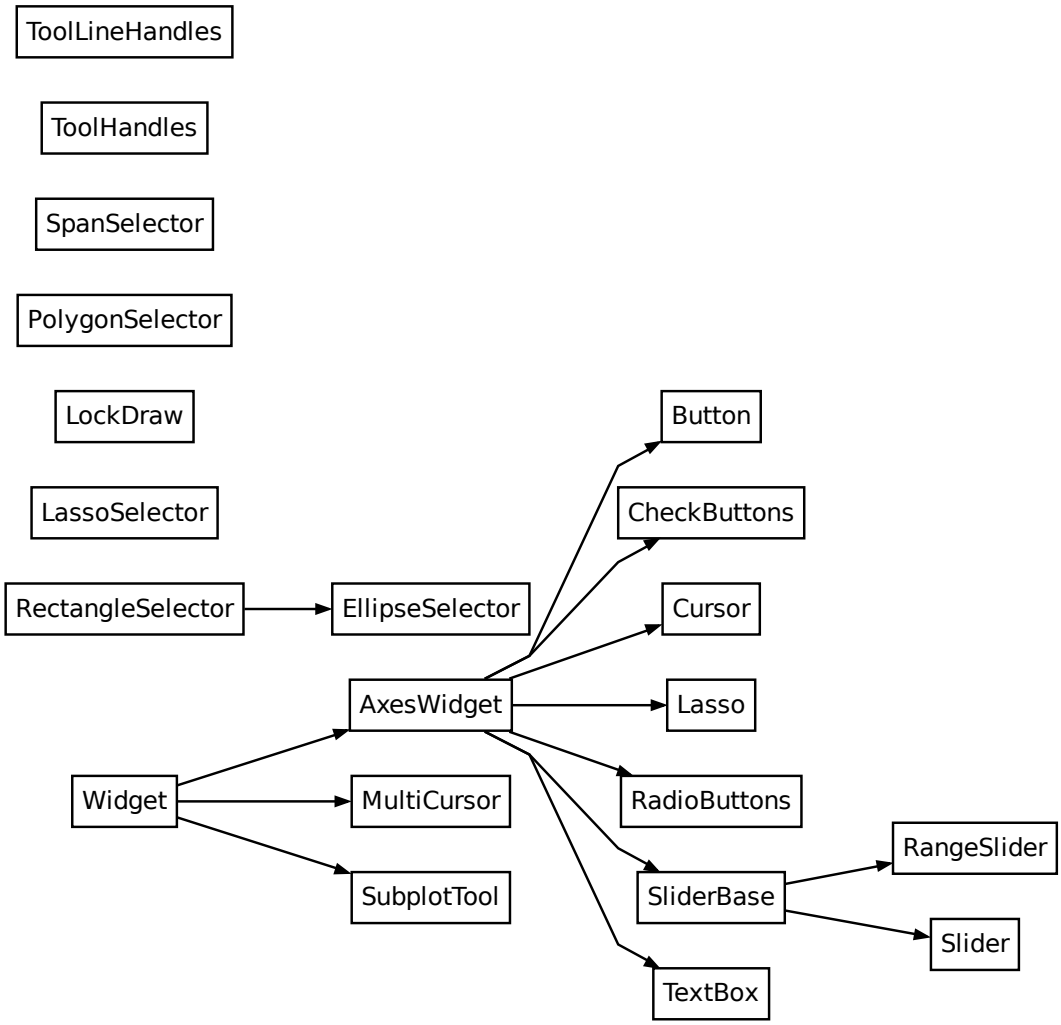
Bases: dict

Register types with conversion interface.

**get_converter**(*x*)

Get the converter interface instance for *x*, or None.

### 7.2.63 matplotlib.widgets




#### GUI neutral widgets

Widgets that are designed to work for any of the GUI backends. All of these widgets require you to predefine an *Axes* instance and pass that as the first parameter. Matplotlib doesn't try to be too smart with respect to layout -- you will have to figure out how wide and tall you want your Axes to be to accommodate your widget.

**class** matplotlib.widgets.**AxesWidget**(*ax*)

Bases: *Widget*

Widget connected to a single *Axes*.

To guarantee that the widget remains responsive and not garbage-collected, a reference to the object should be maintained by the user.

This is necessary because the callback registry maintains only weak-refs to the functions, which are member functions of the widget. If there are no references to the widget object it may be garbage collected which will disconnect the callbacks.

**Attributes**

**ax**

[*Axes*] The parent Axes for the widget.

**canvas**

[*FigureCanvasBase*] The parent figure canvas for the widget.

**active**

[bool] Is the widget active?

**connect_event**(*event*, *callback*)

Connect a callback function with an event.

This should be used in lieu of `figure.canvas.mpl_connect` since this function stores callback ids for later clean up.

**disconnect_events**()

Disconnect all events created by this widget.

**class** matplotlib.widgets.**Button**(*ax*, *label*, *image=None*, *color='0.85'*, *hovercolor='0.95'*, ***, *useblit=True*)

Bases: *AxesWidget*

A GUI neutral button.

For the button to remain responsive you must keep a reference to it. Call *on_clicked* to connect to the button.

**Attributes**

**ax**

The *Axes* the button renders into.

**label**

A *Text* instance.

**color**

The color of the button when not hovering.

**hovercolor**

The color of the button when hovering.

**Parameters**

**ax**

[*Axes*] The *Axes* instance the button will be placed into.

**label**

[str] The button text.

**image**

[array-like or PIL Image] The image to place in the button, if not *None*. The parameter is directly forwarded to *imshow*.

**color**

[*color*] The color of the button when not activated.

**hovercolor**

[*color*] The color of the button when the mouse is over it.

**useblit**

[bool, default: True] Use blitting for faster drawing if supported by the backend. See the tutorial *Faster rendering by using blitting* for details.

New in version 3.7.

**disconnect**(*cid*)

Remove the callback function with connection id *cid*.

**on_clicked**(*func*)

Connect the callback function *func* to button click events.

Returns a connection id, which can be used to disconnect the callback.

**class** matplotlib.widgets.**CheckButtons**(*ax*, *labels*, *actives=None*, ***, *useblit=True*, *label_props=None*, *frame_props=None*, *check_props=None*)

Bases: *AxesWidget*

A GUI neutral set of check buttons.

For the check buttons to remain responsive you must keep a reference to this object.

Connect to the CheckButtons with the *on_clicked* method.

**Attributes**

**ax**

[*Axes*] The parent Axes for the widget.

**labels**

[list of *Text*]

**rectangles**

[list of *Rectangle*]

**lines**

[list of (*Line2D*, *Line2D*) pairs] List of lines for the x's in the checkboxes. These lines exist for each box, but have `set_visible(False)` when its box is not checked.

Add check buttons to *Axes* instance *ax*.

**Parameters**

**ax**

[*Axes*] The parent Axes for the widget.

**labels**

[list of str] The labels of the check buttons.

**actives**

[list of bool, optional] The initial check states of the buttons. The list must have the same length as *labels*. If not given, all buttons are unchecked.

**useblit**

[bool, default: True] Use blitting for faster drawing if supported by the backend. See the tutorial *Faster rendering by using blitting* for details.

New in version 3.7.

**label_props**

[dict, optional] Dictionary of *Text* properties to be used for the labels.

New in version 3.7.

**frame_props**

[dict, optional] Dictionary of scatter *Collection* properties to be used for the check button frame. Defaults (label font size / 2)**2 size, black edgecolor, no facecolor, and 1.0 linewidth.

New in version 3.7.

**check_props**

[dict, optional] Dictionary of scatter *Collection* properties to be used for the check button check. Defaults to (label font size / 2)**2 size, black color, and 1.0 linewidth.

New in version 3.7.

**clear**()

Uncheck all checkboxes.

**disconnect**(*cid*)

Remove the observer with connection id *cid*.

**get_checked_labels**()

Return a list of labels currently checked by user.

**get_status**()

Return a list of the status (True/False) of all of the check buttons.

**on_clicked**(*func*)

Connect the callback function *func* to button click events.

**Parameters**

**func**

[callable] When the button is clicked, call *func* with button label. When all buttons are cleared, call *func* with None. The callback func must have the signature:

```
def func(label: str | None) -> Any
```

Return values may exist, but are ignored.

**Returns**

**A connection id, which can be used to disconnect the callback.**

**set_active**(*index*, *state=None*)

Modify the state of a check button by index.

Callbacks will be triggered if `eventson` is True.

**Parameters**

**index**

[int] Index of the check button to toggle.

**state**

[bool, optional] If a boolean value, set the state explicitly. If no value is provided, the state is toggled.

**Raises**

**ValueError**

If *index* is invalid.

**TypeError**

If *state* is not boolean.

**set_check_props**(*props*)

Set properties of the check button checks.

New in version 3.7.

**Parameters**

**props**

[dict] Dictionary of *Collection* properties to be used for the check button check.

**set_frame_props**(*props*)

Set properties of the check button frames.

New in version 3.7.

**Parameters**

**props**

[dict] Dictionary of *Collection* properties to be used for the check button frames.

**set_label_props**(*props*)

Set properties of the *Text* labels.

New in version 3.7.

**Parameters**

**props**

[dict] Dictionary of *Text* properties to be used for the labels.

**class** matplotlib.widgets.**Cursor**(*ax*, *, *horizOn=True*, *vertOn=True*, *useblit=False*, ***lineprops*)

Bases: *AxesWidget*

A crosshair cursor that spans the Axes and moves with mouse cursor.

For the cursor to remain responsive you must keep a reference to it.

**Parameters**

**ax**

[*Axes*] The *Axes* to attach the cursor to.

**horizOn**

[bool, default: True] Whether to draw the horizontal line.

**vertOn**

[bool, default: True] Whether to draw the vertical line.

**useblit**

[bool, default: False] Use blitting for faster drawing if supported by the backend. See the tutorial *Faster rendering by using blitting* for details.

**Other Parameters**

****lineprops**

*Line2D* properties that control the appearance of the lines. See also *axhline*.

## Examples

See *Cursor*.

**clear**(*event*)

Internal event handler to clear the cursor.

**onmove**(*event*)

Internal event handler to draw the cursor when the mouse moves.

**class** matplotlib.widgets.**EllipseSelector**(*ax*, *onselect*, ***, *minspanx=0*, *minspany=0*, *useblit=False*, *props=None*, *spancoords='data'*, *button=None*, *grab_range=10*, *handle_props=None*, *interactive=False*, *state_modifier_keys=None*, *drag_from_anywhere=False*, *ignore_event_outside=False*, *use_data_coordinates=False*)

Bases: *RectangleSelector*

Select an elliptical region of an Axes.

For the cursor to remain responsive you must keep a reference to it.

Press and release events triggered at the same coordinates outside the selection will clear the selector, except when `ignore_event_outside=True`.

**Parameters**

**ax**

[*Axes*] The parent Axes for the widget.

**onselect**

[function] A callback function that is called after a release event and the selection is created, changed or removed. It must have the signature:

```
def onselect(eclick: MouseEvent, erelease: MouseEvent)
```

where *eclick* and *erelease* are the mouse click and release *MouseEvent*s that start and complete the selection.

**minspanx**

[float, default: 0] Selections with an x-span less than or equal to *minspanx* are removed (when already existing) or cancelled.

**minspany**

[float, default: 0] Selections with an y-span less than or equal to *minspanx* are removed (when already existing) or cancelled.

**useblit**

[bool, default: False] Whether to use blitting for faster drawing (if supported by the backend). See the tutorial *Faster rendering by using blitting* for details.

**props**

[dict, optional] Properties with which the ellipse is drawn. See *Patch* for valid properties. Default:

```
dict(facecolor='red', edgecolor='black', alpha=0.2,
fill=True)
```

**spancoords**

[{"data", "pixels"}, default: "data"] Whether to interpret *minspanx* and *minspany* in data or in pixel coordinates.

**button**

[*MouseButton*, list of *MouseButton*, default: all buttons] Button(s) that trigger rectangle selection.

**grab_range**

[float, default: 10] Distance in pixels within which the interactive tool handles can be activated.

**handle_props**

[dict, optional] Properties with which the interactive handles (marker artists) are drawn. See the marker arguments in *Line2D* for valid properties. Default values are defined in `mpl.rcParams` except for the default value of `markeredgecolor` which will be the same as the `edgecolor` property in *props*.

**interactive**

[bool, default: False] Whether to draw a set of handles that allow interaction with the widget after it is drawn.

**state_modifier_keys**

[dict, optional] Keyboard modifiers which affect the widget's behavior. Values amend the defaults, which are:

- "move": Move the existing shape, default: no modifier.
- "clear": Clear the current shape, default: "escape".
- "square": Make the shape square, default: "shift".
- "center": change the shape around its center, default: "ctrl".
- "rotate": Rotate the shape around its center between -45° and 45°, default: "r".

"square" and "center" can be combined. The square shape can be defined in data or display coordinates as determined by the `use_data_coordinates` argument specified when creating the selector.

**drag_from_anywhere**

[bool, default: False] If `True`, the widget can be moved by clicking anywhere within its bounds.

**ignore_event_outside**

[bool, default: False] If `True`, the event triggered outside the span selector will be ignored.

**use_data_coordinates**

[bool, default: False] If `True`, the "square" shape of the selector is defined in data coordinates instead of display coordinates.

## Examples

*Rectangle and ellipse selectors*

**class** matplotlib.widgets.**Lasso**(*ax*, *xy*, *callback*, *, *useblit=True*, *props=None*)

Bases: *AxesWidget*

Selection curve of an arbitrary shape.

The selected path can be used in conjunction with *contains_point* to select data points from an image.

Unlike *LassoSelector*, this must be initialized with a starting point *xy*, and the *Lasso* events are destroyed upon release.

**Parameters**

**ax**

[*Axes*] The parent Axes for the widget.

**xy**

[(float, float)] Coordinates of the start of the lasso.

**callback**

[callable] Whenever the lasso is released, the *callback* function is called and passed the vertices of the selected path.

**useblit**

[bool, default: True] Whether to use blitting for faster drawing (if supported by the backend). See the tutorial *Faster rendering by using blitting* for details.

**props: dict, optional**

Lasso line properties. See *Line2D* for valid properties. Default *props* are:

```
{'linestyle' : '-', 'color' : 'black', 'lw' : 2}
```

New in version 3.9.

**onmove**(*event*)

**onrelease**(*event*)

**class** matplotlib.widgets.**LassoSelector**(*ax*, *onselect*, *, *useblit=True*, *props=None*, *button=None*)

Bases: _SelectorWidget

Selection curve of an arbitrary shape.

For the selector to remain responsive you must keep a reference to it.

The selected path can be used in conjunction with *contains_point* to select data points from an image.

In contrast to *Lasso*, *LassoSelector* is written with an interface similar to *RectangleSelector* and *SpanSelector*, and will continue to interact with the Axes until disconnected.

Example usage:

```
ax = plt.subplot()
ax.plot(x, y)

def onselect(verts):
    print(verts)
lasso = LassoSelector(ax, onselect)
```

**Parameters**

**ax**

[*Axes*] The parent Axes for the widget.

**onselect**

[function] Whenever the lasso is released, the *onselect* function is called and passed the vertices of the selected path.

**useblit**

[bool, default: True] Whether to use blitting for faster drawing (if supported by the backend). See the tutorial *Faster rendering by using blitting* for details.

**props**

[dict, optional] Properties with which the line is drawn, see *Line2D* for valid properties. Default values are defined in `mpl.rcParams`.

**button**

[*MouseButton* or list of *MouseButton*, optional] The mouse buttons used for rectangle selection. Default is `None`, which corresponds to all buttons.

**class** matplotlib.widgets.**LockDraw**

Bases: `object`

Some widgets, like the cursor, draw onto the canvas, and this is not desirable under all circumstances, like when the toolbar is in zoom-to-rect mode and drawing a rectangle. To avoid this, a widget can acquire a canvas' lock with `canvas.widgetlock(widget)` before drawing on the canvas; this will prevent other widgets from doing so at the same time (if they also try to acquire the lock first).

**available**(*o*)

Return whether drawing is available to *o*.

**isowner**(*o*)

Return whether *o* owns this lock.

**locked**()

Return whether the lock is currently held by an owner.

**release**(*o*)

Release the lock from *o*.

**class** matplotlib.widgets.**MultiCursor**(*canvas*, *axes*, *, *useblit=True*, *horizOn=False*, *vertOn=True*, ***lineprops*)

Bases: *Widget*

Provide a vertical (default) and/or horizontal line cursor shared between multiple Axes.

For the cursor to remain responsive you must keep a reference to it.

**Parameters**

**canvas**

[object] This parameter is entirely unused and only kept for back-compatibility.

**axes**

[list of *Axes*] The *Axes* to attach the cursor to.

**useblit**

[bool, default: True] Use blitting for faster drawing if supported by the backend. See the tutorial *Faster rendering by using blitting* for details.

**horizOn**

[bool, default: False] Whether to draw the horizontal line.

**vertOn**

[bool, default: True] Whether to draw the vertical line.

**Other Parameters**

****lineprops**

*Line2D* properties that control the appearance of the lines. See also *axhline*.

### Examples

See *Multicursor*.

**clear**(*event*)

Clear the cursor.

**connect**()

Connect events.

**disconnect**()

Disconnect events.

**onmove**(*event*)

**class** matplotlib.widgets.**PolygonSelector**(*ax*, *onselect*, ***, *useblit=False*, *props=None*, *handle_props=None*, *grab_range=10*, *draw_bounding_box=False*, *box_handle_props=None*, *box_props=None*)

Bases: _SelectorWidget

Select a polygon region of an Axes.

Place vertices with each mouse click, and make the selection by completing the polygon (clicking on the first vertex). Once drawn individual vertices can be moved by clicking and dragging with the left mouse button, or removed by clicking the right mouse button.

In addition, the following modifier keys can be used:

- Hold *ctrl* and click and drag a vertex to reposition it before the polygon has been completed.
- Hold the *shift* key and click and drag anywhere in the Axes to move all vertices.
- Press the *esc* key to start a new polygon.

For the selector to remain responsive you must keep a reference to it.

**Parameters**

**ax**

[*Axes*] The parent Axes for the widget.

**onselect**

[function] When a polygon is completed or modified after completion, the *onselect* function is called and passed a list of the vertices as `(xdata, ydata)` tuples.

**useblit**

[bool, default: False] Whether to use blitting for faster drawing (if supported by the backend). See the tutorial *Faster rendering by using blitting* for details.

**props**

[dict, optional] Properties with which the line is drawn, see *Line2D* for valid properties. Default:

```
dict(color='k', linestyle='-', linewidth=2, alpha=0.5)
```

**handle_props**

[dict, optional] Artist properties for the markers drawn at the vertices of the polygon. See the marker arguments in *Line2D* for valid properties. Default values are defined in `mpl.rcParams` except for the default value of `markeredgecolor` which will be the same as the `color` property in *props*.

**grab_range**

[float, default: 10] A vertex is selected (to complete the polygon or to move a vertex) if the mouse click is within *grab_range* pixels of the vertex.

**draw_bounding_box**

[bool, optional] If `True`, a bounding box will be drawn around the polygon selector once it is complete. This box can be used to move and resize the selector.

**box_handle_props**

[dict, optional] Properties to set for the box handles. See the documentation for the *handle_props* argument to *RectangleSelector* for more info.

**box_props**

[dict, optional] Properties to set for the box. See the documentation for the *props* argument to *RectangleSelector* for more info.

### Notes

If only one point remains after removing points, the selector reverts to an incomplete state and you can start drawing a new polygon from the existing point.

### Examples

*Polygon Selector Select indices from a collection using polygon selector*

**onmove**(*event*)

Cursor move event handler and validator.

**property verts**

The polygon vertices, as a list of `(x, y)` pairs.

**class** matplotlib.widgets.**RadioButtons**(*ax*, *labels*, *active=0*, *activecolor=None*, ***, *useblit=True*, *label_props=None*, *radio_props=None*)

Bases: *AxesWidget*

A GUI neutral radio button.

For the buttons to remain responsive you must keep a reference to this object.

Connect to the RadioButtons with the *on_clicked* method.

**Attributes**

**ax**

[*Axes*] The parent Axes for the widget.

**activecolor**

[*color*] The color of the selected button.

**labels**

[list of *Text*] The button labels.

**circles**

[list of *Circle*] The buttons.

**value_selected**

[str] The label text of the currently selected button.

**index_selected**

[int] The index of the selected button.

Add radio buttons to an *Axes*.

**Parameters**

**ax**

[*Axes*] The Axes to add the buttons to.

**labels**

[list of str] The button labels.

**active**

[int] The index of the initially selected button.

**activecolor**

[*color*] The color of the selected button. The default is `'blue'` if not specified here or in *radio_props*.

**useblit**

[bool, default: True] Use blitting for faster drawing if supported by the backend. See the tutorial *Faster rendering by using blitting* for details.

New in version 3.7.

**label_props**

[dict or list of dict, optional] Dictionary of *Text* properties to be used for the labels.

New in version 3.7.

**radio_props**

[dict, optional] Dictionary of scatter *Collection* properties to be used for the radio buttons. Defaults to (label font size / 2)**2 size, black edgecolor, and *activecolor* facecolor (when active).

---

**Note:** If a facecolor is supplied in *radio_props*, it will override *activecolor*. This may be used to provide an active color per button.

---

New in version 3.7.

**property activecolor**

**clear**()

Reset the active button to the initially active one.

**disconnect**(*cid*)

Remove the observer with connection id *cid*.

**on_clicked**(*func*)

Connect the callback function *func* to button click events.

**Parameters**

**func**

[callable] When the button is clicked, call *func* with button label. When all buttons are cleared, call *func* with None. The callback func must have the signature:

```
def func(label: str | None) -> Any
```

Return values may exist, but are ignored.

**Returns**

**A connection id, which can be used to disconnect the callback.**

**set_active**(*index*)

Select button with number *index*.

Callbacks will be triggered if `eventson` is True.

**Parameters**

**index**

[int] The index of the button to activate.

**Raises**

**ValueError**

If the index is invalid.

**set_label_props**(*props*)

Set properties of the *Text* labels.

New in version 3.7.

**Parameters**

**props**

[dict] Dictionary of *Text* properties to be used for the labels.

**set_radio_props**(*props*)

Set properties of the *Text* labels.

New in version 3.7.

**Parameters**

**props**

[dict] Dictionary of *Collection* properties to be used for the radio buttons.

**class** matplotlib.widgets.**RangeSlider**(*ax*, *label*, *valmin*, *valmax*, *, *valinit=None*, *valfmt=None*, *closedmin=True*, *closedmax=True*, *dragging=True*, *valstep=None*, *orientation='horizontal'*, *track_color='lightgrey'*, *handle_style=None*, ***kwargs*)

Bases: *SliderBase*

A slider representing a range of floating point values. Defines the min and max of the range via the *val* attribute as a tuple of (min, max).

Create a slider that defines a range contained within [*valmin*, *valmax*] in Axes *ax*. For the slider to remain responsive you must maintain a reference to it. Call *on_changed()* to connect to the slider event.

**Attributes**

**val**

[tuple of float] Slider value.

**Parameters**

**ax**

[Axes] The Axes to put the slider in.

**label**

[str] Slider label.

**valmin**

[float] The minimum value of the slider.

**valmax**

[float] The maximum value of the slider.

**valinit**

[tuple of float or None, default: None] The initial positions of the slider. If None the initial positions will be at the 25th and 75th percentiles of the range.

**valfmt**

[str, default: None] %-format string used to format the slider values. If None, a *ScalarFormatter* is used instead.

**closedmin**

[bool, default: True] Whether the slider interval is closed on the bottom.

**closedmax**

[bool, default: True] Whether the slider interval is closed on the top.

**dragging**

[bool, default: True] If True the slider can be dragged by the mouse.

**valstep**

[float, default: None] If given, the slider will snap to multiples of *valstep*.

**orientation**

[{'horizontal', 'vertical'}, default: 'horizontal'] The orientation of the slider.

**track_color**

[*color*, default: 'lightgrey'] The color of the background track. The track is accessible for further styling via the *track* attribute.

**handle_style**

[dict] Properties of the slider handles. Default values are

| Key | Value | Default | Description |
|---|---|---|---|
| facecolor | color | 'white' | The facecolor of the slider handles. |
| edgecolor | color | '.75' | The edgecolor of the slider handles. |
| size | int | 10 | The size of the slider handles in points. |

Other values will be transformed as marker{foo} and passed to the *Line2D* constructor. e.g. `handle_style = {'style'='x'}` will result in `marker-style = 'x'`.

## Notes

Additional kwargs are passed on to `self.poly` which is the *Polygon* that draws the slider knob. See the *Polygon* documentation for valid property names (`facecolor`, `edgecolor`, `alpha`, etc.).

**on_changed**(*func*)

Connect *func* as callback function to changes of the slider value.

**Parameters**

**func**

[callable] Function to call when slider is changed. The function must accept a 2-tuple of floats as its argument.

**Returns**

**int**

Connection id (which can be used to disconnect *func*).

**set_max**(*max*)

Set the lower value of the slider to *max*.

**Parameters**

**max**

[float]

**set_min**(*min*)

Set the lower value of the slider to *min*.

**Parameters**

**min**

[float]

**set_val**(*val*)

Set slider value to *val*.

**Parameters**

**val**

[tuple or array-like of float]

**class** matplotlib.widgets.**RectangleSelector**(*ax*, *onselect*, *, *minspanx=0*, *minspany=0*, *useblit=False*, *props=None*, *spancoords='data'*, *button=None*, *grab_range=10*, *handle_props=None*, *interactive=False*, *state_modifier_keys=None*, *drag_from_anywhere=False*, *ignore_event_outside=False*, *use_data_coordinates=False*)

Bases: _SelectorWidget

Select a rectangular region of an Axes.

For the cursor to remain responsive you must keep a reference to it.

Press and release events triggered at the same coordinates outside the selection will clear the selector, except when ignore_event_outside=True.

**Parameters**

**ax**

[*Axes*] The parent Axes for the widget.

**onselect**

[function] A callback function that is called after a release event and the selection is created, changed or removed. It must have the signature:

```
def onselect(eclick: MouseEvent, erelease: MouseEvent)
```

where *eclick* and *erelease* are the mouse click and release *MouseEvent*s that start and complete the selection.

**minspanx**

[float, default: 0] Selections with an x-span less than or equal to *minspanx* are removed (when already existing) or cancelled.

**minspany**

[float, default: 0] Selections with an y-span less than or equal to *minspanx* are removed (when already existing) or cancelled.

**useblit**

[bool, default: False] Whether to use blitting for faster drawing (if supported by the backend). See the tutorial *Faster rendering by using blitting* for details.

**props**

[dict, optional] Properties with which the rectangle is drawn. See *Patch* for valid properties. Default:

```
dict(facecolor='red', edgecolor='black', alpha=0.2,
fill=True)
```

**spancoords**

[{"data", "pixels"}, default: "data"] Whether to interpret *minspanx* and *minspany* in data or in pixel coordinates.

**button**

[*MouseButton*, list of *MouseButton*, default: all buttons] Button(s) that trigger rectangle selection.

**grab_range**

[float, default: 10] Distance in pixels within which the interactive tool handles can be activated.

**handle_props**

[dict, optional] Properties with which the interactive handles (marker artists) are drawn. See the marker arguments in *Line2D* for valid properties. Default values are defined in `mpl.rcParams` except for the default value of `markeredgecolor` which will be the same as the `edgecolor` property in *props*.

**interactive**

[bool, default: False] Whether to draw a set of handles that allow interaction with the widget after it is drawn.

**state_modifier_keys**

[dict, optional] Keyboard modifiers which affect the widget's behavior. Values amend the defaults, which are:

- "move": Move the existing shape, default: no modifier.
- "clear": Clear the current shape, default: "escape".
- "square": Make the shape square, default: "shift".
- "center": change the shape around its center, default: "ctrl".
- "rotate": Rotate the shape around its center between -45° and 45°, default: "r".

"square" and "center" can be combined. The square shape can be defined in data or display coordinates as determined by the `use_data_coordinates` argument specified when creating the selector.

**drag_from_anywhere**

[bool, default: False] If `True`, the widget can be moved by clicking anywhere within its bounds.

**ignore_event_outside**

[bool, default: False] If `True`, the event triggered outside the span selector will be ignored.

**use_data_coordinates**

[bool, default: False] If `True`, the "square" shape of the selector is defined in data coordinates instead of display coordinates.

## Examples

```
>>> import matplotlib.pyplot as plt
>>> import matplotlib.widgets as mwidgets
>>> fig, ax = plt.subplots()
>>> ax.plot([1, 2, 3], [10, 50, 100])
>>> def onselect(eclick, erelease):
...     print(eclick.xdata, eclick.ydata)
...     print(erelease.xdata, erelease.ydata)
>>> props = dict(facecolor='blue', alpha=0.5)
>>> rect = mwidgets.RectangleSelector(ax, onselect, interactive=True,
...                                   props=props)
>>> fig.show()
>>> rect.add_state('square')
```

See also: *Rectangle and ellipse selectors*

**property center**

Center of rectangle in data coordinates.

**property corners**

Corners of rectangle in data coordinates from lower left, moving clockwise.

**property edge_centers**

Midpoint of rectangle edges in data coordinates from left, moving anti-clockwise.

**property extents**

Return (xmin, xmax, ymin, ymax) in data coordinates as defined by the bounding box before rotation.

**property geometry**

Return an array of shape (2, 5) containing the x (`RectangleSelector.geometry[1, :]`) and y (`RectangleSelector.geometry[0, :]`) data coordinates of the four corners of the rectangle starting and ending in the top left corner.

**property rotation**

Rotation in degree in interval [-45°, 45°]. The rotation is limited in range to keep the implementation simple.

**class** matplotlib.widgets.**Slider**(*ax*, *label*, *valmin*, *valmax*, *, *valinit=0.5*, *valfmt=None*, *closedmin=True*, *closedmax=True*, *slidermin=None*, *slidermax=None*, *dragging=True*, *valstep=None*, *orientation='horizontal'*, *initcolor='r'*, *track_color='lightgrey'*, *handle_style=None*, ***kwargs*)

Bases: *SliderBase*

A slider representing a floating point range.

Create a slider from *valmin* to *valmax* in Axes *ax*. For the slider to remain responsive you must maintain a reference to it. Call *on_changed()* to connect to the slider event.

**Attributes**

**val**

[float] Slider value.

**Parameters**

**ax**

[Axes] The Axes to put the slider in.

**label**

[str] Slider label.

**valmin**

[float] The minimum value of the slider.

**valmax**

[float] The maximum value of the slider.

**valinit**

[float, default: 0.5] The slider initial position.

**valfmt**

[str, default: None] %-format string used to format the slider value. If None, a *ScalarFormatter* is used instead.

**closedmin**

[bool, default: True] Whether the slider interval is closed on the bottom.

**closedmax**

[bool, default: True] Whether the slider interval is closed on the top.

**slidermin**

[Slider, default: None] Do not allow the current slider to have a value less than the value of the Slider *slidermin*.

**slidermax**

[Slider, default: None] Do not allow the current slider to have a value greater than the value of the Slider *slidermax*.

**dragging**

[bool, default: True] If True the slider can be dragged by the mouse.

**valstep**

[float or array-like, default: None] If a float, the slider will snap to multiples of *valstep*. If an array the slider will snap to the values in the array.

**orientation**

[{'horizontal', 'vertical'}, default: 'horizontal'] The orientation of the slider.

**initcolor**

[*color*, default: 'r'] The color of the line at the *valinit* position. Set to `'none'` for no line.

**track_color**

[*color*, default: 'lightgrey'] The color of the background track. The track is accessible for further styling via the *track* attribute.

**handle_style**

[dict] Properties of the slider handle. Default values are

| Key | Value | Default | Description |
|---|---|---|---|
| facecolor | color | 'white' | The facecolor of the slider handle. |
| edgecolor | color | '.75' | The edgecolor of the slider handle. |
| size | int | 10 | The size of the slider handle in points. |

Other values will be transformed as marker{foo} and passed to the *Line2D* constructor. e.g. `handle_style = {'style'='x'}` will result in `marker-style = 'x'`.

### Notes

Additional kwargs are passed on to `self.poly` which is the *Rectangle* that draws the slider knob. See the *Rectangle* documentation for valid property names (`facecolor`, `edgecolor`, `alpha`, etc.).

**on_changed**(*func*)

Connect *func* as callback function to changes of the slider value.

**Parameters**

**func**

[callable] Function to call when slider is changed. The function must accept a single float as its arguments.

**Returns**

**int**

Connection id (which can be used to disconnect *func*).

**set_val**(*val*)

Set slider value to *val*.

**Parameters**

**val**

[float]

**class** matplotlib.widgets.**SliderBase**(*ax*, *orientation*, *closedmin*, *closedmax*, *valmin*, *valmax*, *valfmt*, *dragging*, *valstep*)

Bases: *AxesWidget*

The base class for constructing Slider widgets. Not intended for direct usage.

For the slider to remain responsive you must maintain a reference to it.

**disconnect**(*cid*)

Remove the observer with connection id *cid*.

**Parameters**

**cid**

[int] Connection id of the observer to be removed.

**reset**()

Reset the slider to the initial value.

**class** matplotlib.widgets.**SpanSelector**(*ax*, *onselect*, *direction*, *, *minspan=0*, *useblit=False*, *props=None*, *onmove_callback=None*, *interactive=False*, *button=None*, *handle_props=None*, *grab_range=10*, *state_modifier_keys=None*, *drag_from_anywhere=False*, *ignore_event_outside=False*, *snap_values=None*)

Bases: _SelectorWidget

Visually select a min/max range on a single axis and call a function with those values.

To guarantee that the selector remains responsive, keep a reference to it.

In order to turn off the SpanSelector, set `span_selector.active` to False. To turn it back on, set it to True.

Press and release events triggered at the same coordinates outside the selection will clear the selector, except when `ignore_event_outside=True`.

**Parameters**

**ax**

[*Axes*]

**onselect**

[callable with signature `func(min: float, max: float)`] A callback function that is called after a release event and the selection is created, changed or removed.

**direction**

[{"horizontal", "vertical"}] The direction along which to draw the span selector.

**minspan**

[float, default: 0] If selection is less than or equal to *minspan*, the selection is removed (when already existing) or cancelled.

**useblit**

[bool, default: False] If True, use the backend-dependent blitting features for faster canvas updates. See the tutorial *Faster rendering by using blitting* for details.

**props**

[dict, default: {'facecolor': 'red', 'alpha': 0.5}] Dictionary of `Patch` properties.

**onmove_callback**

[callable with signature `func(min: float, max: float)`, optional] Called on mouse move while the span is being selected.

**interactive**

[bool, default: False] Whether to draw a set of handles that allow interaction with the widget after it is drawn.

**button**

[`MouseButton` or list of `MouseButton`, default: all buttons] The mouse buttons which activate the span selector.

**handle_props**

[dict, default: None] Properties of the handle lines at the edges of the span. Only used when *interactive* is True. See `Line2D` for valid properties.

**grab_range**

[float, default: 10] Distance in pixels within which the interactive tool handles can be activated.

**state_modifier_keys**

[dict, optional] Keyboard modifiers which affect the widget's behavior. Values amend the defaults, which are:

- "clear": Clear the current shape, default: "escape".

**drag_from_anywhere**

[bool, default: False] If `True`, the widget can be moved by clicking anywhere within its bounds.

**ignore_event_outside**

[bool, default: False] If `True`, the event triggered outside the span selector will be ignored.

**snap_values**

[1D array-like, optional] Snap the selector edges to the given values.

### Examples

```
>>> import matplotlib.pyplot as plt
>>> import matplotlib.widgets as mwidgets
>>> fig, ax = plt.subplots()
>>> ax.plot([1, 2, 3], [10, 50, 100])
>>> def onselect(vmin, vmax):
...     print(vmin, vmax)
>>> span = mwidgets.SpanSelector(ax, onselect, 'horizontal',
...                              props=dict(facecolor='blue', alpha=0.5))
>>> fig.show()
```

See also: *Span Selector*

**connect_default_events**()

Connect the major canvas events to methods.

**property direction**

Direction of the span selector: 'vertical' or 'horizontal'.

**property extents**

**(float, float)**

The values, in data coordinates, for the start and end points of the current selection. If there is no selection then the start and end values will be the same.

**new_axes**(*ax*, *, *_props=None*)

Set SpanSelector to operate on a new Axes.

**class** matplotlib.widgets.**SubplotTool**(*targetfig*, *toolfig*)

Bases: *Widget*

A tool to adjust the subplot params of a *Figure*.

**Parameters**

**targetfig**

[*Figure*] The figure instance to adjust.

**toolfig**

[*Figure*] The figure instance to embed the subplot tool into.

**class** matplotlib.widgets.**TextBox**(*ax*, *label*, *initial=''*, *, *color='.95'*, *hovercolor='1'*, *label_pad=0.01*, *textalignment='left'*)

Bases: *AxesWidget*

A GUI neutral text input box.

For the text box to remain responsive you must keep a reference to it.

Call *on_text_change* to be updated whenever the text changes.

Call *on_submit* to be updated whenever the user hits enter or leaves the text entry field.

**Attributes**

**ax**

[*Axes*] The parent Axes for the widget.

**label**

[*Text*]

**color**

[*color*] The color of the text box when not hovering.

**hovercolor**

[*color*] The color of the text box when hovering.

**Parameters**

**ax**

[*Axes*] The *Axes* instance the button will be placed into.

**label**

[str] Label for this text box.

**initial**

[str] Initial value in the text box.

**color**

[*color*] The color of the box.

**hovercolor**

[*color*] The color of the box when the mouse is over it.

**label_pad**

[float] The distance between the label and the right side of the textbox.

**textalignment**

[{'left', 'center', 'right'}] The horizontal location of the text.

**begin_typing**()

**disconnect**(*cid*)

Remove the observer with connection id *cid*.

**on_submit**(*func*)

When the user hits enter or leaves the submission box, call this *func* with event.

A connection id is returned which can be used to disconnect.

**on_text_change**(*func*)

When the text changes, call this *func* with event.

A connection id is returned which can be used to disconnect.

**set_val**(*val*)

**stop_typing**()

**property text**

**class** matplotlib.widgets.**ToolHandles**(*ax*, *x*, *y*, *, *marker='o'*, *marker_props=None*, *useblit=True*)

Bases: object

Control handles for canvas tools.

**Parameters**

**ax**

[*Axes*] Matplotlib Axes where tool handles are displayed.

**x, y**

[1D arrays] Coordinates of control handles.

**marker**

[str, default: 'o'] Shape of marker used to display handle. See *plot*.

**marker_props**

[dict, optional] Additional marker properties. See *Line2D*.

**useblit**

[bool, default: True] Whether to use blitting for faster drawing (if supported by the backend). See the tutorial *Faster rendering by using blitting* for details.

**property artists**

**closest**(*x*, *y*)

Return index and pixel distance to closest index.

**set_animated**(*val*)

**set_data**(*pts*, *y=None*)

Set x and y positions of handles.

**set_visible**(*val*)

**property x**

**property y**

**class** matplotlib.widgets.**ToolLineHandles**(*ax*, *positions*, *direction*, *, *line_props=None*, *useblit=True*)

Bases: object

Control handles for canvas tools.

**Parameters**

**ax**

[*Axes*] Matplotlib Axes where tool handles are displayed.

**positions**

[1D array] Positions of handles in data coordinates.

**direction**

[{"horizontal", "vertical"}] Direction of handles, either 'vertical' or 'horizontal'

**line_props**

[dict, optional] Additional line properties. See *Line2D*.

**useblit**

[bool, default: True] Whether to use blitting for faster drawing (if supported by the backend). See the tutorial *Faster rendering by using blitting* for details.

**property artists**

**closest**(*x*, *y*)

Return index and pixel distance to closest handle.

**Parameters**

**x, y**

[float] x, y position from which the distance will be calculated to determinate the closest handle

**Returns**

**index, distance**

[index of the handle and its distance from] position x, y

**property direction**

Direction of the handle: 'vertical' or 'horizontal'.

**property positions**

Positions of the handle in data coordinates.

**remove**()

Remove the handles artist from the figure.

**set_animated**(*value*)

Set the animated state of the handles artist.

**set_data**(*positions*)

Set x- or y-positions of handles, depending on if the lines are vertical or horizontal.

**Parameters**

**positions**

[tuple of length 2] Set the positions of the handle in data coordinates

**set_visible**(*value*)

Set the visibility state of the handles artist.

**class** matplotlib.widgets.**Widget**

Bases: object

Abstract base class for GUI neutral widgets.

**property active**

Is the widget active?

**drawon = True**

**eventson = True**

**get_active**()

Get whether the widget is active.

**ignore**(*event*)

Return whether *event* should be ignored.

This method should be called at the beginning of any event callback.

**set_active**(*active*)

Set whether the widget is active.

### 7.2.64 matplotlib._api

Helper functions for managing the Matplotlib API.

This documentation is only relevant for Matplotlib developers, not for users.

**Warning:** This module and its submodules are for internal use only. Do not use them in your own code. We may change the API at any time with no warning.

matplotlib._api.**caching_module_getattr**(*cls*)

Helper decorator for implementing module-level __getattr__ as a class.

This decorator must be used at the module toplevel as follows:

```
@caching_module_getattr
class __getattr__:  # The class *must* be named ``__getattr__``.
    @property  # Only properties are taken into account.
    def name(self): ...
```

The __getattr__ class will be replaced by a __getattr__ function such that trying to access name on the module will resolve the corresponding property (which may be decorated e.g. with _api.deprecated for deprecating module globals). The properties are all implicitly cached. Moreover, a suitable AttributeError is generated and raised if no property with the given name exists.

matplotlib._api.**check_getitem**(*mapping*, */*, ***kwargs*)

*kwargs* must consist of a single *key, value* pair. If *key* is in *mapping*, return mapping[value]; else, raise an appropriate ValueError.

### Examples

```
>>> _api.check_getitem({"foo": "bar"}, arg=arg)
```

matplotlib._api.**check_in_list**(*values*, */*, *, *_print_supported_values=True*, ***kwargs*)

For each *key, value* pair in *kwargs*, check that *value* is in *values*; if not, raise an appropriate ValueError.

**Parameters**

**values**

[iterable] Sequence of values to check on.

**_print_supported_values**

[bool, default: True] Whether to print *values* when raising ValueError.

****kwargs**

[dict] *key, value* pairs as keyword arguments to find in *values*.

**Raises**

**ValueError**

If any *value* in *kwargs* is not found in *values*.

### Examples

```
>>> _api.check_in_list(["foo", "bar"], arg=arg, other_arg=other_arg)
```

matplotlib._api.**check_isinstance**(*types*, */*, ***kwargs*)

For each *key, value* pair in *kwargs*, check that *value* is an instance of one of *types*; if not, raise an appropriate TypeError.

As a special case, a `None` entry in *types* is treated as NoneType.

### Examples

```
>>> _api.check_isinstance((SomeClass, None), arg=arg)
```

matplotlib._api.**check_shape**(*shape*, */*, ***kwargs*)

For each *key, value* pair in *kwargs*, check that *value* has the shape *shape*; if not, raise an appropriate ValueError.

*None* in the shape is treated as a "free" size that can have any length. e.g. (None, 2) -> (N, 2)

The values checked must be numpy arrays.

### Examples

To check for (N, 2) shaped arrays

```
>>> _api.check_shape((None, 2), arg=arg, other_arg=other_arg)
```

**class** matplotlib._api.**classproperty**(*fget*, *fset=None*, *fdel=None*, *doc=None*)

Bases: `object`

Like `property`, but also triggers on access via the class, and it is the *class* that's passed as argument.

### Examples

```
class C:
    @classproperty
    def foo(cls):
        return cls.__name__

assert C.foo == "C"
```

**property fget**

matplotlib._api.**define_aliases**(*alias_d*, *cls=None*)

Class decorator for defining property aliases.

Use as

```
@_api.define_aliases({"property": ["alias", ...], ...})
class C: ...
```

For each property, if the corresponding `get_property` is defined in the class so far, an alias named `get_alias` will be defined; the same will be done for setters. If neither the getter nor the setter exists, an exception will be raised.

The alias map is stored as the `_alias_map` attribute on the class and can be used by *normalize_kwargs* (which assumes that higher priority aliases come last).

matplotlib._api.**kwarg_error**(*name*, *kw*)

Generate a TypeError to be raised by function calls with wrong kwarg.

**Parameters**

**name**

[str] The name of the calling function.

**kw**

[str or Iterable[str]] Either the invalid keyword argument name, or an iterable yielding invalid keyword arguments (e.g., a `kwargs` dict).

matplotlib._api.**nargs_error**(*name*, *takes*, *given*)

Generate a TypeError to be raised by function calls with wrong arity.

matplotlib._api.**recursive_subclasses**(*cls*)

Yield *cls* and direct and indirect subclasses of *cls*.

matplotlib._api.**select_matching_signature**(*funcs*, **args*, ***kwargs*)

Select and call the function that accepts `*args, **kwargs`.

*funcs* is a list of functions which should not raise any exception (other than `TypeError` if the arguments passed do not match their signature).

*select_matching_signature* tries to call each of the functions in *funcs* with `*args, **kwargs` (in the order in which they are given). Calls that fail with a `TypeError` are silently skipped. As soon as a call succeeds, *select_matching_signature* returns its return value. If no function accepts `*args, **kwargs`, then the `TypeError` raised by the last failing call is re-raised.

Callers should normally make sure that any `*args, **kwargs` can only bind a single *func* (to avoid any ambiguity), although this is not checked by *select_matching_signature*.

### Notes

*select_matching_signature* is intended to help implementing signature-overloaded functions. In general, such functions should be avoided, except for back-compatibility concerns. A typical use pattern is

```
def my_func(*args, **kwargs):
    params = select_matching_signature(
        [lambda old1, old2: locals(), lambda new: locals()],
        *args, **kwargs)
    if "old1" in params:
        warn_deprecated(...)
        old1, old2 = params.values()  # note that locals() is ordered.
    else:
        new, = params.values()
    # do things with params
```

which allows *my_func* to be called either with two parameters (*old1* and *old2*) or a single one (*new*). Note that the new signature is given last, so that callers get a `TypeError` corresponding to the new signature if the arguments they passed in do not match any signature.

`matplotlib._api.`**`warn_external`**(*message*, *category=None*)

`warnings.warn` wrapper that sets *stacklevel* to "outside Matplotlib".

The original emitter of the warning can be obtained by patching this function back to `warnings.warn`, i.e. `_api.warn_external = warnings.warn` (or `functools.partial(warnings.warn, stacklevel=2)`, etc.).

Helper functions for deprecating parts of the Matplotlib API.

This documentation is only relevant for Matplotlib developers, not for users.

**Warning:** This module is for internal use only. Do not use it in your own code. We may change the API at any time with no warning.

**exception** `matplotlib._api.deprecation.`**`MatplotlibDeprecationWarning`**

Bases: `DeprecationWarning`

A class for issuing deprecation warnings for Matplotlib users.

`matplotlib._api.deprecation.`**`delete_parameter`**(*since*, *name*, *func=None*, ***kwargs*)

Decorator indicating that parameter *name* of *func* is being deprecated.

The actual implementation of *func* should keep the *name* parameter in its signature, or accept a `**kwargs` argument (through which *name* would be passed).

Parameters that come after the deprecated parameter effectively become keyword-only (as they cannot be passed positionally without triggering the DeprecationWarning on the deprecated parameter), and should be marked as such after the deprecation period has passed and the deprecated parameter is removed.

Parameters other than *since*, *name*, and *func* are keyword-only and forwarded to *warn_deprecated*.

### Examples

```
@_api.delete_parameter("3.1", "unused")
def func(used_arg, other_arg, unused, more_args): ...
```

matplotlib._api.deprecation.**deprecate_method_override**(*method*, *obj*, *, *allow_empty=False*, ***kwargs*)

Return `obj.method` with a deprecation if it was overridden, else None.

**Parameters**

**method**

An unbound method, i.e. an expression of the form `Class.method_name`. Remember that within the body of a method, one can always use `__class__` to refer to the class that is currently being defined.

**obj**

Either an object of the class where *method* is defined, or a subclass of that class.

**allow_empty**

[bool, default: False] Whether to allow overrides by "empty" methods without emitting a warning.

****kwargs**

Additional parameters passed to *warn_deprecated* to generate the deprecation warning; must at least include the "since" key.

**class** matplotlib._api.deprecation.**deprecate_privatize_attribute**(**args*, ***kwargs*)

Bases: `object`

Helper to deprecate public access to an attribute (or method).

This helper should only be used at class scope, as follows:

```
class Foo:
    attr = _deprecate_privatize_attribute(*args, **kwargs)
```

where *all* parameters are forwarded to *deprecated*. This form makes `attr` a property which forwards read and write access to `self._attr` (same name but with a leading underscore), with a deprecation warning. Note that the attribute name is derived from *the name this helper is assigned to*. This helper also works for deprecating methods.

matplotlib._api.deprecation.**deprecated**(*since*, *, *message=''*, *name=''*, *alternative=''*, *pending=False*, *obj_type=None*, *addendum=''*, *removal=''*)

Decorator to mark a function, a class, or a property as deprecated.

When deprecating a classmethod, a staticmethod, or a property, the `@deprecated` decorator should go *under* `@classmethod` and `@staticmethod` (i.e., *deprecated* should directly decorate the underlying callable), but *over* `@property`.

When deprecating a class `C` intended to be used as a base class in a multiple inheritance hierarchy, `C` *must* define an `__init__` method (if `C` instead inherited its `__init__` from its own base class, then `@deprecated` would mess up `__init__` inheritance when installing its own (deprecation-emitting) `C.__init__`).

Parameters are the same as for *warn_deprecated*, except that *obj_type* defaults to 'class' if decorating a class, 'attribute' if decorating a property, and 'function' otherwise.

### Examples

```
@deprecated('1.4.0')
def the_function_to_deprecate():
    pass
```

matplotlib._api.deprecation.**make_keyword_only**(*since*, *name*, *func=None*)

Decorator indicating that passing parameter *name* (or any of the following ones) positionally to *func* is being deprecated.

When used on a method that has a pyplot wrapper, this should be the outermost decorator, so that `boilerplate.py` can access the original signature.

matplotlib._api.deprecation.**rename_parameter**(*since*, *old*, *new*, *func=None*)

Decorator indicating that parameter *old* of *func* is renamed to *new*.

The actual implementation of *func* should use *new*, not *old*. If *old* is passed to *func*, a DeprecationWarning is emitted, and its value is used, even if *new* is also passed by keyword (this is to simplify pyplot wrapper functions, which always pass *new* explicitly to the Axes method). If *new* is also passed but positionally, a TypeError will be raised by the underlying function during argument binding.

### Examples

```
@_api.rename_parameter("3.1", "bad_name", "good_name")
def func(good_name): ...
```

matplotlib._api.deprecation.**suppress_matplotlib_deprecation_warning**()

matplotlib._api.deprecation.**warn_deprecated**(*since*, *, *message=''*, *name=''*, *alternative=''*, *pending=False*, *obj_type=''*, *addendum=''*, *removal=''*)

Display a standardized deprecation.

**Parameters**

**since**

[str] The release at which this API became deprecated.

**message**

[str, optional] Override the default deprecation message. The `%(since)s`, `%(name)s`, `%(alternative)s`, `%(obj_type)s`, `%(addendum)s`, and `%(removal)s` format specifiers will be replaced by the values of the respective arguments passed to this function.

**name**

[str, optional] The name of the deprecated object.

**alternative**

[str, optional] An alternative API that the user may use in place of the deprecated API. The deprecation warning will tell the user about this alternative if provided.

**pending**

[bool, optional] If True, uses a PendingDeprecationWarning instead of a DeprecationWarning. Cannot be used together with *removal*.

**obj_type**

[str, optional] The object type being deprecated.

**addendum**

[str, optional] Additional text appended directly to the final message.

**removal**

[str, optional] The expected removal version. With the default (an empty string), a removal version is automatically computed from *since*. Set to other Falsy values to not schedule a removal date. Cannot be used together with *pending*.

## Examples

```
# To warn of the deprecation of "matplotlib.name_of_module"
warn_deprecated('1.4.0', name='matplotlib.name_of_module',
                obj_type='module')
```

### 7.2.65 matplotlib._enums

Enums representing sets of strings that Matplotlib uses as input parameters.

Matplotlib often uses simple data types like strings or tuples to define a concept; e.g. the line capstyle can be specified as one of 'butt', 'round', or 'projecting'. The classes in this module are used internally and serve to document these concepts formally.

As an end-user you will not use these classes directly, but only the values they define.

**class** matplotlib._enums.**JoinStyle**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Define how the connection between two line segments is drawn.

For a visual impression of each *JoinStyle*, *view these docs online*, or run *JoinStyle.demo*.

Lines in Matplotlib are typically defined by a 1D *Path* and a finite `linewidth`, where the underlying 1D *Path* represents the center of the stroked line.

By default, *GraphicsContextBase* defines the boundaries of a stroked line to simply be every point within some radius, `linewidth/2`, away from any point of the center line. However, this results in corners appearing "rounded", which may not be the desired behavior if you are drawing, for example, a polygon or pointed star.

**Supported values:**

**'miter'**

the "arrow-tip" style. Each boundary of the filled-in area will extend in a straight line parallel to the tangent vector of the centerline at the point it meets the corner, until they meet in a sharp point.

**'round'**

stokes every point within a radius of `linewidth/2` of the center lines.

**'bevel'**

the "squared-off" style. It can be thought of as a rounded corner where the "circular" part of the corner has been cut off.

---

**Note:** Very long miter tips are cut off (to form a *bevel*) after a backend-dependent limit called the "miter limit", which specifies the maximum allowed ratio of miter length to line width. For example, the PDF backend uses the default value of 10 specified by the PDF standard, while the SVG backend does not even specify the miter limit, resulting in a default value of 4 per the SVG specification. Matplotlib does not currently allow the user to adjust this parameter.

A more detailed description of the effect of a miter limit can be found in the Mozilla Developer Docs

---

**static demo**()

Demonstrate how each JoinStyle looks for various join angles.

**class** matplotlib._enums.**CapStyle**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Define how the two endpoints (caps) of an unclosed line are drawn.

How to draw the start and end points of lines that represent a closed curve (i.e. that end in a *CLOSEPOLY*) is controlled by the line's *JoinStyle*. For all other lines, how the start and end points are drawn is controlled by the *CapStyle*.

For a visual impression of each *CapStyle*, *view these docs online* or run *CapStyle.demo*.

By default, *GraphicsContextBase* draws a stroked line as squared off at its endpoints.

**Supported values:**

**'butt'**

the line is squared off at its endpoint.

**'projecting'**

the line is squared off as in *butt*, but the filled in area extends beyond the endpoint a distance of `linewidth/2`.

**'round'**

like *butt*, but a semicircular cap is added to the end of the line, of radius `linewidth/2`.

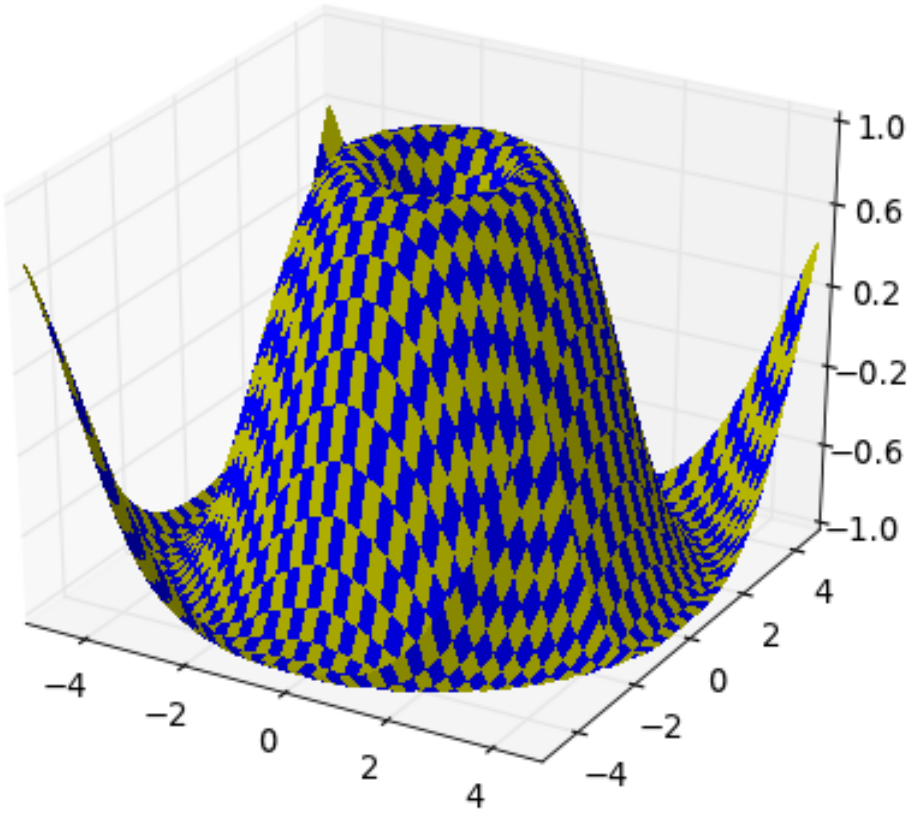
**static demo**()

Demonstrate how each CapStyle looks for a thick line segment.

### 7.2.66 mpl_toolkits.mplot3d

The mplot3d toolkit adds simple 3D plotting capabilities (scatter, surface, line, mesh, etc.) to Matplotlib by supplying an Axes object that can create a 2D projection of a 3D scene. The resulting graph will have the same look and feel as regular 2D plots. Not the fastest or most feature complete 3D library out there, but it ships with Matplotlib and thus may be a lighter weight solution for some use cases.

See the *mplot3d tutorial* for more information.




The interactive backends also provide the ability to rotate and zoom the 3D scene. One can rotate the 3D scene by simply clicking-and-dragging the scene. Panning is done by clicking the middle mouse button, and zooming is done by right-clicking the scene and dragging the mouse up and down. Unlike 2D plots, the toolbar pan and zoom buttons are not used.

### mplot3d FAQ

#### How is mplot3d different from Mayavi?

Mayavi is a very powerful and featureful 3D graphing library. For advanced 3D scenes and excellent rendering capabilities, it is highly recommended to use Mayavi.

mplot3d was intended to allow users to create simple 3D graphs with the same "look-and-feel" as matplotlib's 2D plots. Furthermore, users can use the same toolkit that they are already familiar with to generate both their 2D and 3D plots.

### My 3D plot doesn't look right at certain viewing angles

This is probably the most commonly reported issue with mplot3d. The problem is that -- from some viewing angles -- a 3D object would appear in front of another object, even though it is physically behind it. This can result in plots that do not look "physically correct."

Unfortunately, while some work is being done to reduce the occurrence of this artifact, it is currently an intractable problem, and cannot be fully solved until matplotlib supports 3D graphics rendering at its core.

The problem occurs due to the reduction of 3D data down to 2D + z-order scalar. A single value represents the 3rd dimension for all parts of 3D objects in a collection. Therefore, when the bounding boxes of two collections intersect, it becomes possible for this artifact to occur. Furthermore, the intersection of two 3D objects (such as polygons or patches) cannot be rendered properly in matplotlib's 2D rendering engine.

This problem will likely not be solved until OpenGL support is added to all of the backends (patches are greatly welcomed). Until then, if you need complex 3D scenes, we recommend using MayaVi.

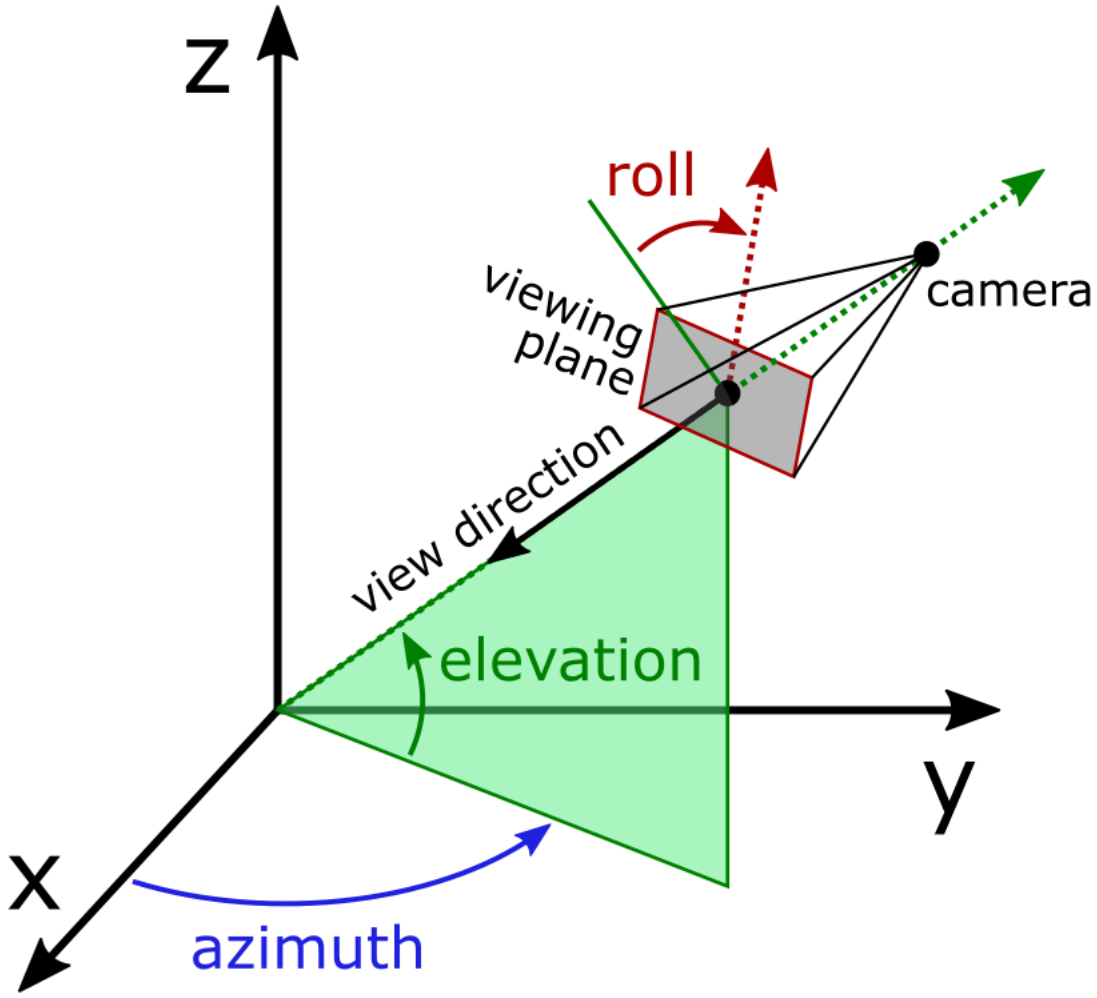
### I don't like how the 3D plot is laid out, how do I change that?

Historically, mplot3d has suffered from a hard-coding of parameters used to control visuals such as label spacing, tick length, and grid line width. Work is being done to eliminate this issue. For matplotlib v1.1.0, there is a semi-official manner to modify these parameters. See the note in the `mplot3d.axis3d` section of the mplot3d API documentation for more information.

### mplot3d View Angles

### How to define the view angle

The position of the viewport "camera" in a 3D plot is defined by three angles: *elevation*, *azimuth*, and *roll*. From the resulting position, it always points towards the center of the plot box volume. The angle direction is a common convention, and is shared with PyVista and MATLAB (though MATLAB lacks a roll angle). Note that a positive roll angle rotates the viewing plane clockwise, so the 3d axes will appear to rotate counter-clockwise.




Rotating the plot using the mouse will control only the azimuth and elevation, but all three angles can be set programmatically:

```
import matplotlib.pyplot as plt
ax = plt.figure().add_subplot(projection='3d')
ax.view_init(elev=30, azim=45, roll=15)
```

### Primary view planes

To look directly at the primary view planes, the required elevation, azimuth, and roll angles are shown in the diagram of an "unfolded" plot below. These are further documented in the *mplot3d.axes3d.Axes3D.view_init* API.

### mpl_toolkits.mplot3d.axes3d.Axes3D

**class** mpl_toolkits.mplot3d.axes3d.**Axes3D**(*fig*, *rect=None*, **args*, *elev=30*, *azim=-60*, *roll=0*, *sharez=None*, *proj_type='persp'*, *box_aspect=None*, *computed_zorder=True*, *focal_length=None*, *shareview=None*, ***kwargs*)

Bases: *Axes*

3D Axes object.

---

**Note:** As a user, you do not instantiate Axes directly, but use Axes creation methods instead; e.g. from *pyplot* or *Figure*: *subplots*, *subplot_mosaic* or *Figure.add_axes*.

---

**Parameters**

**fig**

[Figure] The parent figure.

**rect**

[tuple (left, bottom, width, height), default: None.] The `(left, bottom, width, height)` Axes position.

**elev**

[float, default: 30] The elevation angle in degrees rotates the camera above and below the x-y plane, with a positive angle corresponding to a location above the plane.

**azim**

[float, default: -60] The azimuthal angle in degrees rotates the camera about the z axis, with a positive angle corresponding to a right-handed rotation. In other words, a positive azimuth rotates the camera about the origin from its location along the +x axis towards the +y axis.

**roll**

[float, default: 0] The roll angle in degrees rotates the camera about the viewing axis. A positive angle spins the camera clockwise, causing the scene to rotate counter-clockwise.

**sharez**

[Axes3D, optional] Other Axes to share z-limits with.

**proj_type**

[{'persp', 'ortho'}] The projection type, default 'persp'.

**box_aspect**

[3-tuple of floats, default: None] Changes the physical dimensions of the Axes3D, such that the ratio of the axis lengths in display units is x:y:z. If None, defaults to 4:4:3

**computed_zorder**

[bool, default: True] If True, the draw order is computed based on the average position of the *Artist*s along the view direction. Set to False if you want to manually control the order in which Artists are drawn on top of each other using their *zorder* attribute. This can be used for fine-tuning if the automatic order does not produce the desired result. Note however, that a manual zorder will only be correct for a limited view angle. If the figure is rotated by the user, it will look wrong from certain angles.

**focal_length**

[float, default: None] For a projection type of 'persp', the focal length of the virtual camera. Must be > 0. If None, defaults to 1. For a projection type of 'ortho', must be set to either None or infinity (numpy.inf). If None, defaults to infinity. The focal length can be computed from a desired Field Of View via the equation: focal_length = 1/tan(FOV/2)

**shareview**

[Axes3D, optional] Other Axes to share view angles with.

****kwargs**

Other optional keyword arguments:

| Property | Description |
|---|---|
| *adjustable* | {'box', 'datalim'} |
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi v |
| *alpha* | scalar or None |
| *anchor* | (float, float) or {'C', 'SW', 'S', 'SE', 'E', 'NE', ...} |
| *animated* | bool |
| *aspect* | {'auto', 'equal', 'equalxy', 'equalxz', 'equalyz'} |
| *autoscale_on* | bool |
| *autoscalex_on* | unknown |
| *autoscaley_on* | unknown |
| *autoscalez_on* | unknown |
| *axes_locator* | Callable[[Axes, Renderer], Bbox] |
| *axisbelow* | bool or 'line' |
| *box_aspect* | 3-tuple of floats or None |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *facecolor* or fc | *color* |

Table 162 – continued fro

| Property | Description |
|---|---|
| figure | Figure |
| forward_navigation_events | bool or "auto" |
| gid | str |
| in_layout | bool |
| label | object |
| mouseover | bool |
| navigate | bool |
| navigate_mode | unknown |
| path_effects | list of AbstractPathEffect |
| picker | None or bool or float or callable |
| position | [left, bottom, width, height] or Bbox |
| proj_type | {'persp', 'ortho'} |
| prop_cycle | Cycler |
| rasterization_zorder | float or None |
| rasterized | bool |
| sketch_params | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| subplotspec | unknown |
| title | unknown |
| transform | Transform |
| url | str |
| visible | bool |
| xbound | unknown |
| xlabel | str |
| xlim or xlim3d | (left: float, right: float) |
| xmargin | float greater than -0.5 |
| xscale | unknown |
| xticklabels | unknown |
| xticks | unknown |
| ybound | unknown |
| ylabel | str |
| ylim or ylim3d | (bottom: float, top: float) |
| ymargin | float greater than -0.5 |
| yscale | unknown |
| yticklabels | unknown |
| yticks | unknown |
| zbound | unknown |
| zlabel | unknown |
| zlim or zlim3d | (bottom: float, top: float) |
| zmargin | float greater than -0.5 |
| zorder | float |
| zscale | unknown |
| zticklabels | unknown |
| zticks | unknown |

Table 162 – continued fro

| Property | Description |
|---|---|

### Plotting

| | |
|---|---|
| *plot* | Plot 2D or 3D data. |
| *scatter* | Create a scatter plot. |
| *bar* | Add 2D bar(s). |
| *bar3d* | Generate a 3D barplot. |
| *plot_surface* | Create a surface plot. |
| *plot_wireframe* | Plot a 3D wireframe. |
| *plot_trisurf* | Plot a triangulated surface. |
| *clabel* | Currently not implemented for 3D Axes, and returns *None*. |
| *contour* | Create a 3D contour plot. |
| *tricontour* | Create a 3D contour plot. |
| *contourf* | Create a 3D filled contour plot. |
| *tricontourf* | Create a 3D filled contour plot. |
| *quiver* | Plot a 3D field of arrows. |
| *voxels* | Plot a set of filled voxels |
| *errorbar* | Plot lines and/or markers with errorbars around them. |
| *stem* | Create a 3D stem plot. |

### mpl_toolkits.mplot3d.axes3d.Axes3D.plot

Axes3D.**plot**(*xs*, *ys*, **args*, *zdir='z'*, ***kwargs*)

Plot 2D or 3D data.

**Parameters**

**xs**

[1D array-like] x coordinates of vertices.

**ys**

[1D array-like] y coordinates of vertices.

**zs**

[float or 1D array-like] z coordinates of vertices; either one for all points or one for each point.

**zdir**

[{'x', 'y', 'z'}, default: 'z'] When plotting 2D data, the direction to use as z.

****kwargs**

Other arguments are forwarded to `matplotlib.axes.Axes.plot`.

## mpl_toolkits.mplot3d.axes3d.Axes3D.scatter

Axes3D.**scatter**(*xs*, *ys*, *zs=0*, *zdir='z'*, *s=20*, *c=None*, *depthshade=True*, **args*, *data=None*, ***kwargs*)

Create a scatter plot.

**Parameters**

**xs, ys**

[array-like] The data positions.

**zs**

[float or array-like, default: 0] The z-positions. Either an array of the same length as *xs* and *ys* or a single value to place all points in the same plane.

**zdir**

[{'x', 'y', 'z', '-x', '-y', '-z'}, default: 'z'] The axis direction for the *zs*. This is useful when plotting 2D data on a 3D Axes. The data must be passed as *xs*, *ys*. Setting *zdir* to 'y' then plots the data to the x-z-plane.

See also *Plot 2D data on 3D plot*.

**s**

[float or array-like, default: 20] The marker size in points**2. Either an array of the same length as *xs* and *ys* or a single value to make all markers the same size.

**c**

[*color*, sequence, or sequence of colors, optional] The marker color. Possible values:

- A single color format string.
- A sequence of colors of length n.
- A sequence of n numbers to be mapped to colors using *cmap* and *norm*.
- A 2D array in which the rows are RGB or RGBA.

For more details see the *c* argument of `scatter`.

**depthshade**

[bool, default: True] Whether to shade the scatter markers to give the appearance of depth. Each call to `scatter()` will perform its depthshading independently.

**data**

[indexable object, optional] If given, the following parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception):

*xs*, *ys*, *zs*, *s*, *edgecolors*, *c*, *facecolor*, *facecolors*, *color*

****kwargs**

All other keyword arguments are passed on to *scatter*.

**Returns**

**paths**

[*PathCollection*]

### Examples using `mpl_toolkits.mplot3d.axes3d.Axes3D.scatter`

- *scatter(xs, ys, zs)*

## mpl_toolkits.mplot3d.axes3d.Axes3D.bar

`Axes3D.`**`bar`**(*left*, *height*, *zs=0*, *zdir='z'*, **args*, *data=None*, ***kwargs*)

Add 2D bar(s).

**Parameters**

**left**

[1D array-like] The x coordinates of the left sides of the bars.

**height**

[1D array-like] The height of the bars.

**zs**

[float or 1D array-like, default: 0] Z coordinate of bars; if a single value is specified, it will be used for all bars.

**zdir**

[{'x', 'y', 'z'}, default: 'z'] When plotting 2D data, the direction to use as z ('x', 'y' or 'z').

**data**

[indexable object, optional] If given, all parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception).

****kwargs**

Other keyword arguments are forwarded to *matplotlib.axes.Axes.bar*.

**Returns**

**mpl_toolkits.mplot3d.art3d.Patch3DCollection**

## mpl_toolkits.mplot3d.axes3d.Axes3D.bar3d

Axes3D.**bar3d**(*x*, *y*, *z*, *dx*, *dy*, *dz*, *color=None*, *zsort='average'*, *shade=True*, *lightsource=None*, **args*, *data=None*, ***kwargs*)

Generate a 3D barplot.

This method creates three-dimensional barplot where the width, depth, height, and color of the bars can all be uniquely set.

**Parameters**

**x, y, z**

[array-like] The coordinates of the anchor point of the bars.

**dx, dy, dz**

[float or array-like] The width, depth, and height of the bars, respectively.

**color**

[sequence of colors, optional] The color of the bars can be specified globally or individually. This parameter can be:

- A single color, to color all bars the same color.
- An array of colors of length N bars, to color each bar independently.
- An array of colors of length 6, to color the faces of the bars similarly.
- An array of colors of length 6 * N bars, to color each face independently.

When coloring the faces of the boxes specifically, this is the order of the coloring:

1. -Z (bottom of box)
2. +Z (top of box)
3. -Y
4. +Y
5. -X
6. +X

**zsort**

[{'average', 'min', 'max'}, default: 'average'] The z-axis sorting scheme passed onto *Poly3DCollection*

**shade**

[bool, default: True] When true, this shades the dark sides of the bars (relative to the plot's source of light).

**lightsource**

[*LightSource*, optional] The lightsource to use when *shade* is True.

**data**

[indexable object, optional] If given, all parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception).

****kwargs**

Any additional keyword arguments are passed onto *Poly3DCollection*.

**Returns**

**collection**

[*Poly3DCollection*] A collection of three-dimensional polygons representing the bars.

## mpl_toolkits.mplot3d.axes3d.Axes3D.plot_surface

Axes3D.**plot_surface**(*X*, *Y*, *Z*, ***, *norm=None*, *vmin=None*, *vmax=None*, *lightsource=None*, ***kwargs*)

Create a surface plot.

By default, it will be colored in shades of a solid color, but it also supports colormapping by supplying the *cmap* argument.

---

**Note:** The *rcount* and *ccount* kwargs, which both default to 50, determine the maximum number of samples used in each direction. If the input data is larger, it will be downsampled (by slicing) to these numbers of points.

---

---

**Note:** To maximize rendering speed consider setting *rstride* and *cstride* to divisors of the number of rows minus 1 and columns minus 1 respectively. For example, given 51 rows rstride can be any of the divisors of 50.

Similarly, a setting of *rstride* and *cstride* equal to 1 (or *rcount* and *ccount* equal the number of rows and columns) can use the optimized path.

---

**Parameters**

**X, Y, Z**

[2D arrays] Data values.

**rcount, ccount**

[int] Maximum number of samples used in each direction. If the input data is larger, it will be downsampled (by slicing) to these numbers of points. Defaults to 50.

**rstride, cstride**

[int] Downsampling stride in each direction. These arguments are mutually exclusive with *rcount* and *ccount*. If only one of *rstride* or *cstride* is set, the other defaults to 10.

'classic' mode uses a default of `rstride = cstride = 10` instead of the new default of `rcount = ccount = 50`.

**color**

[*color*] Color of the surface patches.

**cmap**

[Colormap, optional] Colormap of the surface patches.

**facecolors**

[list of *color*] Colors of each individual patch.

**norm**

[*Normalize*, optional] Normalization for the colormap.

**vmin, vmax**

[float, optional] Bounds for the normalization.

**shade**

[bool, default: True] Whether to shade the facecolors. Shading is always disabled when *cmap* is specified.

**lightsource**

[*LightSource*, optional] The lightsource to use when *shade* is True.

****kwargs**

Other keyword arguments are forwarded to *Poly3DCollection*.

### Examples using `mpl_toolkits.mplot3d.axes3d.Axes3D.plot_surface`

- *plot_surface(X, Y, Z)*

## mpl_toolkits.mplot3d.axes3d.Axes3D.plot_wireframe

Axes3D.**plot_wireframe**(*X*, *Y*, *Z*, ***kwargs*)

Plot a 3D wireframe.

---

**Note:** The *rcount* and *ccount* kwargs, which both default to 50, determine the maximum number of samples used in each direction. If the input data is larger, it will be downsampled (by slicing) to these numbers of points.

---

**Parameters**

**X, Y, Z**

[2D arrays] Data values.

**rcount, ccount**

[int] Maximum number of samples used in each direction. If the input data is larger, it will be downsampled (by slicing) to these numbers of points. Setting a count to zero causes the data to be not sampled in the corresponding direction, producing a 3D line plot rather than a wireframe plot. Defaults to 50.

**rstride, cstride**

[int] Downsampling stride in each direction. These arguments are mutually exclusive with *rcount* and *ccount*. If only one of *rstride* or *cstride* is set, the other defaults to 1. Setting a stride to zero causes the data to be not sampled in the corresponding direction, producing a 3D line plot rather than a wireframe plot.

'classic' mode uses a default of `rstride = cstride = 1` instead of the new default of `rcount = ccount = 50`.

****kwargs**

Other keyword arguments are forwarded to *Line3DCollection*.

### Examples using `mpl_toolkits.mplot3d.axes3d.Axes3D.plot_wireframe`

- *plot_wireframe(X, Y, Z)*

## mpl_toolkits.mplot3d.axes3d.Axes3D.plot_trisurf

Axes3D.**plot_trisurf**(**args*, *color=None*, *norm=None*, *vmin=None*, *vmax=None*, *lightsource=None*, ***kwargs*)

Plot a triangulated surface.

The (optional) triangulation can be specified in one of two ways; either:

```
plot_trisurf(triangulation, ...)
```

where triangulation is a *Triangulation* object, or:

```
plot_trisurf(X, Y, ...)
plot_trisurf(X, Y, triangles, ...)
plot_trisurf(X, Y, triangles=triangles, ...)
```

in which case a Triangulation object will be created. See *Triangulation* for an explanation of these possibilities.

The remaining arguments are:

```
plot_trisurf(..., Z)
```

where *Z* is the array of values to contour, one per point in the triangulation.

**Parameters**

**X, Y, Z**

[array-like] Data values as 1D arrays.

**color**

Color of the surface patches.

**cmap**

A colormap for the surface patches.

**norm**

[*Normalize*, optional] An instance of Normalize to map values to colors.

**vmin, vmax**

[float, optional] Minimum and maximum value to map.

**shade**

[bool, default: True] Whether to shade the facecolors. Shading is always disabled when *cmap* is specified.

**lightsource**

[*LightSource*, optional] The lightsource to use when *shade* is True.

****kwargs**

All other keyword arguments are passed on to *Poly3DCollection*

### Examples

## Examples using `mpl_toolkits.mplot3d.axes3d.Axes3D.plot_trisurf`

- *plot_trisurf(x, y, z)*

## mpl_toolkits.mplot3d.axes3d.Axes3D.clabel

Axes3D.**clabel**(**args*, ***kwargs*)

Currently not implemented for 3D Axes, and returns *None*.

## mpl_toolkits.mplot3d.axes3d.Axes3D.contour

Axes3D.**contour**(*X*, *Y*, *Z*, **args*, *extend3d=False*, *stride=5*, *zdir='z'*, *offset=None*, *data=None*, ***kwargs*)

Create a 3D contour plot.

**Parameters**

**X, Y, Z**

[array-like,] Input data. See *Axes.contour* for supported data shapes.

**extend3d**

[bool, default: False] Whether to extend contour in 3D.

**stride**

[int, default: 5] Step size for extending contour.

**zdir**

[{'x', 'y', 'z'}, default: 'z'] The direction to use.

**offset**

[float, optional] If specified, plot a projection of the contour lines at this position in a plane normal to *zdir*.

**data**

[indexable object, optional] If given, all parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception).

***args, **kwargs**

Other arguments are forwarded to *matplotlib.axes.Axes.contour*.

**Returns**

**matplotlib.contour.QuadContourSet**

### mpl_toolkits.mplot3d.axes3d.Axes3D.tricontour

`Axes3D.`**`tricontour`**(**args*, *extend3d=False*, *stride=5*, *zdir='z'*, *offset=None*, *data=None*, ***kwargs*)

Create a 3D contour plot.

---

**Note:** This method currently produces incorrect output due to a longstanding bug in 3D PolyCollection rendering.

---

**Parameters**

**X, Y, Z**

[array-like] Input data. See *Axes.tricontour* for supported data shapes.

**extend3d**

[bool, default: False] Whether to extend contour in 3D.

**stride**

[int, default: 5] Step size for extending contour.

**zdir**

[{'x', 'y', 'z'}, default: 'z'] The direction to use.

**offset**

[float, optional] If specified, plot a projection of the contour lines at this position in a plane normal to *zdir*.

**data**

[indexable object, optional] If given, all parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception).

***args, **kwargs**

Other arguments are forwarded to *matplotlib.axes.Axes.tricontour*.

**Returns**

**matplotlib.tri._tricontour.TriContourSet**

### mpl_toolkits.mplot3d.axes3d.Axes3D.contourf

Axes3D.**contourf**(*X*, *Y*, *Z*, **args*, *zdir='z'*, *offset=None*, *data=None*, ***kwargs*)

Create a 3D filled contour plot.

**Parameters**

**X, Y, Z**

[array-like] Input data. See *Axes.contourf* for supported data shapes.

**zdir**

[{'x', 'y', 'z'}, default: 'z'] The direction to use.

**offset**

[float, optional] If specified, plot a projection of the contour lines at this position in a plane normal to *zdir*.

**data**

[indexable object, optional] If given, all parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception).

***args, **kwargs**

Other arguments are forwarded to *matplotlib.axes.Axes.contourf*.

**Returns**

**matplotlib.contour.QuadContourSet**

### mpl_toolkits.mplot3d.axes3d.Axes3D.tricontourf

Axes3D.**tricontourf**(**args*, *zdir='z'*, *offset=None*, *data=None*, ***kwargs*)

Create a 3D filled contour plot.

---

**Note:** This method currently produces incorrect output due to a longstanding bug in 3D PolyCollection rendering.

---

**Parameters**

**X, Y, Z**

[array-like] Input data. See *Axes.tricontourf* for supported data shapes.

**zdir**

[{'x', 'y', 'z'}, default: 'z'] The direction to use.

**offset**

[float, optional] If specified, plot a projection of the contour lines at this position in a plane normal to zdir.

**data**

[indexable object, optional] If given, all parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception).

***args, **kwargs**

Other arguments are forwarded to *matplotlib.axes.Axes.tricontourf*.

**Returns**

**matplotlib.tri._tricontour.TriContourSet**

## mpl_toolkits.mplot3d.axes3d.Axes3D.quiver

Axes3D.**quiver**(*X*, *Y*, *Z*, *U*, *V*, *W*, *, *length=1*, *arrow_length_ratio=0.3*, *pivot='tail'*, *normalize=False*, *data=None*, ***kwargs*)

Plot a 3D field of arrows.

The arguments can be array-like or scalars, so long as they can be broadcast together. The arguments can also be masked arrays. If an element in any of argument is masked, then that corresponding quiver element will not be plotted.

**Parameters**

**X, Y, Z**

[array-like] The x, y and z coordinates of the arrow locations (default is tail of arrow; see *pivot* kwarg).

**U, V, W**

[array-like] The x, y and z components of the arrow vectors.

**length**

[float, default: 1] The length of each quiver.

**arrow_length_ratio**

[float, default: 0.3] The ratio of the arrow head with respect to the quiver.

**pivot**

[{'tail', 'middle', 'tip'}, default: 'tail'] The part of the arrow that is at the grid point; the arrow rotates about this point, hence the name *pivot*.

**normalize**

[bool, default: False] Whether all arrows are normalized to have the same length, or keep the lengths defined by *u*, *v*, and *w*.

**data**

[indexable object, optional] If given, all parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception).

****kwargs**

Any additional keyword arguments are delegated to *Line3DCollection*

### mpl_toolkits.mplot3d.axes3d.Axes3D.voxels

Axes3D.**voxels**([*x*, *y*, *z*, ]/, *filled*, *facecolors=None*, *edgecolors=None*, ***kwargs*)

Plot a set of filled voxels

All voxels are plotted as 1x1x1 cubes on the axis, with `filled[0, 0, 0]` placed with its lower corner at the origin. Occluded faces are not plotted.

**Parameters**

**filled**

[3D np.array of bool] A 3D array of values, with truthy values indicating which voxels to fill

**x, y, z**

[3D np.array, optional] The coordinates of the corners of the voxels. This should broadcast to a shape one larger in every dimension than the shape of *filled*. These can be used to plot non-cubic voxels.

If not specified, defaults to increasing integers along each axis, like those returned by `indices()`. As indicated by the `/` in the function signature, these arguments can only be passed positionally.

**facecolors, edgecolors**

[array-like, optional] The color to draw the faces and edges of the voxels. Can only be passed as keyword arguments. These parameters can be:

- A single color value, to color all voxels the same color. This can be either a string, or a 1D RGB/RGBA array
- `None`, the default, to use a single color for the faces, and the style default for the edges.

- A 3D `ndarray` of color names, with each item the color for the corresponding voxel. The size must match the voxels.
- A 4D `ndarray` of RGB/RGBA data, with the components along the last axis.

**shade**

[bool, default: True] Whether to shade the facecolors.

**lightsource**

[*LightSource*, optional] The lightsource to use when *shade* is True.

****kwargs**

Additional keyword arguments to pass onto *Poly3DCollection*.

**Returns**

**faces**

[dict] A dictionary indexed by coordinate, where `faces[i, j, k]` is a *Poly3DCollection* of the faces drawn for the voxel `filled[i, j, k]`. If no faces were drawn for a given voxel, either because it was not asked to be drawn, or it is fully occluded, then `(i, j, k) not in faces`.

### Examples

## Examples using `mpl_toolkits.mplot3d.axes3d.Axes3D.voxels`

- *voxels([x, y, z], filled)*

## mpl_toolkits.mplot3d.axes3d.Axes3D.errorbar

`Axes3D.`**`errorbar`**(*x*, *y*, *z*, *zerr=None*, *yerr=None*, *xerr=None*, *fmt=''*, *barsabove=False*, *errorevery=1*, *ecolor=None*, *elinewidth=None*, *capsize=None*, *capthick=None*, *xlolims=False*, *xuplims=False*, *ylolims=False*, *yuplims=False*, *zlolims=False*, *zuplims=False*, ***, *data=None*, ***kwargs*)

Plot lines and/or markers with errorbars around them.

*x*/*y*/*z* define the data locations, and *xerr*/*yerr*/*zerr* define the errorbar sizes. By default, this draws the data markers/lines as well the errorbars. Use fmt='none' to draw errorbars only.

**Parameters**

**x, y, z**

[float or array-like] The data positions.

**xerr, yerr, zerr**

[float or array-like, shape (N,) or (2, N), optional] The errorbar sizes:

- scalar: Symmetric +/- values for all data points.
- shape(N,): Symmetric +/-values for each data point.
- shape(2, N): Separate - and + values for each bar. First row contains the lower errors, the second row contains the upper errors.
- *None*: No errorbar.

Note that all error arrays should have *positive* values.

**fmt**

[str, default: ''] The format for the data points / data lines. See `plot` for details.

Use 'none' (case-insensitive) to plot errorbars without any data markers.

**ecolor**

[*color*, default: None] The color of the errorbar lines. If None, use the color of the line connecting the markers.

**elinewidth**

[float, default: None] The linewidth of the errorbar lines. If None, the linewidth of the current style is used.

**capsize**

[float, default: `rcParams["errorbar.capsize"]` (default: `0.0`)] The length of the error bar caps in points.

**capthick**

[float, default: None] An alias to the keyword argument *markeredgewidth* (a.k.a. *mew*). This setting is a more sensible name for the property that controls the thickness of the error bar cap in points. For backwards compatibility, if *mew* or *markeredgewidth* are given, then they will over-ride *capthick*. This may change in future releases.

**barsabove**

[bool, default: False] If True, will plot the errorbars above the plot symbols. Default is below.

**xlolims, ylolims, zlolims**

[bool, default: False] These arguments can be used to indicate that a value gives only lower limits. In that case a caret symbol is used to indicate this. *lims*-arguments may be scalars, or array-likes of the same length as the errors. To use

limits with inverted axes, *set_xlim*, *set_ylim*, or *set_zlim* must be called before *errorbar*. Note the tricky parameter names: setting e.g. *ylolims* to True means that the y-value is a *lower* limit of the True value, so, only an *upward*-pointing arrow will be drawn!

**xuplims, yuplims, zuplims**

[bool, default: False] Same as above, but for controlling the upper limits.

**errorevery**

[int or (int, int), default: 1] draws error bars on a subset of the data. *errorevery* =N draws error bars on the points (x[::N], y[::N], z[::N]). *errorevery* =(start, N) draws error bars on the points (x[start::N], y[start::N], z[start::N]). e.g. *errorevery* =(6, 3) adds error bars to the data at (x[6], x[9], x[12], x[15], ...). Used to avoid overlapping error bars when two series share x-axis values.

**Returns**

**errlines**

[list] List of *Line3DCollection* instances each containing an errorbar line.

**caplines**

[list] List of *Line3D* instances each containing a capline object.

**limmarks**

[list] List of *Line3D* instances each containing a marker with an upper or lower limit.

**Other Parameters**

**data**

[indexable object, optional] If given, the following parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception):

*x*, *y*, *z*, *xerr*, *yerr*, *zerr*

****kwargs**

All other keyword arguments for styling errorbar lines are passed *Line3DCollection*.

### Examples

## mpl_toolkits.mplot3d.axes3d.Axes3D.stem

Axes3D.**stem**(*x*, *y*, *z*, *, *linefmt='C0-'*, *markerfmt='C0o'*, *basefmt='C3-'*, *bottom=0*, *label=None*, *orientation='z'*, *data=None*)

Create a 3D stem plot.

A stem plot draws lines perpendicular to a baseline, and places markers at the heads. By default, the baseline is defined by *x* and *y*, and stems are drawn vertically from *bottom* to *z*.

**Parameters**

**x, y, z**

[array-like] The positions of the heads of the stems. The stems are drawn along the *orientation*-direction from the baseline at *bottom* (in the *orientation*-coordinate) to the heads. By default, the *x* and *y* positions are used for the baseline and *z* for the head position, but this can be changed by *orientation*.

**linefmt**

[str, default: 'C0-'] A string defining the properties of the vertical lines. Usually, this will be a color or a color and a linestyle:

| Character | Line Style |
|---|---|
| `'-'` | solid line |
| `'--'` | dashed line |
| `'-.'` | dash-dot line |
| `':'` | dotted line |

Note: While it is technically possible to specify valid formats other than color or color and linestyle (e.g. 'rx' or '-.'), this is beyond the intention of the method and will most likely not result in a reasonable plot.

**markerfmt**

[str, default: 'C0o'] A string defining the properties of the markers at the stem heads.

**basefmt**

[str, default: 'C3-'] A format string defining the properties of the baseline.

**bottom**

[float, default: 0] The position of the baseline, in *orientation*-coordinates.

**label**

[str, optional] The label to use for the stems in legends.

**orientation**

[{'x', 'y', 'z'}, default: 'z'] The direction along which stems are drawn.

**data**

[indexable object, optional] If given, all parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception).

**Returns**

***StemContainer***

The container may be treated like a tuple (*markerline*, *stemlines*, *baseline*)

### Examples

## Examples using `mpl_toolkits.mplot3d.axes3d.Axes3D.stem`

- *3D stem*

## Text and annotations

| | |
|---|---|
| *text* | Add the text *s* to the 3D Axes at location *x*, *y*, *z* in data coordinates. |
| *text2D* | Add text to the Axes. |

## mpl_toolkits.mplot3d.axes3d.Axes3D.text

Axes3D.**text**(*x*, *y*, *z*, *s*, *zdir=None*, ***kwargs*)

Add the text *s* to the 3D Axes at location *x*, *y*, *z* in data coordinates.

**Parameters**

**x, y, z**

[float] The position to place the text.

**s**

[str] The text.

**zdir**

[{'x', 'y', 'z', 3-tuple}, optional] The direction to be used as the z-direction. Default: 'z'. See *get_dir_vector* for a description of the values.

****kwargs**

Other arguments are forwarded to *matplotlib.axes.Axes.text*.

**Returns**

***Text3D***

The created *Text3D* instance.

## mpl_toolkits.mplot3d.axes3d.Axes3D.text2D

Axes3D.**text2D**(*x*, *y*, *s*, *fontdict=None*, ***kwargs*)

Add text to the Axes.

Add the text *s* to the Axes at location *x*, *y* in data coordinates, with a default `horizontalalignment` on the `left` and `verticalalignment` at the `baseline`. See *Text alignment*.

**Parameters**

**x, y**

[float] The position to place the text. By default, this is in data coordinates. The coordinate system can be changed using the *transform* parameter.

**s**

[str] The text.

**fontdict**

[dict, default: None]

---

**Discouraged**

The use of *fontdict* is discouraged. Parameters should be passed as individual keyword arguments or using dictionary-unpacking `text(..., **fontdict)`.

---

A dictionary to override the default text properties. If fontdict is None, the defaults are determined by *rcParams*.

**Returns**

***Text***

The created *Text* instance.

**Other Parameters**

****kwargs**

[*Text* properties.] Other miscellaneous text parameters.

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array a |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* | bool |
| *backgroundcolor* | *color* |
| *bbox* | dict with properties for *patches.FancyBboxPa* |
| *clip_box* | unknown |
| *clip_on* | unknown |
| *clip_path* | unknown |
| *color* or c | *color* |
| *figure* | *Figure* |
| *fontfamily* or family or fontname | {FONTNAME, 'serif', 'sans-serif', 'cursive', 'fantasy' |
| *fontproperties* or font or font_properties | *font_manager.FontProperties* or *str* or |
| *fontsize* or size | float or {'xx-small', 'x-small', 'small', 'medium', 'large |
| *fontstretch* or stretch | {a numeric value in range 0-1000, 'ultra-condensed', |
| *fontstyle* or style | {'normal', 'italic', 'oblique'} |
| *fontvariant* or variant | {'normal', 'small-caps'} |
| *fontweight* or weight | {a numeric value in range 0-1000, 'ultralight', 'light', |
| *gid* | str |
| *horizontalalignment* or ha | {'left', 'center', 'right'} |
| *in_layout* | bool |
| *label* | object |
| *linespacing* | float (multiple of font size) |
| *math_fontfamily* | str |
| *mouseover* | bool |
| *multialignment* or ma | {'left', 'right', 'center'} |
| *parse_math* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *position* | (float, float) |
| *rasterized* | bool |
| *rotation* | float or {'vertical', 'horizontal'} |
| *rotation_mode* | {None, 'default', 'anchor'} |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *text* | object |

Table 163 -

| Property | Description |
|---|---|
| *transform* | *Transform* |
| *transform_rotates_text* | bool |
| *url* | str |
| *usetex* | bool, default: *rcParams["text.usetex"]* (de |
| *verticalalignment* or va | {'baseline', 'bottom', 'center', 'center_baseline', 'top'} |
| *visible* | bool |
| *wrap* | bool |
| *x* | float |
| *y* | float |
| *zorder* | float |

### Examples

Individual keyword arguments can be used to override any given parameter:

```
>>> text(x, y, s, fontsize=12)
```

The default transform specifies that text is in data coords, alternatively, you can specify text in axis coords ((0, 0) is lower-left and (1, 1) is upper-right). The example below places text in the center of the Axes:

```
>>> text(0.5, 0.5, 'matplotlib', horizontalalignment='center',
...      verticalalignment='center', transform=ax.transAxes)
```

You can put a rectangular box around the text instance (e.g., to set a background color) by using the keyword *bbox*. *bbox* is a dictionary of *Rectangle* properties. For example:

```
>>> text(x, y, s, bbox=dict(facecolor='red', alpha=0.5))
```

## Clearing

| | |
|---|---|
| *clear* | Clear the Axes. |

### mpl_toolkits.mplot3d.axes3d.Axes3D.clear

Axes3D.**clear**()

Clear the Axes.

### Appearance

| | |
|---|---|
| *set_axis_off* | Hide all visual components of the x- and y-axis. |
| *set_axis_on* | Do not hide all visual components of the x- and y-axis. |
| *grid* | Set / unset 3D grid. |

#### mpl_toolkits.mplot3d.axes3d.Axes3D.set_axis_off

Axes3D.**set_axis_off**()

Hide all visual components of the x- and y-axis.

This sets a flag to suppress drawing of all axis decorations, i.e. axis labels, axis spines, and the axis tick component (tick markers, tick labels, and grid lines). Individual visibility settings of these components are ignored as long as *set_axis_off()* is in effect.

#### mpl_toolkits.mplot3d.axes3d.Axes3D.set_axis_on

Axes3D.**set_axis_on**()

Do not hide all visual components of the x- and y-axis.

This reverts the effect of a prior *set_axis_off()* call. Whether the individual axis decorations are drawn is controlled by their respective visibility settings.

This is on by default.

#### mpl_toolkits.mplot3d.axes3d.Axes3D.grid

Axes3D.**grid**(*visible=True*, ***kwargs*)

Set / unset 3D grid.

**Note:** Currently, this function does not behave the same as *axes.Axes.grid*, but it is intended to eventually support that behavior.

### Axis

### Axis limits and direction

| | |
|---|---|
| *get_zaxis* | Return the ZAxis (*Axis*) instance. |
| *get_xlim* | Return the x-axis view limits. |
| *set_xlim* | Set the 3D x-axis view limits. |
| *get_ylim* | Return the y-axis view limits. |
| *set_ylim* | Set the 3D y-axis view limits. |
| *get_zlim* | Return the 3D z-axis view limits. |
| *set_zlim* | Set the 3D z-axis view limits. |
| *get_w_lims* | Get 3D world limits. |
| *invert_xaxis* | Invert the x-axis. |
| *xaxis_inverted* | Return whether the xaxis is oriented in the "inverse" direction. |
| *invert_yaxis* | Invert the y-axis. |
| *yaxis_inverted* | Return whether the yaxis is oriented in the "inverse" direction. |
| *invert_zaxis* | Invert the z-axis. |
| *zaxis_inverted* | Return whether the zaxis is oriented in the "inverse" direction. |
| *get_xbound* | Return the lower and upper x-axis bounds, in increasing order. |
| *set_xbound* | Set the lower and upper numerical bounds of the x-axis. |
| *get_ybound* | Return the lower and upper y-axis bounds, in increasing order. |
| *set_ybound* | Set the lower and upper numerical bounds of the y-axis. |
| *get_zbound* | Return the lower and upper z-axis bounds, in increasing order. |
| *set_zbound* | Set the lower and upper numerical bounds of the z-axis. |

### mpl_toolkits.mplot3d.axes3d.Axes3D.get_zaxis

Axes3D.**get_zaxis**()

Return the ZAxis (*Axis*) instance.

### mpl_toolkits.mplot3d.axes3d.Axes3D.get_xlim

Axes3D.**get_xlim**()

Return the x-axis view limits.

**Returns**

**left, right**

[(float, float)] The current x-axis limits in data coordinates.

**See also:**

***Axes.set_xlim***
***Axes.set_xbound***, ***Axes.get_xbound***
***Axes.invert_xaxis***, ***Axes.xaxis_inverted***

#### Notes

The x-axis may be inverted, in which case the *left* value will be greater than the *right* value.

### mpl_toolkits.mplot3d.axes3d.Axes3D.set_xlim

Axes3D.**set_xlim**(*left=None*, *right=None*, *, *emit=True*, *auto=False*, *view_margin=None*, *xmin=None*, *xmax=None*)

Set the 3D x-axis view limits.

**Parameters**

**left**

[float, optional] The left xlim in data coordinates. Passing *None* leaves the limit unchanged.

The left and right xlims may also be passed as the tuple (*left*, *right*) as the first positional argument (or as the *left* keyword argument).

**right**

[float, optional] The right xlim in data coordinates. Passing *None* leaves the limit unchanged.

**emit**

[bool, default: True] Whether to notify observers of limit change.

**auto**

[bool or None, default: False] Whether to turn on autoscaling of the x-axis. *True* turns on, *False* turns off, *None* leaves unchanged.

**view_margin**

[float, optional] The additional margin to apply to the limits.

**xmin, xmax**

[float, optional] They are equivalent to left and right respectively, and it is an error to pass both *xmin* and *left* or *xmax* and *right*.

**Returns**

**left, right**

[(float, float)] The new x-axis limits in data coordinates.

**See also:**

***get_xlim***
***set_xbound*, *get_xbound***
***invert_xaxis*, *xaxis_inverted***

## Notes

The *left* value may be greater than the *right* value, in which case the x-axis values will decrease from *left* to *right*.

## Examples

```
>>> set_xlim(left, right)
>>> set_xlim((left, right))
>>> left, right = set_xlim(left, right)
```

One limit may be left unchanged.

```
>>> set_xlim(right=right_lim)
```

Limits may be passed in reverse order to flip the direction of the x-axis. For example, suppose `x` represents depth of the ocean in m. The x-axis limits might be set like the following so 5000 m depth is at the left of the plot and the surface, 0 m, is at the right.

```
>>> set_xlim(5000, 0)
```

### mpl_toolkits.mplot3d.axes3d.Axes3D.get_ylim

Axes3D.**get_ylim**()

Return the y-axis view limits.

**Returns**

**bottom, top**

[(float, float)] The current y-axis limits in data coordinates.

**See also:**

***Axes.set_ylim***
***Axes.set_ybound*, *Axes.get_ybound***
***Axes.invert_yaxis*, *Axes.yaxis_inverted***

#### Notes

The y-axis may be inverted, in which case the *bottom* value will be greater than the *top* value.

### mpl_toolkits.mplot3d.axes3d.Axes3D.set_ylim

Axes3D.**set_ylim**(*bottom=None*, *top=None*, *, *emit=True*, *auto=False*, *view_margin=None*, *ymin=None*, *ymax=None*)

Set the 3D y-axis view limits.

**Parameters**

**bottom**

[float, optional] The bottom ylim in data coordinates. Passing *None* leaves the limit unchanged.

The bottom and top ylims may also be passed as the tuple (*bottom*, *top*) as the first positional argument (or as the *bottom* keyword argument).

**top**

[float, optional] The top ylim in data coordinates. Passing *None* leaves the limit unchanged.

**emit**

[bool, default: True] Whether to notify observers of limit change.

**auto**

[bool or None, default: False] Whether to turn on autoscaling of the y-axis. *True* turns on, *False* turns off, *None* leaves unchanged.

**view_margin**

[float, optional] The additional margin to apply to the limits.

**ymin, ymax**

[float, optional] They are equivalent to bottom and top respectively, and it is an error to pass both *ymin* and *bottom* or *ymax* and *top*.

**Returns**

**bottom, top**

[(float, float)] The new y-axis limits in data coordinates.

**See also:**

***get_ylim***
***set_ybound*, *get_ybound***
***invert_yaxis*, *yaxis_inverted***

## Notes

The *bottom* value may be greater than the *top* value, in which case the y-axis values will decrease from *bottom* to *top*.

## Examples

```
>>> set_ylim(bottom, top)
>>> set_ylim((bottom, top))
>>> bottom, top = set_ylim(bottom, top)
```

One limit may be left unchanged.

```
>>> set_ylim(top=top_lim)
```

Limits may be passed in reverse order to flip the direction of the y-axis. For example, suppose `y` represents depth of the ocean in m. The y-axis limits might be set like the following so 5000 m depth is at the bottom of the plot and the surface, 0 m, is at the top.

```
>>> set_ylim(5000, 0)
```

### mpl_toolkits.mplot3d.axes3d.Axes3D.get_zlim

Axes3D.**get_zlim**()

Return the 3D z-axis view limits.

**Returns**

**left, right**

[(float, float)] The current z-axis limits in data coordinates.

**See also:**

***set_zlim***
***set_zbound***, ***get_zbound***
***invert_zaxis***, ***zaxis_inverted***

#### Notes

The z-axis may be inverted, in which case the *left* value will be greater than the *right* value.

### mpl_toolkits.mplot3d.axes3d.Axes3D.set_zlim

Axes3D.**set_zlim**(*bottom=None*, *top=None*, *, *emit=True*, *auto=False*, *view_margin=None*, *zmin=None*, *zmax=None*)

Set the 3D z-axis view limits.

**Parameters**

**bottom**

[float, optional] The bottom zlim in data coordinates. Passing *None* leaves the limit unchanged.

The bottom and top zlims may also be passed as the tuple (*bottom*, *top*) as the first positional argument (or as the *bottom* keyword argument).

**top**

[float, optional] The top zlim in data coordinates. Passing *None* leaves the limit unchanged.

**emit**

[bool, default: True] Whether to notify observers of limit change.

**auto**

[bool or None, default: False] Whether to turn on autoscaling of the z-axis. *True* turns on, *False* turns off, *None* leaves unchanged.

**view_margin**

[float, optional] The additional margin to apply to the limits.

**zmin, zmax**

[float, optional] They are equivalent to bottom and top respectively, and it is an error to pass both *zmin* and *bottom* or *zmax* and *top*.

**Returns**

**bottom, top**

[(float, float)] The new z-axis limits in data coordinates.

**See also:**

***get_zlim***
***set_zbound***, ***get_zbound***
***invert_zaxis***, ***zaxis_inverted***

## Notes

The *bottom* value may be greater than the *top* value, in which case the z-axis values will decrease from *bottom* to *top*.

## Examples

```
>>> set_zlim(bottom, top)
>>> set_zlim((bottom, top))
>>> bottom, top = set_zlim(bottom, top)
```

One limit may be left unchanged.

```
>>> set_zlim(top=top_lim)
```

Limits may be passed in reverse order to flip the direction of the z-axis. For example, suppose `z` represents depth of the ocean in m. The z-axis limits might be set like the following so 5000 m depth is at the bottom of the plot and the surface, 0 m, is at the top.

```
>>> set_zlim(5000, 0)
```

### mpl_toolkits.mplot3d.axes3d.Axes3D.get_w_lims

Axes3D.**get_w_lims**()

Get 3D world limits.

### mpl_toolkits.mplot3d.axes3d.Axes3D.invert_xaxis

Axes3D.**invert_xaxis**()

Invert the x-axis.

**See also:**

*xaxis_inverted*
*get_xlim*, *set_xlim*
*get_xbound*, *set_xbound*

### mpl_toolkits.mplot3d.axes3d.Axes3D.xaxis_inverted

Axes3D.**xaxis_inverted**()

Return whether the xaxis is oriented in the "inverse" direction.

The "normal" direction is increasing to the right for the x-axis and to the top for the y-axis; the "inverse" direction is increasing to the left for the x-axis and to the bottom for the y-axis.

### mpl_toolkits.mplot3d.axes3d.Axes3D.invert_yaxis

Axes3D.**invert_yaxis**()

Invert the y-axis.

**See also:**

*yaxis_inverted*
*get_ylim*, *set_ylim*
*get_ybound*, *set_ybound*

### mpl_toolkits.mplot3d.axes3d.Axes3D.yaxis_inverted

Axes3D.**yaxis_inverted**()

Return whether the yaxis is oriented in the "inverse" direction.

The "normal" direction is increasing to the right for the x-axis and to the top for the y-axis; the "inverse" direction is increasing to the left for the x-axis and to the bottom for the y-axis.

### mpl_toolkits.mplot3d.axes3d.Axes3D.invert_zaxis

Axes3D.**invert_zaxis**()

Invert the z-axis.

**See also:**

*zaxis_inverted*
*get_zlim*, *set_zlim*
*get_zbound*, *set_zbound*

### mpl_toolkits.mplot3d.axes3d.Axes3D.zaxis_inverted

Axes3D.**zaxis_inverted**()

Return whether the zaxis is oriented in the "inverse" direction.

The "normal" direction is increasing to the right for the x-axis and to the top for the y-axis; the "inverse" direction is increasing to the left for the x-axis and to the bottom for the y-axis.

### mpl_toolkits.mplot3d.axes3d.Axes3D.get_xbound

Axes3D.**get_xbound**()

Return the lower and upper x-axis bounds, in increasing order.

**See also:**

*set_xbound*
*get_xlim*, *set_xlim*
*invert_xaxis*, *xaxis_inverted*

### mpl_toolkits.mplot3d.axes3d.Axes3D.set_xbound

Axes3D.**set_xbound**(*lower=None*, *upper=None*, *view_margin=None*)

Set the lower and upper numerical bounds of the x-axis.

This method will honor axis inversion regardless of parameter order. It will not change the autoscaling setting (*get_autoscalex_on()*).

**Parameters**

**lower, upper**

[float or None] The lower and upper bounds. If *None*, the respective axis bound is not modified.

**view_margin**

[float or None] The margin to apply to the bounds. If *None*, the margin is handled by *set_xlim*.

**See also:**

***get_xbound***
***get_xlim*, *set_xlim***
***invert_xaxis*, *xaxis_inverted***

### mpl_toolkits.mplot3d.axes3d.Axes3D.get_ybound

Axes3D.**get_ybound**()

Return the lower and upper y-axis bounds, in increasing order.

**See also:**

***set_ybound***
***get_ylim*, *set_ylim***
***invert_yaxis*, *yaxis_inverted***

### mpl_toolkits.mplot3d.axes3d.Axes3D.set_ybound

Axes3D.**set_ybound**(*lower=None*, *upper=None*, *view_margin=None*)

Set the lower and upper numerical bounds of the y-axis.

This method will honor axis inversion regardless of parameter order. It will not change the autoscaling setting (*get_autoscaley_on()*).

**Parameters**

**lower, upper**

[float or None] The lower and upper bounds. If *None*, the respective axis bound is not modified.

**view_margin**

[float or None] The margin to apply to the bounds. If *None*, the margin is handled by *set_ylim*.

**See also:**

***get_ybound***
***get_ylim*, *set_ylim***
***invert_yaxis*, *yaxis_inverted***

### mpl_toolkits.mplot3d.axes3d.Axes3D.get_zbound

Axes3D.**get_zbound**()

Return the lower and upper z-axis bounds, in increasing order.

**See also:**

***set_zbound***
***get_zlim***, ***set_zlim***
***invert_zaxis***, ***zaxis_inverted***

### mpl_toolkits.mplot3d.axes3d.Axes3D.set_zbound

Axes3D.**set_zbound**(*lower=None*, *upper=None*, *view_margin=None*)

Set the lower and upper numerical bounds of the z-axis. This method will honor axis inversion regardless of parameter order. It will not change the autoscaling setting (*get_autoscaley_on()*).

**Parameters**

**lower, upper**

[float or None] The lower and upper bounds. If *None*, the respective axis bound is not modified.

**view_margin**

[float or None] The margin to apply to the bounds. If *None*, the margin is handled by *set_zlim*.

**See also:**

***get_zbound***
***get_zlim***, ***set_zlim***
***invert_zaxis***, ***zaxis_inverted***

### Axis labels and title

| | |
|---|---|
| *set_zlabel* | Set zlabel. |
| *get_zlabel* | Get the z-label text string. |
| *set_title* | Set a title for the Axes. |

### mpl_toolkits.mplot3d.axes3d.Axes3D.set_zlabel

Axes3D.**set_zlabel**(*zlabel*, *fontdict=None*, *labelpad=None*, ***kwargs*)

Set zlabel. See doc for *set_ylabel* for description.

### mpl_toolkits.mplot3d.axes3d.Axes3D.get_zlabel

Axes3D.**get_zlabel**()

Get the z-label text string.

### mpl_toolkits.mplot3d.axes3d.Axes3D.set_title

Axes3D.**set_title**(*label*, *fontdict=None*, *loc='center'*, ***kwargs*)

Set a title for the Axes.

Set one of the three available Axes titles. The available titles are positioned above the Axes in the center, flush with the left edge, and flush with the right edge.

**Parameters**

**label**

[str] Text to use for the title

**fontdict**

[dict]

---

**Discouraged**

The use of *fontdict* is discouraged. Parameters should be passed as individual keyword arguments or using dictionary-unpacking `set_title(..., **fontdict)`.

---

A dictionary controlling the appearance of the title text, the default *fontdict* is:

```
{'fontsize': rcParams['axes.titlesize'],
 'fontweight': rcParams['axes.titleweight'],
 'color': rcParams['axes.titlecolor'],
 'verticalalignment': 'baseline',
 'horizontalalignment': loc}
```

**loc**

[{'center', 'left', 'right'}, default: *rcParams["axes.titlelocation"]* (default: `'center'`)] Which title to set.

**y**

[float, default: *rcParams["axes.titley"]* (default: None)] Vertical Axes location for the title (1.0 is the top). If None (the default) and *rcParams["axes.titley"]* (default: None) is also None, y is determined automatically to avoid decorators on the Axes.

**pad**

[float, default: *rcParams["axes.titlepad"]* (default: 6.0)] The offset of the title from the top of the Axes, in points.

**Returns**

***Text***

The matplotlib text instance representing the title

**Other Parameters**

****kwargs**

[*Text* properties] Other keyword arguments are text properties, see *Text* for a list of valid text properties.

## Axis scales

| | |
|---|---|
| *set_xscale* | Set the x-axis scale. |
| *set_yscale* | Set the y-axis scale. |
| *set_zscale* | Set the z-axis scale. |
| *get_zscale* | Return the zaxis' scale (as a str). |

### mpl_toolkits.mplot3d.axes3d.Axes3D.set_xscale

Axes3D.**set_xscale**(*value*, ***kwargs*)

Set the x-axis scale.

**Parameters**

**value**

[{"linear"}] The axis scale type to apply. 3D Axes currently only support linear scales; other scales yield nonsensical results.

****kwargs**

Keyword arguments are nominally forwarded to the scale class, but none of them is applicable for linear scales.

### mpl_toolkits.mplot3d.axes3d.Axes3D.set_yscale

Axes3D.**set_yscale**(*value*, ***kwargs*)

Set the y-axis scale.

**Parameters**

**value**

[{"linear"}] The axis scale type to apply. 3D Axes currently only support linear scales; other scales yield nonsensical results.

****kwargs**

Keyword arguments are nominally forwarded to the scale class, but none of them is applicable for linear scales.

### mpl_toolkits.mplot3d.axes3d.Axes3D.set_zscale

Axes3D.**set_zscale**(*value*, ***kwargs*)

Set the z-axis scale.

**Parameters**

**value**

[{"linear"}] The axis scale type to apply. 3D Axes currently only support linear scales; other scales yield nonsensical results.

****kwargs**

Keyword arguments are nominally forwarded to the scale class, but none of them is applicable for linear scales.

### mpl_toolkits.mplot3d.axes3d.Axes3D.get_zscale

Axes3D.**get_zscale**()

Return the zaxis' scale (as a str).

### Autoscaling and margins

| | |
|---|---|
| *get_zmargin* | Retrieve autoscaling margin of the z-axis. |
| *set_zmargin* | Set padding of Z data limits prior to autoscaling. |
| *margins* | Set or retrieve autoscaling margins. |
| *autoscale* | Convenience method for simple axis view autoscaling. |
| *autoscale_view* | Autoscale the view limits using the data limits. |
| *set_autoscalez_on* | Set whether the zaxis is autoscaled when drawing or by *Axes.autoscale_view*. |
| *get_autoscalez_on* | Return whether the zaxis is autoscaled. |
| *auto_scale_xyz* | |

## mpl_toolkits.mplot3d.axes3d.Axes3D.get_zmargin

Axes3D.**get_zmargin**()

Retrieve autoscaling margin of the z-axis.

New in version 3.9.

**Returns**

**zmargin**

[float]

**See also:**

***mpl_toolkits.mplot3d.axes3d.Axes3D.set_zmargin***

## mpl_toolkits.mplot3d.axes3d.Axes3D.set_zmargin

Axes3D.**set_zmargin**(*m*)

Set padding of Z data limits prior to autoscaling.

*m* times the data interval will be added to each end of that interval before it is used in autoscaling. If *m* is negative, this will clip the data range instead of expanding it.

For example, if your data is in the range [0, 2], a margin of 0.1 will result in a range [-0.2, 2.2]; a margin of -0.1 will result in a range of [0.2, 1.8].

**Parameters**

**m**

[float greater than -0.5]

### mpl_toolkits.mplot3d.axes3d.Axes3D.margins

Axes3D.**margins**(**margins*, *x=None*, *y=None*, *z=None*, *tight=True*)

Set or retrieve autoscaling margins.

See *Axes.margins* for full documentation. Because this function applies to 3D Axes, it also takes a *z* argument, and returns `(xmargin, ymargin, zmargin)`.

### mpl_toolkits.mplot3d.axes3d.Axes3D.autoscale

Axes3D.**autoscale**(*enable=True*, *axis='both'*, *tight=None*)

Convenience method for simple axis view autoscaling.

See *Axes.autoscale* for full documentation. Because this function applies to 3D Axes, *axis* can also be set to 'z', and setting *axis* to 'both' autoscales all three axes.

### mpl_toolkits.mplot3d.axes3d.Axes3D.autoscale_view

Axes3D.**autoscale_view**(*tight=None*, *scalex=True*, *scaley=True*, *scalez=True*)

Autoscale the view limits using the data limits.

See *Axes.autoscale_view* for full documentation. Because this function applies to 3D Axes, it also takes a *scalez* argument.

### mpl_toolkits.mplot3d.axes3d.Axes3D.set_autoscalez_on

Axes3D.**set_autoscalez_on**(*b*)

Set whether the zaxis is autoscaled when drawing or by *Axes.autoscale_view*. If b is None, then the value is not changed.

**Parameters**

**b**

[bool]

### mpl_toolkits.mplot3d.axes3d.Axes3D.get_autoscalez_on

Axes3D.**get_autoscalez_on**()

Return whether the zaxis is autoscaled.

### mpl_toolkits.mplot3d.axes3d.Axes3D.auto_scale_xyz

Axes3D.**auto_scale_xyz**(*X*, *Y*, *Z=None*, *had_data=None*)

### Aspect ratio

| | |
|---|---|
| *set_aspect* | Set the aspect ratios. |
| *set_box_aspect* | Set the Axes box aspect. |
| *apply_aspect* | Adjust the Axes for a specified data aspect ratio. |

### mpl_toolkits.mplot3d.axes3d.Axes3D.set_aspect

Axes3D.**set_aspect**(*aspect*, *adjustable=None*, *anchor=None*, *share=False*)

Set the aspect ratios.

**Parameters**

**aspect**

[{'auto', 'equal', 'equalxy', 'equalxz', 'equalyz'}] Possible values:

| value | description |
|---|---|
| 'auto' | automatic; fill the position rectangle with data. |
| 'equal' | adapt all the axes to have equal aspect ratios. |
| 'equalxy' | adapt the x and y axes to have equal aspect ratios. |
| 'equalxz' | adapt the x and z axes to have equal aspect ratios. |
| 'equalyz' | adapt the y and z axes to have equal aspect ratios. |

**adjustable**

[None or {'box', 'datalim'}, optional] If not *None*, this defines which parameter will be adjusted to meet the required aspect. See *set_adjustable* for further details.

**anchor**

[None or str or 2-tuple of float, optional] If not *None*, this defines where the Axes will be drawn if there is extra space due to aspect constraints. The most common way to specify the anchor are abbreviations of cardinal directions:

| value | description |
|---|---|
| 'C' | centered |
| 'SW' | lower left corner |
| 'S' | middle of bottom edge |
| 'SE' | lower right corner |
| etc. | |

See *set_anchor* for further details.

**share**

[bool, default: False] If `True`, apply the settings to all shared Axes.

**See also:**

***mpl_toolkits.mplot3d.axes3d.Axes3D.set_box_aspect***

## mpl_toolkits.mplot3d.axes3d.Axes3D.set_box_aspect

`Axes3D.`**`set_box_aspect`**(*aspect*, *, *zoom=1*)

Set the Axes box aspect.

The box aspect is the ratio of height to width in display units for each face of the box when viewed perpendicular to that face. This is not to be confused with the data aspect (see *set_aspect*). The default ratios are 4:4:3 (x:y:z).

To simulate having equal aspect in data space, set the box aspect to match your data range in each dimension.

*zoom* controls the overall size of the Axes3D in the figure.

**Parameters**

**aspect**

[3-tuple of floats or None] Changes the physical dimensions of the Axes3D, such that the ratio of the axis lengths in display units is x:y:z. If None, defaults to (4, 4, 3).

**zoom**

[float, default: 1] Control overall size of the Axes3D in the figure. Must be > 0.

## mpl_toolkits.mplot3d.axes3d.Axes3D.apply_aspect

Axes3D.**apply_aspect**(*position=None*)

Adjust the Axes for a specified data aspect ratio.

Depending on *get_adjustable* this will modify either the Axes box (position) or the view limits. In the former case, *get_anchor* will affect the position.

**Parameters**

**position**

[None or .Bbox] If not `None`, this defines the position of the Axes within the figure as a Bbox. See *get_position* for further details.

**See also:**

***matplotlib.axes.Axes.set_aspect***

For a description of aspect ratio handling.

***matplotlib.axes.Axes.set_adjustable***

Set how the Axes adjusts to achieve the required aspect ratio.

***matplotlib.axes.Axes.set_anchor***

Set the position in case of extra space.

### Notes

This is called automatically when each Axes is drawn. You may need to call it yourself if you need to update the Axes position and/or view limits before the Figure is drawn.

### Ticks

| | |
|---|---|
| *tick_params* | Convenience method for changing the appearance of ticks and tick labels. |
| *set_zticks* | Set the zaxis' tick locations and optionally tick labels. |
| *get_zticks* | Return the zaxis' tick locations in data coordinates. |
| *set_zticklabels* | [*Discouraged*] Set the zaxis' tick labels with list of string labels. |
| *get_zticklines* | Return the zaxis' tick lines as a list of *Line2D*s. |
| *get_zgridlines* | Return the zaxis' grid lines as a list of *Line2D*s. |
| *get_zminorticklabels* | Return the zaxis' minor tick labels, as a list of *Text*. |
| *get_zmajorticklabels* | Return the zaxis' major tick labels, as a list of *Text*. |
| *zaxis_date* | Set up axis ticks and labels to treat data along the zaxis as dates. |

### mpl_toolkits.mplot3d.axes3d.Axes3D.tick_params

Axes3D.**tick_params**(*axis='both'*, ***kwargs*)

Convenience method for changing the appearance of ticks and tick labels.

See *Axes.tick_params* for full documentation. Because this function applies to 3D Axes, *axis* can also be set to 'z', and setting *axis* to 'both' autoscales all three axes.

Also, because of how Axes3D objects are drawn very differently from regular 2D Axes, some of these settings may have ambiguous meaning. For simplicity, the 'z' axis will accept settings as if it was like the 'y' axis.

**Note:** Axes3D currently ignores some of these settings.

### mpl_toolkits.mplot3d.axes3d.Axes3D.set_zticks

Axes3D.**set_zticks**(*ticks*, *labels=None*, ***, *minor=False*, ***kwargs*)

Set the zaxis' tick locations and optionally tick labels.

If necessary, the view limits of the Axis are expanded so that all given ticks are visible.

**Parameters**

**ticks**

[1D array-like] Array of tick locations (either floats or in axis units). The axis *Locator* is replaced by a *FixedLocator*.

Pass an empty list (`set_ticks([])`) to remove all ticks.

Some tick formatters will not label arbitrary tick positions; e.g. log formatters only label decade ticks by default. In such a case you can set a formatter explicitly on the axis using *Axis.set_major_formatter* or provide formatted *labels* yourself.

**labels**

[list of str, optional] Tick labels for each location in *ticks*; must have the same length as *ticks*. If set, the labels are used as is, via a *FixedFormatter*. If not set, the labels are generated using the axis tick *Formatter*.

**minor**

[bool, default: False] If `False`, set only the major ticks; if `True`, only the minor ticks.

****kwargs**

*Text* properties for the labels. Using these is only allowed if you pass *labels*. In other cases, please use *tick_params*.

### Notes

The mandatory expansion of the view limits is an intentional design choice to prevent the surprise of a non-visible tick. If you need other limits, you should set the limits explicitly after setting the ticks.

## mpl_toolkits.mplot3d.axes3d.Axes3D.get_zticks

Axes3D.**get_zticks**(*, *minor=False*)

Return the zaxis' tick locations in data coordinates.

The locations are not clipped to the current axis limits and hence may contain locations that are not visible in the output.

**Parameters**

**minor**

[bool, default: False] True to return the minor tick directions, False to return the major tick directions.

**Returns**

**array of tick locations**

### mpl_toolkits.mplot3d.axes3d.Axes3D.set_zticklabels

Axes3D.**set_zticklabels**(*labels*, *, *minor=False*, *fontdict=None*, ***kwargs*)

[*Discouraged*] Set the zaxis' tick labels with list of string labels.

---

**Discouraged**

The use of this method is discouraged, because of the dependency on tick positions. In most cases, you'll want to use `Axes.set_[x/y/z]ticks(positions, labels)` or `Axes3D.set_zticks` instead.

If you are using this method, you should always fix the tick positions before, e.g. by using *Axes3D.set_zticks* or by explicitly setting a *FixedLocator*. Otherwise, ticks are free to move and the labels may end up in unexpected positions.

---

**Parameters**

**labels**

[sequence of str or of *Text*s] Texts for labeling each tick location in the sequence set by *Axes3D.set_zticks*; the number of labels must match the number of locations. The labels are used as is, via a *FixedFormatter* (without further formatting).

**minor**

[bool] If True, set minor ticks instead of major ticks.

**fontdict**

[dict, optional]

---

**Discouraged**

The use of *fontdict* is discouraged. Parameters should be passed as individual keyword arguments or using dictionary-unpacking `set_ticklabels(..., **fontdict)`.

---

A dictionary controlling the appearance of the ticklabels. The default *fontdict* is:

```
{'fontsize': rcParams['axes.titlesize'],
 'fontweight': rcParams['axes.titleweight'],
 'verticalalignment': 'baseline',
 'horizontalalignment': loc}
```

****kwargs**

Text properties.

**Warning:** This only sets the properties of the current ticks. Ticks are not guaranteed to be persistent. Various operations can create, delete and modify the Tick instances. There is an imminent risk that these settings can get lost if you work on the figure further (including also panning/zooming on a displayed figure).

Use *set_tick_params* instead if possible.

**Returns**

**list of *Text*s**

For each tick, includes `tick.label1` if it is visible, then `tick.label2` if it is visible, in that order.

### mpl_toolkits.mplot3d.axes3d.Axes3D.get_zticklines

`Axes3D.`**`get_zticklines`**`(`*minor=False*`)`

Return the zaxis' tick lines as a list of *Line2D*s.

### mpl_toolkits.mplot3d.axes3d.Axes3D.get_zgridlines

`Axes3D.`**`get_zgridlines`**`()`

Return the zaxis' grid lines as a list of *Line2D*s.

### mpl_toolkits.mplot3d.axes3d.Axes3D.get_zminorticklabels

`Axes3D.`**`get_zminorticklabels`**`()`

Return the zaxis' minor tick labels, as a list of *Text*.

### mpl_toolkits.mplot3d.axes3d.Axes3D.get_zmajorticklabels

`Axes3D.`**`get_zmajorticklabels`**`()`

Return the zaxis' major tick labels, as a list of *Text*.

### mpl_toolkits.mplot3d.axes3d.Axes3D.zaxis_date

Axes3D.**zaxis_date**(*tz=None*)

Set up axis ticks and labels to treat data along the zaxis as dates.

**Parameters**

**tz**

[str or datetime.tzinfo, default: *rcParams["timezone"]* (default: 'UTC')] The timezone used to create date labels.

#### Notes

This function is merely provided for completeness, but 3D Axes do not support dates for ticks, and so this may not work as expected.

## Units

| | |
|---|---|
| *convert_zunits* | For artists in an Axes, if the zaxis has units support, convert *z* using zaxis unit type |

### mpl_toolkits.mplot3d.axes3d.Axes3D.convert_zunits

Axes3D.**convert_zunits**(*z*)

For artists in an Axes, if the zaxis has units support, convert *z* using zaxis unit type

## Adding artists

| | |
|---|---|
| *add_collection3d* | Add a 3D collection object to the plot. |

### mpl_toolkits.mplot3d.axes3d.Axes3D.add_collection3d

Axes3D.**add_collection3d**(*col*, *zs=0*, *zdir='z'*)

Add a 3D collection object to the plot.

2D collection types are converted to a 3D version by modifying the object and adding z coordinate information, *zs* and *zdir*.

Supported 2D collection types are:

- *PolyCollection*

- *LineCollection*
- *PatchCollection*

### Sharing

| | |
|---|---|
| *sharez* | Share the z-axis with *other*. |
| *shareview* | Share the view angles with *other*. |

### mpl_toolkits.mplot3d.axes3d.Axes3D.sharez

Axes3D.**sharez**(*other*)

Share the z-axis with *other*.

This is equivalent to passing `sharez=other` when constructing the Axes, and cannot be used if the z-axis is already being shared with another Axes.

### mpl_toolkits.mplot3d.axes3d.Axes3D.shareview

Axes3D.**shareview**(*other*)

Share the view angles with *other*.

This is equivalent to passing `shareview=other` when constructing the Axes, and cannot be used if the view angles are already being shared with another Axes.

### Interactive

| | |
|---|---|
| *can_zoom* | Return whether this Axes supports the zoom box button functionality. |
| *can_pan* | Return whether this Axes supports any pan/zoom button functionality. |
| *disable_mouse_rotation* | Disable mouse buttons for 3D rotation, panning, and zooming. |
| *mouse_init* | Set the mouse buttons for 3D rotation and zooming. |
| *drag_pan* | Called when the mouse moves during a pan operation. |
| *format_zdata* | Return *z* string formatted. |
| *format_coord* | Return a string giving the current view rotation angles, or the x, y, z coordinates of the point on the nearest axis pane underneath the mouse cursor, depending on the mouse button pressed. |

### mpl_toolkits.mplot3d.axes3d.Axes3D.can_zoom

Axes3D.**can_zoom**()

Return whether this Axes supports the zoom box button functionality.

### mpl_toolkits.mplot3d.axes3d.Axes3D.can_pan

Axes3D.**can_pan**()

Return whether this Axes supports any pan/zoom button functionality.

### mpl_toolkits.mplot3d.axes3d.Axes3D.disable_mouse_rotation

Axes3D.**disable_mouse_rotation**()

Disable mouse buttons for 3D rotation, panning, and zooming.

### mpl_toolkits.mplot3d.axes3d.Axes3D.mouse_init

Axes3D.**mouse_init**(*rotate_btn=1*, *pan_btn=2*, *zoom_btn=3*)

Set the mouse buttons for 3D rotation and zooming.

**Parameters**

**rotate_btn**

[int or list of int, default: 1] The mouse button or buttons to use for 3D rotation of the Axes.

**pan_btn**

[int or list of int, default: 2] The mouse button or buttons to use to pan the 3D Axes.

**zoom_btn**

[int or list of int, default: 3] The mouse button or buttons to use to zoom the 3D Axes.

### mpl_toolkits.mplot3d.axes3d.Axes3D.drag_pan

Axes3D.**drag_pan**(*button*, *key*, *x*, *y*)

Called when the mouse moves during a pan operation.

**Parameters**

**button**

[*MouseButton*] The pressed mouse button.

**key**

[str or None] The pressed key, if any.

**x, y**

[float] The mouse coordinates in display coords.

### Notes

This is intended to be overridden by new projection types.

## mpl_toolkits.mplot3d.axes3d.Axes3D.format_zdata

Axes3D.**format_zdata**(*z*)

Return *z* string formatted. This function will use the `fmt_zdata` attribute if it is callable, else will fall back on the zaxis major formatter

## mpl_toolkits.mplot3d.axes3d.Axes3D.format_coord

Axes3D.**format_coord**(*xv*, *yv*, *renderer=None*)

Return a string giving the current view rotation angles, or the x, y, z coordinates of the point on the nearest axis pane underneath the mouse cursor, depending on the mouse button pressed.

## Projection and perspective

| | |
|---|---|
| *view_init* | Set the elevation and azimuth of the Axes in degrees (not radians). |
| *set_proj_type* | Set the projection type. |
| *get_proj* | Create the projection matrix from the current viewing position. |
| *set_top_view* | |

## mpl_toolkits.mplot3d.axes3d.Axes3D.view_init

Axes3D.**view_init**(*elev=None*, *azim=None*, *roll=None*, *vertical_axis='z'*, *share=False*)

Set the elevation and azimuth of the Axes in degrees (not radians).

This can be used to rotate the Axes programmatically.

To look normal to the primary planes, the following elevation and azimuth angles can be used. A roll angle of 0, 90, 180, or 270 deg will rotate these views while keeping the axes at right angles.

| view plane | elev | azim |
|---|---|---|
| XY | 90 | -90 |
| XZ | 0 | -90 |
| YZ | 0 | 0 |
| -XY | -90 | 90 |
| -XZ | 0 | 90 |
| -YZ | 0 | 180 |

**Parameters**

**elev**

[float, default: None] The elevation angle in degrees rotates the camera above the plane pierced by the vertical axis, with a positive angle corresponding to a location above that plane. For example, with the default vertical axis of 'z', the elevation defines the angle of the camera location above the x-y plane. If None, then the initial value as specified in the *Axes3D* constructor is used.

**azim**

[float, default: None] The azimuthal angle in degrees rotates the camera about the vertical axis, with a positive angle corresponding to a right-handed rotation. For example, with the default vertical axis of 'z', a positive azimuth rotates the camera about the origin from its location along the +x axis towards the +y axis. If None, then the initial value as specified in the *Axes3D* constructor is used.

**roll**

[float, default: None] The roll angle in degrees rotates the camera about the viewing axis. A positive angle spins the camera clockwise, causing the scene to rotate counter-clockwise. If None, then the initial value as specified in the *Axes3D* constructor is used.

**vertical_axis**

[{"z", "x", "y"}, default: "z"] The axis to align vertically. *azim* rotates about this axis.

**share**

[bool, default: False] If `True`, apply the settings to all Axes with shared views.

### mpl_toolkits.mplot3d.axes3d.Axes3D.set_proj_type

Axes3D.**set_proj_type**(*proj_type*, *focal_length=None*)

Set the projection type.

**Parameters**

**proj_type**

[{'persp', 'ortho'}] The projection type.

**focal_length**

[float, default: None] For a projection type of 'persp', the focal length of the virtual camera. Must be $> 0$. If None, defaults to 1. The focal length can be computed from a desired Field Of View via the equation: focal_length = 1/tan(FOV/2)

### mpl_toolkits.mplot3d.axes3d.Axes3D.get_proj

Axes3D.**get_proj**()

Create the projection matrix from the current viewing position.

### mpl_toolkits.mplot3d.axes3d.Axes3D.set_top_view

Axes3D.**set_top_view**()

## Drawing

| | |
|---|---|
| *draw* | Draw the Artist (and its children) using the given renderer. |
| *get_tightbbox* | Return the tight bounding box of the Axes, including axis and their decorators (xlabel, title, etc). |

### mpl_toolkits.mplot3d.axes3d.Axes3D.draw

Axes3D.**draw**(*renderer*)

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

**Parameters**

**renderer**

[*RendererBase* subclass.]

### Notes

This method is overridden in the Artist subclasses.

## mpl_toolkits.mplot3d.axes3d.Axes3D.get_tightbbox

Axes3D.**get_tightbbox**(*renderer=None*, *, *call_axes_locator=True*, *bbox_extra_artists=None*, *for_layout_only=False*)

Return the tight bounding box of the Axes, including axis and their decorators (xlabel, title, etc).

Artists that have `artist.set_in_layout(False)` are not included in the bbox.

**Parameters**

**renderer**

[*RendererBase* subclass] renderer that will be used to draw the figures (i.e. `fig.canvas.get_renderer()`)

**bbox_extra_artists**

[list of *Artist* or `None`] List of artists to include in the tight bounding box. If `None` (default), then all artist children of the Axes are included in the tight bounding box.

**call_axes_locator**

[bool, default: True] If *call_axes_locator* is `False`, it does not call the `_axes_locator` attribute, which is necessary to get the correct bounding box. `call_axes_locator=False` can be used if the caller is only interested in the relative size of the tightbbox compared to the Axes bbox.

**for_layout_only**

[default: False] The bounding box will *not* include the x-extent of the title and the xlabel, or the y-extent of the ylabel.

**Returns**

***BboxBase***

Bounding box in figure pixel coordinates.

**See also:**

***matplotlib.axes.Axes.get_window_extent***
***matplotlib.axis.Axis.get_tightbbox***
***matplotlib.spines.Spine.get_window_extent***

### Aliases and deprecated methods

| | |
|---|---|
| *set_zlim3d* | Alias for *set_zlim*. |
| *stem3D* | Create a 3D stem plot. |
| *text3D* | Add the text *s* to the 3D Axes at location *x*, *y*, *z* in data coordinates. |

### mpl_toolkits.mplot3d.axes3d.Axes3D.set_zlim3d

Axes3D.**set_zlim3d**(*bottom=None*, *top=None*, ***, *emit=True*, *auto=False*, *view_margin=None*, *zmin=None*, *zmax=None*)

Alias for *set_zlim*.

### mpl_toolkits.mplot3d.axes3d.Axes3D.stem3D

Axes3D.**stem3D**(*x*, *y*, *z*, ***, *linefmt='C0-'*, *markerfmt='C0o'*, *basefmt='C3-'*, *bottom=0*, *label=None*, *orientation='z'*, *data=None*)

Create a 3D stem plot.

A stem plot draws lines perpendicular to a baseline, and places markers at the heads. By default, the baseline is defined by *x* and *y*, and stems are drawn vertically from *bottom* to *z*.

**Parameters**

**x, y, z**

[array-like] The positions of the heads of the stems. The stems are drawn along the *orientation*-direction from the baseline at *bottom* (in the *orientation*-coordinate) to the heads. By default, the *x* and *y* positions are used for the baseline and *z* for the head position, but this can be changed by *orientation*.

**linefmt**

[str, default: 'C0-'] A string defining the properties of the vertical lines. Usually, this will be a color or a color and a linestyle:

| Character | Line Style |
|---|---|
| `'-'` | solid line |
| `'--'` | dashed line |
| `'-.'` | dash-dot line |
| `':'` | dotted line |

Note: While it is technically possible to specify valid formats other than color or color and linestyle (e.g. 'rx' or '-.'), this is beyond the intention of the method and will most likely not result in a reasonable plot.

**markerfmt**

[str, default: 'C0o'] A string defining the properties of the markers at the stem heads.

**basefmt**

[str, default: 'C3-'] A format string defining the properties of the baseline.

**bottom**

[float, default: 0] The position of the baseline, in *orientation*-coordinates.

**label**

[str, optional] The label to use for the stems in legends.

**orientation**

[{'x', 'y', 'z'}, default: 'z'] The direction along which stems are drawn.

**data**

[indexable object, optional] If given, all parameters also accept a string `s`, which is interpreted as `data[s]` (unless this raises an exception).

**Returns**

***StemContainer***

The container may be treated like a tuple (*markerline*, *stemlines*, *baseline*)

### Examples

## mpl_toolkits.mplot3d.axes3d.Axes3D.text3D

`Axes3D.`**`text3D`**(*x*, *y*, *z*, *s*, *zdir=None*, ***kwargs*)

Add the text *s* to the 3D Axes at location *x*, *y*, *z* in data coordinates.

**Parameters**

**x, y, z**

[float] The position to place the text.

**s**

[str] The text.

**zdir**

[{'x', 'y', 'z', 3-tuple}, optional] The direction to be used as the z-direction. Default: 'z'. See *get_dir_vector* for a description of the values.

****kwargs**

Other arguments are forwarded to *matplotlib.axes.Axes.text*.

**Returns**

***Text3D***

The created *Text3D* instance.

## Other

| | |
|---|---|
| *get_axis_position* | |
| *add_contour_set* | |
| *add_contourf_set* | |
| *update_datalim* | Not implemented in *Axes3D*. |

### mpl_toolkits.mplot3d.axes3d.Axes3D.get_axis_position

Axes3D.**get_axis_position**()

### mpl_toolkits.mplot3d.axes3d.Axes3D.add_contour_set

Axes3D.**add_contour_set**(*cset*, *extend3d=False*, *stride=5*, *zdir='z'*, *offset=None*)

### mpl_toolkits.mplot3d.axes3d.Axes3D.add_contourf_set

Axes3D.**add_contourf_set**(*cset*, *zdir='z'*, *offset=None*)

### mpl_toolkits.mplot3d.axes3d.Axes3D.update_datalim

Axes3D.**update_datalim**(*xys*, ***kwargs*)

Not implemented in *Axes3D*.

### Sample 3D data

| | |
|---|---|
| *axes3d.get_test_data* | Return a tuple X, Y, Z with a test data set. |

### mpl_toolkits.mplot3d.axes3d.get_test_data

mpl_toolkits.mplot3d.axes3d.**get_test_data**(*delta=0.05*)

Return a tuple X, Y, Z with a test data set.

### Examples using `mpl_toolkits.mplot3d.axes3d.get_test_data`

- *Plot contour (level) curves in 3D*
- *Plot contour (level) curves in 3D using the extend3d option*
- *Project contour profiles onto a graph*
- *Filled contours*
- *Project filled contour onto a graph*
- *3D plot projection types*
- *Rotating a 3D plot*
- *3D plots as subplots*
- *3D wireframe plot*
- *3D wireframe plots in one direction*
- *plot_wireframe(X, Y, Z)*

**Note:** *pyplot* cannot be used to add content to 3D plots, because its function signatures are strictly 2D and cannot handle the additional information needed for 3D. Instead, use the explicit API by calling the respective methods on the *Axes3D* object.

### axes3d

**Note:** 3D plotting in Matplotlib is still not as mature as the 2D case. Please report any functions that do not behave as expected as a bug. In addition, help and patches would be greatly appreciated!

| | |
|---|---|
| *axes3d.Axes3D*(fig[, rect, elev, azim, roll, ...]) | 3D Axes object. |

### axis3d

**Note:** See `mpl_toolkits.mplot3d.axis3d._axinfo` for a dictionary containing constants that may be modified for controlling the look and feel of mplot3d axes (e.g., label spacing, font colors and panel colors). Historically, axis3d has suffered from having hard-coded constants that precluded user adjustments, and this dictionary was implemented in version 1.1 as a stop-gap measure.

| | |
|---|---|
| *axis3d.Axis*(axes, *[, rotate_label]) | An Axis class for the 3D plots. |

#### mpl_toolkits.mplot3d.axis3d.Axis

**class** mpl_toolkits.mplot3d.axis3d.**Axis**(*axes*, *, *rotate_label=None*, ***kwargs*)

Bases: *XAxis*

An Axis class for the 3D plots.

**Parameters**

**axes**

[*Axes*] The *Axes* to which the created Axis belongs.

**pickradius**

[float] The acceptance radius for containment tests. See also *Axis.contains*.

**clear**

[bool, default: True] Whether to clear the Axis on creation. This is not required, e.g., when creating an Axis as part of an Axes, as `Axes.clear` will call `Axis.clear`. .. versionadded:: 3.8

**active_pane**()

**property adir**

[*Deprecated*]

### Notes

Deprecated since version 3.6:

**property d_interval**

[*Deprecated*]

### Notes

Deprecated since version 3.6: Use get_data_interval instead.

**draw**(*renderer*)

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

**Parameters**

**renderer**

[*RendererBase* subclass.]

### Notes

This method is overridden in the Artist subclasses.

**draw_grid**(*renderer*)

**draw_pane**(*renderer*)

Draw pane.

**Parameters**

**renderer**

[*RendererBase* subclass]

**get_label_position**()

Get the label position.

**Returns**

**str**

[{'lower', 'upper', 'both', 'default', 'none'}] The position of the axis label.

**get_major_ticks**(*numticks=None*)

Return the list of major *Tick*s.

> **Warning:** Ticks are not guaranteed to be persistent. Various operations can create, delete and modify the Tick instances. There is an imminent risk that changes to individual ticks will not survive if you work on the figure further (including also panning/zooming on a displayed figure).
>
> Working on the individual ticks is a method of last resort. Use *set_tick_params* instead if possible.

**get_minor_ticks**(*numticks=None*)

Return the list of minor *Tick*s.

> **Warning:** Ticks are not guaranteed to be persistent. Various operations can create, delete and modify the Tick instances. There is an imminent risk that changes to individual ticks will not survive if you work on the figure further (including also panning/zooming on a displayed figure).
>
> Working on the individual ticks is a method of last resort. Use *set_tick_params* instead if possible.

**get_rotate_label**(*text*)

**get_ticks_position**()

Get the ticks position.

**Returns**

**str**

[{'lower', 'upper', 'both', 'default', 'none'}] The position of the bolded axis lines, ticks, and tick labels.

**get_tightbbox**(*renderer=None*, *, *for_layout_only=False*)

Return a bounding box that encloses the axis. It only accounts tick labels, axis label, and offset-Text.

If *for_layout_only* is True, then the width of the label (if this is an x-axis) or the height of the label (if this is a y-axis) is collapsed to near zero. This allows tight/constrained_layout to ignore too-long labels when doing their layout.

**init3d**()

[*Deprecated*]

### Notes

Deprecated since version 3.6:

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *data_interval=<UNSET>*, *gid=<UNSET>*, *in_layout=<UNSET>*, *inverted=<UNSET>*, *label=<UNSET>*, *label_coords=<UNSET>*, *label_position=<UNSET>*, *label_text=<UNSET>*, *major_formatter=<UNSET>*, *major_locator=<UNSET>*, *minor_formatter=<UNSET>*, *minor_locator=<UNSET>*, *mouseover=<UNSET>*, *pane_color=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *pickradius=<UNSET>*, *rasterized=<UNSET>*, *remove_overlapping_locs=<UNSET>*, *rotate_label=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *tick_params=<UNSET>*, *ticklabels=<UNSET>*, *ticks=<UNSET>*, *ticks_position=<UNSET>*, *transform=<UNSET>*, *units=<UNSET>*, *url=<UNSET>*, *view_interval=<UNSET>*, *visible=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, a |
| *alpha* | scalar or None |
| *animated* | bool |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *data_interval* | unknown |
| *figure* | *Figure* |
| *gid* | str |
| *in_layout* | bool |
| *inverted* | unknown |
| *label* | object |
| *label_coords* | unknown |
| *label_position* | {'lower', 'upper', 'both', 'default', 'none'} |
| *label_text* | str |
| *major_formatter* | *Formatter*, `str`, or function |
| *major_locator* | *Locator* |
| *minor_formatter* | *Formatter*, `str`, or function |
| *minor_locator* | *Locator* |
| *mouseover* | bool |
| *pane_color* | *color* |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *pickradius* | float |
| *rasterized* | bool |
| *remove_overlapping_locs* | unknown |

Table 164 – continued from pre

| Property | Description |
|---|---|
| *rotate_label* | unknown |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *tick_params* | unknown |
| *ticklabels* | sequence of str or of *Text*s |
| *ticks* | 1D array-like |
| *ticks_position* | {'lower', 'upper', 'both', 'default', 'none'} |
| *transform* | *Transform* |
| *units* | units tag |
| *url* | str |
| *view_interval* | unknown |
| *visible* | bool |
| *zorder* | float |

**set_label_position**(*position*)

Set the label position.

**Parameters**

**position**

[{'lower', 'upper', 'both', 'default', 'none'}] The position of the axis label.

**set_pane_color**(*color*, *alpha=None*)

Set pane color.

**Parameters**

**color**

[*color*] Color for axis pane.

**alpha**

[float, optional] Alpha value for axis pane. If None, base it on *color*.

**set_rotate_label**(*val*)

Whether to rotate the axis label: True, False or None. If set to None the label will be rotated if longer than 4 chars.

**set_ticks_position**(*position*)

Set the ticks position.

**Parameters**

**position**

[{'lower', 'upper', 'both', 'default', 'none'}] The position of the bolded axis lines, ticks, and tick labels.

**property v_interval**

[*Deprecated*]

### Notes

Deprecated since version 3.6: Use get_view_interval instead.

## art3d

| | |
|---|---|
| art3d.Line3D(xs, ys, zs, *args, **kwargs) | 3D line object. |
| art3d.Line3DCollection(segments, *[, zorder]) | A collection of 3D lines. |
| art3d.Patch3D(*args[, zs, zdir]) | 3D patch object. |
| art3d.Patch3DCollection(*args[, zs, zdir, ...]) | A collection of 3D patches. |
| art3d.Path3DCollection(*args[, zs, zdir, ...]) | A collection of 3D paths. |
| art3d.PathPatch3D(path, *[, zs, zdir]) | 3D PathPatch object. |
| art3d.Poly3DCollection(verts, *args[, ...]) | A collection of 3D polygons. |
| art3d.Text3D([x, y, z, text, zdir]) | Text object with 3D position and direction. |
| art3d.get_dir_vector(zdir) | Return a direction vector. |
| art3d.juggle_axes(xs, ys, zs, zdir) | Reorder coordinates so that 2D *xs*, *ys* can be plotted in the plane orthogonal to *zdir*. |
| art3d.line_2d_to_3d(line[, zs, zdir]) | Convert a *Line2D* to a *Line3D* object. |
| art3d.line_collection_2d_to_3d(col[, zs, zdir]) | Convert a *LineCollection* to a *Line3DCollection* object. |
| art3d.patch_2d_to_3d(patch[, z, zdir]) | Convert a *Patch* to a *Patch3D* object. |
| art3d.patch_collection_2d_to_3d(col[, zs, ...]) | Convert a *PatchCollection* into a *Patch3DCollection* object (or a *Path-Collection* into a *Path3DCollection* object). |
| art3d.pathpatch_2d_to_3d(pathpatch[, z, zdir]) | Convert a *PathPatch* to a *PathPatch3D* object. |
| art3d.poly_collection_2d_to_3d(col[, zs, zdir]) | Convert a *PolyCollection* into a *Poly3DCollection* object. |
| art3d.rotate_axes(xs, ys, zs, zdir) | Reorder coordinates so that the axes are rotated with *zdir* along the original z axis. |
| art3d.text_2d_to_3d(obj[, z, zdir]) | Convert a *Text* to a *Text3D* object. |

### mpl_toolkits.mplot3d.art3d.Line3D

**class** mpl_toolkits.mplot3d.art3d.**Line3D**(*xs*, *ys*, *zs*, **args*, ***kwargs*)

Bases: *Line2D*

3D line object.

---

**Note:** Use *get_data_3d* to obtain the data associated with the line. *get_data*, *get_xdata*, and *get_ydata* return the x- and y-coordinates of the projected 2D-line, not the x- and y-data of the 3D-line. Similarly, use *set_data_3d* to set the data, not *set_data*, *set_xdata*, and *set_ydata*.

---

**Parameters**

**xs**

[array-like] The x-data to be plotted.

**ys**

[array-like] The y-data to be plotted.

**zs**

[array-like] The z-data to be plotted.

***args, **kwargs**

Additional arguments are passed to *Line2D*.

**draw**(*renderer*)

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

**Parameters**

**renderer**

[*RendererBase* subclass.]

#### Notes

This method is overridden in the Artist subclasses.

**get_data_3d**()

Get the current data

**Returns**

**verts3d**

[length-3 tuple or array-like] The current data as a tuple or array-like.

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *antialiased=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *color=<UNSET>*, *dash_capstyle=<UNSET>*, *dash_joinstyle=<UNSET>*, *dashes=<UNSET>*, *data=<UNSET>*, *data_3d=<UNSET>*, *drawstyle=<UNSET>*, *fillstyle=<UNSET>*, *gapcolor=<UNSET>*, *gid=<UNSET>*, *in_layout=<UNSET>*, *label=<UNSET>*, *linestyle=<UNSET>*, *linewidth=<UNSET>*, *marker=<UNSET>*, *markeredgecolor=<UNSET>*, *markeredgewidth=<UNSET>*, *markerfacecolor=<UNSET>*, *markerfacecoloralt=<UNSET>*, *markersize=<UNSET>*, *markevery=<UNSET>*, *mouseover=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *pickradius=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *solid_capstyle=<UNSET>*, *solid_joinstyle=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *xdata=<UNSET>*, *ydata=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *3d_properties* | float or array of floats |
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* or aa | bool |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* or c | *color* |
| *dash_capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *dash_joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *dashes* | sequence of floats (on/off ink in points) or (None, None) |
| *data* | (2, N) array or two 1D arrays |
| *data_3d* | unknown |
| *drawstyle* or ds | {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'defa |
| *figure* | *Figure* |
| *fillstyle* | {'full', 'left', 'right', 'bottom', 'top', 'none'} |
| *gapcolor* | *color* or None |
| *gid* | str |
| *in_layout* | bool |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float |
| *marker* | marker style string, *Path* or *MarkerStyle* |
| *markeredgecolor* or mec | *color* |

Table 165 – continued from pr

| Property | Description |
|---|---|
| markeredgewidth or mew | float |
| markerfacecolor or mfc | color |
| markerfacecoloralt or mfcalt | color |
| markersize or ms | float |
| markevery | None or int or (int, int) or slice or list[int] or float or (float, float) o |
| mouseover | bool |
| path_effects | list of AbstractPathEffect |
| picker | float or callable[[Artist, Event], tuple[bool, dict]] |
| pickradius | float |
| rasterized | bool |
| sketch_params | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| solid_capstyle | CapStyle or {'butt', 'projecting', 'round'} |
| solid_joinstyle | JoinStyle or {'miter', 'round', 'bevel'} |
| transform | Transform |
| url | str |
| visible | bool |
| xdata | 1D array |
| ydata | 1D array |
| zorder | float |

**set_3d_properties**(*zs=0*, *zdir='z'*)

Set the *z* position and direction of the line.

**Parameters**

**zs**

[float or array of floats] The location along the *zdir* axis in 3D space to position the line.

**zdir**

[{'x', 'y', 'z'}] Plane to plot line orthogonal to. Default: 'z'. See *get_dir_vector* for a description of the values.

**set_data_3d**(**args*)

Set the x, y and z data

**Parameters**

**x**

[array-like] The x-data to be plotted.

**y**

[array-like] The y-data to be plotted.

**z**

[array-like] The z-data to be plotted.

### Notes

Accepts x, y, z arguments or a single array-like (x, y, z)

## Examples using `mpl_toolkits.mplot3d.art3d.Line3D`

- *3D stem*

## mpl_toolkits.mplot3d.art3d.Line3DCollection

**class** mpl_toolkits.mplot3d.art3d.**Line3DCollection**(*segments*, *, *zorder=2*, ***kwargs*)

Bases: *LineCollection*

A collection of 3D lines.

**Parameters**

**segments**

[list of array-like] A sequence (*line0*, *line1*, *line2*) of lines, where each line is a list of points:

```
lineN = [(x0, y0), (x1, y1), ... (xm, ym)]
```

or the equivalent Mx2 numpy array with two columns. Each line can have a different number of segments.

**linewidths**

[float or list of float, default: *rcParams["lines.linewidth"]* (default: `1.5`)] The width of each line in points.

**colors**

[*color* or list of color, default: *rcParams["lines.color"]* (default: `'C0'`)] A sequence of RGBA tuples (e.g., arbitrary color strings, etc, not allowed).

**antialiaseds**

[bool or list of bool, default: *rcParams["lines.antialiased"]* (default: `True`)] Whether to use antialiasing for each line.

**zorder**

[float, default: 2] zorder of the lines once drawn.

**facecolors**

[*color* or list of *color*, default: 'none'] When setting *facecolors*, each line is interpreted as a boundary for an area, implicitly closing the path from the last point to the first point. The enclosed area is filled with *facecolor*. In order to manually specify what should count as the "interior" of each line, please use `PathCollection` instead, where the "interior" can be specified by appropriate usage of `CLOSEPOLY`.

****kwargs**

Forwarded to `Collection`.

**do_3d_projection**()

Project the points according to renderer matrix.

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *antialiased=<UNSET>*, *array=<UNSET>*, *capstyle=<UNSET>*, *clim=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *cmap=<UNSET>*, *color=<UNSET>*, *colors=<UNSET>*, *edgecolor=<UNSET>*, *facecolor=<UNSET>*, *gapcolor=<UNSET>*, *gid=<UNSET>*, *hatch=<UNSET>*, *in_layout=<UNSET>*, *joinstyle=<UNSET>*, *label=<UNSET>*, *linestyle=<UNSET>*, *linewidth=<UNSET>*, *mouseover=<UNSET>*, *norm=<UNSET>*, *offset_transform=<UNSET>*, *offsets=<UNSET>*, *path_effects=<UNSET>*, *paths=<UNSET>*, *picker=<UNSET>*, *pickradius=<UNSET>*, *rasterized=<UNSET>*, *segments=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *sort_zpos=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *urls=<UNSET>*, *verts=<UNSET>*, *visible=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| `agg_filter` | a filter function, which takes a (m, n, 3) float array and a dpi v |
| `alpha` | array-like or scalar or None |
| `animated` | bool |
| `antialiased` or aa or antialiaseds | bool or list of bools |
| `array` | array-like or None |
| `capstyle` | `CapStyle` or {'butt', 'projecting', 'round'} |
| `clim` | (vmin: float, vmax: float) |
| `clip_box` | `BboxBase` or None |
| `clip_on` | bool |
| `clip_path` | Patch or (Path, Transform) or None |
| `cmap` | `Colormap` or str or None |
| `color` | *color* or list of *color* |
| `colors` | *color* or list of *color* |
| `edgecolor` or ec or edgecolors | *color* or list of *color* or 'face' |
| `facecolor` or facecolors or fc | *color* or list of *color* |
| `figure` | `Figure` |

Table 166 – continued from

| Property | Description |
|---|---|
| *gapcolor* | *color* or list of *color* or None |
| *gid* | str |
| *hatch* | {'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or dashes or linestyles or ls | str or tuple or list thereof |
| *linewidth* or linewidths or lw | float or list of floats |
| *mouseover* | bool |
| *norm* | *Normalize* or str or None |
| *offset_transform* or transOffset | *Transform* |
| *offsets* | (N, 2) or (2,) array-like |
| *path_effects* | list of *AbstractPathEffect* |
| *paths* | unknown |
| *picker* | None or bool or float or callable |
| *pickradius* | float |
| *rasterized* | bool |
| *segments* | unknown |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *sort_zpos* | unknown |
| *transform* | *Transform* |
| *url* | str |
| *urls* | list of str or None |
| *verts* | unknown |
| *visible* | bool |
| *zorder* | float |

**set_segments**(*segments*)

Set 3D segments.

**set_sort_zpos**(*val*)

Set the position to use for z-sorting.

### Examples using `mpl_toolkits.mplot3d.art3d.Line3DCollection`

- *3D stem*

## mpl_toolkits.mplot3d.art3d.Patch3D

**class** mpl_toolkits.mplot3d.art3d.**Patch3D**(**args*, *zs=()*, *zdir='z'*, ***kwargs*)

Bases: *Patch*

3D patch object.

**Parameters**

**verts**

**zs**

[float] The location along the *zdir* axis in 3D space to position the patch.

**zdir**

[{'x', 'y', 'z'}] Plane to plot patch orthogonal to. Default: 'z'. See *get_dir_vector* for a description of the values.

**do_3d_projection**()

**get_path**()

Return the path of this patch.

**set**(***, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *antialiased=<UNSET>*, *capstyle=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *color=<UNSET>*, *edgecolor=<UNSET>*, *facecolor=<UNSET>*, *fill=<UNSET>*, *gid=<UNSET>*, *hatch=<UNSET>*, *in_layout=<UNSET>*, *joinstyle=<UNSET>*, *label=<UNSET>*, *linestyle=<UNSET>*, *linewidth=<UNSET>*, *mouseover=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *3d_properties* | unknown |
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* or aa | bool or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clip_box* | *BboxBase* or None |

Table 167 – continued from previous

| Property | Description |
|---|---|
| clip_on | bool |
| clip_path | Patch or (Path, Transform) or None |
| color | color |
| edgecolor or ec | color or None |
| facecolor or fc | color or None |
| figure | Figure |
| fill | bool |
| gid | str |
| hatch | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| in_layout | bool |
| joinstyle | JoinStyle or {'miter', 'round', 'bevel'} |
| label | object |
| linestyle or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| linewidth or lw | float or None |
| mouseover | bool |
| path_effects | list of AbstractPathEffect |
| picker | None or bool or float or callable |
| rasterized | bool |
| sketch_params | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| transform | Transform |
| url | str |
| visible | bool |
| zorder | float |

**set_3d_properties**(*verts*, *zs=0*, *zdir='z'*)

Set the *z* position and direction of the patch.

**Parameters**

**verts**

**zs**

[float] The location along the *zdir* axis in 3D space to position the patch.

**zdir**

[{'x', 'y', 'z'}] Plane to plot patch orthogonal to. Default: 'z'. See get_dir_vector for a description of the values.

### mpl_toolkits.mplot3d.art3d.Patch3DCollection

**class** mpl_toolkits.mplot3d.art3d.**Patch3DCollection**(**args*, *zs=0*, *zdir='z'*, *depthshade=True*, ***kwargs*)

Bases: *PatchCollection*

A collection of 3D patches.

Create a collection of flat 3D patches with its normal vector pointed in *zdir* direction, and located at *zs* on the *zdir* axis. 'zs' can be a scalar or an array-like of the same length as the number of patches in the collection.

Constructor arguments are the same as for *PatchCollection*. In addition, keywords *zs=0* and *zdir='z'* are available.

Also, the keyword argument *depthshade* is available to indicate whether to shade the patches in order to give the appearance of depth (default is *True*). This is typically desired in scatter plots.

**do_3d_projection**()

**get_depthshade**()

**get_edgecolor**()

**get_facecolor**()

**set**(***, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *antialiased=<UNSET>*, *array=<UNSET>*, *capstyle=<UNSET>*, *clim=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *cmap=<UNSET>*, *color=<UNSET>*, *depthshade=<UNSET>*, *edgecolor=<UNSET>*, *facecolor=<UNSET>*, *gid=<UNSET>*, *hatch=<UNSET>*, *in_layout=<UNSET>*, *joinstyle=<UNSET>*, *label=<UNSET>*, *linestyle=<UNSET>*, *linewidth=<UNSET>*, *mouseover=<UNSET>*, *norm=<UNSET>*, *offset_transform=<UNSET>*, *offsets=<UNSET>*, *path_effects=<UNSET>*, *paths=<UNSET>*, *picker=<UNSET>*, *pickradius=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *sort_zpos=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *urls=<UNSET>*, *visible=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *3d_properties* | float or array of floats |
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi v |
| *alpha* | array-like or scalar or None |
| *animated* | bool |
| *antialiased* or aa or antialiaseds | bool or list of bools |
| *array* | array-like or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clim* | (vmin: float, vmax: float) |

Table 168 – continued from

| Property | Description |
|---|---|
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *cmap* | *Colormap* or str or None |
| *color* | *color* or list of RGBA tuples |
| *depthshade* | bool |
| *edgecolor* or ec or edgecolors | *color* or list of *color* or 'face' |
| *facecolor* or facecolors or fc | *color* or list of *color* |
| *figure* | *Figure* |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or dashes or linestyles or ls | str or tuple or list thereof |
| *linewidth* or linewidths or lw | float or list of floats |
| *mouseover* | bool |
| *norm* | *Normalize* or str or None |
| *offset_transform* or transOffset | *Transform* |
| *offsets* | (N, 2) or (2,) array-like |
| *path_effects* | list of *AbstractPathEffect* |
| *paths* | unknown |
| *picker* | None or bool or float or callable |
| *pickradius* | float |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *sort_zpos* | unknown |
| *transform* | *Transform* |
| *url* | str |
| *urls* | list of str or None |
| *visible* | bool |
| *zorder* | float |

**set_3d_properties**(*zs*, *zdir*)

Set the *z* positions and direction of the patches.

**Parameters**

**zs**

[float or array of floats] The location or locations to place the patches in the collection along the *zdir* axis.

**zdir**

[{'x', 'y', 'z'}] Plane to plot patches orthogonal to. All patches must have the same direction. See *get_dir_vector* for a description of the values.

**set_depthshade**(*depthshade*)

Set whether depth shading is performed on collection members.

**Parameters**

**depthshade**

[bool] Whether to shade the patches in order to give the appearance of depth.

**set_sort_zpos**(*val*)

Set the position to use for z-sorting.

### mpl_toolkits.mplot3d.art3d.Path3DCollection

**class** mpl_toolkits.mplot3d.art3d.**Path3DCollection**(**args*, *zs=0*, *zdir='z'*, *depthshade=True*, ***kwargs*)

Bases: *PathCollection*

A collection of 3D paths.

Create a collection of flat 3D paths with its normal vector pointed in *zdir* direction, and located at *zs* on the *zdir* axis. 'zs' can be a scalar or an array-like of the same length as the number of paths in the collection.

Constructor arguments are the same as for *PathCollection*. In addition, keywords *zs=0* and *zdir='z'* are available.

Also, the keyword argument *depthshade* is available to indicate whether to shade the patches in order to give the appearance of depth (default is *True*). This is typically desired in scatter plots.

**do_3d_projection**()

**draw**(*renderer*)

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

**Parameters**

**renderer**

[*RendererBase* subclass.]

### Notes

This method is overridden in the Artist subclasses.

**get_depthshade**()

**get_edgecolor**()

**get_facecolor**()

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *antialiased=<UNSET>*, *array=<UNSET>*, *capstyle=<UNSET>*, *clim=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *cmap=<UNSET>*, *color=<UNSET>*, *depthshade=<UNSET>*, *edgecolor=<UNSET>*, *facecolor=<UNSET>*, *gid=<UNSET>*, *hatch=<UNSET>*, *in_layout=<UNSET>*, *joinstyle=<UNSET>*, *label=<UNSET>*, *linestyle=<UNSET>*, *linewidth=<UNSET>*, *mouseover=<UNSET>*, *norm=<UNSET>*, *offset_transform=<UNSET>*, *offsets=<UNSET>*, *path_effects=<UNSET>*, *paths=<UNSET>*, *picker=<UNSET>*, *pickradius=<UNSET>*, *rasterized=<UNSET>*, *sizes=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *sort_zpos=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *urls=<UNSET>*, *visible=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *3d_properties* | float or array of floats |
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi v |
| *alpha* | array-like or scalar or None |
| *animated* | bool |
| *antialiased* or aa or antialiaseds | bool or list of bools |
| *array* | array-like or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clim* | (vmin: float, vmax: float) |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *cmap* | *Colormap* or str or None |
| *color* | *color* or list of RGBA tuples |
| *depthshade* | bool |
| *edgecolor* or ec or edgecolors | *color* or list of *color* or 'face' |
| *facecolor* or facecolors or fc | *color* or list of *color* |
| *figure* | *Figure* |
| *gid* | str |
| *hatch* | {'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |

Table 169 – continued from

| Property | Description |
|---|---|
| label | object |
| linestyle or dashes or linestyles or ls | str or tuple or list thereof |
| linewidth or linewidths or lw | unknown |
| mouseover | bool |
| norm | Normalize or str or None |
| offset_transform or transOffset | Transform |
| offsets | (N, 2) or (2,) array-like |
| path_effects | list of AbstractPathEffect |
| paths | unknown |
| picker | None or bool or float or callable |
| pickradius | float |
| rasterized | bool |
| sizes | unknown |
| sketch_params | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| sort_zpos | unknown |
| transform | Transform |
| url | str |
| urls | list of str or None |
| visible | bool |
| zorder | float |

**set_3d_properties**(*zs*, *zdir*)

Set the *z* positions and direction of the paths.

**Parameters**

**zs**

[float or array of floats] The location or locations to place the paths in the collection along the *zdir* axis.

**zdir**

[{'x', 'y', 'z'}] Plane to plot paths orthogonal to. All paths must have the same direction. See get_dir_vector for a description of the values.

**set_depthshade**(*depthshade*)

Set whether depth shading is performed on collection members.

**Parameters**

**depthshade**

[bool] Whether to shade the patches in order to give the appearance of depth.

**set_linewidth**(*lw*)

Set the linewidth(s) for the collection. *lw* can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

**Parameters**

**lw**

[float or list of floats]

**set_sizes**(*sizes*, *dpi=72.0*)

Set the sizes of each member of the collection.

**Parameters**

**sizes**

[numpy.ndarray or None] The size to set for each element of the collection. The value is the 'area' of the element.

**dpi**

[float, default: 72] The dpi of the canvas.

**set_sort_zpos**(*val*)

Set the position to use for z-sorting.

### mpl_toolkits.mplot3d.art3d.PathPatch3D

**class** mpl_toolkits.mplot3d.art3d.**PathPatch3D**(*path*, *, *zs=()*, *zdir='z'*, ***kwargs*)

Bases: *Patch3D*

3D PathPatch object.

**Parameters**

**path**

**zs**

[float] The location along the *zdir* axis in 3D space to position the path patch.

**zdir**

[{'x', 'y', 'z', 3-tuple}] Plane to plot path patch orthogonal to. Default: 'z'. See *get_dir_vector* for a description of the values.

**do_3d_projection**()

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *antialiased=<UNSET>*, *capstyle=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *color=<UNSET>*, *edgecolor=<UNSET>*, *facecolor=<UNSET>*, *fill=<UNSET>*, *gid=<UNSET>*, *hatch=<UNSET>*, *in_layout=<UNSET>*, *joinstyle=<UNSET>*, *label=<UNSET>*, *linestyle=<UNSET>*, *linewidth=<UNSET>*, *mouseover=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *3d_properties* | unknown |
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* or aa | bool or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* | *color* |
| *edgecolor* or ec | *color* or None |
| *facecolor* or fc | *color* or None |
| *figure* | *Figure* |
| *fill* | bool |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float or None |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *zorder* | float |

**set_3d_properties**(*path*, *zs=0*, *zdir='z'*)

Set the *z* position and direction of the path patch.

**Parameters**

**path**

**zs**

[float] The location along the *zdir* axis in 3D space to position the path patch.

**zdir**

[{'x', 'y', 'z', 3-tuple}] Plane to plot path patch orthogonal to. Default: 'z'. See *get_dir_vector* for a description of the values.

### Examples using mpl_toolkits.mplot3d.art3d.PathPatch3D

- *Draw flat objects in 3D plot*

## mpl_toolkits.mplot3d.art3d.Poly3DCollection

**class** mpl_toolkits.mplot3d.art3d.**Poly3DCollection**(*verts*, **args*, *zsort='average'*, *shade=False*, *lightsource=None*, ***kwargs*)

Bases: *PolyCollection*

A collection of 3D polygons.

---

**Note: Filling of 3D polygons**

There is no simple definition of the enclosed surface of a 3D polygon unless the polygon is planar.

In practice, Matplotlib fills the 2D projection of the polygon. This gives a correct filling appearance only for planar polygons. For all other polygons, you'll find orientations in which the edges of the polygon intersect in the projection. This will lead to an incorrect visualization of the 3D area.

If you need filled areas, it is recommended to create them via *plot_trisurf*, which creates a triangulation and thus generates consistent surfaces.

---

**Parameters**

**verts**

[list of (N, 3) array-like] The sequence of polygons [*verts0*, *verts1*, ...] where each element *verts_i* defines the vertices of polygon *i* as a 2D array-like of shape (N, 3).

**zsort**

[{'average', 'min', 'max'}, default: 'average'] The calculation method for the z-order. See *set_zsort* for details.

**shade**

[bool, default: False] Whether to shade *facecolors* and *edgecolors*. When activating *shade*, *facecolors* and/or *edgecolors* must be provided.

New in version 3.7.

**lightsource**

[*LightSource*, optional] The lightsource to use when *shade* is True.

New in version 3.7.

***args, **kwargs**

All other parameters are forwarded to *PolyCollection*.

## Notes

Note that this class does a bit of magic with the _facecolors and _edgecolors properties.

**do_3d_projection**()

Perform the 3D projection for this object.

**get_edgecolor**()

**get_facecolor**()

**get_vector**(*segments3d*)

Optimize points for projection.

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *antialiased=<UNSET>*, *array=<UNSET>*, *capstyle=<UNSET>*, *clim=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *cmap=<UNSET>*, *color=<UNSET>*, *edgecolor=<UNSET>*, *facecolor=<UNSET>*, *gid=<UNSET>*, *hatch=<UNSET>*, *in_layout=<UNSET>*, *joinstyle=<UNSET>*, *label=<UNSET>*, *linestyle=<UNSET>*, *linewidth=<UNSET>*, *mouseover=<UNSET>*, *norm=<UNSET>*, *offset_transform=<UNSET>*, *offsets=<UNSET>*, *path_effects=<UNSET>*, *paths=<UNSET>*, *picker=<UNSET>*, *pickradius=<UNSET>*, *rasterized=<UNSET>*, *sizes=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *sort_zpos=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *urls=<UNSET>*, *verts=<UNSET>*, *verts_and_codes=<UNSET>*, *visible=<UNSET>*, *zorder=<UNSET>*, *zsort=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi v |
| *alpha* | unknown |
| *animated* | bool |
| *antialiased* or aa or antialiaseds | bool or list of bools |
| *array* | array-like or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clim* | (vmin: float, vmax: float) |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *cmap* | *Colormap* or str or None |
| *color* | *color* or list of RGBA tuples |
| *edgecolor* or ec or edgecolors | unknown |
| *facecolor* or facecolors or fc | unknown |
| *figure* | *Figure* |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or dashes or linestyles or ls | str or tuple or list thereof |
| *linewidth* or linewidths or lw | float or list of floats |
| *mouseover* | bool |
| *norm* | *Normalize* or str or None |
| *offset_transform* or transOffset | *Transform* |
| *offsets* | (N, 2) or (2,) array-like |
| *path_effects* | list of *AbstractPathEffect* |
| *paths* | list of array-like |
| *picker* | None or bool or float or callable |
| *pickradius* | float |
| *rasterized* | bool |
| sizes | numpy.ndarray or None |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *sort_zpos* | unknown |
| *transform* | *Transform* |
| *url* | str |
| *urls* | list of str or None |
| *verts* | list of (N, 3) array-like |
| *verts_and_codes* | unknown |
| *visible* | bool |
| *zorder* | float |
| *zsort* | {'average', 'min', 'max'} |

**set_3d_properties**()

**set_alpha**(*alpha*)

Set the alpha value used for blending - not supported on all backends.

**Parameters**

**alpha**

[array-like or scalar or None] All values must be within the 0-1 range, inclusive. Masked values and nans are not supported.

**set_edgecolor**(*colors*)

Set the edgecolor(s) of the collection.

**Parameters**

**c**

[*color* or list of *color* or 'face'] The collection edgecolor(s). If a sequence, the patches cycle through it. If 'face', match the facecolor.

**set_facecolor**(*colors*)

Set the facecolor(s) of the collection. *c* can be a color (all patches have same color), or a sequence of colors; if it is a sequence the patches will cycle through the sequence.

If *c* is 'none', the patch will not be filled.

**Parameters**

**c**

[*color* or list of *color*]

**set_sort_zpos**(*val*)

Set the position to use for z-sorting.

**set_verts**(*verts*, *closed=True*)

Set 3D vertices.

**Parameters**

**verts**

[list of (N, 3) array-like] The sequence of polygons [*verts0*, *verts1*, ...] where each element *verts_i* defines the vertices of polygon *i* as a 2D array-like of shape (N, 3).

**closed**

[bool, default: True] Whether the polygon should be closed by adding a CLOSEPOLY connection at the end.

**set_verts_and_codes**(*verts*, *codes*)

Set 3D vertices with path codes.

**set_zsort**(*zsort*)

Set the calculation method for the z-order.

**Parameters**

**zsort**

[{'average', 'min', 'max'}] The function applied on the z-coordinates of the vertices in the viewer's coordinate system, to determine the z-order.

### Examples using `mpl_toolkits.mplot3d.art3d.Poly3DCollection`

- *Custom hillshading in a 3D surface plot*
- *2D and 3D Axes in same figure*
- *Generate polygons to fill under 3D line graph*
- *3D plots as subplots*
- *3D surface (colormap)*
- *3D surface (checkerboard)*

## mpl_toolkits.mplot3d.art3d.Text3D

**class** mpl_toolkits.mplot3d.art3d.**Text3D**(*x=0*, *y=0*, *z=0*, *text=''*, *zdir='z'*, ***kwargs*)

Bases: *Text*

Text object with 3D position and direction.

**Parameters**

**x, y, z**

[float] The position of the text.

**text**

[str] The text string to display.

**zdir**

[{'x', 'y', 'z', None, 3-tuple}] The direction of the text. See *get_dir_vector* for a description of the values.

**Other Parameters**

****kwargs**

All other parameters are passed on to *Text*.

Create a *Text* instance at *x*, *y* with string *text*.

The text is aligned relative to the anchor point (*x*, *y*) according to `horizontalalignment` (default: 'left') and `verticalalignment` (default: 'baseline'). See also *Text alignment*.

While Text accepts the 'label' keyword argument, by default it is not added to the handles of a legend.

Valid keyword arguments are:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* | bool |
| *backgroundcolor* | *color* |
| *bbox* | dict with properties for *patches.FancyBboxPatch* |
| *clip_box* | unknown |
| *clip_on* | unknown |
| *clip_path* | unknown |
| *color* or c | *color* |
| *figure* | *Figure* |
| *fontfamily* or family or fontname | {FONTNAME, 'serif', 'sans-serif', 'cursive', 'fantasy', 'monosp |
| *fontproperties* or font or font_properties | *font_manager.FontProperties* or *str* or *pathli* |
| *fontsize* or size | float or {'xx-small', 'x-small', 'small', 'medium', 'large', 'x-larg |
| *fontstretch* or stretch | {a numeric value in range 0-1000, 'ultra-condensed', 'extra-co |
| *fontstyle* or style | {'normal', 'italic', 'oblique'} |
| *fontvariant* or variant | {'normal', 'small-caps'} |
| *fontweight* or weight | {a numeric value in range 0-1000, 'ultralight', 'light', 'normal' |
| *gid* | str |
| *horizontalalignment* or ha | {'left', 'center', 'right'} |
| *in_layout* | bool |
| *label* | object |
| *linespacing* | float (multiple of font size) |
| *math_fontfamily* | str |
| *mouseover* | bool |
| *multialignment* or ma | {'left', 'right', 'center'} |
| *parse_math* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *position* | (float, float) |
| *rasterized* | bool |
| *rotation* | float or {'vertical', 'horizontal'} |
| *rotation_mode* | {None, 'default', 'anchor'} |
| *sketch_params* | (scale: float, length: float, randomness: float) |

Table 172 – continu

| Property | Description |
|---|---|
| *snap* | bool or None |
| *text* | object |
| *transform* | *Transform* |
| *transform_rotates_text* | bool |
| *url* | str |
| *usetex* | bool, default: *rcParams["text.usetex"]* (default: Fa |
| *verticalalignment* or va | {'baseline', 'bottom', 'center', 'center_baseline', 'top'} |
| *visible* | bool |
| *wrap* | bool |
| *x* | float |
| *y* | float |
| *zorder* | float |

**draw**(*renderer*)

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

**Parameters**

**renderer**

[*RendererBase* subclass.]

### Notes

This method is overridden in the Artist subclasses.

**get_position_3d**()

Return the (x, y, z) position of the text.

**get_tightbbox**(*renderer=None*)

Like *Artist.get_window_extent*, but includes any clipping.

**Parameters**

**renderer**

[*RendererBase* subclass, optional] renderer that will be used to draw the figures (i.e. `fig.canvas.get_renderer()`)

**Returns**

***Bbox*** **or None**

The enclosing bounding box (in figure pixel coordinates). Returns None if clipping results in no intersection.

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *antialiased=<UNSET>*, *backgroundcolor=<UNSET>*, *bbox=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *color=<UNSET>*, *fontfamily=<UNSET>*, *fontproperties=<UNSET>*, *fontsize=<UNSET>*, *fontstretch=<UNSET>*, *fontstyle=<UNSET>*, *fontvariant=<UNSET>*, *fontweight=<UNSET>*, *gid=<UNSET>*, *horizontalalignment=<UNSET>*, *in_layout=<UNSET>*, *label=<UNSET>*, *linespacing=<UNSET>*, *math_fontfamily=<UNSET>*, *mouseover=<UNSET>*, *multialignment=<UNSET>*, *parse_math=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *position=<UNSET>*, *position_3d=<UNSET>*, *rasterized=<UNSET>*, *rotation=<UNSET>*, *rotation_mode=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *text=<UNSET>*, *transform=<UNSET>*, *transform_rotates_text=<UNSET>*, *url=<UNSET>*, *usetex=<UNSET>*, *verticalalignment=<UNSET>*, *visible=<UNSET>*, *wrap=<UNSET>*, *x=<UNSET>*, *y=<UNSET>*, *z=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *3d_properties* | float |
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* | bool |
| *backgroundcolor* | *color* |
| *bbox* | dict with properties for *patches.FancyBboxPatch* |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* or c | *color* |
| *figure* | *Figure* |
| *fontfamily* or family or fontname | {FONTNAME, 'serif', 'sans-serif', 'cursive', 'fantasy', 'mo |
| *fontproperties* or font or font_properties | *font_manager.FontProperties* or *str* or *pat* |
| *fontsize* or size | float or {'xx-small', 'x-small', 'small', 'medium', 'large', 'x- |
| *fontstretch* or stretch | {a numeric value in range 0-1000, 'ultra-condensed', 'ext |
| *fontstyle* or style | {'normal', 'italic', 'oblique'} |
| *fontvariant* or variant | {'normal', 'small-caps'} |
| *fontweight* or weight | {a numeric value in range 0-1000, 'ultralight', 'light', 'nor |
| *gid* | str |
| *horizontalalignment* or ha | {'left', 'center', 'right'} |
| *in_layout* | bool |
| *label* | object |
| *linespacing* | float (multiple of font size) |
| *math_fontfamily* | str |

Table 173 – co

| Property | Description |
|---|---|
| *mouseover* | bool |
| *multialignment* or ma | {'left', 'right', 'center'} |
| *parse_math* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *position* | (float, float) |
| *position_3d* | (float, float, float) |
| *rasterized* | bool |
| *rotation* | float or {'vertical', 'horizontal'} |
| *rotation_mode* | {None, 'default', 'anchor'} |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *text* | object |
| *transform* | *Transform* |
| *transform_rotates_text* | bool |
| *url* | str |
| *usetex* | bool, default: *rcParams["text.usetex"]* (defaul |
| *verticalalignment* or va | {'baseline', 'bottom', 'center', 'center_baseline', 'top'} |
| *visible* | bool |
| *wrap* | bool |
| *x* | float |
| *y* | float |
| *z* | float |
| *zorder* | float |

**set_3d_properties**(*z=0*, *zdir='z'*)

Set the *z* position and direction of the text.

**Parameters**

**z**

[float] The z-position in 3D space.

**zdir**

[{'x', 'y', 'z', 3-tuple}] The direction of the text. Default: 'z'. See *get_dir_vector* for a description of the values.

**set_position_3d**(*xyz*, *zdir=None*)

Set the (*x*, *y*, *z*) position of the text.

**Parameters**

**xyz**

[(float, float, float)] The position in 3D space.

**zdir**

[{'x', 'y', 'z', None, 3-tuple}] The direction of the text. If unspecified, the *zdir* will not be changed. See *get_dir_vector* for a description of the values.

**set_z**(*z*)

Set the *z* position of the text.

**Parameters**

**z**

[float]

## mpl_toolkits.mplot3d.art3d.get_dir_vector

mpl_toolkits.mplot3d.art3d.**get_dir_vector**(*zdir*)

Return a direction vector.

**Parameters**

**zdir**

[{'x', 'y', 'z', None, 3-tuple}] The direction. Possible values are:

- 'x': equivalent to (1, 0, 0)
- 'y': equivalent to (0, 1, 0)
- 'z': equivalent to (0, 0, 1)
- *None*: equivalent to (0, 0, 0)
- an iterable (x, y, z) is converted to an array

**Returns**

**x, y, z**

[array] The direction vector.

### mpl_toolkits.mplot3d.art3d.juggle_axes

mpl_toolkits.mplot3d.art3d.**juggle_axes**(*xs*, *ys*, *zs*, *zdir*)

Reorder coordinates so that 2D *xs*, *ys* can be plotted in the plane orthogonal to *zdir*. *zdir* is normally 'x', 'y' or 'z'. However, if *zdir* starts with a '-' it is interpreted as a compensation for *rotate_axes*.

### mpl_toolkits.mplot3d.art3d.line_2d_to_3d

mpl_toolkits.mplot3d.art3d.**line_2d_to_3d**(*line*, *zs=0*, *zdir='z'*)

Convert a *Line2D* to a *Line3D* object.

**Parameters**

**zs**

[float] The location along the *zdir* axis in 3D space to position the line.

**zdir**

[{'x', 'y', 'z'}] Plane to plot line orthogonal to. Default: 'z'. See *get_dir_vector* for a description of the values.

### mpl_toolkits.mplot3d.art3d.line_collection_2d_to_3d

mpl_toolkits.mplot3d.art3d.**line_collection_2d_to_3d**(*col*, *zs=0*, *zdir='z'*)

Convert a *LineCollection* to a *Line3DCollection* object.

### mpl_toolkits.mplot3d.art3d.patch_2d_to_3d

mpl_toolkits.mplot3d.art3d.**patch_2d_to_3d**(*patch*, *z=0*, *zdir='z'*)

Convert a *Patch* to a *Patch3D* object.

### mpl_toolkits.mplot3d.art3d.patch_collection_2d_to_3d

mpl_toolkits.mplot3d.art3d.**patch_collection_2d_to_3d**(*col*, *zs=0*, *zdir='z'*, *depthshade=True*)

Convert a *PatchCollection* into a *Patch3DCollection* object (or a *PathCollection* into a *Path3DCollection* object).

**Parameters**

**col**

[*PatchCollection* or *PathCollection*] The collection to convert.

**zs**

[float or array of floats] The location or locations to place the patches in the collection along the *zdir* axis. Default: 0.

**zdir**

[{'x', 'y', 'z'}] The axis in which to place the patches. Default: "z". See `get_dir_vector` for a description of the values.

**depthshade**

[bool, default: True] Whether to shade the patches to give a sense of depth.

### mpl_toolkits.mplot3d.art3d.pathpatch_2d_to_3d

mpl_toolkits.mplot3d.art3d.**pathpatch_2d_to_3d**(*pathpatch*, *z=0*, *zdir='z'*)

Convert a `PathPatch` to a `PathPatch3D` object.

### Examples using `mpl_toolkits.mplot3d.art3d.pathpatch_2d_to_3d`

- *Draw flat objects in 3D plot*

### mpl_toolkits.mplot3d.art3d.poly_collection_2d_to_3d

mpl_toolkits.mplot3d.art3d.**poly_collection_2d_to_3d**(*col*, *zs=0*, *zdir='z'*)

Convert a `PolyCollection` into a `Poly3DCollection` object.

**Parameters**

**col**

[`PolyCollection`] The collection to convert.

**zs**

[float or array of floats] The location or locations to place the polygons in the collection along the *zdir* axis. Default: 0.

**zdir**

[{'x', 'y', 'z'}] The axis in which to place the patches. Default: 'z'. See `get_dir_vector` for a description of the values.

### mpl_toolkits.mplot3d.art3d.rotate_axes

mpl_toolkits.mplot3d.art3d.**rotate_axes**(*xs*, *ys*, *zs*, *zdir*)

Reorder coordinates so that the axes are rotated with *zdir* along the original z axis. Prepending the axis with a '-' does the inverse transform, so *zdir* can be 'x', '-x', 'y', '-y', 'z' or '-z'.

### mpl_toolkits.mplot3d.art3d.text_2d_to_3d

mpl_toolkits.mplot3d.art3d.**text_2d_to_3d**(*obj*, *z=0*, *zdir='z'*)

Convert a *Text* to a *Text3D* object.

**Parameters**

**z**

[float] The z-position in 3D space.

**zdir**

[{'x', 'y', 'z', 3-tuple}] The direction of the text. Default: 'z'. See *get_dir_vector* for a description of the values.

### proj3d

| | |
|---|---|
| *proj3d.inv_transform*(xs, ys, zs, invM) | Transform the points by the inverse of the projection matrix, *invM*. |
| *proj3d.persp_transformation*(zfront, zback, ...) | [*Deprecated*] |
| *proj3d.proj_points*(points, M) | [*Deprecated*] |
| *proj3d.proj_trans_points*(points, M) | [*Deprecated*] |
| *proj3d.proj_transform*(xs, ys, zs, M) | Transform the points by the projection matrix *M*. |
| *proj3d.proj_transform_clip*(xs, ys, zs, M) | Transform the points by the projection matrix and return the clipping result returns txs, tys, tzs, tis |
| *proj3d.rot_x*(V, alpha) | [*Deprecated*] |
| *proj3d.transform*(xs, ys, zs, M) | [*Deprecated*] Transform the points by the projection matrix *M*. |
| *proj3d.view_transformation*(E, R, V, roll) | [*Deprecated*] Return the view transformation matrix. |
| *proj3d.world_transformation*(xmin, xmax, ...) | Produce a matrix that scales homogeneous coords in the specified ranges to [0, 1], or [0, pb_aspect[i]] if the plotbox aspect ratio is specified. |

### mpl_toolkits.mplot3d.proj3d.inv_transform

mpl_toolkits.mplot3d.proj3d.**inv_transform**(*xs*, *ys*, *zs*, *invM*)

Transform the points by the inverse of the projection matrix, *invM*.

### mpl_toolkits.mplot3d.proj3d.persp_transformation

mpl_toolkits.mplot3d.proj3d.**persp_transformation**(*zfront*, *zback*, *focal_length*)

[*Deprecated*]

#### Notes

Deprecated since version 3.8:

### mpl_toolkits.mplot3d.proj3d.proj_points

mpl_toolkits.mplot3d.proj3d.**proj_points**(*points*, *M*)

[*Deprecated*]

#### Notes

Deprecated since version 3.8:

### mpl_toolkits.mplot3d.proj3d.proj_trans_points

mpl_toolkits.mplot3d.proj3d.**proj_trans_points**(*points*, *M*)

[*Deprecated*]

#### Notes

Deprecated since version 3.8:

### mpl_toolkits.mplot3d.proj3d.proj_transform

mpl_toolkits.mplot3d.proj3d.**proj_transform**(*xs*, *ys*, *zs*, *M*)

Transform the points by the projection matrix *M*.

### mpl_toolkits.mplot3d.proj3d.proj_transform_clip

mpl_toolkits.mplot3d.proj3d.**proj_transform_clip**(*xs*, *ys*, *zs*, *M*)

Transform the points by the projection matrix and return the clipping result returns txs, tys, tzs, tis

### mpl_toolkits.mplot3d.proj3d.rot_x

mpl_toolkits.mplot3d.proj3d.**rot_x**(*V*, *alpha*)

[*Deprecated*]

#### Notes

Deprecated since version 3.8:

### mpl_toolkits.mplot3d.proj3d.transform

mpl_toolkits.mplot3d.proj3d.**transform**(*xs*, *ys*, *zs*, *M*)

[*Deprecated*] Transform the points by the projection matrix *M*.

#### Notes

Deprecated since version 3.8: Use proj_transform instead.

### mpl_toolkits.mplot3d.proj3d.view_transformation

mpl_toolkits.mplot3d.proj3d.**view_transformation**(*E*, *R*, *V*, *roll*)

[*Deprecated*] Return the view transformation matrix.

**Parameters**

**E**

[3-element numpy array] The coordinates of the eye/camera.

**R**

[3-element numpy array] The coordinates of the center of the view box.

**V**

[3-element numpy array] Unit vector in the direction of the vertical axis.

**roll**

[float] The roll angle in radians.

### Notes

Deprecated since version 3.8.

### mpl_toolkits.mplot3d.proj3d.world_transformation

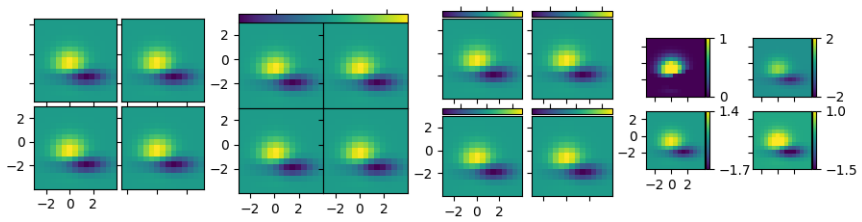
mpl_toolkits.mplot3d.proj3d.**world_transformation**(*xmin*, *xmax*, *ymin*, *ymax*, *zmin*, *zmax*, *pb_aspect=None*)

Produce a matrix that scales homogeneous coords in the specified ranges to [0, 1], or [0, pb_aspect[i]] if the plotbox aspect ratio is specified.

## 7.2.67 mpl_toolkits.axes_grid1

*mpl_toolkits.axes_grid1* provides a framework of helper classes to adjust the positioning of multiple fixed-aspect Axes (e.g., displaying images). It can be contrasted with the `aspect` property of Matplotlib Axes, which adjusts the position of a single Axes.

See *The axes_grid1 toolkit* for a guide on the usage of axes_grid1.



**Note:** This module contains classes and function that were formerly part of the `mpl_toolkits.axes_grid` module that was removed in 3.6. Additional classes from that older module may also be found in *mpl_toolkits.axisartist*.

**The submodules of the axes_grid1 API are:**

| | |
|---|---|
| *axes_grid1.anchored_artists* | |
| *axes_grid1.axes_divider* | Helper classes to adjust the positions of multiple axes at drawing time. |
| *axes_grid1.axes_grid* | |
| *axes_grid1.axes_rgb* | |
| *axes_grid1.axes_size* | Provides classes of simple units that will be used with *AxesDivider* class (or others) to determine the size of each Axes. |
| *axes_grid1.inset_locator* | A collection of functions and objects for creating or placing inset axes. |
| *axes_grid1.mpl_axes* | |
| *axes_grid1.parasite_axes* | |

### mpl_toolkits.axes_grid1.anchored_artists

#### Classes

| | |
|---|---|
| *AnchoredAuxTransformBox*(transform, loc[, ...]) | An anchored container with transformed coordinates. |
| *AnchoredDirectionArrows*(transform, label_x, ...) | Draw two perpendicular arrows to indicate directions. |
| *AnchoredDrawingArea*(width, height, xdescent, ...) | An anchored container with a fixed size and fillable *DrawingArea*. |
| *AnchoredEllipse*(transform, width, height, ...) | [*Deprecated*] |
| *AnchoredSizeBar*(transform, size, label, loc) | Draw a horizontal scale bar with a center-aligned label underneath. |

### mpl_toolkits.axes_grid1.anchored_artists.AnchoredAuxTransformBox

**class** mpl_toolkits.axes_grid1.anchored_artists.**AnchoredAuxTransformBox**(*transform*, *loc*, *pad=0.4*, *borderpad=0.5*, *prop=None*, *frameon=True*, ***kwargs*)

Bases: *AnchoredOffsetbox*

An anchored container with transformed coordinates.

Artists added to the *drawing_area* are scaled according to the coordinates of the transformation used. The dimensions of this artist will scale to contain the artists added.

**Parameters**

**transform**

[*Transform*] The transformation object for the coordinate system in use, i.e., matplotlib.axes.Axes.transData.

**loc**

[str] Location of this artist. Valid locations are 'upper left', 'upper center', 'upper right', 'center left', 'center', 'center right', 'lower left', 'lower center', 'lower right'. For backward compatibility, numeric values are accepted as well. See the parameter *loc* of *Legend* for details.

**pad**

[float, default: 0.4] Padding around the child objects, in fraction of the font size.

**borderpad**

[float, default: 0.5] Border padding, in fraction of the font size.

**prop**

[*FontProperties*, optional] Font property used as a reference for paddings.

**frameon**

[bool, default: True] If True, draw a box around this artist.

****kwargs**

Keyword arguments forwarded to *AnchoredOffsetbox*.

### Examples

To display an ellipse in the upper left, with a width of 0.1 and height of 0.4 in data coordinates:

```
>>> box = AnchoredAuxTransformBox(ax.transData, loc='upper left')
>>> el = Ellipse((0, 0), width=0.1, height=0.4, angle=30)
>>> box.drawing_area.add_artist(el)
>>> ax.add_artist(box)
```

**Attributes**

**drawing_area**

[*AuxTransformBox*] A container for artists to display.

**set** (*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *bbox_to_anchor=<UNSET>*, *child=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *gid=<UNSET>*, *height=<UNSET>*, *in_layout=<UNSET>*, *label=<UNSET>*, *mouseover=<UNSET>*, *offset=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *width=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| agg_filt | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image |
| alpha | scalar or None |
| animated | bool |
| bbox_to_ | unknown |
| child | unknown |
| clip_box | BboxBase or None |
| clip_on | bool |
| clip_pat | Patch or (Path, Transform) or None |
| figure | Figure |
| gid | str |
| height | float |
| in_layou | bool |
| label | object |
| mouseove | bool |
| offset | (float, float) or callable |
| path_eff | list of AbstractPathEffect |
| picker | None or bool or float or callable |
| rasterized | bool |
| sketch_p | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| transform | Transform |
| url | str |
| visible | bool |
| width | float |
| zorder | float |

### Examples using `mpl_toolkits.axes_grid1.anchored_artists.AnchoredAuxTransformBox`

- *Annotations*

### mpl_toolkits.axes_grid1.anchored_artists.AnchoredDirectionArrows

**class** mpl_toolkits.axes_grid1.anchored_artists.**AnchoredDirectionArrows**(*transform*, *label_x*, *label_y*, *length=0.15*, *fontsize=0.08*, *loc='upper left'*, *angle=0*, *aspect_ratio=1*, *pad=0.4*, *borderpad=0.4*, *frameon=False*, *color='w'*, *alpha=1*, *sep_x=0.01*, *sep_y=0*, *fontproperties=None*, *back_length=0.1*, *head_width=10*, *head_length=15*, *tail_width=2*, *text_props=None*, *arrow_props=None*, ***kwargs*)

Bases: *AnchoredOffsetbox*

Draw two perpendicular arrows to indicate directions.

**Parameters**

**transform**

[*Transform*] The transformation object for the coordinate system in use, i.e., matplotlib.axes.Axes.transAxes.

**label_x, label_y**

[str] Label text for the x and y arrows

**length**

[float, default: 0.15] Length of the arrow, given in coordinates of *transform*.

**fontsize**

[float, default: 0.08] Size of label strings, given in coordinates of *transform*.

**loc**

[str, default: 'upper left'] Location of the arrow. Valid locations are 'upper left', 'upper center', 'upper right', 'center left', 'center', 'center right', 'lower left', 'lower center', 'lower right'. For backward compatibility, numeric values are accepted as well. See the parameter *loc* of `Legend` for details.

**angle**

[float, default: 0] The angle of the arrows in degrees.

**aspect_ratio**

[float, default: 1] The ratio of the length of arrow_x and arrow_y. Negative numbers can be used to change the direction.

**pad**

[float, default: 0.4] Padding around the labels and arrows, in fraction of the font size.

**borderpad**

[float, default: 0.4] Border padding, in fraction of the font size.

**frameon**

[bool, default: False] If True, draw a box around the arrows and labels.

**color**

[str, default: 'white'] Color for the arrows and labels.

**alpha**

[float, default: 1] Alpha values of the arrows and labels

**sep_x, sep_y**

[float, default: 0.01 and 0 respectively] Separation between the arrows and labels in coordinates of *transform*.

**fontproperties**

[`FontProperties`, optional] Font properties for the label text.

**back_length**

[float, default: 0.15] Fraction of the arrow behind the arrow crossing.

**head_width**

[float, default: 10] Width of arrow head, sent to *ArrowStyle*.

**head_length**

[float, default: 15] Length of arrow head, sent to *ArrowStyle*.

**tail_width**

[float, default: 2] Width of arrow tail, sent to *ArrowStyle*.

**text_props, arrow_props**

[dict] Properties of the text and arrows, passed to *TextPath* and *FancyArrowPatch*.

****kwargs**

Keyword arguments forwarded to *AnchoredOffsetbox*.

## Notes

If *prop* is passed as a keyword argument, but *fontproperties* is not, then *prop* is assumed to be the intended *fontproperties*. Using both *prop* and *fontproperties* is not supported.

## Examples

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> from mpl_toolkits.axes_grid1.anchored_artists import (
...     AnchoredDirectionArrows)
>>> fig, ax = plt.subplots()
>>> ax.imshow(np.random.random((10, 10)))
>>> arrows = AnchoredDirectionArrows(ax.transAxes, '111', '110')
>>> ax.add_artist(arrows)
>>> fig.show()
```

Using several of the optional parameters, creating downward pointing arrow and high contrast text labels.

```
>>> import matplotlib.font_manager as fm
>>> fontprops = fm.FontProperties(family='monospace')
>>> arrows = AnchoredDirectionArrows(ax.transAxes, 'East', 'South',
...                                  loc='lower left', color='k',
...                                  aspect_ratio=-1, sep_x=0.02,
...                                  sep_y=-0.01,
...                                  text_props={'ec':'w', 'fc':'k'},
...                                  fontproperties=fontprops)
```

**Attributes**

**arrow_x, arrow_y**

[*FancyArrowPatch*] Arrow x and y

**text_path_x, text_path_y**

[*TextPath*] Path for arrow labels

**p_x, p_y**

[*PathPatch*] Patch for arrow labels

**box**

[*AuxTransformBox*] Container for the arrows and labels.

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *bbox_to_anchor=<UNSET>*, *child=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *gid=<UNSET>*, *height=<UNSET>*, *in_layout=<UNSET>*, *label=<UNSET>*, *mouseover=<UNSET>*, *offset=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *width=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| agg_filt | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image |
| alpha | scalar or None |
| animated | bool |
| bbox_to_ | unknown |
| child | unknown |
| clip_box | BboxBase or None |
| clip_on | bool |
| clip_pat | Patch or (Path, Transform) or None |
| figure | Figure |
| gid | str |
| height | float |
| in_layou | bool |
| label | object |
| mouseove | bool |
| offset | (float, float) or callable |
| path_eff | list of AbstractPathEffect |
| picker | None or bool or float or callable |
| rasterized | bool |
| sketch_p | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| transform | Transform |
| url | str |
| visible | bool |
| width | float |
| zorder | float |

### Examples using `mpl_toolkits.axes_grid1.anchored_artists.AnchoredDirectionArrows`

- *Anchored Direction Arrow*

### mpl_toolkits.axes_grid1.anchored_artists.AnchoredDrawingArea

**class** mpl_toolkits.axes_grid1.anchored_artists.**AnchoredDrawingArea**(*width*, *height*, *xdescent*, *ydescent*, *loc*, *pad=0.4*, *borderpad=0.5*, *prop=None*, *frameon=True*, ***kwargs*)

Bases: *AnchoredOffsetbox*

An anchored container with a fixed size and fillable *DrawingArea*.

Artists added to the *drawing_area* will have their coordinates interpreted as pixels. Any transformations set on the artists will be overridden.

**Parameters**

**width, height**

[float] Width and height of the container, in pixels.

**xdescent, ydescent**

[float] Descent of the container in the x- and y- direction, in pixels.

**loc**

[str] Location of this artist. Valid locations are 'upper left', 'upper center', 'upper right', 'center left', 'center', 'center right', 'lower left', 'lower center', 'lower right'. For backward compatibility, numeric values are accepted as well. See the parameter *loc* of *Legend* for details.

**pad**

[float, default: 0.4] Padding around the child objects, in fraction of the font size.

**borderpad**

[float, default: 0.5] Border padding, in fraction of the font size.

**prop**

[*FontProperties*, optional] Font property used as a reference for paddings.

**frameon**

[bool, default: True] If True, draw a box around this artist.

****kwargs**

Keyword arguments forwarded to *AnchoredOffsetbox*.

## Examples

To display blue and red circles of different sizes in the upper right of an Axes *ax*:

```
>>> ada = AnchoredDrawingArea(20, 20, 0, 0,
...                           loc='upper right', frameon=False)
>>> ada.drawing_area.add_artist(Circle((10, 10), 10, fc="b"))
>>> ada.drawing_area.add_artist(Circle((30, 10), 5, fc="r"))
>>> ax.add_artist(ada)
```

**Attributes**

**drawing_area**

[*DrawingArea*] A container for artists to display.

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *bbox_to_anchor=<UNSET>*, *child=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *gid=<UNSET>*, *height=<UNSET>*, *in_layout=<UNSET>*, *label=<UNSET>*, *mouseover=<UNSET>*, *offset=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *width=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| agg_filt | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image |
| alpha | scalar or None |
| animated | bool |
| bbox_to_ | unknown |
| child | unknown |
| clip_box | BboxBase or None |
| clip_on | bool |
| clip_pat | Patch or (Path, Transform) or None |
| figure | Figure |
| gid | str |
| height | float |
| in_layou | bool |
| label | object |
| mouseove | bool |
| offset | (float, float) or callable |
| path_eff | list of AbstractPathEffect |
| picker | None or bool or float or callable |
| rasterized | bool |
| sketch_p | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| transform | Transform |
| url | str |
| visible | bool |
| width | float |
| zorder | float |

### Examples using `mpl_toolkits.axes_grid1.anchored_artists.AnchoredDrawingArea`

- *Simple Anchored Artists*
- *Annotations*

### mpl_toolkits.axes_grid1.anchored_artists.AnchoredEllipse

**class** mpl_toolkits.axes_grid1.anchored_artists.**AnchoredEllipse**(*transform*, *width*, *height*, *angle*, *loc*, *pad=0.1*, *borderpad=0.1*, *prop=None*, *frameon=True*, ***kwargs*)

Bases: *AnchoredOffsetbox*

[*Deprecated*]

#### Notes

Deprecated since version 3.8:

Draw an anchored ellipse of a given size.

**Parameters**

**transform**

[*Transform*] The transformation object for the coordinate system in use, i.e., matplotlib.axes.Axes.transData.

**width, height**

[float] Width and height of the ellipse, given in coordinates of *transform*.

**angle**

[float] Rotation of the ellipse, in degrees, anti-clockwise.

**loc**

[str] Location of the ellipse. Valid locations are 'upper left', 'upper center', 'upper right', 'center left', 'center', 'center right', 'lower left', 'lower center', 'lower right'. For backward compatibility, numeric values are accepted as well. See the parameter *loc* of *Legend* for details.

**pad**

[float, default: 0.1] Padding around the ellipse, in fraction of the font size.

**borderpad**

[float, default: 0.1] Border padding, in fraction of the font size.

**frameon**

[bool, default: True] If True, draw a box around the ellipse.

**prop**

[*FontProperties*, optional] Font property used as a reference for paddings.

****kwargs**

Keyword arguments forwarded to *AnchoredOffsetbox*.

**Attributes**

**ellipse**

[*Ellipse*] Ellipse patch drawn.

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *bbox_to_anchor=<UNSET>*, *child=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *gid=<UNSET>*, *height=<UNSET>*, *in_layout=<UNSET>*, *label=<UNSET>*, *mouseover=<UNSET>*, *offset=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *width=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| agg_filt | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image |
| alpha | scalar or None |
| animated | bool |
| bbox_to_ | unknown |
| child | unknown |
| clip_box | BboxBase or None |
| clip_on | bool |
| clip_pat | Patch or (Path, Transform) or None |
| figure | Figure |
| gid | str |
| height | float |
| in_layou | bool |
| label | object |
| mouseove | bool |
| offset | (float, float) or callable |
| path_eff | list of AbstractPathEffect |
| picker | None or bool or float or callable |
| rasterized | bool |
| sketch_p | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| transform | Transform |
| url | str |
| visible | bool |
| width | float |
| zorder | float |

### mpl_toolkits.axes_grid1.anchored_artists.AnchoredSizeBar

**class** mpl_toolkits.axes_grid1.anchored_artists.**AnchoredSizeBar**(*transform*, *size*, *label*, *loc*, *pad=0.1*, *borderpad=0.1*, *sep=2*, *frameon=True*, *size_vertical=0*, *color='black'*, *label_top=False*, *fontproperties=None*, *fill_bar=None*, ***kwargs*)

Bases: *AnchoredOffsetbox*

Draw a horizontal scale bar with a center-aligned label underneath.

**Parameters**

**transform**

[*Transform*] The transformation object for the coordinate system in use, i.e., matplotlib.axes.Axes.transData.

**size**

[float] Horizontal length of the size bar, given in coordinates of *transform*.

**label**

[str] Label to display.

**loc**

[str] Location of the size bar. Valid locations are 'upper left', 'upper center', 'upper right', 'center left', 'center', 'center right', 'lower left', 'lower center', 'lower right'. For backward compatibility, numeric values are accepted as well. See the parameter *loc* of *Legend* for details.

**pad**

[float, default: 0.1] Padding around the label and size bar, in fraction of the font size.

**borderpad**

[float, default: 0.1] Border padding, in fraction of the font size.

**sep**

[float, default: 2] Separation between the label and the size bar, in points.

**frameon**

[bool, default: True] If True, draw a box around the horizontal bar and label.

**size_vertical**

[float, default: 0] Vertical length of the size bar, given in coordinates of *transform*.

**color**

[str, default: 'black'] Color for the size bar and label.

**label_top**

[bool, default: False] If True, the label will be over the size bar.

**fontproperties**

[*FontProperties*, optional] Font properties for the label text.

**fill_bar**

[bool, optional] If True and if *size_vertical* is nonzero, the size bar will be filled in with the color specified by the size bar. Defaults to True if *size_vertical* is greater than zero and False otherwise.

****kwargs**

Keyword arguments forwarded to *AnchoredOffsetbox*.

## Notes

If *prop* is passed as a keyword argument, but *fontproperties* is not, then *prop* is assumed to be the intended *fontproperties*. Using both *prop* and *fontproperties* is not supported.

## Examples

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> from mpl_toolkits.axes_grid1.anchored_artists import (
...     AnchoredSizeBar)
>>> fig, ax = plt.subplots()
>>> ax.imshow(np.random.random((10, 10)))
>>> bar = AnchoredSizeBar(ax.transData, 3, '3 data units', 4)
>>> ax.add_artist(bar)
>>> fig.show()
```

Using all the optional parameters

```
>>> import matplotlib.font_manager as fm
>>> fontprops = fm.FontProperties(size=14, family='monospace')
>>> bar = AnchoredSizeBar(ax.transData, 3, '3 units', 4, pad=0.5,
...                       sep=5, borderpad=0.5, frameon=False,
```

(continues on next page)

(continued from previous page)

```
...                         size_vertical=0.5, color='white',
...                         fontproperties=fontprops)
```

**Attributes**

**size_bar**

[*AuxTransformBox*] Container for the size bar.

**txt_label**

[*TextArea*] Container for the label of the size bar.

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *bbox_to_anchor=<UNSET>*, *child=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *gid=<UNSET>*, *height=<UNSET>*, *in_layout=<UNSET>*, *label=<UNSET>*, *mouseover=<UNSET>*, *offset=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *width=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| agg_filt | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image |
| alpha | scalar or None |
| animated | bool |
| bbox_to_ | unknown |
| child | unknown |
| clip_box | BboxBase or None |
| clip_on | bool |
| clip_pat | Patch or (Path, Transform) or None |
| figure | Figure |
| gid | str |
| height | float |
| in_layou | bool |
| label | object |
| mouseove | bool |
| offset | (float, float) or callable |
| path_eff | list of AbstractPathEffect |
| picker | None or bool or float or callable |
| rasterized | bool |
| sketch_p | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| transform | Transform |
| url | str |
| visible | bool |
| width | float |
| zorder | float |

### Examples using `mpl_toolkits.axes_grid1.anchored_artists.AnchoredSizeBar`

- *Inset locator demo 2*
- *Simple Anchored Artists*

## mpl_toolkits.axes_grid1.axes_divider

Helper classes to adjust the positions of multiple axes at drawing time.

### Classes

| | |
|---|---|
| *AxesDivider*(axes[, xref, yref]) | Divider based on the preexisting axes. |
| *AxesLocator*(axes_divider, nx, ny[, nx1, ny1]) | [*Deprecated*] A callable object which returns the position and size of a given *AxesDivider* cell. |
| *Divider*(fig, pos, horizontal, vertical[, ...]) | An Axes positioning class. |
| *HBoxDivider*(fig, *args[, horizontal, ...]) | A *SubplotDivider* for laying out axes horizontally, while ensuring that they have equal heights. |
| *SubplotDivider*(fig, *args[, horizontal, ...]) | The Divider class whose rectangle area is specified as a subplot geometry. |
| *VBoxDivider*(fig, *args[, horizontal, ...]) | A *SubplotDivider* for laying out axes vertically, while ensuring that they have equal widths. |

## mpl_toolkits.axes_grid1.axes_divider.AxesDivider

**class** mpl_toolkits.axes_grid1.axes_divider.**AxesDivider**(*axes*, *xref=None*, *yref=None*)

Bases: *Divider*

Divider based on the preexisting axes.

**Parameters**

**axes**

[*Axes*]

**xref**
**yref**

**append_axes**(*position*, *size*, *pad=None*, ***, *axes_class=None*, ***kwargs*)

Add a new axes on a given side of the main axes.

**Parameters**

**position**

[{"left", "right", "bottom", "top"}] Where the new axes is positioned relative to the main axes.

**size**

[*axes_size* or float or str] The axes width or height. float or str arguments are interpreted as `axes_size.from_any(size, AxesX(<main_axes>))` for left or right axes, and likewise with `AxesY` for bottom or top axes.

**pad**

[*axes_size* or float or str] Padding between the axes. float or str arguments are interpreted as for *size*. Defaults to *rcParams["figure.subplot.wspace"]* (default: `0.2`) times the main Axes width (left or right axes) or *rcParams["figure.subplot.hspace"]* (default: `0.2`) times the main Axes height (bottom or top axes).

**axes_class**

[subclass type of *Axes*, optional] The type of the new axes. Defaults to the type of the main axes.

****kwargs**

All extra keywords arguments are passed to the created axes.

**get_anchor**()

Return the anchor.

**get_aspect**()

Return aspect.

**get_position**()

Return the position of the rectangle.

**get_subplotspec**()

### Examples using `mpl_toolkits.axes_grid1.axes_divider.AxesDivider`

- *Colorbar with AxesDivider*
- *Make room for ylabel using axes_grid*
- *Scatter Histogram (Locatable Axes)*
- *Tight layout guide*

## mpl_toolkits.axes_grid1.axes_divider.AxesLocator

**class** mpl_toolkits.axes_grid1.axes_divider.**AxesLocator**(*axes_divider*, *nx*, *ny*, *nx1=None*, *ny1=None*)

Bases: `object`

[*Deprecated*] A callable object which returns the position and size of a given *AxesDivider* cell.

### Notes

Deprecated since version 3.8.

**Parameters**

**axes_divider**

[*AxesDivider*]

**nx, nx1**

[int] Integers specifying the column-position of the cell. When *nx1* is None, a single *nx*-th column is specified. Otherwise, location of columns spanning between *nx* to *nx1* (but excluding *nx1*-th column) is specified.

**ny, ny1**

[int] Same as *nx* and *nx1*, but for row positions.

**__call__**(*axes*, *renderer*)

Call self as a function.

**get_subplotspec**()

## mpl_toolkits.axes_grid1.axes_divider.Divider

**class** mpl_toolkits.axes_grid1.axes_divider.**Divider**(*fig*, *pos*, *horizontal*, *vertical*, *aspect=None*, *anchor='C'*)

Bases: object

An Axes positioning class.

The divider is initialized with lists of horizontal and vertical sizes (*mpl_toolkits.axes_grid1.axes_size*) based on which a given rectangular area will be divided.

The *new_locator* method then creates a callable object that can be used as the *axes_locator* of the axes.

**Parameters**

**fig**

[Figure]

**pos**

[tuple of 4 floats] Position of the rectangle that will be divided.

**horizontal**

[list of *axes_size*] Sizes for horizontal division.

**vertical**

[list of *axes_size*] Sizes for vertical division.

**aspect**

[bool, optional] Whether overall rectangular area is reduced so that the relative part of the horizontal and vertical scales have the same scale.

**anchor**

[(float, float) or {'C', 'SW', 'S', 'SE', 'E', 'NE', 'N', 'NW', 'W'}, default: 'C'] Placement of the reduced rectangle, when *aspect* is True.

**add_auto_adjustable_area**(*use_axes*, *pad=0.1*, *adjust_dirs=None*)

Add auto-adjustable padding around *use_axes* to take their decorations (title, labels, ticks, ticklabels) into account during layout.

**Parameters**

**use_axes**

[*Axes* or list of *Axes*] The Axes whose decorations are taken into account.

**pad**

[float, default: 0.1] Additional padding in inches.

**adjust_dirs**

[list of {"left", "right", "bottom", "top"}, optional] The sides where padding is added; defaults to all four sides.

**append_size**(*position*, *size*)

**get_anchor**()

Return the anchor.

**get_aspect**()

Return aspect.

**get_horizontal**()

Return horizontal sizes.

**get_horizontal_sizes**(*renderer*)

**get_locator**()

**get_position**()

Return the position of the rectangle.

**get_position_runtime**(*ax*, *renderer*)

**get_subplotspec**()

**get_vertical**()

Return vertical sizes.

**get_vertical_sizes**(*renderer*)

**locate**(*nx*, *ny*, *nx1=None*, *ny1=None*, *axes=None*, *renderer=None*)

[*Deprecated*] Implementation of `divider.new_locator().__call__`.

**Parameters**

**nx, nx1**

[int] Integers specifying the column-position of the cell. When *nx1* is None, a single *nx*-th column is specified. Otherwise, the location of columns spanning between *nx* to *nx1* (but excluding *nx1*-th column) is specified.

**ny, ny1**

[int] Same as *nx* and *nx1*, but for row positions.

**axes**
**renderer**

### Notes

Deprecated since version 3.8: Use divider.new_locator(...)(ax, renderer) instead.

**new_locator**(*nx*, *ny*, *nx1=None*, *ny1=None*)

Return an axes locator callable for the specified cell.

**Parameters**

**nx, nx1**

[int] Integers specifying the column-position of the cell. When *nx1* is None, a single *nx*-th column is specified. Otherwise, location of columns spanning between *nx* to *nx1* (but excluding *nx1*-th column) is specified.

**ny, ny1**

[int] Same as *nx* and *nx1*, but for row positions.

**set_anchor**(*anchor*)

**Parameters**

**anchor**

[(float, float) or {'C', 'SW', 'S', 'SE', 'E', 'NE', 'N', 'NW', 'W'}] Either an (*x*, *y*) pair of relative coordinates (0 is left or bottom, 1 is right or top), 'C' (center), or a cardinal direction ('SW', southwest, is bottom left, etc.).

**See also:**

*Axes.set_anchor*

**set_aspect**(*aspect=False*)

**Parameters**

**aspect**

[bool]

**set_horizontal**(*h*)

**Parameters**

**h**

[list of *axes_size*] sizes for horizontal division

**set_locator**(*_locator*)

**set_position**(*pos*)

Set the position of the rectangle.

**Parameters**

**pos**

[tuple of 4 floats] position of the rectangle that will be divided

**set_vertical**(*v*)

**Parameters**

**v**

[list of *axes_size*] sizes for vertical division

### mpl_toolkits.axes_grid1.axes_divider.HBoxDivider

**class** mpl_toolkits.axes_grid1.axes_divider.**HBoxDivider**(*fig*, **args*, *horizontal=None*, *vertical=None*, *aspect=None*, *anchor='C'*)

Bases: *SubplotDivider*

A *SubplotDivider* for laying out axes horizontally, while ensuring that they have equal heights.

### Examples

**Parameters**

**fig**

[*Figure*]

***args**

[tuple (*nrows*, *ncols*, *index*) or int] The array of subplots in the figure has dimensions `(nrows, ncols)`, and *index* is the index of the subplot being created. *index* starts at 1 in the upper left corner and increases to the right.

If *nrows*, *ncols*, and *index* are all single digit numbers, then *args* can be passed as a single 3-digit number (e.g. 234 for (2, 3, 4)).

**horizontal**

[list of *axes_size*, optional] Sizes for horizontal division.

**vertical**

[list of *axes_size*, optional] Sizes for vertical division.

**aspect**

[bool, optional] Whether overall rectangular area is reduced so that the relative part of the horizontal and vertical scales have the same scale.

**anchor**

[(float, float) or {'C', 'SW', 'S', 'SE', 'E', 'NE', 'N', 'NW', 'W'}, default: 'C'] Placement of the reduced rectangle, when *aspect* is True.

**new_locator**(*nx*, *nx1=None*)

Create an axes locator callable for the specified cell.

**Parameters**

**nx, nx1**

[int] Integers specifying the column-position of the cell. When *nx1* is None, a single *nx*-th column is specified. Otherwise, location of columns spanning between *nx* to *nx1* (but excluding *nx1*-th column) is specified.

### Examples using `mpl_toolkits.axes_grid1.axes_divider.HBoxDivider`

- *HBoxDivider and VBoxDivider demo*

## mpl_toolkits.axes_grid1.axes_divider.SubplotDivider

**class** mpl_toolkits.axes_grid1.axes_divider.**SubplotDivider**(*fig*, **args*, *horizontal=None*, *vertical=None*, *aspect=None*, *anchor='C'*)

Bases: *Divider*

The Divider class whose rectangle area is specified as a subplot geometry.

**Parameters**

**fig**

[*Figure*]

***args**

[tuple (*nrows*, *ncols*, *index*) or int] The array of subplots in the figure has dimensions `(nrows, ncols)`, and *index* is the index of the subplot being created. *index* starts at 1 in the upper left corner and increases to the right.

If *nrows*, *ncols*, and *index* are all single digit numbers, then *args* can be passed as a single 3-digit number (e.g. 234 for (2, 3, 4)).

**horizontal**

[list of *axes_size*, optional] Sizes for horizontal division.

**vertical**

[list of *axes_size*, optional] Sizes for vertical division.

**aspect**

[bool, optional] Whether overall rectangular area is reduced so that the relative part of the horizontal and vertical scales have the same scale.

**anchor**

[(float, float) or {'C', 'SW', 'S', 'SE', 'E', 'NE', 'N', 'NW', 'W'}, default: 'C'] Placement of the reduced rectangle, when *aspect* is True.

**get_position**()

Return the bounds of the subplot box.

**get_subplotspec**()

Get the SubplotSpec instance.

**set_subplotspec**(*subplotspec*)

Set the SubplotSpec instance.

### mpl_toolkits.axes_grid1.axes_divider.VBoxDivider

**class** mpl_toolkits.axes_grid1.axes_divider.**VBoxDivider**(*fig*, **args*, *horizontal=None*, *vertical=None*, *aspect=None*, *anchor='C'*)

Bases: *SubplotDivider*

A *SubplotDivider* for laying out axes vertically, while ensuring that they have equal widths.

**Parameters**

**fig**

[*Figure*]

***args**

[tuple (*nrows*, *ncols*, *index*) or int] The array of subplots in the figure has dimensions `(nrows, ncols)`, and *index* is the index of the subplot being created. *index* starts at 1 in the upper left corner and increases to the right.

If *nrows*, *ncols*, and *index* are all single digit numbers, then *args* can be passed as a single 3-digit number (e.g. 234 for (2, 3, 4)).

**horizontal**

[list of *axes_size*, optional] Sizes for horizontal division.

**vertical**

[list of *axes_size*, optional] Sizes for vertical division.

**aspect**

[bool, optional] Whether overall rectangular area is reduced so that the relative part of the horizontal and vertical scales have the same scale.

**anchor**

[(float, float) or {'C', 'SW', 'S', 'SE', 'E', 'NE', 'N', 'NW', 'W'}, default: 'C'] Placement of the reduced rectangle, when *aspect* is True.

**new_locator**(*ny*, *ny1=None*)

Create an axes locator callable for the specified cell.

**Parameters**

**ny, ny1**

[int] Integers specifying the row-position of the cell. When *ny1* is None, a single *ny*-th row is specified. Otherwise, location of rows spanning between *ny* to *ny1* (but excluding *ny1*-th row) is specified.

### Examples using `mpl_toolkits.axes_grid1.axes_divider.VBoxDivider`

- *HBoxDivider and VBoxDivider demo*

### Functions

| | |
|---|---|
| `make_axes_area_auto_adjustable`(ax[, ...]) | Add auto-adjustable padding around *ax* to take its decorations (title, labels, ticks, ticklabels) into account during layout, using `Divider.add_auto_adjustable_area`. |
| `make_axes_locatable`(axes) | |

## mpl_toolkits.axes_grid1.axes_divider.make_axes_area_auto_adjustable

mpl_toolkits.axes_grid1.axes_divider.**make_axes_area_auto_adjustable**(*ax*, *use_axes=None*, *pad=0.1*, *adjust_dirs=None*)

Add auto-adjustable padding around *ax* to take its decorations (title, labels, ticks, ticklabels) into account during layout, using `Divider.add_auto_adjustable_area`.

By default, padding is determined from the decorations of *ax*. Pass *use_axes* to consider the decorations of other Axes instead.

### Examples using `mpl_toolkits.axes_grid1.axes_divider.make_axes_area_auto_adjustable`

- *Make room for ylabel using axes_grid*

### mpl_toolkits.axes_grid1.axes_divider.make_axes_locatable

mpl_toolkits.axes_grid1.axes_divider.**make_axes_locatable**(*axes*)

### mpl_toolkits.axes_grid1.axes_grid

### Classes

| | |
|---|---|
| *AxesGrid* | alias of *ImageGrid* |
| *CbarAxesBase*(*args, orientation, **kwargs) | |
| *Grid*(fig, rect, nrows_ncols[, ngrids, ...]) | A grid of Axes. |
| *ImageGrid*(fig, rect, nrows_ncols[, ngrids, ...]) | A grid of Axes for Image display. |

### mpl_toolkits.axes_grid1.axes_grid.AxesGrid

mpl_toolkits.axes_grid1.axes_grid.**AxesGrid**

alias of *ImageGrid*

### mpl_toolkits.axes_grid1.axes_grid.CbarAxesBase

**class** mpl_toolkits.axes_grid1.axes_grid.**CbarAxesBase**(**args*, *orientation*, ***kwargs*)

Bases: object

**colorbar**(*mappable*, ***kwargs*)

**toggle_label**(*b*)

[*Deprecated*]

#### Notes

Deprecated since version 3.8: Use ax.tick_params and colorbar.set_label instead.

### mpl_toolkits.axes_grid1.axes_grid.Grid

**class** mpl_toolkits.axes_grid1.axes_grid.**Grid**(*fig*, *rect*, *nrows_ncols*, *ngrids=None*, *direction='row'*, *axes_pad=0.02*, *, *share_all=False*, *share_x=True*, *share_y=True*, *label_mode='L'*, *axes_class=None*, *aspect=False*)

Bases: `object`

A grid of Axes.

In Matplotlib, the Axes location (and size) is specified in normalized figure coordinates. This may not be ideal for images that needs to be displayed with a given aspect ratio; for example, it is difficult to display multiple images of a same size with some fixed padding between them. AxesGrid can be used in such case.

**Attributes**

**axes_all**

[list of Axes] A flat list of Axes. Note that you can also access this directly from the grid. The following is equivalent

```
grid[i] == grid.axes_all[i]
len(grid) == len(grid.axes_all)
```

**axes_column**

[list of list of Axes] A 2D list of Axes where the first index is the column. This results in the usage pattern `grid.axes_column[col][row]`.

**axes_row**

[list of list of Axes] A 2D list of Axes where the first index is the row. This results in the usage pattern `grid.axes_row[row][col]`.

**axes_llc**

[Axes] The Axes in the lower left corner.

**ngrids**

[int] Number of Axes in the grid.

**Parameters**

**fig**

[*Figure*] The parent figure.

**rect**

[(float, float, float, float), (int, int, int), int, or *SubplotSpec*] The axes position, as a `(left, bottom, width, height)` tuple, as a three-digit subplot position code (e.g., `(1, 2, 1)` or `121`), or as a *SubplotSpec*.

**nrows_ncols**

[(int, int)] Number of rows and columns in the grid.

**ngrids**

[int or None, default: None] If not None, only the first *ngrids* axes in the grid are created.

**direction**

[{"row", "column"}, default: "row"] Whether axes are created in row-major ("row by row") or column-major order ("column by column"). This also affects the order in which axes are accessed using indexing (`grid[index]`).

**axes_pad**

[float or (float, float), default: 0.02] Padding or (horizontal padding, vertical padding) between axes, in inches.

**share_all**

[bool, default: False] Whether all axes share their x- and y-axis. Overrides *share_x* and *share_y*.

**share_x**

[bool, default: True] Whether all axes of a column share their x-axis.

**share_y**

[bool, default: True] Whether all axes of a row share their y-axis.

**label_mode**

[{"L", "1", "all", "keep"}, default: "L"] Determines which axes will get tick labels:

- "L": All axes on the left column get vertical tick labels; all axes on the bottom row get horizontal tick labels.
- "1": Only the bottom left axes is labelled.
- "all": All axes are labelled.
- "keep": Do not do anything.

**axes_class**

[subclass of *matplotlib.axes.Axes*, default: *mpl_axes.Axes*] The type of Axes to create.

**aspect**

[bool, default: False] Whether the axes aspect ratio follows the aspect ratio of the data limits.

**get_aspect**()

Return the aspect of the SubplotDivider.

**get_axes_locator**()

**get_axes_pad**()

Return the axes padding.

**Returns**

**hpad, vpad**

Padding (horizontal pad, vertical pad) in inches.

**get_divider**()

**get_geometry**()

Return the number of rows and columns of the grid as (nrows, ncols).

**set_aspect**(*aspect*)

Set the aspect of the SubplotDivider.

**set_axes_locator**(*locator*)

**set_axes_pad**(*axes_pad*)

Set the padding between the axes.

**Parameters**

**axes_pad**

[(float, float)] The padding (horizontal pad, vertical pad) in inches.

**set_label_mode**(*mode*)

Define which axes have tick labels.

**Parameters**

**mode**

[{"L", "1", "all", "keep"}] The label mode:

- "L": All axes on the left column get vertical tick labels; all axes on the bottom row get horizontal tick labels.
- "1": Only the bottom left axes is labelled.
- "all": All axes are labelled.
- "keep": Do not do anything.

### mpl_toolkits.axes_grid1.axes_grid.ImageGrid

**class** mpl_toolkits.axes_grid1.axes_grid.**ImageGrid**(*fig*, *rect*, *nrows_ncols*, *ngrids=None*, *direction='row'*, *axes_pad=0.02*, ***, *share_all=False*, *aspect=True*, *label_mode='L'*, *cbar_mode=None*, *cbar_location='right'*, *cbar_pad=None*, *cbar_size='5%'*, *cbar_set_cax=True*, *axes_class=None*)

Bases: *Grid*

A grid of Axes for Image display.

This class is a specialization of *Grid* for displaying a grid of images. In particular, it forces all axes in a column to share their x-axis and all axes in a row to share their y-axis. It further provides helpers to add colorbars to some or all axes.

**Parameters**

**fig**

[*Figure*] The parent figure.

**rect**

[(float, float, float, float) or int] The axes position, as a `(left, bottom, width, height)` tuple or as a three-digit subplot position code (e.g., "121").

**nrows_ncols**

[(int, int)] Number of rows and columns in the grid.

**ngrids**

[int or None, default: None] If not None, only the first *ngrids* axes in the grid are created.

**direction**

[{"row", "column"}, default: "row"] Whether axes are created in row-major ("row by row") or column-major order ("column by column"). This also affects the order in which axes are accessed using indexing (`grid[index]`).

**axes_pad**

[float or (float, float), default: 0.02in] Padding or (horizontal padding, vertical padding) between axes, in inches.

**share_all**

[bool, default: False] Whether all axes share their x- and y-axis. Note that in any case, all axes in a column share their x-axis and all axes in a row share their y-axis.

**aspect**

[bool, default: True] Whether the axes aspect ratio follows the aspect ratio of the data limits.

**label_mode**

[{"L", "1", "all"}, default: "L"] Determines which axes will get tick labels:

- "L": All axes on the left column get vertical tick labels; all axes on the bottom row get horizontal tick labels.
- "1": Only the bottom left axes is labelled.
- "all": all axes are labelled.

**cbar_mode**

[{"each", "single", "edge", None}, default: None] Whether to create a colorbar for "each" axes, a "single" colorbar for the entire grid, colorbars only for axes on the "edge" determined by *cbar_location*, or no colorbars. The colorbars are stored in the `cbar_axes` attribute.

**cbar_location**

[{"left", "right", "bottom", "top"}, default: "right"]

**cbar_pad**

[float, default: None] Padding between the image axes and the colorbar axes.

**cbar_size**

[size specification (see `Size.from_any`), default: "5%"] Colorbar size.

**cbar_set_cax**

[bool, default: True] If True, each axes in the grid has a *cax* attribute that is bound to associated *cbar_axes*.

**axes_class**

[subclass of *matplotlib.axes.Axes*, default: None]

## mpl_toolkits.axes_grid1.axes_rgb

### Classes

| | |
|---|---|
| *RGBAxes*(*args[, pad]) | 4-panel *imshow* (RGB, R, G, B). |

## mpl_toolkits.axes_grid1.axes_rgb.RGBAxes

**class** mpl_toolkits.axes_grid1.axes_rgb.**RGBAxes**(**args*, *pad=0*, ***kwargs*)

Bases: object

4-panel *imshow* (RGB, R, G, B).

Layout:

Subclasses can override the _defaultAxesClass attribute. By default RGBAxes uses *mpl_axes.Axes*.

**Attributes**

**RGB**

[_defaultAxesClass] The Axes object for the three-channel *imshow*.

**R**

[_defaultAxesClass] The Axes object for the red channel *imshow*.

**G**

[_defaultAxesClass] The Axes object for the green channel *imshow*.

**B**

[_defaultAxesClass] The Axes object for the blue channel *imshow*.

**Parameters**

**pad**

[float, default: 0] Fraction of the Axes height to put as padding.

**axes_class**

[*Axes*] Axes class to use. If not provided, _defaultAxesClass is used.

***args**

Forwarded to *axes_class* init for the RGB Axes

****kwargs**

Forwarded to *axes_class* init for the RGB, R, G, and B Axes

**imshow_rgb**(*r*, *g*, *b*, ***kwargs*)

Create the four images {rgb, r, g, b}.

**Parameters**

**r, g, b**

[array-like] The red, green, and blue arrays.

****kwargs**

Forwarded to *imshow* calls for the four images.

**Returns**

**rgb**

[*AxesImage*]

**r**

[*AxesImage*]

**g**

[*AxesImage*]

**b**

[*AxesImage*]

## Examples using `mpl_toolkits.axes_grid1.axes_rgb.RGBAxes`

- *Showing RGB channels using RGBAxes*

## Functions

| |
|---|
| *make_rgb_axes*(ax[, pad, axes_class]) |
| **Parameters** |

### mpl_toolkits.axes_grid1.axes_rgb.make_rgb_axes

mpl_toolkits.axes_grid1.axes_rgb.**make_rgb_axes**(*ax*, *pad=0.01*, *axes_class=None*, ***kwargs*)

**Parameters**

**ax**

[*Axes*] Axes instance to create the RGB Axes in.

**pad**

[float, optional] Fraction of the Axes height to pad.

**axes_class**

[*matplotlib.axes.Axes* or None, optional] Axes class to use for the R, G, and B Axes. If None, use the same class as *ax*.

****kwargs**

Forwarded to *axes_class* init for the R, G, and B Axes.

### Examples using `mpl_toolkits.axes_grid1.axes_rgb.make_rgb_axes`

- *Showing RGB channels using RGBAxes*

### mpl_toolkits.axes_grid1.axes_size

Provides classes of simple units that will be used with *AxesDivider* class (or others) to determine the size of each Axes. The unit classes define `get_size` method that returns a tuple of two floats, meaning relative and absolute sizes, respectively.

Note that this class is nothing more than a simple tuple of two floats. Take a look at the Divider class to see how these two values are used.

### Classes

| | |
|---|---|
| *Add*(a, b) | Sum of two sizes. |
| *AxesX*(axes[, aspect, ref_ax]) | Scaled size whose relative part corresponds to the data width of the *axes* multiplied by the *aspect*. |
| *AxesY*(axes[, aspect, ref_ax]) | Scaled size whose relative part corresponds to the data height of the *axes* multiplied by the *aspect*. |
| *Fixed*(fixed_size) | Simple fixed size with absolute part = *fixed_size* and relative part = 0. |
| *Fraction*(fraction, ref_size) | An instance whose size is a *fraction* of the *ref_size*. |
| *MaxExtent*(artist_list, w_or_h) | Size whose absolute part is either the largest width or the largest height of the given *artist_list*. |
| *MaxHeight*(artist_list) | Size whose absolute part is the largest height of the given *artist_list*. |
| *MaxWidth*(artist_list) | Size whose absolute part is the largest width of the given *artist_list*. |
| *Scalable* | alias of *Scaled* |
| *Scaled*(scalable_size) | Simple scaled(?) size with absolute part = 0 and relative part = *scalable_size*. |

### mpl_toolkits.axes_grid1.axes_size.Add

**class** mpl_toolkits.axes_grid1.axes_size.**Add**(*a*, *b*)

Bases: _Base

Sum of two sizes.

**get_size**(*renderer*)

Return two-float tuple with relative and absolute sizes.

### mpl_toolkits.axes_grid1.axes_size.AxesX

**class** mpl_toolkits.axes_grid1.axes_size.**AxesX**(*axes*, *aspect=1.0*, *ref_ax=None*)

Bases: _Base

Scaled size whose relative part corresponds to the data width of the *axes* multiplied by the *aspect*.

**get_size**(*renderer*)

Return two-float tuple with relative and absolute sizes.

### Examples using `mpl_toolkits.axes_grid1.axes_size.AxesX`

- *HBoxDivider and VBoxDivider demo*
- *Simple axes divider 3*

### mpl_toolkits.axes_grid1.axes_size.AxesY

**class** mpl_toolkits.axes_grid1.axes_size.**AxesY**(*axes*, *aspect=1.0*, *ref_ax=None*)

Bases: `_Base`

Scaled size whose relative part corresponds to the data height of the *axes* multiplied by the *aspect*.

**get_size**(*renderer*)

Return two-float tuple with relative and absolute sizes.

### Examples using `mpl_toolkits.axes_grid1.axes_size.AxesY`

- *HBoxDivider and VBoxDivider demo*
- *Simple axes divider 3*

### mpl_toolkits.axes_grid1.axes_size.Fixed

**class** mpl_toolkits.axes_grid1.axes_size.**Fixed**(*fixed_size*)

Bases: `_Base`

Simple fixed size with absolute part = *fixed_size* and relative part = 0.

**get_size**(*renderer*)

Return two-float tuple with relative and absolute sizes.

### Examples using `mpl_toolkits.axes_grid1.axes_size.Fixed`

- *HBoxDivider and VBoxDivider demo*
- *Axes with a fixed physical size*
- *Simple Axes Divider 1*
- *Simple axes divider 3*

### mpl_toolkits.axes_grid1.axes_size.Fraction

**class** mpl_toolkits.axes_grid1.axes_size.**Fraction**(*fraction*, *ref_size*)

Bases: _Base

An instance whose size is a *fraction* of the *ref_size*.

```
>>> s = Fraction(0.3, AxesX(ax))
```

**get_size**(*renderer*)

Return two-float tuple with relative and absolute sizes.

### mpl_toolkits.axes_grid1.axes_size.MaxExtent

**class** mpl_toolkits.axes_grid1.axes_size.**MaxExtent**(*artist_list*, *w_or_h*)

Bases: _Base

Size whose absolute part is either the largest width or the largest height of the given *artist_list*.

**add_artist**(*a*)

**get_size**(*renderer*)

Return two-float tuple with relative and absolute sizes.

### mpl_toolkits.axes_grid1.axes_size.MaxHeight

**class** mpl_toolkits.axes_grid1.axes_size.**MaxHeight**(*artist_list*)

Bases: *MaxExtent*

Size whose absolute part is the largest height of the given *artist_list*.

### mpl_toolkits.axes_grid1.axes_size.MaxWidth

**class** mpl_toolkits.axes_grid1.axes_size.**MaxWidth**(*artist_list*)

Bases: *MaxExtent*

Size whose absolute part is the largest width of the given *artist_list*.

### mpl_toolkits.axes_grid1.axes_size.Scalable

mpl_toolkits.axes_grid1.axes_size.**Scalable**

alias of *Scaled*

### mpl_toolkits.axes_grid1.axes_size.Scaled

**class** mpl_toolkits.axes_grid1.axes_size.**Scaled**(*scalable_size*)

Bases: _Base

Simple scaled(?) size with absolute part = 0 and relative part = *scalable_size*.

**get_size**(*renderer*)

Return two-float tuple with relative and absolute sizes.

### Examples using mpl_toolkits.axes_grid1.axes_size.Scaled

- *HBoxDivider and VBoxDivider demo*
- *Axes with a fixed physical size*
- *Simple Axes Divider 1*

### Functions

| | |
|---|---|
| *from_any*(size[, fraction_ref]) | Create a Fixed unit when the first argument is a float, or a Fraction unit if that is a string that ends with %. |

### mpl_toolkits.axes_grid1.axes_size.from_any

mpl_toolkits.axes_grid1.axes_size.**from_any**(*size*, *fraction_ref=None*)

Create a Fixed unit when the first argument is a float, or a Fraction unit if that is a string that ends with %. The second argument is only meaningful when Fraction unit is created.

```
>>> from mpl_toolkits.axes_grid1.axes_size import from_any
>>> a = from_any(1.2) # => Fixed(1.2)
>>> from_any("50%", a) # => Fraction(0.5, a)
```

## mpl_toolkits.axes_grid1.inset_locator

A collection of functions and objects for creating or placing inset axes.

### Classes

| | |
|---|---|
| *AnchoredLocatorBase*(bbox_to_anchor, ...[, ...]) | **Parameters** |
| *AnchoredSizeLocator*(bbox_to_anchor, x_size, ...) | **Parameters** |
| *AnchoredZoomLocator*(parent_axes, zoom, loc) | **Parameters** |
| *BboxConnector*(bbox1, bbox2, loc1[, loc2]) | Connect two bboxes with a straight line. |
| *BboxConnectorPatch*(bbox1, bbox2, loc1a, ...) | Connect two bboxes with a quadrilateral. |
| *BboxPatch*(bbox, **kwargs) | Patch showing the shape bounded by a Bbox. |
| *InsetPosition*(parent, lbwh) | [*Deprecated*] |

## mpl_toolkits.axes_grid1.inset_locator.AnchoredLocatorBase

**class** mpl_toolkits.axes_grid1.inset_locator.**AnchoredLocatorBase**(*bbox_to_anchor*, *offsetbox*, *loc*, *borderpad=0.5*, *bbox_transform=None*)

Bases: *AnchoredOffsetbox*

**Parameters**

**loc**

[str] The box location. Valid locations are 'upper left', 'upper center', 'upper right', 'center left', 'center', 'center right', 'lower left', 'lower center', 'lower right'. For backward compatibility, numeric values are accepted as well. See the parameter *loc* of *Legend* for details.

**pad**

[float, default: 0.4] Padding around the child as fraction of the fontsize.

**borderpad**

[float, default: 0.5] Padding between the offsetbox frame and the *bbox_to_anchor*.

**child**

[*OffsetBox*] The box that will be anchored.

**prop**

[*FontProperties*] This is only used as a reference for paddings. If not given, *rcParams["legend.fontsize"]* (default: `'medium'`) is used.

**frameon**

[bool] Whether to draw a frame around the box.

**bbox_to_anchor**

[*BboxBase*, 2-tuple, or 4-tuple of floats] Box that is used to position the legend in conjunction with *loc*.

**bbox_transform**

[None or *matplotlib.transforms.Transform*] The transform for the bounding box (*bbox_to_anchor*).

****kwargs**

All other parameters are passed on to *OffsetBox*.

## Notes

See *Legend* for a detailed description of the anchoring mechanism.

**__call__**(*ax*, *renderer*)

Call self as a function.

**draw**(*renderer*)

Update the location of children if necessary and draw them to the given *renderer*.

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *bbox_to_anchor=<UNSET>*, *child=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *gid=<UNSET>*, *height=<UNSET>*, *in_layout=<UNSET>*, *label=<UNSET>*, *mouseover=<UNSET>*, *offset=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *width=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| agg_filt | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image |
| alpha | scalar or None |
| animated | bool |
| bbox_to_ | unknown |
| child | unknown |
| clip_box | BboxBase or None |
| clip_on | bool |
| clip_path | Patch or (Path, Transform) or None |
| figure | Figure |
| gid | str |
| height | float |
| in_layout | bool |
| label | object |
| mouseover | bool |
| offset | (float, float) or callable |
| path_eff | list of AbstractPathEffect |
| picker | None or bool or float or callable |
| rasterized | bool |
| sketch_p | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| transform | Transform |
| url | str |
| visible | bool |
| width | float |
| zorder | float |

### mpl_toolkits.axes_grid1.inset_locator.AnchoredSizeLocator

**class** mpl_toolkits.axes_grid1.inset_locator.**AnchoredSizeLocator**(*bbox_to_anchor*, *x_size*, *y_size*, *loc*, *borderpad=0.5*, *bbox_transform=None*)

Bases: AnchoredLocatorBase

**Parameters**

**loc**

[str] The box location. Valid locations are 'upper left', 'upper center', 'upper right', 'center left', 'center', 'center right', 'lower left', 'lower center', 'lower right'. For backward compatibility, numeric values are accepted as well. See the parameter *loc* of `Legend` for details.

**pad**

[float, default: 0.4] Padding around the child as fraction of the fontsize.

**borderpad**

[float, default: 0.5] Padding between the offsetbox frame and the *bbox_to_anchor*.

**child**

[`OffsetBox`] The box that will be anchored.

**prop**

[`FontProperties`] This is only used as a reference for paddings. If not given, `rcParams["legend.fontsize"]` (default: `'medium'`) is used.

**frameon**

[bool] Whether to draw a frame around the box.

**bbox_to_anchor**

[`BboxBase`, 2-tuple, or 4-tuple of floats] Box that is used to position the legend in conjunction with *loc*.

**bbox_transform**

[None or `matplotlib.transforms.Transform`] The transform for the bounding box (*bbox_to_anchor*).

****kwargs**

All other parameters are passed on to `OffsetBox`.

## Notes

See `Legend` for a detailed description of the anchoring mechanism.

**get_bbox**(*renderer*)

Return the bbox of the offsetbox, ignoring parent offsets.

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *bbox_to_anchor=<UNSET>*, *child=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *gid=<UNSET>*, *height=<UNSET>*, *in_layout=<UNSET>*, *label=<UNSET>*, *mouseover=<UNSET>*, *offset=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *width=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| agg_filt | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image |
| alpha | scalar or None |
| animated | bool |
| bbox_to_ | unknown |
| child | unknown |
| clip_box | BboxBase or None |
| clip_on | bool |
| clip_pat | Patch or (Path, Transform) or None |
| figure | Figure |
| gid | str |
| height | float |
| in_layou | bool |
| label | object |
| mouseove | bool |
| offset | (float, float) or callable |
| path_eff | list of AbstractPathEffect |
| picker | None or bool or float or callable |
| rasterized | bool |
| sketch_p | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| transform | Transform |
| url | str |
| visible | bool |
| width | float |
| zorder | float |

### mpl_toolkits.axes_grid1.inset_locator.AnchoredZoomLocator

**class** mpl_toolkits.axes_grid1.inset_locator.**AnchoredZoomLocator**(*parent_axes*, *zoom*, *loc*, *borderpad=0.5*, *bbox_to_anchor=None*, *bbox_transform=None*)

Bases: *AnchoredLocatorBase*

**Parameters**

**loc**

[str] The box location. Valid locations are 'upper left', 'upper center', 'upper right', 'center left', 'center', 'center right', 'lower left', 'lower center', 'lower right'. For backward compatibility, numeric values are accepted as well. See the parameter *loc* of *Legend* for details.

**pad**

[float, default: 0.4] Padding around the child as fraction of the fontsize.

**borderpad**

[float, default: 0.5] Padding between the offsetbox frame and the *bbox_to_anchor*.

**child**

[*OffsetBox*] The box that will be anchored.

**prop**

[*FontProperties*] This is only used as a reference for paddings. If not given, *rcParams["legend.fontsize"]* (default: `'medium'`) is used.

**frameon**

[bool] Whether to draw a frame around the box.

**bbox_to_anchor**

[*BboxBase*, 2-tuple, or 4-tuple of floats] Box that is used to position the legend in conjunction with *loc*.

**bbox_transform**

[None or *matplotlib.transforms.Transform*] The transform for the bounding box (*bbox_to_anchor*).

****kwargs**

All other parameters are passed on to *OffsetBox*.

### Notes

See *Legend* for a detailed description of the anchoring mechanism.

**get_bbox**(*renderer*)

Return the bbox of the offsetbox, ignoring parent offsets.

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *bbox_to_anchor=<UNSET>*, *child=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *gid=<UNSET>*, *height=<UNSET>*, *in_layout=<UNSET>*, *label=<UNSET>*, *mouseover=<UNSET>*, *offset=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *width=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| agg_filt | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image |
| alpha | scalar or None |
| animated | bool |
| bbox_to_ | unknown |
| child | unknown |
| clip_box | BboxBase or None |
| clip_on | bool |
| clip_pat | Patch or (Path, Transform) or None |
| figure | Figure |
| gid | str |
| height | float |
| in_layou | bool |
| label | object |
| mouseove | bool |
| offset | (float, float) or callable |
| path_eff | list of AbstractPathEffect |
| picker | None or bool or float or callable |
| rasterized | bool |
| sketch_p | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| transform | Transform |
| url | str |
| visible | bool |
| width | float |
| zorder | float |

## mpl_toolkits.axes_grid1.inset_locator.BboxConnector

**class** mpl_toolkits.axes_grid1.inset_locator.**BboxConnector**(*bbox1*, *bbox2*, *loc1*, *loc2=None*, ***kwargs*)

Bases: Patch

Connect two bboxes with a straight line.

**Parameters**

**bbox1, bbox2**

[*Bbox*] Bounding boxes to connect.

**loc1, loc2**

[{1, 2, 3, 4}] Corner of *bbox1* and *bbox2* to draw the line. Valid values are:

```
'upper right'  : 1,
'upper left'   : 2,
'lower left'   : 3,
'lower right'  : 4
```

*loc2* is optional and defaults to *loc1*.

****kwargs**

Patch properties for the line drawn. Valid arguments include:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and returr |
| *alpha* | unknown |
| *animated* | bool |
| *antialiased* or aa | bool or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* | *color* |
| *edgecolor* or ec | *color* or None |
| *facecolor* or fc | *color* or None |
| *figure* | *Figure* |
| *fill* | bool |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float or None |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |

Table 174 – continued from previ

| Property | Description |
|---|---|
| *zorder* | float |

**static connect_bbox**(*bbox1*, *bbox2*, *loc1*, *loc2=None*)

Construct a *Path* connecting corner *loc1* of *bbox1* to corner *loc2* of *bbox2*, where parameters behave as documented as for the *BboxConnector* constructor.

**static get_bbox_edge_pos**(*bbox*, *loc*)

Return the `(x, y)` coordinates of corner *loc* of *bbox*; parameters behave as documented for the *BboxConnector* constructor.

**get_path**()

Return the path of this patch.

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *antialiased=<UNSET>*, *capstyle=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *color=<UNSET>*, *edgecolor=<UNSET>*, *facecolor=<UNSET>*, *fill=<UNSET>*, *gid=<UNSET>*, *hatch=<UNSET>*, *in_layout=<UNSET>*, *joinstyle=<UNSET>*, *label=<UNSET>*, *linestyle=<UNSET>*, *linewidth=<UNSET>*, *mouseover=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* or aa | bool or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* | *color* |
| *edgecolor* or ec | *color* or None |
| *facecolor* or fc | *color* or None |
| *figure* | *Figure* |
| *fill* | bool |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |

Table 175 – continued from previous

| Property | Description |
|---|---|
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float or None |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *zorder* | float |

### Examples using `mpl_toolkits.axes_grid1.inset_locator.BboxConnector`

- *Axes Zoom Effect*

### mpl_toolkits.axes_grid1.inset_locator.BboxConnectorPatch

**class** mpl_toolkits.axes_grid1.inset_locator.**BboxConnectorPatch**(*bbox1*, *bbox2*, *loc1a*, *loc2a*, *loc1b*, *loc2b*, ***kwargs*)

Bases: *BboxConnector*

Connect two bboxes with a quadrilateral.

The quadrilateral is specified by two lines that start and end at corners of the bboxes. The four sides of the quadrilateral are defined by the two lines given, the line between the two corners specified in *bbox1* and the line between the two corners specified in *bbox2*.

**Parameters**

**bbox1, bbox2**

[*Bbox*] Bounding boxes to connect.

**loc1a, loc2a, loc1b, loc2b**

[{1, 2, 3, 4}] The first line connects corners *loc1a* of *bbox1* and *loc2a* of *bbox2*; the second line connects corners *loc1b* of *bbox1* and *loc2b* of *bbox2*. Valid values are:

```
'upper right'  : 1,
'upper left'   : 2,
'lower left'   : 3,
'lower right'  : 4
```

****kwargs**

Patch properties for the line drawn:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and return |
| *alpha* | unknown |
| *animated* | bool |
| *antialiased* or aa | bool or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* | *color* |
| *edgecolor* or ec | *color* or None |
| *facecolor* or fc | *color* or None |
| *figure* | *Figure* |
| *fill* | bool |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float or None |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *zorder* | float |

**get_path**()

Return the path of this patch.

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *antialiased=<UNSET>*, *capstyle=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *color=<UNSET>*, *edgecolor=<UNSET>*, *facecolor=<UNSET>*, *fill=<UNSET>*, *gid=<UNSET>*, *hatch=<UNSET>*, *in_layout=<UNSET>*, *joinstyle=<UNSET>*, *label=<UNSET>*, *linestyle=<UNSET>*, *linewidth=<UNSET>*, *mouseover=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* or aa | bool or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* | *color* |
| *edgecolor* or ec | *color* or None |
| *facecolor* or fc | *color* or None |
| *figure* | *Figure* |
| *fill* | bool |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float or None |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *zorder* | float |

### Examples using `mpl_toolkits.axes_grid1.inset_locator.BboxConnectorPatch`

- *Axes Zoom Effect*

## mpl_toolkits.axes_grid1.inset_locator.BboxPatch

**class** mpl_toolkits.axes_grid1.inset_locator.**BboxPatch**(*bbox*, ***kwargs*)

Bases: *Patch*

Patch showing the shape bounded by a Bbox.

**Parameters**

**bbox**

[*Bbox*] Bbox to use for the extents of this patch.

****kwargs**

Patch properties. Valid arguments include:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and retur |
| *alpha* | unknown |
| *animated* | bool |
| *antialiased* or aa | bool or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* | *color* |
| *edgecolor* or ec | *color* or None |
| *facecolor* or fc | *color* or None |
| *figure* | *Figure* |
| *fill* | bool |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float or None |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |

Table 178 – continued from previ

| Property | Description |
|---|---|
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *zorder* | float |

**get_path**()

Return the path of this patch.

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *antialiased=<UNSET>*, *capstyle=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *color=<UNSET>*, *edgecolor=<UNSET>*, *facecolor=<UNSET>*, *fill=<UNSET>*, *gid=<UNSET>*, *hatch=<UNSET>*, *in_layout=<UNSET>*, *joinstyle=<UNSET>*, *label=<UNSET>*, *linestyle=<UNSET>*, *linewidth=<UNSET>*, *mouseover=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* or aa | bool or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* | *color* |
| *edgecolor* or ec | *color* or None |
| *facecolor* or fc | *color* or None |
| *figure* | *Figure* |
| *fill* | bool |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float or None |
| *mouseover* | bool |

Table 179 – continued from previous

| Property | Description |
|---|---|
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *zorder* | float |

### Examples using `mpl_toolkits.axes_grid1.inset_locator.BboxPatch`

- *Axes Zoom Effect*

## mpl_toolkits.axes_grid1.inset_locator.InsetPosition

**class** mpl_toolkits.axes_grid1.inset_locator.**InsetPosition**(*parent*, *lbwh*)

Bases: object

[*Deprecated*]

### Notes

Deprecated since version 3.8: Use Axes.inset_axes instead.

An object for positioning an inset axes.

This is created by specifying the normalized coordinates in the axes, instead of the figure.

**Parameters**

**parent**

[*Axes*] Axes to use for normalizing coordinates.

**lbwh**

[iterable of four floats] The left edge, bottom edge, width, and height of the inset axes, in units of the normalized coordinate of the *parent* axes.

**See also:**

***matplotlib.axes.Axes.set_axes_locator()***

### Examples

The following bounds the inset axes to a box with 20% of the parent axes height and 40% of the width. The size of the axes specified ([0, 0, 1, 1]) ensures that the axes completely fills the bounding box:

```
>>> parent_axes = plt.gca()
>>> ax_ins = plt.axes([0, 0, 1, 1])
>>> ip = InsetPosition(parent_axes, [0.5, 0.1, 0.4, 0.2])
>>> ax_ins.set_axes_locator(ip)
```

**__call__**(*ax*, *renderer*)

Call self as a function.

## Functions

| | |
|---|---|
| *inset_axes*(parent_axes, width, height[, ...]) | Create an inset axes with a given width and height. |
| *mark_inset*(parent_axes, inset_axes, loc1, ...) | Draw a box to mark the location of an area represented by an inset axes. |
| *zoomed_inset_axes*(parent_axes, zoom[, loc, ...]) | Create an anchored inset axes by scaling a parent axes. |

### mpl_toolkits.axes_grid1.inset_locator.inset_axes

mpl_toolkits.axes_grid1.inset_locator.**inset_axes**(*parent_axes*, *width*, *height*, *loc='upper right'*, *bbox_to_anchor=None*, *bbox_transform=None*, *axes_class=None*, *axes_kwargs=None*, *borderpad=0.5*)

Create an inset axes with a given width and height.

Both sizes used can be specified either in inches or percentage. For example,:

```
inset_axes(parent_axes, width='40%', height='30%', loc='lower left')
```

creates in inset axes in the lower left corner of *parent_axes* which spans over 30% in height and 40% in width of the *parent_axes*. Since the usage of *inset_axes* may become slightly tricky when exceeding such standard cases, it is recommended to read *the examples*.

**Parameters**

**parent_axes**

[*matplotlib.axes.Axes*] Axes to place the inset axes.

**width, height**

[float or str] Size of the inset axes to create. If a float is provided, it is the size in inches, e.g. *width=1.3*. If a string is provided, it is the size in relative units, e.g. *width='40%'*. By default, i.e. if neither *bbox_to_anchor* nor *bbox_transform* are specified, those are relative to the parent_axes. Otherwise, they are to be understood relative to the bounding box provided via *bbox_to_anchor*.

**loc**

[str, default: 'upper right'] Location to place the inset axes. Valid locations are 'upper left', 'upper center', 'upper right', 'center left', 'center', 'center right', 'lower left', 'lower center', 'lower right'. For backward compatibility, numeric values are accepted as well. See the parameter *loc* of `Legend` for details.

**bbox_to_anchor**

[tuple or `BboxBase`, optional] Bbox that the inset axes will be anchored to. If None, a tuple of (0, 0, 1, 1) is used if *bbox_transform* is set to *parent_axes.transAxes* or *parent_axes.figure.transFigure*. Otherwise, *parent_axes.bbox* is used. If a tuple, can be either [left, bottom, width, height], or [left, bottom]. If the kwargs *width* and/or *height* are specified in relative units, the 2-tuple [left, bottom] cannot be used. Note that, unless *bbox_transform* is set, the units of the bounding box are interpreted in the pixel coordinate. When using *bbox_to_anchor* with tuple, it almost always makes sense to also specify a *bbox_transform*. This might often be the axes transform *parent_axes.transAxes*.

**bbox_transform**

[`Transform`, optional] Transformation for the bbox that contains the inset axes. If None, a `transforms.IdentityTransform` is used. The value of *bbox_to_anchor* (or the return value of its get_points method) is transformed by the *bbox_transform* and then interpreted as points in the pixel coordinate (which is dpi dependent). You may provide *bbox_to_anchor* in some normalized coordinate, and give an appropriate transform (e.g., *parent_axes.transAxes*).

**axes_class**

[`Axes` type, default: `HostAxes`] The type of the newly created inset axes.

**axes_kwargs**

[dict, optional] Keyword arguments to pass to the constructor of the inset axes. Valid arguments include:

| Property | Description |
|---|---|
| `adjustable` | {'box', 'datalim'} |
| `agg_filter` | a filter function, which takes a (m, n, 3) float array and a dpi v |
| `alpha` | scalar or None |
| `anchor` | (float, float) or {'C', 'SW', 'S', 'SE', 'E', 'NE', ...} |
| `animated` | bool |

Table 180 – continued fro

| Property | Description |
|---|---|
| *aspect* | {'auto', 'equal'} or float |
| *autoscale_on* | bool |
| *autoscalex_on* | unknown |
| *autoscaley_on* | unknown |
| *axes_locator* | Callable[[Axes, Renderer], Bbox] |
| *axisbelow* | bool or 'line' |
| *box_aspect* | float or None |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *facecolor* or fc | *color* |
| *figure* | *Figure* |
| *forward_navigation_events* | bool or "auto" |
| *frame_on* | bool |
| *gid* | str |
| *in_layout* | bool |
| *label* | object |
| *mouseover* | bool |
| *navigate* | bool |
| *navigate_mode* | unknown |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *position* | [left, bottom, width, height] or *Bbox* |
| *prop_cycle* | *Cycler* |
| *rasterization_zorder* | float or None |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *subplotspec* | unknown |
| *title* | str |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *xbound* | (lower: float, upper: float) |
| *xlabel* | str |
| *xlim* | (left: float, right: float) |
| *xmargin* | float greater than -0.5 |
| *xscale* | unknown |
| *xticklabels* | unknown |
| *xticks* | unknown |
| *ybound* | (lower: float, upper: float) |
| *ylabel* | str |
| *ylim* | (bottom: float, top: float) |
| *ymargin* | float greater than -0.5 |

Table 180 – continued fro

| Property | Description |
|---|---|
| *yscale* | unknown |
| *yticklabels* | unknown |
| *yticks* | unknown |
| *zorder* | float |

**borderpad**

[float, default: 0.5] Padding between inset axes and the bbox_to_anchor. The units are axes font size, i.e. for a default font size of 10 points *borderpad = 0.5* is equivalent to a padding of 5 points.

**Returns**

**inset_axes**

[*axes_class*] Inset axes object created.

## Notes

The meaning of *bbox_to_anchor* and *bbox_to_transform* is interpreted differently from that of legend. The value of bbox_to_anchor (or the return value of its get_points method; the default is *parent_axes.bbox*) is transformed by the bbox_transform (the default is Identity transform) and then interpreted as points in the pixel coordinate (which is dpi dependent).

Thus, following three calls are identical and creates an inset axes with respect to the *parent_axes*:

```
axins = inset_axes(parent_axes, "30%", "40%")
axins = inset_axes(parent_axes, "30%", "40%",
                   bbox_to_anchor=parent_axes.bbox)
axins = inset_axes(parent_axes, "30%", "40%",
                   bbox_to_anchor=(0, 0, 1, 1),
                   bbox_transform=parent_axes.transAxes)
```

## Examples using `mpl_toolkits.axes_grid1.inset_locator.inset_axes`

- *Controlling the position and size of colorbars with Inset Axes*
- *Inset locator demo*

### mpl_toolkits.axes_grid1.inset_locator.mark_inset

mpl_toolkits.axes_grid1.inset_locator.**mark_inset**(*parent_axes*, *inset_axes*, *loc1*, *loc2*, ***kwargs*)

Draw a box to mark the location of an area represented by an inset axes.

This function draws a box in *parent_axes* at the bounding box of *inset_axes*, and shows a connection with the inset axes by drawing lines at the corners, giving a "zoomed in" effect.

**Parameters**

**parent_axes**

[*Axes*] Axes which contains the area of the inset axes.

**inset_axes**

[*Axes*] The inset axes.

**loc1, loc2**

[{1, 2, 3, 4}] Corners to use for connecting the inset axes and the area in the parent axes.

****kwargs**

Patch properties for the lines and box drawn:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, and return |
| *alpha* | unknown |
| *animated* | bool |
| *antialiased* or aa | bool or None |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* | *color* |
| *edgecolor* or ec | *color* or None |
| *facecolor* or fc | *color* or None |
| *figure* | *Figure* |
| *fill* | bool |
| *gid* | str |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float or None |
| *mouseover* | bool |

Table 181 – continued from previ

| Property | Description |
|---|---|
| path_effects | list of AbstractPathEffect |
| picker | None or bool or float or callable |
| rasterized | bool |
| sketch_params | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| transform | Transform |
| url | str |
| visible | bool |
| zorder | float |

**Returns**

**pp**

[Patch] The patch drawn to represent the area of the inset axes.

**p1, p2**

[Patch] The patches connecting two corners of the inset axes and its area.

### Examples using `mpl_toolkits.axes_grid1.inset_locator.mark_inset`

- *Inset locator demo 2*

## mpl_toolkits.axes_grid1.inset_locator.zoomed_inset_axes

mpl_toolkits.axes_grid1.inset_locator.**zoomed_inset_axes**(*parent_axes*, *zoom*, *loc='upper right'*, *bbox_to_anchor=None*, *bbox_transform=None*, *axes_class=None*, *axes_kwargs=None*, *borderpad=0.5*)

Create an anchored inset axes by scaling a parent axes. For usage, also see *the examples*.

**Parameters**

**parent_axes**

[Axes] Axes to place the inset axes.

**zoom**

[float] Scaling factor of the data axes. *zoom* > 1 will enlarge the coordinates (i.e., "zoomed in"), while *zoom* < 1 will shrink the coordinates (i.e., "zoomed out").

**loc**

[str, default: 'upper right'] Location to place the inset axes. Valid locations are 'upper left', 'upper center', 'upper right', 'center left', 'center', 'center right', 'lower left', 'lower center', 'lower right'. For backward compatibility, numeric values are accepted as well. See the parameter *loc* of `Legend` for details.

**bbox_to_anchor**

[tuple or `BboxBase`, optional] Bbox that the inset axes will be anchored to. If None, *parent_axes.bbox* is used. If a tuple, can be either [left, bottom, width, height], or [left, bottom]. If the kwargs *width* and/or *height* are specified in relative units, the 2-tuple [left, bottom] cannot be used. Note that the units of the bounding box are determined through the transform in use. When using *bbox_to_anchor* it almost always makes sense to also specify a *bbox_transform*. This might often be the axes transform *parent_axes.transAxes*.

**bbox_transform**

[`Transform`, optional] Transformation for the bbox that contains the inset axes. If None, a `transforms.IdentityTransform` is used (i.e. pixel coordinates). This is useful when not providing any argument to *bbox_to_anchor*. When using *bbox_to_anchor* it almost always makes sense to also specify a *bbox_transform*. This might often be the axes transform *parent_axes.transAxes*. Inversely, when specifying the axes- or figure-transform here, be aware that not specifying *bbox_to_anchor* will use *parent_axes.bbox*, the units of which are in display (pixel) coordinates.

**axes_class**

[`Axes` type, default: `HostAxes`] The type of the newly created inset axes.

**axes_kwargs**

[dict, optional] Keyword arguments to pass to the constructor of the inset axes. Valid arguments include:

| Property | Description |
|---|---|
| `adjustable` | {'box', 'datalim'} |
| `agg_filter` | a filter function, which takes a (m, n, 3) float array and a dpi v |
| `alpha` | scalar or None |
| `anchor` | (float, float) or {'C', 'SW', 'S', 'SE', 'E', 'NE', ...} |
| `animated` | bool |
| `aspect` | {'auto', 'equal'} or float |
| `autoscale_on` | bool |
| `autoscalex_on` | unknown |
| `autoscaley_on` | unknown |
| `axes_locator` | Callable[[Axes, Renderer], Bbox] |
| `axisbelow` | bool or 'line' |
| `box_aspect` | float or None |

Table 182 – continued fro

| Property | Description |
|---|---|
| clip_box | BboxBase or None |
| clip_on | bool |
| clip_path | Patch or (Path, Transform) or None |
| facecolor or fc | color |
| figure | Figure |
| forward_navigation_events | bool or "auto" |
| frame_on | bool |
| gid | str |
| in_layout | bool |
| label | object |
| mouseover | bool |
| navigate | bool |
| navigate_mode | unknown |
| path_effects | list of AbstractPathEffect |
| picker | None or bool or float or callable |
| position | [left, bottom, width, height] or Bbox |
| prop_cycle | Cycler |
| rasterization_zorder | float or None |
| rasterized | bool |
| sketch_params | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| subplotspec | unknown |
| title | str |
| transform | Transform |
| url | str |
| visible | bool |
| xbound | (lower: float, upper: float) |
| xlabel | str |
| xlim | (left: float, right: float) |
| xmargin | float greater than -0.5 |
| xscale | unknown |
| xticklabels | unknown |
| xticks | unknown |
| ybound | (lower: float, upper: float) |
| ylabel | str |
| ylim | (bottom: float, top: float) |
| ymargin | float greater than -0.5 |
| yscale | unknown |
| yticklabels | unknown |
| yticks | unknown |
| zorder | float |

**borderpad**

[float, default: 0.5] Padding between inset axes and the bbox_to_anchor. The units

are axes font size, i.e. for a default font size of 10 points *borderpad = 0.5* is equivalent to a padding of 5 points.

**Returns**

**inset_axes**

[*axes_class*] Inset axes object created.

### Examples using `mpl_toolkits.axes_grid1.inset_locator.zoomed_inset_axes`

- *Inset locator demo 2*

## mpl_toolkits.axes_grid1.mpl_axes

### Classes

| | |
|---|---|
| *Axes*(fig, *args[, facecolor, frameon, ...]) | Build an Axes in a figure. |
| *SimpleAxisArtist*(axis, axisnum, spine) | |
| *SimpleChainedObjects*(objects) | |

## mpl_toolkits.axes_grid1.mpl_axes.Axes

**class** mpl_toolkits.axes_grid1.mpl_axes.**Axes**(*fig*, **args*, *facecolor=None*, *frameon=True*, *sharex=None*, *sharey=None*, *label=''*, *xscale=None*, *yscale=None*, *box_aspect=None*, *forward_navigation_events='auto'*, ***kwargs*)

Bases: *Axes*

Build an Axes in a figure.

**Parameters**

**fig**

[*Figure*] The Axes is built in the *Figure* *fig*.

***args**

`*args` can be a single `(left, bottom, width, height)` rectangle or a single *Bbox*. This specifies the rectangle (in figure coordinates) where the Axes is positioned.

`*args` can also consist of three numbers or a single three-digit number; in the latter case, the digits are considered as independent numbers. The numbers are interpreted as `(nrows, ncols, index)`: `(nrows, ncols)` specifies the size of an array of subplots, and `index` is the 1-based index of the subplot being created. Finally, `*args` can also directly be a *SubplotSpec* instance.

**sharex, sharey**

[*Axes*, optional] The x- or y-*axis* is shared with the x- or y-axis in the input *Axes*.

**frameon**

[bool, default: True] Whether the Axes frame is visible.

**box_aspect**

[float, optional] Set a fixed aspect for the Axes box, i.e. the ratio of height to width. See *set_box_aspect* for details.

**forward_navigation_events**

[bool or "auto", default: "auto"] Control whether pan/zoom events are passed through to Axes below this one. "auto" is *True* for axes with an invisible patch and *False* otherwise.

****kwargs**

Other optional keyword arguments:

| Property | Description |
|---|---|
| *adjustable* | {'box', 'datalim'} |
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi v |
| *alpha* | scalar or None |
| *anchor* | (float, float) or {'C', 'SW', 'S', 'SE', 'E', 'NE', ...} |
| *animated* | bool |
| *aspect* | {'auto', 'equal'} or float |
| *autoscale_on* | bool |
| *autoscalex_on* | unknown |
| *autoscaley_on* | unknown |
| *axes_locator* | Callable[[Axes, Renderer], Bbox] |
| *axisbelow* | bool or 'line' |
| *box_aspect* | float or None |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *facecolor* or fc | *color* |
| *figure* | *Figure* |
| *forward_navigation_events* | bool or "auto" |
| *frame_on* | bool |
| *gid* | str |

Table 183 – continued fro

| Property | Description |
|---|---|
| *in_layout* | bool |
| *label* | object |
| *mouseover* | bool |
| *navigate* | bool |
| *navigate_mode* | unknown |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *position* | [left, bottom, width, height] or *Bbox* |
| *prop_cycle* | *Cycler* |
| *rasterization_zorder* | float or None |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *subplotspec* | unknown |
| *title* | str |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *xbound* | (lower: float, upper: float) |
| *xlabel* | str |
| *xlim* | (left: float, right: float) |
| *xmargin* | float greater than -0.5 |
| *xscale* | unknown |
| *xticklabels* | unknown |
| *xticks* | unknown |
| *ybound* | (lower: float, upper: float) |
| *ylabel* | str |
| *ylim* | (bottom: float, top: float) |
| *ymargin* | float greater than -0.5 |
| *yscale* | unknown |
| *yticklabels* | unknown |
| *yticks* | unknown |
| *zorder* | float |

**Returns**

***Axes***

The new *Axes* object.

**class AxisDict**(*axes*)

Bases: dict

**__call__**(**v*, ***kwargs*)

Call self as a function.

**property axis**

Convenience method to get or set some axis properties.

Call signatures:

```
xmin, xmax, ymin, ymax = axis()
xmin, xmax, ymin, ymax = axis([xmin, xmax, ymin, ymax])
xmin, xmax, ymin, ymax = axis(option)
xmin, xmax, ymin, ymax = axis(**kwargs)
```

**Parameters**

**xmin, xmax, ymin, ymax**

[float, optional] The axis limits to be set. This can also be achieved using

```
ax.set(xlim=(xmin, xmax), ylim=(ymin, ymax))
```

**option**

[bool or str] If a bool, turns axis lines and labels on or off. If a string, possible values are:

| Value | Description |
|---|---|
| 'off' or False | Hide all axis decorations, i.e. axis labels, spines, tick marks, tick labels, and grid lines. This is the same as *set_axis_off()*. |
| 'on' or True | Do not hide all axis decorations, i.e. axis labels, spines, tick marks, tick labels, and grid lines. This is the same as *set_axis_on()*. |
| 'equal | Set equal scaling (i.e., make circles circular) by changing the axis limits. This is the same as `ax.set_aspect('equal', adjustable='datalim')`. Explicit data limits may not be respected in this case. |
| 'scale | Set equal scaling (i.e., make circles circular) by changing dimensions of the plot box. This is the same as `ax.set_aspect('equal', adjustable='box', anchor='C')`. Additionally, further autoscaling will be disabled. |
| 'tight' | Set limits just large enough to show all data, then disable further autoscaling. |
| 'auto' | Automatic scaling (fill plot box with data). |
| 'image' | 'scaled' with axis limits equal to data limits. |
| 'squar | Square plot; similar to 'scaled', but initially forcing `xmax-xmin == ymax-ymin`. |

**emit**

[bool, default: True] Whether observers are notified of the axis limit change. This option is passed on to *set_xlim* and *set_ylim*.

**Returns**

**xmin, xmax, ymin, ymax**

[float] The axis limits.

**See also:**

***matplotlib.axes.Axes.set_xlim***
***matplotlib.axes.Axes.set_ylim***

## Notes

For 3D Axes, this method additionally takes *zmin*, *zmax* as parameters and likewise returns them.

**clear**()

Clear the Axes.

**set**(*, *adjustable=<UNSET>*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *anchor=<UNSET>*, *animated=<UNSET>*, *aspect=<UNSET>*, *autoscale_on=<UNSET>*, *autoscalex_on=<UNSET>*, *autoscaley_on=<UNSET>*, *axes_locator=<UNSET>*, *axisbelow=<UNSET>*, *box_aspect=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *facecolor=<UNSET>*, *forward_navigation_events=<UNSET>*, *frame_on=<UNSET>*, *gid=<UNSET>*, *in_layout=<UNSET>*, *label=<UNSET>*, *mouseover=<UNSET>*, *navigate=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *position=<UNSET>*, *prop_cycle=<UNSET>*, *rasterization_zorder=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *subplotspec=<UNSET>*, *title=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *xbound=<UNSET>*, *xlabel=<UNSET>*, *xlim=<UNSET>*, *xmargin=<UNSET>*, *xscale=<UNSET>*, *xticklabels=<UNSET>*, *xticks=<UNSET>*, *ybound=<UNSET>*, *ylabel=<UNSET>*, *ylim=<UNSET>*, *ymargin=<UNSET>*, *yscale=<UNSET>*, *yticklabels=<UNSET>*, *yticks=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *adjustable* | {'box', 'datalim'} |
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value |
| *alpha* | scalar or None |
| *anchor* | (float, float) or {'C', 'SW', 'S', 'SE', 'E', 'NE', ...} |
| *animated* | bool |
| *aspect* | {'auto', 'equal'} or float |

Table 184 – continued from pr

| Property | Description |
|---|---|
| *autoscale_on* | bool |
| *autoscalex_on* | unknown |
| *autoscaley_on* | unknown |
| *axes_locator* | Callable[[Axes, Renderer], Bbox] |
| *axisbelow* | bool or 'line' |
| *box_aspect* | float or None |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *facecolor* or fc | *color* |
| *figure* | *Figure* |
| *forward_navigation_events* | bool or "auto" |
| *frame_on* | bool |
| *gid* | str |
| *in_layout* | bool |
| *label* | object |
| *mouseover* | bool |
| *navigate* | bool |
| *navigate_mode* | unknown |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *position* | [left, bottom, width, height] or *Bbox* |
| *prop_cycle* | *Cycler* |
| *rasterization_zorder* | float or None |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *subplotspec* | unknown |
| *title* | str |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *xbound* | (lower: float, upper: float) |
| *xlabel* | str |
| *xlim* | (left: float, right: float) |
| *xmargin* | float greater than -0.5 |
| *xscale* | unknown |
| *xticklabels* | unknown |
| *xticks* | unknown |
| *ybound* | (lower: float, upper: float) |
| *ylabel* | str |
| *ylim* | (bottom: float, top: float) |
| *ymargin* | float greater than -0.5 |
| *yscale* | unknown |

Table 184 – continued from pr

| Property | Description |
|---|---|
| *yticklabels* | unknown |
| *yticks* | unknown |
| *zorder* | float |

### Examples using `mpl_toolkits.axes_grid1.mpl_axes.Axes`

- *Demo Axes Grid*
- *Axes Grid2*
- *Parasite Simple2*
- *Simple ImageGrid*
- *Simple ImageGrid 2*
- *Tight layout guide*

## mpl_toolkits.axes_grid1.mpl_axes.SimpleAxisArtist

**class** mpl_toolkits.axes_grid1.mpl_axes.**SimpleAxisArtist**(*axis*, *axisnum*, *spine*)

Bases: *Artist*

**draw**(*renderer*)

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

**Parameters**

**renderer**

[*RendererBase* subclass.]

### Notes

This method is overridden in the Artist subclasses.

**property label**

**property major_ticklabels**

**property major_ticks**

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *gid=<UNSET>*, *in_layout=<UNSET>*, *label=<UNSET>*, *mouseover=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| agg_fil | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image |
| alpha | scalar or None |
| animated | bool |
| clip_bo | BboxBase or None |
| clip_on | bool |
| clip_pa | Patch or (Path, Transform) or None |
| figure | Figure |
| gid | str |
| in_layo | bool |
| label | unknown |
| mouseov | bool |
| path_ef | list of AbstractPathEffect |
| picker | None or bool or float or callable |
| rasterized | bool |
| sketch_ | (scale: float, length: float, randomness: float) |
| snap | bool or None |
| transform | Transform |
| url | str |
| visible | unknown |
| zorder | float |

**set_label**(*txt*)

Set a label that will be displayed in the legend.

**Parameters**

**s**

[object] *s* will be converted to a string by calling `str`.

**set_visible**(*b*)

Set the artist's visibility.

**Parameters**

**b**

[bool]

**toggle**(*all=None*, *ticks=None*, *ticklabels=None*, *label=None*)

### mpl_toolkits.axes_grid1.mpl_axes.SimpleChainedObjects

**class** mpl_toolkits.axes_grid1.mpl_axes.**SimpleChainedObjects**(*objects*)

Bases: `object`

**__call__**(**args*, ***kwargs*)

Call self as a function.

### mpl_toolkits.axes_grid1.parasite_axes

### Classes

| | |
|---|---|
| *HostAxes* | alias of AxesHostAxes |
| *HostAxesBase*(*args, **kwargs) | |
| *ParasiteAxes* | alias of AxesParasite |
| *ParasiteAxesBase*(parent_axes[, ...]) | |
| *SubplotHost* | alias of AxesHostAxes |

### mpl_toolkits.axes_grid1.parasite_axes.HostAxes

mpl_toolkits.axes_grid1.parasite_axes.**HostAxes**

alias of AxesHostAxes

### mpl_toolkits.axes_grid1.parasite_axes.HostAxesBase

**class** mpl_toolkits.axes_grid1.parasite_axes.**HostAxesBase**(**args*, ***kwargs*)

Bases: object

**clear**()

**draw**(*renderer*)

**get_aux_axes**(*tr=None*, *viewlim_mode='equal'*, *axes_class=None*, ***kwargs*)

Add a parasite axes to this host.

Despite this method's name, this should actually be thought of as an `add_parasite_axes` method.

Changed in version 3.7: Defaults to same base axes class as host axes.

**Parameters**

**tr**

[*Transform* or None, default: None] If a *Transform*, the following relation will hold: `parasite.transData = tr + host.transData`. If None, the parasite's and the host's `transData` are unrelated.

**viewlim_mode**

[{"equal", "transform", None}, default: "equal"] How the parasite's view limits are set: directly equal to the parent axes ("equal"), equal after application of *tr* ("transform"), or independently (None).

**axes_class**

[subclass type of *Axes*, optional] The *Axes* subclass that is instantiated. If None, the base class of the host axes is used.

****kwargs**

Other parameters are forwarded to the parasite axes constructor.

**get_tightbbox**(*renderer=None*, ***, *call_axes_locator=True*, *bbox_extra_artists=None*)

**pick**(*mouseevent*)

**twin**(*aux_trans=None*, *axes_class=None*)

Create a twin of Axes with no shared axis.

While self will have ticks on the left and bottom axis, the returned axes will have ticks on the top and right axis.

**twinx**(*axes_class=None*)

Create a twin of Axes with a shared x-axis but independent y-axis.

The y-axis of self will have ticks on the left and the returned axes will have ticks on the right.

**twiny**(*axes_class=None*)

Create a twin of Axes with a shared y-axis but independent x-axis.

The x-axis of self will have ticks on the bottom and the returned axes will have ticks on the top.

### Examples using `mpl_toolkits.axes_grid1.parasite_axes.HostAxesBase`

- *Parasite Simple2*
- *Curvilinear grid demo*
- *floating_axes features*
- *floating_axis demo*
- *Parasite Axes demo*

## mpl_toolkits.axes_grid1.parasite_axes.ParasiteAxes

mpl_toolkits.axes_grid1.parasite_axes.**ParasiteAxes**

alias of `AxesParasite`

## mpl_toolkits.axes_grid1.parasite_axes.ParasiteAxesBase

**class** mpl_toolkits.axes_grid1.parasite_axes.**ParasiteAxesBase**(*parent_axes*, *aux_transform=None*, ***, *viewlim_mode=None*, ***kwargs*)

Bases: `object`

**clear**()

**get_viewlim_mode**()

**pick**(*mouseevent*)

**set_viewlim_mode**(*mode*)

### mpl_toolkits.axes_grid1.parasite_axes.SubplotHost

mpl_toolkits.axes_grid1.parasite_axes.**SubplotHost**

alias of AxesHostAxes

### Functions

| | |
|---|---|
| *host_axes*(*args[, axes_class, figure]) | Create axes that can act as a hosts to parasitic axes. |
| *host_axes_class_factory*(axes_class) | |
| *host_subplot*(*args[, axes_class, figure]) | Create axes that can act as a hosts to parasitic axes. |
| *host_subplot_class_factory*(axes_class) | |
| *parasite_axes_class_factory*(axes_class) | |

### mpl_toolkits.axes_grid1.parasite_axes.host_axes

mpl_toolkits.axes_grid1.parasite_axes.**host_axes**(**args*, *axes_class=<class 'mpl_toolkits.axes_grid1.mpl_axes.Axes'>*, *figure=None*, ***kwargs*)

Create axes that can act as a hosts to parasitic axes.

**Parameters**

**figure**

[*Figure*] Figure to which the axes will be added. Defaults to the current figure *pyplot.gcf()*.

***args, **kwargs**

Will be passed on to the underlying *Axes* object creation.

### mpl_toolkits.axes_grid1.parasite_axes.host_axes_class_factory

mpl_toolkits.axes_grid1.parasite_axes.**host_axes_class_factory**(*axes_class*)

### mpl_toolkits.axes_grid1.parasite_axes.host_subplot

mpl_toolkits.axes_grid1.parasite_axes.**host_subplot**(**args*, *axes_class=<class 'mpl_toolkits.axes_grid1.mpl_axes.Axes'>*, *figure=None*, ***kwargs*)

Create axes that can act as a hosts to parasitic axes.

**Parameters**

**figure**

[*Figure*] Figure to which the axes will be added. Defaults to the current figure *pyplot.gcf()*.

***args, **kwargs**

Will be passed on to the underlying *Axes* object creation.

### mpl_toolkits.axes_grid1.parasite_axes.host_subplot_class_factory

mpl_toolkits.axes_grid1.parasite_axes.**host_subplot_class_factory**(*axes_class*)

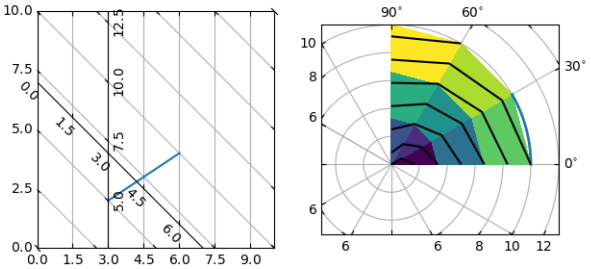
### mpl_toolkits.axes_grid1.parasite_axes.parasite_axes_class_factory

mpl_toolkits.axes_grid1.parasite_axes.**parasite_axes_class_factory**(*axes_class*)

## 7.2.68 mpl_toolkits.axisartist

The *axisartist* namespace provides a derived Axes implementation (mpl_toolkits.axisartist.Axes), designed to support curvilinear grids. The biggest difference is that the artists that are responsible for drawing axis lines, ticks, ticklabels, and axis labels are separated out from Matplotlib's Axis class.

You can find a tutorial describing usage of axisartist at the *axisartist* user guide.



**Note:** This module contains classes and function that were formerly part of the `mpl_toolkits.axes_grid` module that was removed in 3.6. Additional classes from that older module may also be found in *mpl_toolkits.axes_grid1*.

**The submodules of the axisartist API are:**

| | |
|---|---|
| *axisartist.angle_helper* | |
| *axisartist.axes_divider* | |
| *axisartist.axes_grid* | |
| *axisartist.axes_rgb* | |
| *axisartist.axis_artist* | The *axis_artist* module implements custom artists to draw axis elements (axis lines and labels, tick lines and labels, grid lines). |
| *axisartist.axisline_style* | Provides classes to style the axis lines. |
| *axisartist.axislines* | Axislines includes modified implementation of the Axes class. |
| *axisartist.floating_axes* | An experimental support for curvilinear grid. |
| *axisartist.grid_finder* | |
| *axisartist.grid_helper_curvelinear* | An experimental support for curvilinear grid. |
| *axisartist.parasite_axes* | |

## mpl_toolkits.axisartist.angle_helper

### Classes

| | |
|---|---|
| *ExtremeFinderCycle*(nx, ny[, lon_cycle, ...]) | This subclass handles the case where one or both coordinates should be taken modulo 360, or be restricted to not exceed a specific range. |
| *FormatterDMS*() | |
| *FormatterHMS*() | |
| *LocatorBase*(nbins[, include_last]) | |
| *LocatorD*(nbins[, include_last]) | |
| *LocatorDM*(nbins[, include_last]) | |
| *LocatorDMS*(nbins[, include_last]) | |
| *LocatorH*(nbins[, include_last]) | |
| *LocatorHM*(nbins[, include_last]) | |
| *LocatorHMS*(nbins[, include_last]) | |

### mpl_toolkits.axisartist.angle_helper.ExtremeFinderCycle

**class** mpl_toolkits.axisartist.angle_helper.**ExtremeFinderCycle**(*nx*, *ny*, *lon_cycle=360.0*, *lat_cycle=None*, *lon_minmax=None*, *lat_minmax=(-90, 90)*)

Bases: *ExtremeFinderSimple*

This subclass handles the case where one or both coordinates should be taken modulo 360, or be restricted to not exceed a specific range.

**Parameters**

**nx, ny**

[int] The number of samples in each direction.

**lon_cycle, lat_cycle**

[360 or None] If not None, values in the corresponding direction are taken modulo *lon_cycle* or *lat_cycle*; in theory this can be any number but the implementation

actually assumes that it is 360 (if not None); other values give nonsensical results.

This is done by "unwrapping" the transformed grid coordinates so that jumps are less than a half-cycle; then normalizing the span to no more than a full cycle.

For example, if values are in the union of the [0, 2] and [358, 360] intervals (typically, angles measured modulo 360), the values in the second interval are normalized to [-2, 0] instead so that the values now cover [-2, 2]. If values are in a range of [5, 1000], this gets normalized to [5, 365].

**lon_minmax, lat_minmax**

[(float, float) or None] If not None, the computed bounding box is clipped to the given range in the corresponding direction.

**__call__**(*transform_xy*, *x1*, *y1*, *x2*, *y2*)

Compute an approximation of the bounding box obtained by applying *transform_xy* to the box delimited by `(x1, y1, x2, y2)`.

The intended use is to have `(x1, y1, x2, y2)` in axes coordinates, and have *transform_xy* be the transform from axes coordinates to data coordinates; this method then returns the range of data coordinates that span the actual axes.

The computation is done by sampling `nx * ny` equispaced points in the `(x1, y1, x2, y2)` box and finding the resulting points with extremal coordinates; then adding some padding to take into account the finite sampling.

As each sampling step covers a relative range of *1/nx* or *1/ny*, the padding is computed by expanding the span covered by the extremal coordinates by these fractions.

### Examples using `mpl_toolkits.axisartist.angle_helper.ExtremeFinderCycle`

- *axis_direction demo*
- *Curvilinear grid demo*
- *floating_axis demo*
- *Simple Axis Pad*

## mpl_toolkits.axisartist.angle_helper.FormatterDMS

**class** mpl_toolkits.axisartist.angle_helper.**FormatterDMS**

Bases: `object`

**__call__**(*direction*, *factor*, *values*)

Call self as a function.

**deg_mark = '^{\\circ}'**

**fmt_d = '$%d^{\\circ}$'**

**fmt_d_m = '$%s%d^{\\circ}\\,%02d^{\\prime}$'**

**fmt_d_m_partial = '$%s%d^{\\circ}\\,%02d^{\\prime}\\,'**

**fmt_d_ms = '$%s%d^{\\circ}\\,%02d.%s^{\\prime}$'**

**fmt_ds = '$%d.%s^{\\circ}$'**

**fmt_s_partial = '%02d^{\\prime\\prime}$'**

**fmt_ss_partial = '%02d.%s^{\\prime\\prime}$'**

**min_mark = '^{\\prime}'**

**sec_mark = '^{\\prime\\prime}'**

### Examples using `mpl_toolkits.axisartist.angle_helper.FormatterDMS`

- *axis_direction demo*
- *Curvilinear grid demo*
- *floating_axes features*
- *floating_axis demo*
- *Simple Axis Pad*

## mpl_toolkits.axisartist.angle_helper.FormatterHMS

**class** mpl_toolkits.axisartist.angle_helper.**FormatterHMS**

Bases: *FormatterDMS*

**__call__**(*direction*, *factor*, *values*)

Call self as a function.

**deg_mark = '^\\mathrm{h}'**

**fmt_d = '$%d^\\mathrm{h}$'**

**fmt_d_m = '$%s%d^\\mathrm{h}\\,%02d^\\mathrm{m}$'**

**fmt_d_m_partial = '$%s%d^\\mathrm{h}\\,%02d^\\mathrm{m}\\,'**

**fmt_d_ms = '$%s%d^\\mathrm{h}\\,%02d.%s^\\mathrm{m}$'**

**fmt_ds = '$%d.%s^\\mathrm{h}$'**

**fmt_s_partial = '%02d^\\mathrm{s}$'**

```
fmt_ss_partial = '%02d.%s^\\mathrm{s}$'

min_mark = '^\\mathrm{m}'

sec_mark = '^\\mathrm{s}'
```

### Examples using `mpl_toolkits.axisartist.angle_helper.FormatterHMS`

- *floating_axes features*

### mpl_toolkits.axisartist.angle_helper.LocatorBase

**class** mpl_toolkits.axisartist.angle_helper.**LocatorBase**(*nbins*, *include_last=True*)

Bases: `object`

**set_params**(*nbins=None*)

### Examples using `mpl_toolkits.axisartist.angle_helper.LocatorBase`

- *axis_direction demo*
- *Curvilinear grid demo*
- *floating_axes features*
- *floating_axis demo*
- *Simple Axis Pad*

### mpl_toolkits.axisartist.angle_helper.LocatorD

**class** mpl_toolkits.axisartist.angle_helper.**LocatorD**(*nbins*, *include_last=True*)

Bases: *LocatorBase*

**__call__**(*v1*, *v2*)

Call self as a function.

### mpl_toolkits.axisartist.angle_helper.LocatorDM

**class** mpl_toolkits.axisartist.angle_helper.**LocatorDM**(*nbins*, *include_last=True*)

Bases: *LocatorBase*

**__call__**(*v1*, *v2*)

Call self as a function.

### mpl_toolkits.axisartist.angle_helper.LocatorDMS

**class** mpl_toolkits.axisartist.angle_helper.**LocatorDMS**(*nbins*, *include_last=True*)

Bases: *LocatorBase*

**__call__**(*v1*, *v2*)

Call self as a function.

### Examples using `mpl_toolkits.axisartist.angle_helper.LocatorDMS`

- *axis_direction demo*
- *Curvilinear grid demo*
- *floating_axis demo*
- *Simple Axis Pad*

### mpl_toolkits.axisartist.angle_helper.LocatorH

**class** mpl_toolkits.axisartist.angle_helper.**LocatorH**(*nbins*, *include_last=True*)

Bases: *LocatorBase*

**__call__**(*v1*, *v2*)

Call self as a function.

### mpl_toolkits.axisartist.angle_helper.LocatorHM

**class** mpl_toolkits.axisartist.angle_helper.**LocatorHM**(*nbins*, *include_last=True*)

Bases: *LocatorBase*

**__call__**(*v1*, *v2*)

Call self as a function.

### mpl_toolkits.axisartist.angle_helper.LocatorHMS

**class** mpl_toolkits.axisartist.angle_helper.**LocatorHMS**(*nbins*, *include_last=True*)

Bases: *LocatorBase*

**__call__**(*v1*, *v2*)

Call self as a function.

### Examples using `mpl_toolkits.axisartist.angle_helper.LocatorHMS`

- *floating_axes features*

### Functions

| | |
|---|---|
| *select_step*(v1, v2, nv[, hour, ...]) | |
| *select_step24*(v1, v2, nv[, include_last, ...]) | |
| *select_step360*(v1, v2, nv[, include_last, ...]) | |
| *select_step_degree*(dv) | |
| *select_step_hour*(dv) | |
| *select_step_sub*(dv) | |

### mpl_toolkits.axisartist.angle_helper.select_step

mpl_toolkits.axisartist.angle_helper.**select_step**(*v1*, *v2*, *nv*, *hour=False*, *include_last=True*, *threshold_factor=3600.0*)

### mpl_toolkits.axisartist.angle_helper.select_step24

mpl_toolkits.axisartist.angle_helper.**select_step24**(*v1*, *v2*, *nv*, *include_last=True*, *threshold_factor=3600*)

### mpl_toolkits.axisartist.angle_helper.select_step360

mpl_toolkits.axisartist.angle_helper.**select_step360**(*v1*, *v2*, *nv*, *include_last=True*, *threshold_factor=3600*)

### mpl_toolkits.axisartist.angle_helper.select_step_degree

mpl_toolkits.axisartist.angle_helper.**select_step_degree**(*dv*)

### mpl_toolkits.axisartist.angle_helper.select_step_hour

mpl_toolkits.axisartist.angle_helper.**select_step_hour**(*dv*)

### mpl_toolkits.axisartist.angle_helper.select_step_sub

mpl_toolkits.axisartist.angle_helper.**select_step_sub**(*dv*)

### mpl_toolkits.axisartist.axes_divider

### mpl_toolkits.axisartist.axes_grid

### Classes

| | |
|---|---|
| *AxesGrid* | alias of *ImageGrid* |
| *Grid*(fig, rect, nrows_ncols[, ngrids, ...]) | [*Deprecated*] |
| *ImageGrid*(fig, rect, nrows_ncols[, ngrids, ...]) | [*Deprecated*] |

### mpl_toolkits.axisartist.axes_grid.AxesGrid

mpl_toolkits.axisartist.axes_grid.**AxesGrid**
alias of *ImageGrid*

### mpl_toolkits.axisartist.axes_grid.Grid

**class** mpl_toolkits.axisartist.axes_grid.**Grid**(*fig*, *rect*, *nrows_ncols*, *ngrids=None*, *direction='row'*, *axes_pad=0.02*, *, *share_all=False*, *share_x=True*, *share_y=True*, *label_mode='L'*, *axes_class=None*, *aspect=False*)

Bases: *Grid*

[*Deprecated*]

#### Notes

Deprecated since version 3.8: Use axes_grid1.axes_grid.Grid(..., axes_class=axislines.Axes instead.

**Parameters**

**fig**

[*Figure*] The parent figure.

**rect**

[(float, float, float, float), (int, int, int), int, or *SubplotSpec*] The axes position, as a `(left, bottom, width, height)` tuple, as a three-digit subplot position code (e.g., `(1, 2, 1)` or `121`), or as a *SubplotSpec*.

**nrows_ncols**

[(int, int)] Number of rows and columns in the grid.

**ngrids**

[int or None, default: None] If not None, only the first *ngrids* axes in the grid are created.

**direction**

[{"row", "column"}, default: "row"] Whether axes are created in row-major ("row by row") or column-major order ("column by column"). This also affects the order in which axes are accessed using indexing (`grid[index]`).

**axes_pad**

[float or (float, float), default: 0.02] Padding or (horizontal padding, vertical padding) between axes, in inches.

**share_all**

[bool, default: False] Whether all axes share their x- and y-axis. Overrides *share_x* and *share_y*.

**share_x**

[bool, default: True] Whether all axes of a column share their x-axis.

**share_y**

[bool, default: True] Whether all axes of a row share their y-axis.

**label_mode**

[{"L", "1", "all", "keep"}, default: "L"] Determines which axes will get tick labels:

- "L": All axes on the left column get vertical tick labels; all axes on the bottom row get horizontal tick labels.
- "1": Only the bottom left axes is labelled.
- "all": All axes are labelled.
- "keep": Do not do anything.

**axes_class**

[subclass of *matplotlib.axes.Axes*, default: *mpl_axes.Axes*] The type of Axes to create.

**aspect**

[bool, default: False] Whether the axes aspect ratio follows the aspect ratio of the data limits.

### mpl_toolkits.axisartist.axes_grid.ImageGrid

**class** mpl_toolkits.axisartist.axes_grid.**ImageGrid**(*fig*, *rect*, *nrows_ncols*, *ngrids=None*, *direction='row'*, *axes_pad=0.02*, *, *share_all=False*, *aspect=True*, *label_mode='L'*, *cbar_mode=None*, *cbar_location='right'*, *cbar_pad=None*, *cbar_size='5%'*, *cbar_set_cax=True*, *axes_class=None*)

Bases: *ImageGrid*

[*Deprecated*]

### Notes

Deprecated since version 3.8: Use axes_grid1.axes_grid.ImageGrid(..., axes_class=axislines.Axes instead.

**Parameters**

**fig**

[*Figure*] The parent figure.

**rect**

[(float, float, float, float) or int] The axes position, as a `(left, bottom, width, height)` tuple or as a three-digit subplot position code (e.g., "121").

**nrows_ncols**

[(int, int)] Number of rows and columns in the grid.

**ngrids**

[int or None, default: None] If not None, only the first *ngrids* axes in the grid are created.

**direction**

[{"row", "column"}, default: "row"] Whether axes are created in row-major ("row by row") or column-major order ("column by column"). This also affects the order in which axes are accessed using indexing (`grid[index]`).

**axes_pad**

[float or (float, float), default: 0.02in] Padding or (horizontal padding, vertical padding) between axes, in inches.

**share_all**

[bool, default: False] Whether all axes share their x- and y-axis. Note that in any case, all axes in a column share their x-axis and all axes in a row share their y-axis.

**aspect**

[bool, default: True] Whether the axes aspect ratio follows the aspect ratio of the data limits.

**label_mode**

[{"L", "1", "all"}, default: "L"] Determines which axes will get tick labels:

- "L": All axes on the left column get vertical tick labels; all axes on the bottom row get horizontal tick labels.
- "1": Only the bottom left axes is labelled.
- "all": all axes are labelled.

**cbar_mode**

[{"each", "single", "edge", None}, default: None] Whether to create a colorbar for "each" axes, a "single" colorbar for the entire grid, colorbars only for axes on the "edge" determined by *cbar_location*, or no colorbars. The colorbars are stored in the `cbar_axes` attribute.

**cbar_location**

[{"left", "right", "bottom", "top"}, default: "right"]

**cbar_pad**

[float, default: None] Padding between the image axes and the colorbar axes.

**cbar_size**

[size specification (see `Size.from_any`), default: "5%"] Colorbar size.

**cbar_set_cax**

[bool, default: True] If True, each axes in the grid has a *cax* attribute that is bound to associated *cbar_axes*.

**axes_class**

[subclass of *matplotlib.axes.Axes*, default: None]

## mpl_toolkits.axisartist.axes_rgb

## Classes

| | |
|---|---|
| *RGBAxes*(*args[, pad]) | [*Deprecated*] Subclass of *RGBAxes* with _defaultAxesClass = *axislines.Axes*. |

## mpl_toolkits.axisartist.axes_rgb.RGBAxes

**class** mpl_toolkits.axisartist.axes_rgb.**RGBAxes**(**args*, *pad=0*, ***kwargs*)

Bases: *RGBAxes*

[*Deprecated*] Subclass of *RGBAxes* with _defaultAxesClass = *axislines.Axes*.

### Notes

Deprecated since version 3.8: Use axes_grid1.axes_rgb.RGBAxes(..., axes_class=axislines.Axes instead.

**Parameters**

**pad**

[float, default: 0] Fraction of the Axes height to put as padding.

**axes_class**

[*Axes*] Axes class to use. If not provided, `_defaultAxesClass` is used.

***args**

Forwarded to *axes_class* init for the RGB Axes

****kwargs**

Forwarded to *axes_class* init for the RGB, R, G, and B Axes

## mpl_toolkits.axisartist.axis_artist

The *axis_artist* module implements custom artists to draw axis elements (axis lines and labels, tick lines and labels, grid lines).

Axis lines and labels and tick lines and labels are managed by the *AxisArtist* class; grid lines are managed by the *GridlinesCollection* class.

There is one *AxisArtist* per Axis; it can be accessed through the `axis` dictionary of the parent Axes (which should be a `mpl_toolkits.axislines.Axes`), e.g. `ax.axis["bottom"]`.

Children of the AxisArtist are accessed as attributes: `.line` and `.label` for the axis line and label, `.major_ticks`, `.major_ticklabels`, `.minor_ticks`, `.minor_ticklabels` for the tick lines and labels (e.g. `ax.axis["bottom"].line`).

Children properties (colors, fonts, line widths, etc.) can be set using setters, e.g.

```
# Make the major ticks of the bottom axis red.
ax.axis["bottom"].major_ticks.set_color("red")
```

However, things like the locations of ticks, and their ticklabels need to be changed from the side of the grid_helper.

### axis_direction

*AxisArtist*, *AxisLabel*, *TickLabels* have an *axis_direction* attribute, which adjusts the location, angle, etc. The *axis_direction* must be one of "left", "right", "bottom", "top", and follows the Matplotlib convention for rectangular axis.

For example, for the *bottom* axis (the left and right is relative to the direction of the increasing coordinate),

- ticklabels and axislabel are on the right
- ticklabels and axislabel have text angle of 0
- ticklabels are baseline, center-aligned
- axislabel is top, center-aligned

The text angles are actually relative to (90 + angle of the direction to the ticklabel), which gives 0 for bottom axis.

| Property | left | bottom | right | top |
|---|---|---|---|---|
| ticklabel location | left | right | right | left |
| axislabel location | left | right | right | left |
| ticklabel angle | 90 | 0 | -90 | 180 |
| axislabel angle | 180 | 0 | 0 | 180 |
| ticklabel va | center | baseline | center | baseline |
| axislabel va | center | top | center | bottom |
| ticklabel ha | right | center | right | center |
| axislabel ha | right | center | right | center |

Ticks are by default direct opposite side of the ticklabels. To make ticks to the same side of the ticklabels,

```
ax.axis["bottom"].major_ticks.set_tick_out(True)
```

The following attributes can be customized (use the `set_xxx` methods):

- *Ticks*: ticksize, tick_out
- *TickLabels*: pad
- *AxisLabel*: pad

### Classes

| | |
|---|---|
| *AttributeCopier*() | |
| *AxisArtist*(axes, helper[, offset, ...]) | An artist which draws axis (a line along which the n-th axes coord is constant) line, ticks, tick labels, and axis label. |
| *AxisLabel*(*args[, axis_direction, axis]) | Axis label. |
| *GridlinesCollection*(*args[, which, axis]) | Collection of grid lines. |
| *LabelBase*(*args, **kwargs) | A base class for *AxisLabel* and *TickLabels*. |
| *TickLabels*(*[, axis_direction]) | Tick labels. |
| *Ticks*(ticksize[, tick_out, axis]) | Ticks are derived from *Line2D*, and note that ticks themselves are markers. |

### mpl_toolkits.axisartist.axis_artist.AttributeCopier

**class** mpl_toolkits.axisartist.axis_artist.**AttributeCopier**

Bases: object

**get_attribute_from_ref_artist**(*attr_name*)

**get_ref_artist**()

Return the underlying artist that actually defines some properties (e.g., color) of this artist.

### mpl_toolkits.axisartist.axis_artist.AxisArtist

**class** mpl_toolkits.axisartist.axis_artist.**AxisArtist**(*axes*, *helper*, *offset=None*, *axis_direction='bottom'*, ***kwargs*)

Bases: *Artist*

An artist which draws axis (a line along which the n-th axes coord is constant) line, ticks, tick labels, and axis label.

**Parameters**

**axes**

[*mpl_toolkits.axisartist.axislines.Axes*]

**helper**

[*AxisArtistHelper*]

**property LABELPAD**

**draw**(*renderer*)

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

**Parameters**

**renderer**

[*RendererBase* subclass.]

### Notes

This method is overridden in the Artist subclasses.

**get_axisline_style**()

Return the current axisline style.

**get_helper**()

Return axis artist helper instance.

**get_tightbbox**(*renderer=None*)

Like *Artist.get_window_extent*, but includes any clipping.

**Parameters**

**renderer**

[*RendererBase* subclass, optional] renderer that will be used to draw the figures (i.e. `fig.canvas.get_renderer()`)

**Returns**

***Bbox*** **or None**

The enclosing bounding box (in figure pixel coordinates). Returns None if clipping results in no intersection.

**get_transform**()

Return the *Transform* instance used by this artist.

**invert_ticklabel_direction**()

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *axis_direction=<UNSET>*, *axislabel_direction=<UNSET>*, *axisline_style=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *gid=<UNSET>*, *in_layout=<UNSET>*, *label=<UNSET>*, *mouseover=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *ticklabel_direction=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *agg_filte* | a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image |
| *alpha* | scalar or None |
| *animated* | bool |
| *axis_direc* | {"left", "bottom", "right", "top"} |
| *axisla-bel_direc* | {"+", "-"} |
| *axis-line_style* | str or None |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *figure* | *Figure* |
| *gid* | str |
| *in_layout* | bool |
| *label* | unknown |
| *mouseover* | bool |
| *path_effe* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *raster-ized* | bool |
| *sketch_pa* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *tickla-bel_direc* | {"+", "-"} |
| *trans-form* | *Transform* |
| *url* | str |
| *visible* | bool |
| *zorder* | float |

**set_axis_direction**(*axis_direction*)

Adjust the direction, text angle, and text alignment of tick labels and axis labels following the Matplotlib convention for the rectangle axes.

The *axis_direction* must be one of [left, right, bottom, top].

| Property | left | bottom | right | top |
|---|---|---|---|---|
| ticklabel direction | "-" | "+" | "+" | "-" |
| axislabel direction | "-" | "+" | "+" | "-" |
| ticklabel angle | 90 | 0 | -90 | 180 |
| ticklabel va | center | baseline | center | baseline |
| ticklabel ha | right | center | right | center |
| axislabel angle | 180 | 0 | 0 | 180 |
| axislabel va | center | top | center | bottom |
| axislabel ha | right | center | right | center |

Note that the direction "+" and "-" are relative to the direction of the increasing coordinate. Also, the text angles are actually relative to (90 + angle of the direction to the ticklabel), which gives 0 for bottom axis.

**Parameters**

**axis_direction**

[{"left", "bottom", "right", "top"}]

**set_axislabel_direction**(*label_direction*)

Adjust the direction of the axis label.

Note that the *label_direction*s '+' and '-' are relative to the direction of the increasing coordinate.

**Parameters**

**label_direction**

[{"+", "-"}]

**set_axisline_style**(*axisline_style=None*, ***kwargs*)

Set the axisline style.

The new style is completely defined by the passed attributes. Existing style attributes are forgotten.

**Parameters**

**axisline_style**

[str or None] The line style, e.g. '->', optionally followed by a comma-separated list of attributes. Alternatively, the attributes can be provided as keywords.

If *None* this returns a string containing the available styles.

### Examples

The following two commands are equal:

```
>>> set_axisline_style("->,size=1.5")
>>> set_axisline_style("->", size=1.5)
```

**set_label**(*s*)

Set a label that will be displayed in the legend.

**Parameters**

**s**

[object] *s* will be converted to a string by calling `str`.

**set_ticklabel_direction**(*tick_direction*)

Adjust the direction of the tick labels.

Note that the *tick_direction*s '+' and '-' are relative to the direction of the increasing coordinate.

**Parameters**

**tick_direction**

[{"+", "-"}]

**toggle**(*all=None*, *ticks=None*, *ticklabels=None*, *label=None*)

Toggle visibility of ticks, ticklabels, and (axis) label. To turn all off,

```
axis.toggle(all=False)
```

To turn all off but ticks on

```
axis.toggle(all=False, ticks=True)
```

To turn all on but (axis) label off

```
axis.toggle(all=True, label=False)
```

**zorder = 2.5**

### mpl_toolkits.axisartist.axis_artist.AxisLabel

**class** mpl_toolkits.axisartist.axis_artist.**AxisLabel**(**args*, *axis_direction='bottom'*, *axis=None*, ***kwargs*)

Bases: *AttributeCopier*, *LabelBase*

Axis label. Derived from *Text*. The position of the text is updated in the fly, so changing text position has no effect. Otherwise, the properties can be changed as a normal *Text*.

To change the pad between tick labels and axis label, use *set_pad*.

**draw**(*renderer*)

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

**Parameters**

**renderer**

[*RendererBase* subclass.]

#### Notes

This method is overridden in the Artist subclasses.

**get_color**()

Return the color of the text.

**get_pad**()

Return the internal pad in points.

See *set_pad* for more details.

**get_ref_artist**()

Return the underlying artist that actually defines some properties (e.g., color) of this artist.

**get_text**()

Return the text string.

**get_window_extent**(*renderer=None*)

Return the *Bbox* bounding the text, in display units.

In addition to being used internally, this is useful for specifying clickable regions in a png file on a web page.

**Parameters**

**renderer**

[Renderer, optional] A renderer is needed to compute the bounding box. If the artist has already been drawn, the renderer is cached; thus, it is only necessary to pass this argument when calling *get_window_extent* before the first draw. In practice, it is usually easier to trigger a draw first, e.g. by calling *draw_without_rendering* or `plt.show()`.

**dpi**

[float, optional] The dpi value for computing the bbox, defaults to `self.figure.dpi` (*not* the renderer dpi); should be set e.g. if to match regions with a figure saved with a custom dpi value.

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *antialiased=<UNSET>*, *axis_direction=<UNSET>*, *backgroundcolor=<UNSET>*, *bbox=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *color=<UNSET>*, *default_alignment=<UNSET>*, *default_angle=<UNSET>*, *fontfamily=<UNSET>*, *fontproperties=<UNSET>*, *fontsize=<UNSET>*, *fontstretch=<UNSET>*, *fontstyle=<UNSET>*, *fontvariant=<UNSET>*, *fontweight=<UNSET>*, *gid=<UNSET>*, *horizontalalignment=<UNSET>*, *in_layout=<UNSET>*, *label=<UNSET>*, *linespacing=<UNSET>*, *math_fontfamily=<UNSET>*, *mouseover=<UNSET>*, *multialignment=<UNSET>*, *pad=<UNSET>*, *parse_math=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *position=<UNSET>*, *rasterized=<UNSET>*, *rotation=<UNSET>*, *rotation_mode=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *text=<UNSET>*, *transform=<UNSET>*, *transform_rotates_text=<UNSET>*, *url=<UNSET>*, *usetex=<UNSET>*, *verticalalignment=<UNSET>*, *visible=<UNSET>*, *wrap=<UNSET>*, *x=<UNSET>*, *y=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* | bool |
| *axis_direction* | {"left", "bottom", "right", "top"} |
| *backgroundcolor* | *color* |
| *bbox* | dict with properties for *patches.FancyBboxPatch* |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* or c | *color* |
| *default_alignment* | {"left", "bottom", "right", "top"} |
| *default_angle* | {"left", "bottom", "right", "top"} |
| *figure* | *Figure* |
| *fontfamily* or family or fontname | {FONTNAME, 'serif', 'sans-serif', 'cursive', 'fantasy', 'mo |

Table 185 – co

| Property | Description |
|---|---|
| *fontproperties* or font or font_properties | *font_manager.FontProperties* or str or pat |
| *fontsize* or size | float or {'xx-small', 'x-small', 'small', 'medium', 'large', 'x |
| *fontstretch* or stretch | {a numeric value in range 0-1000, 'ultra-condensed', 'ext |
| *fontstyle* or style | {'normal', 'italic', 'oblique'} |
| *fontvariant* or variant | {'normal', 'small-caps'} |
| *fontweight* or weight | {a numeric value in range 0-1000, 'ultralight', 'light', 'nor |
| *gid* | str |
| *horizontalalignment* or ha | {'left', 'center', 'right'} |
| *in_layout* | bool |
| *label* | object |
| *linespacing* | float (multiple of font size) |
| *math_fontfamily* | str |
| *mouseover* | bool |
| *multialignment* or ma | {'left', 'right', 'center'} |
| *pad* | float |
| *parse_math* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *position* | (float, float) |
| *rasterized* | bool |
| *rotation* | float or {'vertical', 'horizontal'} |
| *rotation_mode* | {None, 'default', 'anchor'} |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *text* | object |
| *transform* | *Transform* |
| *transform_rotates_text* | bool |
| *url* | str |
| *usetex* | bool, default: *rcParams["text.usetex"]* (defaul |
| *verticalalignment* or va | {'baseline', 'bottom', 'center', 'center_baseline', 'top'} |
| *visible* | bool |
| *wrap* | bool |
| *x* | float |
| *y* | float |
| *zorder* | float |

**set_axis_direction**(*d*)

Adjust the text angle and text alignment of axis label according to the matplotlib convention.

| Property | left | bottom | right | top |
|---|---|---|---|---|
| axislabel angle | 180 | 0 | 0 | 180 |
| axislabel va | center | top | center | bottom |
| axislabel ha | right | center | right | center |

Note that the text angles are actually relative to (90 + angle of the direction to the ticklabel), which gives 0 for bottom axis.

**Parameters**

**d**

[{"left", "bottom", "right", "top"}]

**set_default_alignment**(*d*)

Set the default alignment. See *set_axis_direction* for details.

**Parameters**

**d**

[{"left", "bottom", "right", "top"}]

**set_default_angle**(*d*)

Set the default angle. See *set_axis_direction* for details.

**Parameters**

**d**

[{"left", "bottom", "right", "top"}]

**set_pad**(*pad*)

Set the internal pad in points.

The actual pad will be the sum of the internal pad and the external pad (the latter is set automatically by the *AxisArtist*).

**Parameters**

**pad**

[float] The internal pad in points.

### mpl_toolkits.axisartist.axis_artist.GridlinesCollection

**class** mpl_toolkits.axisartist.axis_artist.**GridlinesCollection**(**args*, *which='major'*, *axis='both'*, ***kwargs*)

Bases: *LineCollection*

Collection of grid lines.

**Parameters**

**which**

[{"major", "minor"}] Which grid to consider.

**axis**

[{"both", "x", "y"}] Which axis to consider.

***args, **kwargs**

Passed to *LineCollection*.

**draw**(*renderer*)

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

**Parameters**

**renderer**

[*RendererBase* subclass.]

#### Notes

This method is overridden in the Artist subclasses.

**set**(***, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *antialiased=<UNSET>*, *array=<UNSET>*, *axis=<UNSET>*, *capstyle=<UNSET>*, *clim=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *cmap=<UNSET>*, *color=<UNSET>*, *colors=<UNSET>*, *edgecolor=<UNSET>*, *facecolor=<UNSET>*, *gapcolor=<UNSET>*, *gid=<UNSET>*, *grid_helper=<UNSET>*, *hatch=<UNSET>*, *in_layout=<UNSET>*, *joinstyle=<UNSET>*, *label=<UNSET>*, *linestyle=<UNSET>*, *linewidth=<UNSET>*, *mouseover=<UNSET>*, *norm=<UNSET>*, *offset_transform=<UNSET>*, *offsets=<UNSET>*, *path_effects=<UNSET>*, *paths=<UNSET>*, *picker=<UNSET>*, *pickradius=<UNSET>*, *rasterized=<UNSET>*, *segments=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *urls=<UNSET>*, *verts=<UNSET>*, *visible=<UNSET>*, *which=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi v |
| *alpha* | array-like or scalar or None |
| *animated* | bool |
| *antialiased* or aa or antialiaseds | bool or list of bools |
| *array* | array-like or None |
| *axis* | {"both", "x", "y"} |
| *capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *clim* | (vmin: float, vmax: float) |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *cmap* | *Colormap* or str or None |
| *color* | *color* or list of *color* |
| *colors* | *color* or list of *color* |
| *edgecolor* or ec or edgecolors | *color* or list of *color* or 'face' |
| *facecolor* or facecolors or fc | *color* or list of *color* |
| *figure* | *Figure* |
| *gapcolor* | *color* or list of *color* or None |
| *gid* | str |
| *grid_helper* | *GridHelperBase* subclass |
| *hatch* | {'/', '\', '\|', '-', '+', 'x', 'o', 'O', '.', '*'} |
| *in_layout* | bool |
| *joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *label* | object |
| *linestyle* or dashes or linestyles or ls | str or tuple or list thereof |
| *linewidth* or linewidths or lw | float or list of floats |
| *mouseover* | bool |
| *norm* | *Normalize* or str or None |
| *offset_transform* or transOffset | *Transform* |
| *offsets* | (N, 2) or (2,) array-like |
| *path_effects* | list of *AbstractPathEffect* |
| *paths* | unknown |
| *picker* | None or bool or float or callable |
| *pickradius* | float |
| *rasterized* | bool |
| *segments* | unknown |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *transform* | *Transform* |
| *url* | str |

Table 186 – continued from

| Property | Description |
|---|---|
| *urls* | list of str or None |
| *verts* | unknown |
| *visible* | bool |
| *which* | {"major", "minor"} |
| *zorder* | float |

**set_axis**(*axis*)

Select axis.

**Parameters**

**axis**

[{"both", "x", "y"}]

**set_grid_helper**(*grid_helper*)

Set grid helper.

**Parameters**

**grid_helper**

[*GridHelperBase* subclass]

**set_which**(*which*)

Select major or minor grid lines.

**Parameters**

**which**

[{"major", "minor"}]

## mpl_toolkits.axisartist.axis_artist.LabelBase

**class** mpl_toolkits.axisartist.axis_artist.**LabelBase**(**args*, ***kwargs*)

Bases: *Text*

A base class for *AxisLabel* and *TickLabels*. The position and angle of the text are calculated by the offset_ref_angle, text_ref_angle, and offset_radius attributes.

Create a *Text* instance at *x*, *y* with string *text*.

The text is aligned relative to the anchor point (*x*, *y*) according to `horizontalalignment` (default: 'left') and `verticalalignment` (default: 'baseline'). See also *Text alignment*.

While Text accepts the 'label' keyword argument, by default it is not added to the handles of a legend.

Valid keyword arguments are:

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* | bool |
| *backgroundcolor* | *color* |
| *bbox* | dict with properties for *patches.FancyBboxPatch* |
| *clip_box* | unknown |
| *clip_on* | unknown |
| *clip_path* | unknown |
| *color* or c | *color* |
| *figure* | *Figure* |
| *fontfamily* or family or fontname | {FONTNAME, 'serif', 'sans-serif', 'cursive', 'fantasy', 'monosp |
| *fontproperties* or font or font_properties | *font_manager.FontProperties* or *str* or *pathli* |
| *fontsize* or size | float or {'xx-small', 'x-small', 'small', 'medium', 'large', 'x-larg |
| *fontstretch* or stretch | {a numeric value in range 0-1000, 'ultra-condensed', 'extra-co |
| *fontstyle* or style | {'normal', 'italic', 'oblique'} |
| *fontvariant* or variant | {'normal', 'small-caps'} |
| *fontweight* or weight | {a numeric value in range 0-1000, 'ultralight', 'light', 'normal' |
| *gid* | str |
| *horizontalalignment* or ha | {'left', 'center', 'right'} |
| *in_layout* | bool |
| *label* | object |
| *linespacing* | float (multiple of font size) |
| *math_fontfamily* | str |
| *mouseover* | bool |
| *multialignment* or ma | {'left', 'right', 'center'} |
| *parse_math* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *position* | (float, float) |
| *rasterized* | bool |
| *rotation* | float or {'vertical', 'horizontal'} |
| *rotation_mode* | {None, 'default', 'anchor'} |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *text* | object |
| *transform* | *Transform* |
| *transform_rotates_text* | bool |
| *url* | str |
| *usetex* | bool, default: *rcParams["text.usetex"]* (default: Fa |
| *verticalalignment* or va | {'baseline', 'bottom', 'center', 'center_baseline', 'top'} |

Table 187 – continu

| Property | Description |
|---|---|
| *visible* | bool |
| *wrap* | bool |
| *x* | float |
| *y* | float |
| *zorder* | float |

**draw**(*renderer*)

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

**Parameters**

**renderer**

[*RendererBase* subclass.]

### Notes

This method is overridden in the Artist subclasses.

**get_window_extent**(*renderer=None*)

Return the *Bbox* bounding the text, in display units.

In addition to being used internally, this is useful for specifying clickable regions in a png file on a web page.

**Parameters**

**renderer**

[Renderer, optional] A renderer is needed to compute the bounding box. If the artist has already been drawn, the renderer is cached; thus, it is only necessary to pass this argument when calling *get_window_extent* before the first draw. In practice, it is usually easier to trigger a draw first, e.g. by calling *draw_without_rendering* or `plt.show()`.

**dpi**

[float, optional] The dpi value for computing the bbox, defaults to `self.figure.dpi` (*not* the renderer dpi); should be set e.g. if to match regions with a figure saved with a custom dpi value.

**set** (*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *antialiased=<UNSET>*, *backgroundcolor=<UNSET>*, *bbox=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *color=<UNSET>*, *fontfamily=<UNSET>*, *fontproperties=<UNSET>*, *fontsize=<UNSET>*, *fontstretch=<UNSET>*, *fontstyle=<UNSET>*, *fontvariant=<UNSET>*, *fontweight=<UNSET>*, *gid=<UNSET>*, *horizontalalignment=<UNSET>*, *in_layout=<UNSET>*, *label=<UNSET>*, *linespacing=<UNSET>*, *math_fontfamily=<UNSET>*, *mouseover=<UNSET>*, *multialignment=<UNSET>*, *parse_math=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *position=<UNSET>*, *rasterized=<UNSET>*, *rotation=<UNSET>*, *rotation_mode=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *text=<UNSET>*, *transform=<UNSET>*, *transform_rotates_text=<UNSET>*, *url=<UNSET>*, *usetex=<UNSET>*, *verticalalignment=<UNSET>*, *visible=<UNSET>*, *wrap=<UNSET>*, *x=<UNSET>*, *y=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* | bool |
| *backgroundcolor* | *color* |
| *bbox* | dict with properties for *patches.FancyBboxPatch* |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* or c | *color* |
| *figure* | *Figure* |
| *fontfamily* or family or fontname | {FONTNAME, 'serif', 'sans-serif', 'cursive', 'fantasy', 'mo |
| *fontproperties* or font or font_properties | *font_manager.FontProperties* or *str* or *pat* |
| *fontsize* or size | float or {'xx-small', 'x-small', 'small', 'medium', 'large', 'x |
| *fontstretch* or stretch | {a numeric value in range 0-1000, 'ultra-condensed', 'ext |
| *fontstyle* or style | {'normal', 'italic', 'oblique'} |
| *fontvariant* or variant | {'normal', 'small-caps'} |
| *fontweight* or weight | {a numeric value in range 0-1000, 'ultralight', 'light', 'nor |
| *gid* | str |
| *horizontalalignment* or ha | {'left', 'center', 'right'} |
| *in_layout* | bool |
| *label* | object |
| *linespacing* | float (multiple of font size) |
| *math_fontfamily* | str |
| *mouseover* | bool |
| *multialignment* or ma | {'left', 'right', 'center'} |
| *parse_math* | bool |
| *path_effects* | list of *AbstractPathEffect* |

Table 188 – co

| Property | Description |
|---|---|
| *picker* | None or bool or float or callable |
| *position* | (float, float) |
| *rasterized* | bool |
| *rotation* | float or {'vertical', 'horizontal'} |
| *rotation_mode* | {None, 'default', 'anchor'} |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *text* | object |
| *transform* | *Transform* |
| *transform_rotates_text* | bool |
| *url* | str |
| *usetex* | bool, default: *rcParams["text.usetex"]* (defaul |
| *verticalalignment* or va | {'baseline', 'bottom', 'center', 'center_baseline', 'top'} |
| *visible* | bool |
| *wrap* | bool |
| *x* | float |
| *y* | float |
| *zorder* | float |

## mpl_toolkits.axisartist.axis_artist.TickLabels

**class** mpl_toolkits.axisartist.axis_artist.**TickLabels**(*, *axis_direction='bottom'*, ***kwargs*)

Bases: *AxisLabel*

Tick labels. While derived from *Text*, this single artist draws all ticklabels. As in *AxisLabel*, the position of the text is updated in the fly, so changing text position has no effect. Otherwise, the properties can be changed as a normal *Text*. Unlike the ticklabels of the mainline Matplotlib, properties of a single ticklabel alone cannot be modified.

To change the pad between ticks and ticklabels, use *set_pad*.

**draw**(*renderer*)

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

**Parameters**

**renderer**

[*RendererBase* subclass.]

### Notes

This method is overridden in the Artist subclasses.

**get_ref_artist**()

Return the underlying artist that actually defines some properties (e.g., color) of this artist.

**get_texts_widths_heights_descents**(*renderer*)

Return a list of `(width, height, descent)` tuples for ticklabels.

Empty labels are left out.

**get_window_extents**(*renderer=None*)

**invert_axis_direction**()

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *antialiased=<UNSET>*, *axis_direction=<UNSET>*, *backgroundcolor=<UNSET>*, *bbox=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *color=<UNSET>*, *default_alignment=<UNSET>*, *default_angle=<UNSET>*, *fontfamily=<UNSET>*, *fontproperties=<UNSET>*, *fontsize=<UNSET>*, *fontstretch=<UNSET>*, *fontstyle=<UNSET>*, *fontvariant=<UNSET>*, *fontweight=<UNSET>*, *gid=<UNSET>*, *horizontalalignment=<UNSET>*, *in_layout=<UNSET>*, *label=<UNSET>*, *linespacing=<UNSET>*, *locs_angles_labels=<UNSET>*, *math_fontfamily=<UNSET>*, *mouseover=<UNSET>*, *multialignment=<UNSET>*, *pad=<UNSET>*, *parse_math=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *position=<UNSET>*, *rasterized=<UNSET>*, *rotation=<UNSET>*, *rotation_mode=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *text=<UNSET>*, *transform=<UNSET>*, *transform_rotates_text=<UNSET>*, *url=<UNSET>*, *usetex=<UNSET>*, *verticalalignment=<UNSET>*, *visible=<UNSET>*, *wrap=<UNSET>*, *x=<UNSET>*, *y=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* | bool |
| *axis_direction* | {"left", "bottom", "right", "top"} |
| *backgroundcolor* | *color* |
| *bbox* | dict with properties for *patches.FancyBboxPatch* |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* or c | *color* |
| *default_alignment* | {"left", "bottom", "right", "top"} |

Table 189 – co

| Property | Description |
|---|---|
| *default_angle* | {"left", "bottom", "right", "top"} |
| *figure* | *Figure* |
| *fontfamily* or family or fontname | {FONTNAME, 'serif', 'sans-serif', 'cursive', 'fantasy', 'mo |
| *fontproperties* or font or font_properties | *font_manager.FontProperties* or `str` or `pat` |
| *fontsize* or size | float or {'xx-small', 'x-small', 'small', 'medium', 'large', 'x- |
| *fontstretch* or stretch | {a numeric value in range 0-1000, 'ultra-condensed', 'ext |
| *fontstyle* or style | {'normal', 'italic', 'oblique'} |
| *fontvariant* or variant | {'normal', 'small-caps'} |
| *fontweight* or weight | {a numeric value in range 0-1000, 'ultralight', 'light', 'nor |
| *gid* | str |
| *horizontalalignment* or ha | {'left', 'center', 'right'} |
| *in_layout* | bool |
| *label* | object |
| *linespacing* | float (multiple of font size) |
| *locs_angles_labels* | unknown |
| *math_fontfamily* | str |
| *mouseover* | bool |
| *multialignment* or ma | {'left', 'right', 'center'} |
| *pad* | float |
| *parse_math* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *position* | (float, float) |
| *rasterized* | bool |
| *rotation* | float or {'vertical', 'horizontal'} |
| *rotation_mode* | {None, 'default', 'anchor'} |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *text* | object |
| *transform* | *Transform* |
| *transform_rotates_text* | bool |
| *url* | str |
| *usetex* | bool, default: *rcParams["text.usetex"]* (default |
| *verticalalignment* or va | {'baseline', 'bottom', 'center', 'center_baseline', 'top'} |
| *visible* | bool |
| *wrap* | bool |
| *x* | float |
| *y* | float |
| *zorder* | float |

**set_axis_direction**(*label_direction*)

Adjust the text angle and text alignment of ticklabels according to the Matplotlib convention.

The *label_direction* must be one of [left, right, bottom, top].

| Property | left | bottom | right | top |
|---|---|---|---|---|
| ticklabel angle | 90 | 0 | -90 | 180 |
| ticklabel va | center | baseline | center | baseline |
| ticklabel ha | right | center | right | center |

Note that the text angles are actually relative to (90 + angle of the direction to the ticklabel), which gives 0 for bottom axis.

**Parameters**

**label_direction**

[{"left", "bottom", "right", "top"}]

**set_locs_angles_labels**(*locs_angles_labels*)

### mpl_toolkits.axisartist.axis_artist.Ticks

**class** mpl_toolkits.axisartist.axis_artist.**Ticks**(*ticksize*, *tick_out=False*, *, *axis=None*, ***kwargs*)

Bases: *AttributeCopier*, *Line2D*

Ticks are derived from *Line2D*, and note that ticks themselves are markers. Thus, you should use set_mec, set_mew, etc.

To change the tick size (length), you need to use *set_ticksize*. To change the direction of the ticks (ticks are in opposite direction of ticklabels by default), use `set_tick_out(False)`

**draw**(*renderer*)

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (*Artist.get_visible* returns False).

**Parameters**

**renderer**

[*RendererBase* subclass.]

### Notes

This method is overridden in the Artist subclasses.

**get_color**()

Return the line color.

See also *set_color*.

**get_markeredgecolor**()

Return the marker edge color.

See also *set_markeredgecolor*.

**get_markeredgewidth**()

Return the marker edge width in points.

See also *set_markeredgewidth*.

**get_ref_artist**()

Return the underlying artist that actually defines some properties (e.g., color) of this artist.

**get_tick_out**()

Return whether ticks are drawn inside or outside the axes.

**get_ticksize**()

Return length of the ticks in points.

**set**(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *antialiased=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *color=<UNSET>*, *dash_capstyle=<UNSET>*, *dash_joinstyle=<UNSET>*, *dashes=<UNSET>*, *data=<UNSET>*, *drawstyle=<UNSET>*, *fillstyle=<UNSET>*, *gapcolor=<UNSET>*, *gid=<UNSET>*, *in_layout=<UNSET>*, *label=<UNSET>*, *linestyle=<UNSET>*, *linewidth=<UNSET>*, *locs_angles=<UNSET>*, *marker=<UNSET>*, *markeredgecolor=<UNSET>*, *markeredgewidth=<UNSET>*, *markerfacecolor=<UNSET>*, *markerfacecoloralt=<UNSET>*, *markersize=<UNSET>*, *markevery=<UNSET>*, *mouseover=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *pickradius=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *solid_capstyle=<UNSET>*, *solid_joinstyle=<UNSET>*, *tick_out=<UNSET>*, *ticksize=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *xdata=<UNSET>*, *ydata=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value, |
| *alpha* | scalar or None |
| *animated* | bool |
| *antialiased* or aa | bool |

Table 190 – continued from pr

| Property | Description |
|---|---|
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *color* or c | unknown |
| *dash_capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *dash_joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *dashes* | sequence of floats (on/off ink in points) or (None, None) |
| *data* | (2, N) array or two 1D arrays |
| *drawstyle* or ds | {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'defa |
| *figure* | *Figure* |
| *fillstyle* | {'full', 'left', 'right', 'bottom', 'top', 'none'} |
| *gapcolor* | *color* or None |
| *gid* | str |
| *in_layout* | bool |
| *label* | object |
| *linestyle* or ls | {'-', '--', '-.', ':', '', (offset, on-off-seq), ...} |
| *linewidth* or lw | float |
| *locs_angles* | unknown |
| *marker* | marker style string, *Path* or *MarkerStyle* |
| *markeredgecolor* or mec | *color* |
| *markeredgewidth* or mew | float |
| *markerfacecolor* or mfc | *color* |
| *markerfacecoloralt* or mfcalt | *color* |
| *markersize* or ms | float |
| *markevery* | None or int or (int, int) or slice or list[int] or float or (float, float) o |
| *mouseover* | bool |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | float or callable[[Artist, Event], tuple[bool, dict]] |
| *pickradius* | float |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *solid_capstyle* | *CapStyle* or {'butt', 'projecting', 'round'} |
| *solid_joinstyle* | *JoinStyle* or {'miter', 'round', 'bevel'} |
| *tick_out* | unknown |
| *ticksize* | unknown |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *xdata* | 1D array |
| *ydata* | 1D array |
| *zorder* | float |

**set_color**(*color*)

Set the color of the line.

**Parameters**

**color**

[*color*]

**set_locs_angles**(*locs_angles*)

**set_tick_out**(*b*)

Set whether ticks are drawn inside or outside the axes.

**set_ticksize**(*ticksize*)

Set length of the ticks in points.

## mpl_toolkits.axisartist.axisline_style

Provides classes to style the axis lines.

## Classes

| | |
|---|---|
| *AxislineStyle*(stylename, **kwargs) | A container class which defines style classes for AxisArtists. |

## mpl_toolkits.axisartist.axisline_style.AxislineStyle

**class** mpl_toolkits.axisartist.axisline_style.**AxislineStyle**(*stylename*, ***kwargs*)

Bases: `_Style`

A container class which defines style classes for AxisArtists.

An instance of any axisline style class is a callable object, whose call signature is

```
__call__(self, axis_artist, path, transform)
```

When called, this should return an *Artist* with the following methods:

```
def set_path(self, path):
    # set the path for axisline.

def set_line_mutation_scale(self, scale):
    # set the scale
```

(continues on next page)

(continued from previous page)

```
def draw(self, renderer):
    # draw
```

Return the instance of the subclass with the given style name.

**class FilledArrow**(*size=1*, *facecolor=None*)

Bases: *SimpleArrow*

An arrow with a filled head.

**Parameters**

**size**

[float] Size of the arrow as a fraction of the ticklabel size.

**facecolor**

[*color*, default: *rcParams["axes.edgecolor"]* (default: 'black')] Fill color.

New in version 3.7.

**ArrowAxisClass**

alias of FilledArrow

**new_line**(*axis_artist*, *transform*)

**class SimpleArrow**(*size=1*)

Bases: _Base

A simple arrow.

**Parameters**

**size**

[float] Size of the arrow as a fraction of the ticklabel size.

**ArrowAxisClass**

alias of SimpleArrow

**new_line**(*axis_artist*, *transform*)

### mpl_toolkits.axisartist.axislines

Axislines includes modified implementation of the Axes class. The biggest difference is that the artists responsible for drawing the axis spine, ticks, ticklabels and axis labels are separated out from Matplotlib's Axis class. Originally, this change was motivated to support curvilinear grid. Here are a few reasons that I came up with a new axes class:

- "top" and "bottom" x-axis (or "left" and "right" y-axis) can have different ticks (tick locations and labels). This is not possible with the current Matplotlib, although some twin axes trick can help.
- Curvilinear grid.
- angled ticks.

In the new axes class, xaxis and yaxis is set to not visible by default, and new set of artist (AxisArtist) are defined to draw axis line, ticks, ticklabels and axis label. Axes.axis attribute serves as a dictionary of these artists, i.e., ax.axis["left"] is a AxisArtist instance responsible to draw left y-axis. The default Axes.axis contains "bottom", "left", "top" and "right".

AxisArtist can be considered as a container artist and has the following children artists which will draw ticks, labels, etc.

- line
- major_ticks, major_ticklabels
- minor_ticks, minor_ticklabels
- offsetText
- label

Note that these are separate artists from *matplotlib.axis.Axis*, thus most tick-related functions in Matplotlib won't work. For example, color and markerwidth of the `ax.axis["bottom"].major_ticks` will follow those of Axes.xaxis unless explicitly specified.

In addition to AxisArtist, the Axes will have *gridlines* attribute, which obviously draws grid lines. The gridlines needs to be separated from the axis as some gridlines can never pass any axis.

### Classes

| | |
|---|---|
| *Axes*(*args[, grid_helper]) | Build an Axes in a figure. |
| *AxesZero*(*args[, grid_helper]) | Build an Axes in a figure. |
| *AxisArtistHelper*() | |
| *AxisArtistHelperRectlinear*() | |
| *FixedAxisArtistHelperRectilinear*(axes, loc) | nth_coord = along which coordinate value varies in 2D, nth_coord = 0 -> x axis, nth_coord = 1 -> y axis |
| *FloatingAxisArtistHelperRectilinear*(axes, ...) | |
| *GridHelperBase*() | |
| *GridHelperRectlinear*(axes) | |
| *Subplot* | alias of *Axes* |
| *SubplotZero* | alias of *AxesZero* |

### mpl_toolkits.axisartist.axislines.Axes

**class** mpl_toolkits.axisartist.axislines.**Axes**(**args*, *grid_helper=None*, ***kwargs*)

Bases: *Axes*

Build an Axes in a figure.

**Parameters**

**fig**

[*Figure*] The Axes is built in the *Figure* *fig*.

***args**

`*args` can be a single `(left, bottom, width, height)` rectangle or a single *Bbox*. This specifies the rectangle (in figure coordinates) where the Axes is positioned.

`*args` can also consist of three numbers or a single three-digit number; in the latter case, the digits are considered as independent numbers. The numbers are interpreted as `(nrows, ncols, index)`: `(nrows, ncols)` specifies the size of an array of subplots, and `index` is the 1-based index of the subplot being created. Finally, `*args` can also directly be a *SubplotSpec* instance.

**sharex, sharey**

[*Axes*, optional] The x- or y-*axis* is shared with the x- or y-axis in the input *Axes*.

**frameon**

[bool, default: True] Whether the Axes frame is visible.

**box_aspect**

[float, optional] Set a fixed aspect for the Axes box, i.e. the ratio of height to width. See *set_box_aspect* for details.

**forward_navigation_events**

[bool or "auto", default: "auto"] Control whether pan/zoom events are passed through to Axes below this one. "auto" is *True* for axes with an invisible patch and *False* otherwise.

****kwargs**

Other optional keyword arguments:

| Property | Description |
|---|---|
| *adjustable* | {'box', 'datalim'} |
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi v |
| *alpha* | scalar or None |
| *anchor* | (float, float) or {'C', 'SW', 'S', 'SE', 'E', 'NE', ...} |
| *animated* | bool |
| *aspect* | {'auto', 'equal'} or float |
| *autoscale_on* | bool |
| *autoscalex_on* | unknown |
| *autoscaley_on* | unknown |
| *axes_locator* | Callable[[Axes, Renderer], Bbox] |
| *axisbelow* | bool or 'line' |
| *box_aspect* | float or None |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *facecolor* or fc | *color* |
| *figure* | *Figure* |
| *forward_navigation_events* | bool or "auto" |
| *frame_on* | bool |
| *gid* | str |
| *in_layout* | bool |
| *label* | object |
| *mouseover* | bool |
| *navigate* | bool |
| *navigate_mode* | unknown |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *position* | [left, bottom, width, height] or *Bbox* |
| *prop_cycle* | *Cycler* |

Table 191 – continued fro

| Property | Description |
|---|---|
| *rasterization_zorder* | float or None |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *subplotspec* | unknown |
| *title* | str |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *xbound* | (lower: float, upper: float) |
| *xlabel* | str |
| *xlim* | (left: float, right: float) |
| *xmargin* | float greater than -0.5 |
| *xscale* | unknown |
| *xticklabels* | unknown |
| *xticks* | unknown |
| *ybound* | (lower: float, upper: float) |
| *ylabel* | str |
| *ylim* | (bottom: float, top: float) |
| *ymargin* | float greater than -0.5 |
| *yscale* | unknown |
| *yticklabels* | unknown |
| *yticks* | unknown |
| *zorder* | float |

**Returns**

***Axes***

The new *Axes* object.

**__call__**(**args*, ***kwargs*)

[*Deprecated*]

### Notes

Deprecated since version 3.8: Use ax.axis instead.

**property axis**

Convenience method to get or set some axis properties.

Call signatures:

```
xmin, xmax, ymin, ymax = axis()
xmin, xmax, ymin, ymax = axis([xmin, xmax, ymin, ymax])
xmin, xmax, ymin, ymax = axis(option)
xmin, xmax, ymin, ymax = axis(**kwargs)
```

**Parameters**

**xmin, xmax, ymin, ymax**

[float, optional] The axis limits to be set. This can also be achieved using

```
ax.set(xlim=(xmin, xmax), ylim=(ymin, ymax))
```

**option**

[bool or str] If a bool, turns axis lines and labels on or off. If a string, possible values are:

| Value | Description |
|---|---|
| 'off' or False | Hide all axis decorations, i.e. axis labels, spines, tick marks, tick labels, and grid lines. This is the same as *set_axis_off()*. |
| 'on' or True | Do not hide all axis decorations, i.e. axis labels, spines, tick marks, tick labels, and grid lines. This is the same as *set_axis_on()*. |
| 'equal | Set equal scaling (i.e., make circles circular) by changing the axis limits. This is the same as `ax.set_aspect('equal', adjustable='datalim')`. Explicit data limits may not be respected in this case. |
| 'scale | Set equal scaling (i.e., make circles circular) by changing dimensions of the plot box. This is the same as `ax.set_aspect('equal', adjustable='box', anchor='C')`. Additionally, further autoscaling will be disabled. |
| 'tight' | Set limits just large enough to show all data, then disable further autoscaling. |
| 'auto' | Automatic scaling (fill plot box with data). |
| 'image' | 'scaled' with axis limits equal to data limits. |
| 'squar | Square plot; similar to 'scaled', but initially forcing `xmax-xmin == ymax-ymin`. |

**emit**

[bool, default: True] Whether observers are notified of the axis limit change. This option is passed on to *set_xlim* and *set_ylim*.

**Returns**

**xmin, xmax, ymin, ymax**

[float] The axis limits.

**See also:**

*matplotlib.axes.Axes.set_xlim*
*matplotlib.axes.Axes.set_ylim*

### Notes

For 3D Axes, this method additionally takes *zmin*, *zmax* as parameters and likewise returns them.

**clear**()

Clear the Axes.

**get_children**()

Return a list of the child *Artist*s of this *Artist*.

**get_grid_helper**()

**grid**(*visible=None*, *which='major'*, *axis='both'*, ***kwargs*)

Toggle the gridlines, and optionally set the properties of the lines.

**new_fixed_axis**(*loc*, *offset=None*)

**new_floating_axis**(*nth_coord*, *value*, *axis_direction='bottom'*)

**set**(***, *adjustable=<UNSET>*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *anchor=<UNSET>*, *animated=<UNSET>*, *aspect=<UNSET>*, *autoscale_on=<UNSET>*, *autoscalex_on=<UNSET>*, *autoscaley_on=<UNSET>*, *axes_locator=<UNSET>*, *axisbelow=<UNSET>*, *box_aspect=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *facecolor=<UNSET>*, *forward_navigation_events=<UNSET>*, *frame_on=<UNSET>*, *gid=<UNSET>*, *in_layout=<UNSET>*, *label=<UNSET>*, *mouseover=<UNSET>*, *navigate=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *position=<UNSET>*, *prop_cycle=<UNSET>*, *rasterization_zorder=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *subplotspec=<UNSET>*, *title=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *xbound=<UNSET>*, *xlabel=<UNSET>*, *xlim=<UNSET>*, *xmargin=<UNSET>*, *xscale=<UNSET>*, *xticklabels=<UNSET>*, *xticks=<UNSET>*, *ybound=<UNSET>*, *ylabel=<UNSET>*, *ylim=<UNSET>*, *ymargin=<UNSET>*, *yscale=<UNSET>*, *yticklabels=<UNSET>*, *yticks=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *adjustable* | {'box', 'datalim'} |
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value |
| *alpha* | scalar or None |
| *anchor* | (float, float) or {'C', 'SW', 'S', 'SE', 'E', 'NE', ...} |
| *animated* | bool |
| *aspect* | {'auto', 'equal'} or float |
| *autoscale_on* | bool |
| *autoscalex_on* | unknown |
| *autoscaley_on* | unknown |
| *axes_locator* | Callable[[Axes, Renderer], Bbox] |
| *axisbelow* | bool or 'line' |
| *box_aspect* | float or None |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *facecolor* or fc | *color* |
| *figure* | *Figure* |
| *forward_navigation_events* | bool or "auto" |
| *frame_on* | bool |
| *gid* | str |
| *in_layout* | bool |
| *label* | object |
| *mouseover* | bool |
| *navigate* | bool |
| *navigate_mode* | unknown |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *position* | [left, bottom, width, height] or *Bbox* |
| *prop_cycle* | *Cycler* |
| *rasterization_zorder* | float or None |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *subplotspec* | unknown |
| *title* | str |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *xbound* | (lower: float, upper: float) |
| *xlabel* | str |
| *xlim* | (left: float, right: float) |
| *xmargin* | float greater than -0.5 |
| *xscale* | unknown |
| *xticklabels* | unknown |
| *xticks* | unknown |

Table 192 – continued from pr

| Property | Description |
|---|---|
| *ybound* | (lower: float, upper: float) |
| *ylabel* | str |
| *ylim* | (bottom: float, top: float) |
| *ymargin* | float greater than -0.5 |
| *yscale* | unknown |
| *yticklabels* | unknown |
| *yticks* | unknown |
| *zorder* | float |

**toggle_axisline**(*b=None*)

### mpl_toolkits.axisartist.axislines.AxesZero

**class** mpl_toolkits.axisartist.axislines.**AxesZero**(**args*, *grid_helper=None*, ***kwargs*)

Bases: *Axes*

Build an Axes in a figure.

**Parameters**

**fig**

[*Figure*] The Axes is built in the *Figure* *fig*.

***args**

`*args` can be a single `(left, bottom, width, height)` rectangle or a single *Bbox*. This specifies the rectangle (in figure coordinates) where the Axes is positioned.

`*args` can also consist of three numbers or a single three-digit number; in the latter case, the digits are considered as independent numbers. The numbers are interpreted as `(nrows, ncols, index)`: `(nrows, ncols)` specifies the size of an array of subplots, and `index` is the 1-based index of the subplot being created. Finally, `*args` can also directly be a *SubplotSpec* instance.

**sharex, sharey**

[*Axes*, optional] The x- or y-*axis* is shared with the x- or y-axis in the input *Axes*.

**frameon**

[bool, default: True] Whether the Axes frame is visible.

**box_aspect**

[float, optional] Set a fixed aspect for the Axes box, i.e. the ratio of height to width. See *set_box_aspect* for details.

**forward_navigation_events**

[bool or "auto", default: "auto"] Control whether pan/zoom events are passed through to Axes below this one. "auto" is *True* for axes with an invisible patch and *False* otherwise.

****kwargs**

Other optional keyword arguments:

| Property | Description |
|---|---|
| *adjustable* | {'box', 'datalim'} |
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi v |
| *alpha* | scalar or None |
| *anchor* | (float, float) or {'C', 'SW', 'S', 'SE', 'E', 'NE', ...} |
| *animated* | bool |
| *aspect* | {'auto', 'equal'} or float |
| *autoscale_on* | bool |
| *autoscalex_on* | unknown |
| *autoscaley_on* | unknown |
| *axes_locator* | Callable[[Axes, Renderer], Bbox] |
| *axisbelow* | bool or 'line' |
| *box_aspect* | float or None |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *facecolor* or fc | *color* |
| *figure* | *Figure* |
| *forward_navigation_events* | bool or "auto" |
| *frame_on* | bool |
| *gid* | str |
| *in_layout* | bool |
| *label* | object |
| *mouseover* | bool |
| *navigate* | bool |
| *navigate_mode* | unknown |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *position* | [left, bottom, width, height] or *Bbox* |
| *prop_cycle* | *Cycler* |
| *rasterization_zorder* | float or None |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |

Table 193 – continued fro

| Property | Description |
|---|---|
| *snap* | bool or None |
| *subplotspec* | unknown |
| *title* | str |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *xbound* | (lower: float, upper: float) |
| *xlabel* | str |
| *xlim* | (left: float, right: float) |
| *xmargin* | float greater than -0.5 |
| *xscale* | unknown |
| *xticklabels* | unknown |
| *xticks* | unknown |
| *ybound* | (lower: float, upper: float) |
| *ylabel* | str |
| *ylim* | (bottom: float, top: float) |
| *ymargin* | float greater than -0.5 |
| *yscale* | unknown |
| *yticklabels* | unknown |
| *yticks* | unknown |
| *zorder* | float |

**Returns**

***Axes***

The new *Axes* object.

**clear**()

Clear the Axes.

**set**(*, *adjustable=<UNSET>*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *anchor=<UNSET>*, *animated=<UNSET>*, *aspect=<UNSET>*, *autoscale_on=<UNSET>*, *autoscalex_on=<UNSET>*, *autoscaley_on=<UNSET>*, *axes_locator=<UNSET>*, *axisbelow=<UNSET>*, *box_aspect=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *facecolor=<UNSET>*, *forward_navigation_events=<UNSET>*, *frame_on=<UNSET>*, *gid=<UNSET>*, *in_layout=<UNSET>*, *label=<UNSET>*, *mouseover=<UNSET>*, *navigate=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *position=<UNSET>*, *prop_cycle=<UNSET>*, *rasterization_zorder=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *subplotspec=<UNSET>*, *title=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *xbound=<UNSET>*, *xlabel=<UNSET>*, *xlim=<UNSET>*, *xmargin=<UNSET>*, *xscale=<UNSET>*, *xticklabels=<UNSET>*, *xticks=<UNSET>*, *ybound=<UNSET>*, *ylabel=<UNSET>*, *ylim=<UNSET>*, *ymargin=<UNSET>*, *yscale=<UNSET>*, *yticklabels=<UNSET>*, *yticks=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

| Property | Description |
|---|---|
| *adjustable* | {'box', 'datalim'} |
| *agg_filter* | a filter function, which takes a (m, n, 3) float array and a dpi value |
| *alpha* | scalar or None |
| *anchor* | (float, float) or {'C', 'SW', 'S', 'SE', 'E', 'NE', ...} |
| *animated* | bool |
| *aspect* | {'auto', 'equal'} or float |
| *autoscale_on* | bool |
| *autoscalex_on* | unknown |
| *autoscaley_on* | unknown |
| *axes_locator* | Callable[[Axes, Renderer], Bbox] |
| *axisbelow* | bool or 'line' |
| *box_aspect* | float or None |
| *clip_box* | *BboxBase* or None |
| *clip_on* | bool |
| *clip_path* | Patch or (Path, Transform) or None |
| *facecolor* or fc | *color* |
| *figure* | *Figure* |
| *forward_navigation_events* | bool or "auto" |
| *frame_on* | bool |
| *gid* | str |
| *in_layout* | bool |
| *label* | object |
| *mouseover* | bool |
| *navigate* | bool |
| *navigate_mode* | unknown |
| *path_effects* | list of *AbstractPathEffect* |
| *picker* | None or bool or float or callable |
| *position* | [left, bottom, width, height] or *Bbox* |
| *prop_cycle* | *Cycler* |
| *rasterization_zorder* | float or None |
| *rasterized* | bool |
| *sketch_params* | (scale: float, length: float, randomness: float) |
| *snap* | bool or None |
| *subplotspec* | unknown |
| *title* | str |
| *transform* | *Transform* |
| *url* | str |
| *visible* | bool |
| *xbound* | (lower: float, upper: float) |
| *xlabel* | str |

Table 194 – continued from pr

| Property | Description |
|---|---|
| *xlim* | (left: float, right: float) |
| *xmargin* | float greater than -0.5 |
| *xscale* | unknown |
| *xticklabels* | unknown |
| *xticks* | unknown |
| *ybound* | (lower: float, upper: float) |
| *ylabel* | str |
| *ylim* | (bottom: float, top: float) |
| *ymargin* | float greater than -0.5 |
| *yscale* | unknown |
| *yticklabels* | unknown |
| *yticks* | unknown |
| *zorder* | float |

### mpl_toolkits.axisartist.axislines.AxisArtistHelper

**class** mpl_toolkits.axisartist.axislines.**AxisArtistHelper**

Bases: object

**Fixed**

alias of _FixedAxisArtistHelperBase

**Floating**

alias of _FloatingAxisArtistHelperBase

### mpl_toolkits.axisartist.axislines.AxisArtistHelperRectlinear

**class** mpl_toolkits.axisartist.axislines.**AxisArtistHelperRectlinear**

Bases: object

**Fixed**

alias of *FixedAxisArtistHelperRectilinear*

**Floating**

alias of *FloatingAxisArtistHelperRectilinear*

### mpl_toolkits.axisartist.axislines.FixedAxisArtistHelperRectilinear

**class** mpl_toolkits.axisartist.axislines.**FixedAxisArtistHelperRectilinear**(*axes*, *loc*, *nth_coord=<d* *parameter>*)

Bases: `_FixedAxisArtistHelperBase`

nth_coord = along which coordinate value varies in 2D, nth_coord = 0 -> x axis, nth_coord = 1 -> y axis

**get_tick_iterators**(*axes*)

tick_loc, tick_angle, tick_label

### mpl_toolkits.axisartist.axislines.FloatingAxisArtistHelperRectilinear

**class** mpl_toolkits.axisartist.axislines.**FloatingAxisArtistHelperRectilinear**(*axes*, *nth_coord* *passingth-rough_po* *axis_direc*

Bases: `_FloatingAxisArtistHelperBase`

**get_axislabel_pos_angle**(*axes*)

Return the label reference position in transAxes.

get_label_transform() returns a transform of (transAxes+offset)

**get_axislabel_transform**(*axes*)

**get_line**(*axes*)

**get_line_transform**(*axes*)

**get_tick_iterators**(*axes*)

tick_loc, tick_angle, tick_label

**get_tick_transform**(*axes*)

### mpl_toolkits.axisartist.axislines.GridHelperBase

**class** mpl_toolkits.axisartist.axislines.**GridHelperBase**

Bases: object

**get_gridlines**(*which*, *axis*)

Return list of grid lines as a list of paths (list of points).

**Parameters**

**which**

[{"both", "major", "minor"}]

**axis**

[{"both", "x", "y"}]

**update_lim**(*axes*)

### mpl_toolkits.axisartist.axislines.GridHelperRectlinear

**class** mpl_toolkits.axisartist.axislines.**GridHelperRectlinear**(*axes*)

Bases: *GridHelperBase*

**get_gridlines**(*which='major'*, *axis='both'*)

Return list of gridline coordinates in data coordinates.

**Parameters**

**which**

[{"both", "major", "minor"}]

**axis**

[{"both", "x", "y"}]

**new_fixed_axis**(*loc*, *nth_coord=<deprecated parameter>*, *axis_direction=None*, *offset=None*, *axes=None*)

**new_floating_axis**(*nth_coord*, *value*, *axis_direction='bottom'*, *axes=None*)

### mpl_toolkits.axisartist.axislines.Subplot

mpl_toolkits.axisartist.axislines.**Subplot**
alias of *Axes*

### mpl_toolkits.axisartist.axislines.SubplotZero

mpl_toolkits.axisartist.axislines.**SubplotZero**
alias of *AxesZero*

### mpl_toolkits.axisartist.floating_axes

An experimental support for curvilinear grid.

### Classes

| | |
|---|---|
| *ExtremeFinderFixed*(extremes) | This subclass always returns the same bounding box. |
| *FixedAxisArtistHelper*(grid_helper, side[, ...]) | nth_coord = along which coordinate value varies. |
| *FloatingAxes* | alias of FloatingAxesHostAxes |
| *FloatingAxesBase*(*args, grid_helper, **kwargs) | |
| *FloatingAxisArtistHelper*(grid_helper, ...[, ...]) | nth_coord = along which coordinate value varies. |
| *FloatingSubplot* | alias of FloatingAxesHostAxes |
| *GridHelperCurveLinear*(aux_trans, extremes[, ...]) | **Parameters** |

### mpl_toolkits.axisartist.floating_axes.ExtremeFinderFixed

**class** mpl_toolkits.axisartist.floating_axes.**ExtremeFinderFixed**(*extremes*)

Bases: *ExtremeFinderSimple*

This subclass always returns the same bounding box.

**Parameters**

**extremes**

[(float, float, float, float)] The bounding box that this helper always returns.

**__call__**(*transform_xy*, *x1*, *y1*, *x2*, *y2*)

Compute an approximation of the bounding box obtained by applying *transform_xy* to the box delimited by `(x1, y1, x2, y2)`.

The intended use is to have `(x1, y1, x2, y2)` in axes coordinates, and have *transform_xy* be the transform from axes coordinates to data coordinates; this method then returns the range of data coordinates that span the actual axes.

The computation is done by sampling `nx * ny` equispaced points in the `(x1, y1, x2, y2)` box and finding the resulting points with extremal coordinates; then adding some padding to take into account the finite sampling.

As each sampling step covers a relative range of *1/nx* or *1/ny*, the padding is computed by expanding the span covered by the extremal coordinates by these fractions.

### mpl_toolkits.axisartist.floating_axes.FixedAxisArtistHelper

**class** mpl_toolkits.axisartist.floating_axes.**FixedAxisArtistHelper**(*grid_helper*, *side*, *nth_coord_ticks=None*)

Bases: *FloatingAxisArtistHelper*

**nth_coord = along which coordinate value varies.**

nth_coord = 0 -> x axis, nth_coord = 1 -> y axis

**get_line**(*axes*)

**get_tick_iterators**(*axes*)

tick_loc, tick_angle, tick_label, (optionally) tick_label

**update_lim**(*axes*)

### mpl_toolkits.axisartist.floating_axes.FloatingAxes

mpl_toolkits.axisartist.floating_axes.**FloatingAxes**

alias of `FloatingAxesHostAxes`

### mpl_toolkits.axisartist.floating_axes.FloatingAxesBase

**class** mpl_toolkits.axisartist.floating_axes.**FloatingAxesBase**(**args*, *grid_helper*, ***kwargs*)

Bases: `object`

**adjust_axes_lim**()

**clear**()

### Examples using `mpl_toolkits.axisartist.floating_axes.FloatingAxesBase`

- *floating_axes features*

## mpl_toolkits.axisartist.floating_axes.FloatingAxisArtistHelper

**class** mpl_toolkits.axisartist.floating_axes.**FloatingAxisArtistHelper**(*grid_helper*, *nth_coord*, *value*, *axis_direction=Non*

Bases: *FloatingAxisArtistHelper*

**nth_coord = along which coordinate value varies.**

nth_coord = 0 -> x axis, nth_coord = 1 -> y axis

## mpl_toolkits.axisartist.floating_axes.FloatingSubplot

mpl_toolkits.axisartist.floating_axes.**FloatingSubplot**

alias of `FloatingAxesHostAxes`

## mpl_toolkits.axisartist.floating_axes.GridHelperCurveLinear

**class** mpl_toolkits.axisartist.floating_axes.**GridHelperCurveLinear**(*aux_trans*, *extremes*, *grid_locator1=None*, *grid_locator2=None*, *tick_formatter1=None*, *tick_formatter2=None*)

Bases: *GridHelperCurveLinear*

**Parameters**

**aux_trans**

[*Transform* or tuple[Callable, Callable]] The transform from curved coordinates to rectilinear coordinate: either a *Transform* instance (which provides also its inverse), or a pair of callables `(trans, inv_trans)` that define the transform and its inverse. The callables should have signature:

```
x_rect, y_rect = trans(x_curved, y_curved)
x_curved, y_curved = inv_trans(x_rect, y_rect)
```

**extreme_finder**
**grid_locator1, grid_locator2**

Grid locators for each axis.

**tick_formatter1, tick_formatter2**

Tick formatters for each axis.

**get_data_boundary**(*side*)

[*Deprecated*] Return v=0, nth=1.

### Notes

Deprecated since version 3.8.

**get_gridlines**(*which='major'*, *axis='both'*)

Return list of grid lines as a list of paths (list of points).

**Parameters**

**which**

[{"both", "major", "minor"}]

**axis**

[{"both", "x", "y"}]

**new_fixed_axis**(*loc*, *nth_coord=None*, *axis_direction=None*, *offset=None*, *axes=None*)

## Examples using `mpl_toolkits.axisartist.floating_axes.GridHelperCurveLinear`

- *floating_axes features*

## Functions

*floatingaxes_class_factory*(axes_class)

### mpl_toolkits.axisartist.floating_axes.floatingaxes_class_factory

mpl_toolkits.axisartist.floating_axes.**floatingaxes_class_factory**(*axes_class*)

### mpl_toolkits.axisartist.grid_finder

### Classes

| | |
|---|---|
| *DictFormatter*(format_dict[, formatter]) | format_dict : dictionary for format strings to be used. |
| *ExtremeFinderSimple*(nx, ny) | A helper class to figure out the range of grid lines that need to be drawn. |
| *FixedLocator*(locs) | |
| *FormatterPrettyPrint*([useMathText]) | |
| *GridFinder*(transform[, extreme_finder, ...]) | Internal helper for *GridHelperCurveLinear*, with the same constructor parameters; should not be directly instantiated. |
| *MaxNLocator*([nbins, steps, trim, integer, ...]) | **Parameters** |

### mpl_toolkits.axisartist.grid_finder.DictFormatter

**class** mpl_toolkits.axisartist.grid_finder.**DictFormatter**(*format_dict*, *formatter=None*)

Bases: object

format_dict : dictionary for format strings to be used. formatter : fall-back formatter

**__call__**(*direction*, *factor*, *values*)

factor is ignored if value is found in the dictionary

### Examples using mpl_toolkits.axisartist.grid_finder.DictFormatter

- *floating_axes features*

### mpl_toolkits.axisartist.grid_finder.ExtremeFinderSimple

**class** mpl_toolkits.axisartist.grid_finder.**ExtremeFinderSimple**(*nx*, *ny*)

Bases: object

A helper class to figure out the range of grid lines that need to be drawn.

**Parameters**

**nx, ny**

[int] The number of samples in each direction.

**__call__**(*transform_xy*, *x1*, *y1*, *x2*, *y2*)

Compute an approximation of the bounding box obtained by applying *transform_xy* to the box delimited by `(x1, y1, x2, y2)`.

The intended use is to have `(x1, y1, x2, y2)` in axes coordinates, and have *transform_xy* be the transform from axes coordinates to data coordinates; this method then returns the range of data coordinates that span the actual axes.

The computation is done by sampling `nx * ny` equispaced points in the `(x1, y1, x2, y2)` box and finding the resulting points with extremal coordinates; then adding some padding to take into account the finite sampling.

As each sampling step covers a relative range of *1/nx* or *1/ny*, the padding is computed by expanding the span covered by the extremal coordinates by these fractions.

### Examples using `mpl_toolkits.axisartist.grid_finder.ExtremeFinderSimple`

- *axis_direction demo*
- *Curvilinear grid demo*
- *Demo CurveLinear Grid2*
- *floating_axis demo*
- *Simple Axis Pad*

### mpl_toolkits.axisartist.grid_finder.FixedLocator

**class** mpl_toolkits.axisartist.grid_finder.**FixedLocator**(*locs*)

Bases: object

**__call__**(*v1*, *v2*)

Call self as a function.

### Examples using `mpl_toolkits.axisartist.grid_finder.FixedLocator`

- *floating_axes features*

### mpl_toolkits.axisartist.grid_finder.FormatterPrettyPrint

**class** mpl_toolkits.axisartist.grid_finder.**FormatterPrettyPrint**(*useMathText=True*)

Bases: object

**__call__**(*direction*, *factor*, *values*)

Call self as a function.

### mpl_toolkits.axisartist.grid_finder.GridFinder

**class** mpl_toolkits.axisartist.grid_finder.**GridFinder**(*transform*, *extreme_finder=None*, *grid_locator1=None*, *grid_locator2=None*, *tick_formatter1=None*, *tick_formatter2=None*)

Bases: object

Internal helper for *GridHelperCurveLinear*, with the same constructor parameters; should not be directly instantiated.

**get_grid_info**(*x1*, *y1*, *x2*, *y2*)

**lon_values, lat_values**

[list of grid values. if integer is given,] rough number of grids in each direction.

**get_transform**()

**inv_transform_xy**(*x*, *y*)

**set_transform**(*aux_trans*)

**transform_xy**(*x*, *y*)

**update**(***kwargs*)

**update_transform**(*aux_trans*)

### mpl_toolkits.axisartist.grid_finder.MaxNLocator

**class** mpl_toolkits.axisartist.grid_finder.**MaxNLocator**(*nbins=10*, *steps=None*, *trim=True*, *integer=False*, *symmetric=False*, *prune=None*)

Bases: *MaxNLocator*

**Parameters**

**nbins**

[int or 'auto', default: 10] Maximum number of intervals; one less than max number of ticks. If the string 'auto', the number of bins will be automatically determined based on the length of the axis.

**steps**

[array-like, optional] Sequence of acceptable tick multiples, starting with 1 and ending with 10. For example, if `steps=[1, 2, 4, 5, 10]`, `20, 40, 60` or `0.4, 0.6, 0.8` would be possible sets of ticks because they are multiples of 2. `30, 60, 90` would not be generated because 3 does not appear in this example list of steps.

**integer**

[bool, default: False] If True, ticks will take only integer values, provided at least *min_n_ticks* integers are found within the view limits.

**symmetric**

[bool, default: False] If True, autoscaling will result in a range symmetric about zero.

**prune**

[{'lower', 'upper', 'both', None}, default: None] Remove the 'lower' tick, the 'upper' tick, or ticks on 'both' sides *if they fall exactly on an axis' edge* (this typically occurs when *rcParams["axes.autolimit_mode"]* (default: `'data'`) is 'round_numbers'). Removing such ticks is mostly useful for stacked or ganged plots, where the upper tick of an Axes overlaps with the lower tick of the axes above it.

**min_n_ticks**

[int, default: 2] Relax *nbins* and *integer* constraints if necessary to obtain this minimum number of ticks.

**__call__**(*v1*, *v2*)

Return the locations of the ticks.

### Examples using `mpl_toolkits.axisartist.grid_finder.MaxNLocator`

- *axis_direction demo*
- *Demo CurveLinear Grid2*
- *floating_axes features*
- *Simple Axis Pad*

## mpl_toolkits.axisartist.grid_helper_curvelinear

An experimental support for curvilinear grid.

### Classes

| | |
|---|---|
| *FixedAxisArtistHelper*(grid_helper, side[, ...]) | Helper class for a fixed axis. |
| *FloatingAxisArtistHelper*(grid_helper, ...[, ...]) | nth_coord = along which coordinate value varies. |
| *GridHelperCurveLinear*(aux_trans[, ...]) | **Parameters** |

## mpl_toolkits.axisartist.grid_helper_curvelinear.FixedAxisArtistHelper

**class** mpl_toolkits.axisartist.grid_helper_curvelinear.**FixedAxisArtistHelper**(*grid_help*
*side*,
*nth_coord*

Bases: `_FixedAxisArtistHelperBase`

Helper class for a fixed axis.

**nth_coord = along which coordinate value varies.**

nth_coord = 0 -> x axis, nth_coord = 1 -> y axis

**get_tick_iterators**(*axes*)

tick_loc, tick_angle, tick_label

**get_tick_transform**(*axes*)

**update_lim**(*axes*)

### mpl_toolkits.axisartist.grid_helper_curvelinear.FloatingAxisArtistHelper

**class** mpl_toolkits.axisartist.grid_helper_curvelinear.**FloatingAxisArtistHelper**(*grid_*
*nth_c*
*value*
*axis_*

Bases: `_FloatingAxisArtistHelperBase`

**nth_coord = along which coordinate value varies.**

nth_coord = 0 -> x axis, nth_coord = 1 -> y axis

**get_axislabel_pos_angle**(*axes*)

**get_axislabel_transform**(*axes*)

**get_line**(*axes*)

**get_line_transform**(*axes*)

**get_tick_iterators**(*axes*)

tick_loc, tick_angle, tick_label, (optionally) tick_label

**get_tick_transform**(*axes*)

**set_extremes**(*e1*, *e2*)

**update_lim**(*axes*)

### mpl_toolkits.axisartist.grid_helper_curvelinear.GridHelperCurveLinear

**class** mpl_toolkits.axisartist.grid_helper_curvelinear.**GridHelperCurveLinear**(*aux_trans*
*ex-*
*treme_fin*
*grid_loca*
*grid_loca*
*tick_form*
*tick_form*

Bases: *GridHelperBase*

**Parameters**

**aux_trans**

[*Transform* or tuple[Callable, Callable]] The transform from curved coordinates to rectilinear coordinate: either a *Transform* instance (which provides also its inverse), or a pair of callables `(trans, inv_trans)` that define the transform and its inverse. The callables should have signature:

```
x_rect, y_rect = trans(x_curved, y_curved)
x_curved, y_curved = inv_trans(x_rect, y_rect)
```

**extreme_finder**
**grid_locator1, grid_locator2**

Grid locators for each axis.

**tick_formatter1, tick_formatter2**

Tick formatters for each axis.

**get_gridlines**(*which='major'*, *axis='both'*)

Return list of grid lines as a list of paths (list of points).

**Parameters**

**which**

[{"both", "major", "minor"}]

**axis**

[{"both", "x", "y"}]

**get_tick_iterator**(*nth_coord*, *axis_side*, *minor=False*)

[*Deprecated*]

### Notes

Deprecated since version 3.9:

**new_fixed_axis**(*loc*, *, *nth_coord=None*, *axis_direction=None*, *offset=None*, *axes=None*)

**new_floating_axis**(*nth_coord*, *value*, *axes=None*, *axis_direction='bottom'*)

**update_grid_finder**(*aux_trans=None*, **kwargs*)

## mpl_toolkits.axisartist.parasite_axes

## 7.2.69 pylab

*pylab* is a historic interface and its use is strongly discouraged. The equivalent replacement is *matplotlib.pyplot*. See *Matplotlib Application Interfaces (APIs)* for a full overview of Matplotlib interfaces.

*pylab* was designed to support a MATLAB-like way of working with all plotting related functions directly available in the global namespace. This was achieved through a wildcard import (`from pylab import *`).

**Warning:** The use of *pylab* is discouraged for the following reasons:

`from pylab import *` imports all the functions from *matplotlib.pyplot*, `numpy`, `numpy.fft`, `numpy.linalg`, and `numpy.random`, and some additional functions into the global namespace.

Such a pattern is considered bad practice in modern python, as it clutters the global namespace. Even more severely, in the case of *pylab*, this will overwrite some builtin functions (e.g. the builtin `sum` will be replaced by `numpy.sum`), which can lead to unexpected behavior.

Top-level interfaces to create:

- figures: *pyplot.figure*
- subplots: *pyplot.subplots*, *pyplot.subplot_mosaic*

# Part III

# Community

CHAPTER

EIGHT

# EXTERNAL RESOURCES

## 8.1 Books, chapters and articles

- Scientific Visualization: Python + Matplotlib (2021) by Nicolas P. Rougier
- Mastering matplotlib by Duncan M. McGreggor
- Interactive Applications Using Matplotlib by Benjamin Root
- Matplotlib for Python Developers by Sandro Tosi
- Matplotlib chapter by John Hunter and Michael Droettboom in The Architecture of Open Source Applications
- Ten Simple Rules for Better Figures by Nicolas P. Rougier, Michael Droettboom and Philip E. Bourne
- Learning Scientific Programming with Python chapter 7 by Christian Hill
- Hands-On Data Analysis with Pandas, chapters 5 and 6 by Stefanie Molin

## 8.2 Videos

- Plotting with matplotlib by Mike Müller
- Introduction to NumPy and Matplotlib by Eric Jones
- Anatomy of Matplotlib by Benjamin Root
- Data Visualization Basics with Python (O'Reilly) by Randal S. Olson
- Matplotlib Introduction by codebasics
- Matplotlib by Derek Banas

## 8.3 Tutorials

- Matplotlib tutorial by Nicolas P. Rougier
- Anatomy of Matplotlib - IPython Notebooks by Benjamin Root
- Beyond the Basics: Data Visualization in Python by Stefanie Molin

## 8.4 Galleries

- Past winners for JDH plotting contest by Nelle Varoquaux
- The Python Graph Gallery by Yan Holtz

Third-party packages,

provide custom, domain specific, and experimental features, including styles, colors, more plot types and backends, and alternative interfaces.

# Part IV

# What's new

Learn about new features and API changes.

CHAPTER

NINE

# RELEASE NOTES

## 9.1 Version 3.9

### 9.1.1 What's new in Matplotlib 3.9.0 (May 15, 2024)

For a list of all of the issues and pull requests since the last revision, see the *GitHub statistics for 3.9.0 (May 15, 2024)*.

**Table of Contents**


- *What's new in Matplotlib 3.9.0 (May 15, 2024)*
  - *Plotting and Annotation improvements*
    - *`Axes.inset_axes` is no longer experimental*
    - *Legend support for Boxplot*
    - *Percent sign in pie labels auto-escaped with `usetex=True`*
    - *`hatch` parameter for stackplot*
    - *Add option to plot only one half of violin plot*
    - *`axhline` and `axhspan` on polar axes*
    - *Subplot titles can now be automatically aligned*
    - *`axisartist` can now be used together with standard `Formatters`*
    - *Toggle minorticks on Axis*
    - *`StrMethodFormatter` now respects `axes.unicode_minus`*
  - *Figure, Axes, and Legend Layout*
    - *Subfigures now have controllable zorders*
    - *Getters for xmargin, ymargin and zmargin*
  - *Mathtext improvements*
    - *`mathtext` documentation improvements*

        * *mathtext spacing corrections*
    - *Widget Improvements*
        * *Check and Radio Button widgets support clearing*
    - *3D plotting improvements*
        * *Setting 3D axis limits now set the limits exactly*
    - *Other improvements*
        * *BackendRegistry*
        * *Add widths, heights and angles setter to EllipseCollection*
        * *image.interpolation_stage rcParam*
        * *Arrow patch position is now modifiable*
        * *NonUniformImage now has mouseover support*


## Plotting and Annotation improvements

### `Axes.inset_axes` is no longer experimental

`Axes.inset_axes` is considered stable for use.

### Legend support for Boxplot

Boxplots now support a *label* parameter to create legend entries. Legend labels can be passed as a list of strings to label multiple boxes in a single `Axes.boxplot` call:

```
np.random.seed(19680801)
fruit_weights = [
    np.random.normal(130, 10, size=100),
    np.random.normal(125, 20, size=100),
    np.random.normal(120, 30, size=100),
]
labels = ['peaches', 'oranges', 'tomatoes']
colors = ['peachpuff', 'orange', 'tomato']

fig, ax = plt.subplots()
ax.set_ylabel('fruit weight (g)')

bplot = ax.boxplot(fruit_weights,
                   patch_artist=True,  # fill with color
                   label=labels)

# fill with colors
for patch, color in zip(bplot['boxes'], colors):
```

(continues on next page)

(continued from previous page)

```
    patch.set_facecolor(color)

ax.set_xticks([])
ax.legend()
```

Or as a single string to each individual *Axes.boxplot*:

```
fig, ax = plt.subplots()

data_A = np.random.random((100, 3))
data_B = np.random.random((100, 3)) + 0.2
pos = np.arange(3)

ax.boxplot(data_A, positions=pos - 0.2, patch_artist=True, label='Box A',
           boxprops={'facecolor': 'steelblue'})
ax.boxplot(data_B, positions=pos + 0.2, patch_artist=True, label='Box B',
           boxprops={'facecolor': 'lightblue'})

ax.legend()
```

### Percent sign in pie labels auto-escaped with `usetex=True`

It is common, with *Axes.pie*, to specify labels that include a percent sign (`%`), which denotes a comment for LaTeX. When enabling LaTeX with *rcParams["text.usetex"]* (default: `False`) or passing `textprops={"usetex": True}`, this used to cause the percent sign to disappear.

Now, the percent sign is automatically escaped (by adding a preceding backslash) so that it appears regardless of the `usetex` setting. If you have pre-escaped the percent sign, this will be detected, and remain as is.

### `hatch` parameter for stackplot

The *stackplot* *hatch* parameter now accepts a list of strings describing hatching styles that will be applied sequentially to the layers in the stack:

```
fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(10,5))

cols = 10
rows = 4
data = (
np.reshape(np.arange(0, cols, 1), (1, -1)) ** 2
+ np.reshape(np.arange(0, rows), (-1, 1))
+ np.random.random((rows, cols))*5
)
x = range(data.shape[1])
ax1.stackplot(x, data, hatch="x")
```

(continues on next page)

(continued from previous page)

```
ax2.stackplot(x, data, hatch=["//","\\","x","o"])

ax1.set_title("hatch='x'")
ax2.set_title("hatch=['//','\\\\','x','o']")

plt.show()
```

### Add option to plot only one half of violin plot

Setting the parameter *side* to 'low' or 'high' allows to only plot one half of the *Axes.violinplot*.

```
# Fake data with reproducible random state.
np.random.seed(19680801)
data = np.random.normal(0, 8, size=100)

fig, ax = plt.subplots()

ax.violinplot(data, [0], showmeans=True, showextrema=True)
ax.violinplot(data, [1], showmeans=True, showextrema=True, side='low')
ax.violinplot(data, [2], showmeans=True, showextrema=True, side='high')

ax.set_title('Violin Sides Example')
ax.set_xticks([0, 1, 2], ['Default', 'side="low"', 'side="high"'])
ax.set_yticklabels([])
```

### `axhline` and `axhspan` on polar axes

... now draw circles and circular arcs (*axhline*) or annuli and wedges (*axhspan*).

```
fig = plt.figure()
ax = fig.add_subplot(projection="polar")
ax.set_rlim(0, 1.2)

ax.axhline(1, c="C0", alpha=.5)
ax.axhspan(.8, .9, fc="C1", alpha=.5)
ax.axhspan(.6, .7, .8, .9, fc="C2", alpha=.5)
```

### Subplot titles can now be automatically aligned

Subplot axes titles can be misaligned vertically if tick labels or xlabels are placed at the top of one subplot. The new *align_titles* method on the *Figure* class will now align the titles vertically.

```
fig, axs = plt.subplots(1, 2, layout='constrained')

axs[0].plot(np.arange(0, 1e6, 1000))
axs[0].set_title('Title 0')
axs[0].set_xlabel('XLabel 0')

axs[1].plot(np.arange(1, 0, -0.1) * 2000, np.arange(1, 0, -0.1))
axs[1].set_title('Title 1')
axs[1].set_xlabel('XLabel 1')
axs[1].xaxis.tick_top()
axs[1].tick_params(axis='x', rotation=55)
```

```
fig, axs = plt.subplots(1, 2, layout='constrained')

axs[0].plot(np.arange(0, 1e6, 1000))
axs[0].set_title('Title 0')
axs[0].set_xlabel('XLabel 0')

axs[1].plot(np.arange(1, 0, -0.1) * 2000, np.arange(1, 0, -0.1))
axs[1].set_title('Title 1')
axs[1].set_xlabel('XLabel 1')
axs[1].xaxis.tick_top()
axs[1].tick_params(axis='x', rotation=55)

fig.align_labels()
fig.align_titles()
```

### `axisartist` can now be used together with standard `Formatters`

... instead of being limited to axisartist-specific ones.

### Toggle minorticks on Axis

Minor ticks on an *Axis* can be displayed or removed using *minorticks_on* and *minorticks_off*; e.g., `ax.xaxis.minorticks_on()`. See also *minorticks_on*.

#### `StrMethodFormatter` now respects `axes.unicode_minus`

When formatting negative values, *StrMethodFormatter* will now use unicode minus signs if *rcParams["axes.unicode_minus"]* (default: `True`) is set.

```
>>> from matplotlib.ticker import StrMethodFormatter
>>> with plt.rc_context({'axes.unicode_minus': False}):
...     formatter = StrMethodFormatter('{x}')
...     print(formatter.format_data(-10))
-10
```

```
>>> with plt.rc_context({'axes.unicode_minus': True}):
...     formatter = StrMethodFormatter('{x}')
...     print(formatter.format_data(-10))
-10
```

### Figure, Axes, and Legend Layout

#### Subfigures now have controllable zorders

Previously, setting the zorder of a subfigure had no effect, and those were plotted on top of any figure-level artists (i.e for example on top of fig-level legends). Now, subfigures behave like any other artists, and their zorder can be controlled, with default a zorder of 0.

```
x = np.linspace(1, 10, 10)
y1, y2 = x, -x
fig = plt.figure(constrained_layout=True)
subfigs = fig.subfigures(nrows=1, ncols=2)
for subfig in subfigs:
    axarr = subfig.subplots(2, 1)
    for ax in axarr.flatten():
        (l1,) = ax.plot(x, y1, label="line1")
        (l2,) = ax.plot(x, y2, label="line2")
subfigs[0].set_zorder(6)
l = fig.legend(handles=[l1, l2], loc="upper center", ncol=2)
```

#### Getters for xmargin, ymargin and zmargin

*Axes.get_xmargin*, *Axes.get_ymargin* and *Axes3D.get_zmargin* methods have been added to return the margin values set by *Axes.set_xmargin*, *Axes.set_ymargin* and *Axes3D.set_zmargin*, respectively.

### Mathtext improvements

#### `mathtext` documentation improvements

The documentation is updated to take information directly from the parser. This means that (almost) all supported symbols, operators, etc. are shown at *Writing mathematical expressions*.

#### `mathtext` spacing corrections

As consequence of the updated documentation, the spacing on a number of relational and operator symbols were correctly classified and therefore will be spaced properly.

### Widget Improvements

#### Check and Radio Button widgets support clearing

The *CheckButtons* and *RadioButtons* widgets now support clearing their state by calling their `.clear` method. Note that it is not possible to have no selected radio buttons, so the selected option at construction time is selected.

### 3D plotting improvements

#### Setting 3D axis limits now set the limits exactly

Previously, setting the limits of a 3D axis would always add a small margin to the limits. Limits are now set exactly by default. The newly introduced rcparam `axes3d.automargin` can be used to revert to the old behavior where margin is automatically added.

```
fig, axs = plt.subplots(1, 2, subplot_kw={'projection': '3d'})

plt.rcParams['axes3d.automargin'] = True
axs[0].set(xlim=(0, 1), ylim=(0, 1), zlim=(0, 1), title='Old Behavior')

plt.rcParams['axes3d.automargin'] = False  # the default in 3.9.0
axs[1].set(xlim=(0, 1), ylim=(0, 1), zlim=(0, 1), title='New Behavior')
```

### Other improvements

### BackendRegistry

New *BackendRegistry* class is the single source of truth for available backends. The singleton instance is `matplotlib.backends.backend_registry`. It is used internally by Matplotlib, and also IPython (and therefore Jupyter) starting with IPython 8.24.0.

There are three sources of backends: built-in (source code is within the Matplotlib repository), explicit `module://some.backend` syntax (backend is obtained by loading the module), or via an entry point (self-registering backend in an external package).

To obtain a list of all registered backends use:

```
>>> from matplotlib.backends import backend_registry
>>> backend_registry.list_all()
```

### Add `widths`, `heights` and `angles` setter to `EllipseCollection`

The `widths`, `heights` and `angles` values of the *EllipseCollection* can now be changed after the collection has been created.

```
from matplotlib.collections import EllipseCollection

rng = np.random.default_rng(0)

widths = (2, )
heights = (3, )
angles = (45, )
offsets = rng.random((10, 2)) * 10

fig, ax = plt.subplots()

ec = EllipseCollection(
    widths=widths,
    heights=heights,
    angles=angles,
    offsets=offsets,
    units='x',
    offset_transform=ax.transData,
    )

ax.add_collection(ec)
ax.set_xlim(-2, 12)
ax.set_ylim(-2, 12)

new_widths = rng.random((10, 2)) * 2
new_heights = rng.random((10, 2)) * 3
new_angles = rng.random((10, 2)) * 180
```

(continues on next page)

(continued from previous page)

```
ec.set(widths=new_widths, heights=new_heights, angles=new_angles)
```

### `image.interpolation_stage` rcParam

This new rcParam controls whether image interpolation occurs in "data" space or in "rgba" space.

### Arrow patch position is now modifiable

A setter method has been added that allows updating the position of the *patches.Arrow* object without requiring a full re-draw.

```
from matplotlib import animation
from matplotlib.patches import Arrow

fig, ax = plt.subplots()
ax.set_xlim(0, 10)
ax.set_ylim(0, 10)

a = Arrow(2, 0, 0, 10)
ax.add_patch(a)


# code for modifying the arrow
def update(i):
    a.set_data(x=.5, dx=i, dy=6, width=2)


ani = animation.FuncAnimation(fig, update, frames=15, interval=90, blit=False)

plt.show()
```

### NonUniformImage now has mouseover support

When mousing over a *NonUniformImage*, the data values are now displayed.

### 9.1.2 API Changes for 3.9.0


- *Behaviour Changes*
- *Deprecations*
- *Removals*
- *Development changes*


#### Behaviour Changes

##### plot() shorthand format interprets "Cn" (n>9) as a color-cycle color

Previously, `plot(..., "-C11")` would be interpreted as requesting a plot using linestyle "-", color "C1" (color #1 of the color cycle), and marker "1" ("tri-down"). It is now interpreted as requesting linestyle "-" and color "C11" (color #11 of the color cycle).

It is recommended to pass ambiguous markers (such as "1") explicitly using the *marker* keyword argument. If the shorthand form is desired, such markers can also be unambiguously set by putting them *before* the color string.

##### Legend labels for `plot`

Previously if a sequence was passed to the *label* parameter of *plot* when plotting a single dataset, the sequence was automatically cast to string for the legend label. Now, if the sequence has only one element, that element will be the legend label. To keep the old behavior, cast the sequence to string before passing.

##### Boxplots now ignore masked data points

*boxplot* and *boxplot_stats* now ignore any masked points in the input data.

##### `axhspan` and `axvspan` now return `Rectangle`s, not `Polygon`s

This change allows using *axhspan* to draw an annulus on polar axes.

This change also affects other elements built via *axhspan* and *axvspan*, such as `Slider.poly`.

### Improved handling of pan/zoom events of overlapping Axes

The forwarding of pan/zoom events is now determined by the visibility of the background-patch (e.g. `ax.patch.get_visible()`) and by the `zorder` of the axes.

- Axes with a visible patch capture the event and do not pass it on to axes below. Only the Axes with the highest `zorder` that contains the event is triggered (if there are multiple Axes with the same `zorder`, the last added Axes counts)
- Axes with an invisible patch are also invisible to events and they are passed on to the axes below.

To override the default behavior and explicitly set whether an Axes should forward navigation events, use *Axes.set_forward_navigation_events*.

### `loc='best'` for `legend` now considers `Text` and `PolyCollections`

The location selection `legend` now considers the existence of `Text` and `PolyCollections` in the `badness` calculation.

Note: The `best` option can already be quite slow for plots with large amounts of data. For `PolyCollections`, it only considers the `Path` of `PolyCollections` and not the enclosed area when checking for overlap to reduce additional latency. However, it can still be quite slow when there are large amounts of `PolyCollections` in the plot to check for.

### Exception when not passing a Bbox to BboxTransform*-classes

The exception when not passing a Bbox to BboxTransform*-classes that expect one, e.g., *BboxTransform* has changed from `ValueError` to `TypeError`.

### *loc* parameter of `Cell` no longer accepts `None`

The default value of the *loc* parameter has been changed from `None` to `right`, which already was the default location. The behavior of *Cell* didn't change when called without an explicit *loc* parameter.

### `ContourLabeler.add_label` now respects *use_clabeltext*

... and sets *Text.set_transform_rotates_text* accordingly.

#### Line2D

When creating a Line2D or using *Line2D.set_xdata* and *Line2D.set_ydata*, passing x/y data as non sequence is now an error.

#### ScalarMappables auto-scale their norm when an array is set

Collections previously deferred auto-scaling of the norm until draw time. This has been changed to scale the norm whenever the first array is set to align with the docstring and reduce unexpected behavior when accessing the norm before drawing.

#### SubplotParams moved from matplotlib.figure to matplotlib.gridspec

It is still importable from `matplotlib.figure`, so does not require any changes to existing code.

#### PowerNorm no longer clips values below vmin

When `clip=False` is set (the default) on *PowerNorm*, values below `vmin` are now linearly normalised. Previously they were clipped to zero. This fixes issues with the display of colorbars associated with a power norm.

### Image path semantics of toolmanager-based tools

Previously, MEP22 ("toolmanager-based") Tools would try to load their icon (`tool.image`) relative to the current working directory, or, as a fallback, from Matplotlib's own image directory. Because both approaches are problematic for third-party tools (the end-user may change the current working directory at any time, and third-parties cannot add new icons in Matplotlib's image directory), this behavior is deprecated; instead, `tool.image` is now interpreted relative to the directory containing the source file where the `Tool.image` class attribute is defined. (Defining `tool.image` as an absolute path also works and is compatible with both the old and the new semantics.)

## Deprecations

#### plot_date

Use of *plot_date* has been discouraged since Matplotlib 3.5 and the function is now formally deprecated.

- `datetime`-like data should directly be plotted using *plot*.
- If you need to plot plain numeric data as *Matplotlib date format* or need to set a timezone, call `ax.xaxis.axis_date` / `ax.yaxis.axis_date` before *plot*. See *Axis.axis_date*.

### Legend labels for `plot`

Previously if a sequence was passed to the *label* parameter of *plot* when plotting a single dataset, the sequence was automatically cast to string for the legend label. This behavior is now deprecated and in future will error if the sequence length is not one (consistent with multi-dataset behavior, where the number of elements must match the number of datasets). To keep the old behavior, cast the sequence to string before passing.

### `boxplot` tick labels

The parameter *labels* has been renamed to *tick_labels* for clarity and consistency with *bar*.

### Mixing positional and keyword arguments for `legend` handles and labels

This previously only raised a warning, but is now formally deprecated. If passing *handles* and *labels*, they must be passed either both positionally or both as keyword.

### Applying theta transforms in `PolarTransform`

Applying theta transforms in *PolarTransform* and *InvertedPolarTransform* is deprecated, and will be removed in a future version of Matplotlib. This is currently the default behaviour when these transforms are used externally, but only takes affect when:

- An axis is associated with the transform.
- The axis has a non-zero theta offset or has theta values increasing in a clockwise direction.

To silence this warning and adopt future behaviour, set `apply_theta_transforms=False`. If you need to retain the behaviour where theta values are transformed, chain the `PolarTransform` with a *Affine2D* transform that performs the theta shift and/or sign shift.

### *interval* parameter of `TimerBase.start`

Setting the timer *interval* while starting it is deprecated. The interval can be specified instead in the timer constructor, or by setting the `timer.interval` attribute.

### *nth_coord* parameter to axisartist helpers for fixed axis

Helper APIs in *axisartist* for generating a "fixed" axis on rectilinear axes (*FixedAxisArtistHelperRectilinear*) no longer take a *nth_coord* parameter, as that parameter is entirely inferred from the (required) *loc* parameter and having inconsistent *nth_coord* and *loc* is an error.

For curvilinear axes, the *nth_coord* parameter remains supported (it affects the *ticks*, not the axis position itself), but that parameter will become keyword-only, for consistency with the rectilinear case.

### `rcsetup.interactive_bk`, `rcsetup.non_interactive_bk` and `rcsetup.all_backends`

... are deprecated and replaced by `matplotlib.backends.backend_registry.list_builtin` with the following arguments

- `matplotlib.backends.BackendFilter.INTERACTIVE`
- `matplotlib.backends.BackendFilter.NON_INTERACTIVE`
- `None`

respectively.

### Miscellaneous deprecations

- `backend_ps.get_bbox_header` is considered an internal helper
- `BboxTransformToMaxOnly`; if you rely on this, please make a copy of the code
- `ContourLabeler.add_label_clabeltext`
- `TransformNode.is_bbox`; instead check the object using `isinstance(..., BboxBase)`
- `GridHelperCurveLinear.get_tick_iterator`

### Removals

### Top-level cmap registration and access functions in `mpl.cm`

As part of the multi-step refactoring of colormap registration, the following functions have been removed:

- `matplotlib.cm.get_cmap`; use `matplotlib.colormaps[name]` instead if you have a `str`.

  Use *matplotlib.cm.ColormapRegistry.get_cmap* if you have a `str`, `None` or a *matplotlib.colors.Colormap* object that you want to convert to a *Colormap* object.
- `matplotlib.cm.register_cmap`; use *matplotlib.colormaps.register* instead.
- `matplotlib.cm.unregister_cmap`; use *matplotlib.colormaps.unregister* instead.

- `matplotlib.pyplot.register_cmap`; use *matplotlib.colormaps.register* instead.

The *matplotlib.pyplot.get_cmap* function will stay available for backward compatibility.

### Contour labels

`contour.ClabelText` and `ContourLabeler.set_label_props` are removed. Use `Text(..., transform_rotates_text=True)` as a replacement for `contour.ClabelText(...)` and `text.set(text=text, color=color, fontproperties=labeler.labelFontProps, clip_box=labeler.axes.bbox)` as a replacement for the `ContourLabeler.set_label_props(label, text, color)`.

The `labelFontProps`, `labelFontSizeList`, and `labelTextsList` attributes of *ContourLabeler* have been removed. Use the `labelTexts` attribute and the font properties of the corresponding text objects instead.

### `num2julian`, `julian2num` and `JULIAN_OFFSET`

... of the *dates* module are removed without replacements. These were undocumented and not exported.

Julian dates in Matplotlib were calculated from a Julian date epoch: `jdate = (date - np.datetime64(EPOCH)) / np.timedelta64(1, 'D')`. Conversely, a Julian date was converted to datetime as `date = np.timedelta64(int(jdate * 24 * 3600), 's') + np.datetime64(EPOCH)`. Matplotlib was using `EPOCH='-4713-11-24T12:00'` so that 2000-01-01 at 12:00 is 2_451_545.0 (see https://en.wikipedia.org/wiki/Julian_day).

### `offsetbox` methods

`offsetbox.bbox_artist` is removed. This was just a wrapper to call *patches.bbox_artist* if a flag is set in the file, so use that directly if you need the behavior.

`OffsetBox.get_extent_offsets` and `OffsetBox.get_extent` are removed; these methods are also removed on all subclasses of *OffsetBox*. To get the offsetbox extents, instead of `get_extent`, use *OffsetBox.get_bbox*, which directly returns a *Bbox* instance. To also get the child offsets, instead of `get_extent_offsets`, separately call *get_offset* on each children after triggering a draw.

### `parse_fontconfig_pattern` raises on unknown constant names

Previously, in a fontconfig pattern like `DejaVu Sans:foo`, the unknown `foo` constant name would be silently ignored. This now raises an error.

### `tri` submodules

The `matplotlib.tri.*` submodules are removed. All functionality is available in `matplotlib.tri` directly and should be imported from there.

### Widget API

- `CheckButtons.rectangles` and `CheckButtons.lines` are removed; *CheckButtons* now draws itself using *scatter*.
- `RadioButtons.circles` is removed; *RadioButtons* now draws itself using *scatter*.
- `MultiCursor.needclear` is removed with no replacement.
- The unused parameter *x* to `TextBox.begin_typing` was a required argument, and is now removed.

### Most arguments to widgets have been made keyword-only

Passing all but the very few first arguments positionally in the constructors of Widgets is now keyword-only. In general, all optional arguments are keyword-only.

### `Axes3D` API

- `Axes3D.unit_cube`, `Axes3D.tunit_cube`, and `Axes3D.tunit_edges` are removed without replacement.
- `axes3d.vvec`, `axes3d.eye`, `axes3d.sx`, and `axes3d.sy` are removed without replacement.

### Inconsistent *nth_coord* and *loc* passed to `_FixedAxisArtistHelperBase`

The value of the *nth_coord* parameter of `_FixedAxisArtistHelperBase` and its subclasses is now inferred from the value of *loc*; passing inconsistent values (e.g., requesting a "top y axis" or a "left x axis") has no more effect.

### Passing undefined *label_mode* to `Grid`

... is no longer allowed. This includes *mpl_toolkits.axes_grid1.axes_grid.Grid*, *mpl_toolkits.axes_grid1.axes_grid.AxesGrid*, and *mpl_toolkits.axes_grid1.axes_grid.ImageGrid* as well as the corresponding classes imported from *mpl_toolkits.axisartist.axes_grid*.

Pass `label_mode='keep'` instead to get the previous behavior of not modifying labels.

#### draw_gouraud_triangle

... is removed. Use *draw_gouraud_triangles* instead.

A `draw_gouraud_triangle` call in a custom *Artist* can readily be replaced as:

```
self.draw_gouraud_triangles(gc, points.reshape((1, 3, 2)),
                            colors.reshape((1, 3, 4)), trans)
```

A *draw_gouraud_triangles* method can be implemented from an existing `draw_gouraud_triangle` method as:

```
transform = transform.frozen()
for tri, col in zip(triangles_array, colors_array):
    self.draw_gouraud_triangle(gc, tri, col, transform)
```

### Miscellaneous removals

The following items have previously been replaced, and are now removed:

- *ticklabels* parameter of `matplotlib.axis.Axis.set_ticklabels` has been renamed to *labels*.
- `Barbs.barbs_doc` and `Quiver.quiver_doc` are removed. These are the doc-strings and should not be accessible as a named class member, but as normal doc-strings would.
- `collections.PolyCollection.span_where` and `collections.BrokenBarHCollection`; use `fill_between` instead.
- `Legend.legendHandles` was undocumented and has been renamed to `legend_handles`.

The following items have been removed without replacements:

- The attributes `repeat` of *TimedAnimation* and subclasses and `save_count` of *FuncAnimation* are considered private and removed.
- `matplotlib.backend.backend_agg.BufferRegion.to_string`
- `matplotlib.backend.backend_agg.BufferRegion.to_string_argb`
- `matplotlib.backends.backend_ps.PsBackendHelper`
- `matplotlib.backends.backend_webagg.ServerThread`
- *raw* parameter of *GridSpecBase.get_grid_positions*
- `matplotlib.patches.ConnectionStyle._Base.SimpleEvent`
- `passthru_pt` attribute of `mpl_toolkits.axisartist.AxisArtistHelper`

### Development changes

### Build system ported to Meson

The build system of Matplotlib has been ported from setuptools to meson-python and Meson. Consequently, there have been a few changes for development and packaging purposes.

1. Installation by `pip` of packages with `pyproject.toml` use build isolation by default, which interferes with editable installation. Thus for developers using editable installs, it is now necessary to pass the `--no-build-isolation` flag to `pip install`. This means that all build-time requirements must be available in the environment for an editable install.

2. Build configuration has moved from a custom `mplsetup.cfg` (also configurable via `MPLSETUP` environment variable) to Meson options. These may be specified using meson-python's build config settings for `setup-args`. See `meson_options.txt` for all options. For example, a `mplsetup.cfg` containing the following:

   ```
   [rc_options]
   backend=Agg

   [libs]
   system_qhull = True
   ```

   may be replaced by passing the following arguments to `pip`:

   ```
   --config-settings=setup-args="-DrcParams-backend=Agg"
   --config-settings=setup-args="-Dsystem-qhull=true"
   ```

   Note that you must use `pip` >= 23.1 in order to pass more than one setting.

3. Relatedly, Meson's builtin options are now used instead of custom options, e.g., the LTO option is now `b_lto`.

4. On Windows, Meson activates a Visual Studio environment automatically. However, it will not do so if another compiler is available. See Meson's documentation if you wish to change the priority of chosen compilers.

5. Installation of test data was previously controlled by `mplsetup.cfg`, but has now been moved to Meson's install tags. To install test data, add the `tests` tag to the requested install (be sure to include the existing tags as below):

   ```
   --config-settings=install-args="--tags=data,python-runtime,runtime,tests"
   ```

6. Checking typing stubs with `stubtest` does not work easily with editable install. For the time being, we suggest using a normal (non-editable) install if you wish to run `stubtest`.

### Increase to minimum supported versions of dependencies

For Matplotlib 3.9, the *minimum supported versions* are being bumped:

| Dependency | min in mpl3.8 | min in mpl3.9 |
|---|---|---|
| NumPy | 1.21.0 | 1.23.0 |
| setuptools | 42 | 64 |

This is consistent with our *Dependency version policy* and SPEC 0.

To comply with requirements of `setuptools_scm`, the minimum version of `setuptools` has been increased from 42 to 64.

### Extensions require C++17

Matplotlib now requires a compiler that supports C++17 in order to build its extensions. According to SciPy's analysis, this should be available on all supported platforms.

### Windows on ARM64 support

Windows on ARM64 now bundles FreeType 2.6.1 instead of 2.11.1 when building from source. This may cause small changes to text rendering, but should become consistent with all other platforms.

## 9.2 Version 3.8

### 9.2.1 What's new in Matplotlib 3.8.0 (Sept 13, 2023)

For a list of all of the issues and pull requests since the last revision, see the *GitHub statistics for 3.9.0 (May 15, 2024)*.

**Table of Contents**


- *What's new in Matplotlib 3.8.0 (Sept 13, 2023)*
  - *Type Hints*
  - *Plotting and Annotation improvements*
    - *Support customizing antialiasing for text and annotation*
    - *rcParams for `AutoMinorLocator` divisions*
    - *Axline setters and getters*
    - *Clipping for contour plots*

        * *`Axes.ecdf`*
        * *`Figure.get_suptitle()`, `Figure.get_supxlabel()`, `Figure.get_supylabel()`*
        * *`Ellipse.get_vertices()`, `Ellipse.get_co_vertices()`*
        * *Remove inner ticks in `label_outer()`*
        * *Configurable legend shadows*
        * *`offset` parameter for MultipleLocator*
        * *Add a new valid color format `(matplotlib_color, alpha)`*
        * *The pie chart shadow can be controlled*
        * *`PolyQuadMesh` is a new class for drawing quadrilateral meshes*
        * *Shadow shade can be controlled*
        * *`SpinesProxy` now supports calling the `set()` method*
        * *Allow setting the tick label fonts with a keyword argument*
    - *Figure, Axes, and Legend Layout*
        * *pad_inches="layout" for savefig*
        * *Add a public method to modify the location of `Legend`*
        * *`rcParams['legend.loc']` now accepts float-tuple inputs*
    - *Mathtext improvements*
        * *Boldsymbol mathtext command `\boldsymbol`*
        * *`mathtext` has more sizable delimiters*
        * *`mathtext` documentation improvements*
        * *`mathtext` now supports `\substack`*
        * *`mathtext` now supports `\middle` delimiter*
        * *`mathtext` operators*
        * *`mathtext` spacing corrections*
        * *`mathtext` now supports `\text`*
        * *Bold-italic mathtext command `\mathbfit`*
    - *3D plotting improvements*
        * *Specify ticks and axis label positions for 3D plots*
        * *3D hover coordinates*
        * *3D plots can share view angles*
    - *Other improvements*

* *macosx: New figures can be opened in either windows or tabs*
* *`matplotlib.mpl_toolkits` is now an implicit namespace package*
* *Plot Directive now can make responsive images with "srcset"*

### Type Hints

Matplotlib now provides first-party PEP484 style type hints files for most public APIs.

While still considered provisional and subject to change (and sometimes we are not quite able to fully specify what we would like to), they should provide a reasonable basis to type check many common usage patterns, as well as integrating with many editors/IDEs.

### Plotting and Annotation improvements

#### Support customizing antialiasing for text and annotation

`matplotlib.pyplot.annotate()` and `matplotlib.pyplot.text()` now support parameter *antialiased*. When *antialiased* is set to `True`, antialiasing will be applied to the text. When *antialiased* is set to `False`, antialiasing will not be applied to the text. When *antialiased* is not specified, antialiasing will be set by *`rcParams["text.antialiased"]`* (default: `True`) at the creation time of `Text` and `Annotation` object. Examples:

```
mpl.text.Text(.5, .5, "foo\nbar", antialiased=True)
plt.text(0.5, 0.5, '6 inches x 2 inches', antialiased=True)
ax.annotate('local max', xy=(2, 1), xytext=(3, 1.5), antialiased=False)
```

If the text contains math expression, *antialiased* applies to the whole text. Examples:

```
# no part will be antialiased for the text below
plt.text(0.5, 0.25, r"$I'm \sqrt{x}$", antialiased=False)
```

Also note that antialiasing for tick labels will be set with *`rcParams["text.antialiased"]`* (default: `True`) when they are created (usually when a `Figure` is created) and cannot be changed afterwards.

Furthermore, with this new feature, you may want to make sure that you are creating and saving/showing the figure under the same context:

```
# previously this was a no-op, now it is what works
with rccontext(text.antialiased=False):
    fig, ax = plt.subplots()
    ax.annotate('local max', xy=(2, 1), xytext=(3, 1.5))
    fig.savefig('/tmp/test.png')


# previously this had an effect, now this is a no-op
fig, ax = plt.subplots()
```

(continues on next page)

(continued from previous page)

```
ax.annotate('local max', xy=(2, 1), xytext=(3, 1.5))
with rccontext(text.antialiased=False):
    fig.savefig('/tmp/test.png')
```

### rcParams for `AutoMinorLocator` divisions

The rcParams *rcParams["xtick.minor.ndivs"]* (default: `'auto'`) and *rcParams["ytick.minor.ndivs"]* (default: `'auto'`) have been added to enable setting the default number of divisions; if set to `auto`, the number of divisions will be chosen by the distance between major ticks.

### Axline setters and getters

The returned object from *axes.Axes.axline* now supports getter and setter methods for its *xy1*, *xy2* and *slope* attributes:

```
line1.get_xy1()
line1.get_slope()
line2.get_xy2()
```

```
line1.set_xy1(.2, .3)
line1.set_slope(2.4)
line2.set_xy2(.1, .6)
```

### Clipping for contour plots

*contour* and *contourf* now accept the *clip_path* parameter.

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches

x = y = np.arange(-3.0, 3.01, 0.025)
X, Y = np.meshgrid(x, y)
Z1 = np.exp(-X**2 - Y**2)
Z2 = np.exp(-(X - 1)**2 - (Y - 1)**2)
Z = (Z1 - Z2) * 2

fig, ax = plt.subplots()
patch = mpatches.RegularPolygon((0, 0), 5, radius=2,
                                transform=ax.transData)
ax.contourf(X, Y, Z, clip_path=patch)

plt.show()
```

### `Axes.ecdf`

A new Axes method, *ecdf*, allows plotting empirical cumulative distribution functions without any binning.

```
import matplotlib.pyplot as plt
import numpy as np

fig, ax = plt.subplots()
ax.ecdf(np.random.randn(100))
```

### `Figure.get_suptitle()`, `Figure.get_supxlabel()`, `Figure.get_supylabel()`

These methods return the strings set by `Figure.suptitle()`, `Figure.supxlabel()` and `Figure.supylabel()` respectively.

### `Ellipse.get_vertices()`, `Ellipse.get_co_vertices()`

These methods return the coordinates of ellipse vertices of major and minor axis. Additionally, an example gallery demo is added which shows how to add an arrow to an ellipse showing a clockwise or counter-clockwise rotation of the ellipse. To place the arrow exactly on the ellipse, the coordinates of the vertices are used.

### Remove inner ticks in `label_outer()`

Up to now, `label_outer()` has only removed the ticklabels. The ticks lines were left visible. This is now configurable through a new parameter `label_outer(remove_inner_ticks=True)`.

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2 * np.pi, 100)

fig, axs = plt.subplots(2, 2, sharex=True, sharey=True,
                        gridspec_kw=dict(hspace=0, wspace=0))

axs[0, 0].plot(x, np.sin(x))
axs[0, 1].plot(x, np.cos(x))
axs[1, 0].plot(x, -np.cos(x))
axs[1, 1].plot(x, -np.sin(x))

for ax in axs.flat:
    ax.grid(color='0.9')
    ax.label_outer(remove_inner_ticks=True)
```

### Configurable legend shadows

The *shadow* parameter of legends now accepts dicts in addition to booleans. Dictionaries can contain any keywords for *patches.Patch*. For example, this allows one to set the color and/or the transparency of a legend shadow:

```
ax.legend(loc='center left', shadow={'color': 'red', 'alpha': 0.5})
```

and to control the shadow location:

```
ax.legend(loc='center left', shadow={"ox":20, "oy":-20})
```

Configuration is currently not supported via *rcParams["legend.shadow"]* (default: `False`).

### `offset` parameter for MultipleLocator

An *offset* may now be specified to shift all the ticks by the given value.

```
import matplotlib.pyplot as plt
import matplotlib.ticker as mticker

_, ax = plt.subplots()
ax.plot(range(10))
locator = mticker.MultipleLocator(base=3, offset=0.3)
ax.xaxis.set_major_locator(locator)

plt.show()
```

### Add a new valid color format `(matplotlib_color, alpha)`

```
import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle

fig, ax = plt.subplots()

rectangle = Rectangle((.2, .2), .6, .6,
                      facecolor=('blue', 0.2),
                      edgecolor=('green', 0.5))
ax.add_patch(rectangle)
```

Users can define a color using the new color specification, *(matplotlib_color, alpha)*. Note that an explicit alpha keyword argument will override an alpha value from *(matplotlib_color, alpha)*.

### The pie chart shadow can be controlled

The *shadow* argument to *pie* can now be a dict, allowing more control of the *Shadow*-patch used.

### `PolyQuadMesh` is a new class for drawing quadrilateral meshes

*pcolor* previously returned a flattened *PolyCollection* with only the valid polygons (unmasked) contained within it. Now, we return a *PolyQuadMesh*, which is a mixin incorporating the usefulness of 2D array and mesh coordinates handling, but still inheriting the draw methods of *PolyCollection*, which enables more control over the rendering properties than a normal *QuadMesh* that is returned from *pcolormesh*. The new class subclasses *PolyCollection* and thus should still behave the same as before. This new class keeps track of the mask for the user and updates the Polygons that are sent to the renderer appropriately.

### Shadow shade can be controlled

The *Shadow* patch now has a *shade* argument to control the shadow darkness. If 1, the shadow is black, if 0, the shadow has the same color as the patch that is shadowed. The default value, which earlier was fixed, is 0.7.

### `SpinesProxy` now supports calling the `set()` method

One can now call e.g. `ax.spines[:].set(visible=False)`.

### Allow setting the tick label fonts with a keyword argument

`Axes.tick_params` now accepts a *labelfontfamily* keyword that changes the tick label font separately from the rest of the text objects:

```
Axis.tick_params(labelfontfamily='monospace')
```

## Figure, Axes, and Legend Layout

### pad_inches="layout" for savefig

When using constrained or compressed layout,

```
savefig(filename, bbox_inches="tight", pad_inches="layout")
```

will now use the padding sizes defined on the layout engine.

### Add a public method to modify the location of `Legend`

*Legend* locations now can be tweaked after they've been defined.

```
from matplotlib import pyplot as plt

fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)

x = list(range(-100, 101))
y = [i**2 for i in x]

ax.plot(x, y, label="f(x)")
ax.legend()
ax.get_legend().set_loc("right")
# Or
# ax.get_legend().set(loc="right")

plt.show()
```

### `rcParams['legend.loc']` now accepts float-tuple inputs

The *rcParams["legend.loc"]* (default: `'best'`) rcParams now accepts float-tuple inputs, same as the *loc* keyword argument to *Legend*. This allows users to set the location of the legend in a more flexible and consistent way.

### Mathtext improvements

Improvements are to Mathtext, Matplotlib's native TeX-like mathematics parser (see *Writing mathematical expressions*, not to be confused with Matplotlib using LaTeX directly: *Text rendering with LaTeX*).

### Boldsymbol mathtext command `\boldsymbol`

Supports using the `\boldsymbol{}` command in mathtext:

To change symbols to bold enclose the text in a font command as shown:

```
r'$\boldsymbol{a+2+\alpha}$'
```

$$\boldsymbol{a+2+\alpha}$$

### `mathtext` has more sizable delimiters

The `\lgroup` and `\rgroup` sizable delimiters have been added.

The following delimiter names have been supported earlier, but can now be sized with `\left` and `\right`:

- `\lbrace`, `\rbrace`, `\leftbrace`, and `\rightbrace`
- `\lbrack` and `\rbrack`
- `\leftparen` and `\rightparen`

There are really no obvious advantages in using these. Instead, they are are added for completeness.

### `mathtext` documentation improvements

The documentation is updated to take information directly from the parser. This means that (almost) all supported symbols, operators etc are shown at *Writing mathematical expressions*.

### `mathtext` now supports `\substack`

`\substack` can be used to create multi-line subscripts or superscripts within an equation.

To use it to enclose the math in a substack command as shown:

```
r'$\sum_{\substack{1\leq i\leq 3\\ 1\leq j\leq 5}}$'
```

$$\sum_{\substack{1\leq i\leq 3\\ 1\leq j\leq 5}} \tag{9.1}$$

### `mathtext` now supports `\middle` delimiter

The `\middle` delimiter has been added, and can now be used with the `\left` and `\right` delimiters:

To use the middle command enclose it in between the `\left` and `\right` delimiter command as shown:

```
r'$\left( \frac{a}{b} \middle| q \right)$'
```

$$\left(\frac{a}{b}\middle| q\right) \tag{9.2}$$

### `mathtext` operators

There has been a number of operators added and corrected when a Unicode font is used. In addition, correct spacing has been added to a number of the previous operators. Especially, the characters used for `\gnapprox`, `\lnapprox`, `\leftangle`, and `\rightangle` have been corrected.

### `mathtext` spacing corrections

As consequence of the updated documentation, the spacing on a number of relational and operator symbols were classified like that and therefore will be spaced properly.

### `mathtext` now supports `\text`

`\text` can be used to obtain upright text within an equation and to get a plain dash (-).

```
import matplotlib.pyplot as plt
plt.text(0.1, 0.5, r"$a = \sin(\phi) \text{ such that } \phi = \frac{x}{y}$")
plt.text(0.1, 0.3, r"$\text{dashes (-) are retained}$")
```

### Bold-italic mathtext command `\mathbfit`

Supports use of bold-italic font style in mathtext using the `\mathbfit{}` command:

To change font to bold and italic enclose the text in a font command as shown:

```
r'$\mathbfit{\eta \leq C(\delta(\eta))}$
```

$$\boldsymbol{\eta \leq C(\delta(\eta))}$$

### 3D plotting improvements

### Specify ticks and axis label positions for 3D plots

You can now specify the positions of ticks and axis labels for 3D plots.

```
import matplotlib.pyplot as plt

positions = ['lower', 'upper', 'default', 'both', 'none']
fig, axs = plt.subplots(2, 3, figsize=(12, 8),
                        subplot_kw={'projection': '3d'})
for ax, pos in zip(axs.flatten(), positions):
    for axis in ax.xaxis, ax.yaxis, ax.zaxis:
        axis.set_label_position(pos)
```

(continues on next page)

(continued from previous page)

```
        axis.set_ticks_position(pos)
    title = f'position="{pos}"'
    ax.set(xlabel='x', ylabel='y', zlabel='z', title=title)
axs[1, 2].axis('off')
```

### 3D hover coordinates

The x, y, z coordinates displayed in 3D plots were previously showing nonsensical values. This has been fixed to report the coordinate on the view pane directly beneath the mouse cursor. This is likely to be most useful when viewing 3D plots along a primary axis direction when using an orthographic projection, or when a 2D plot has been projected onto one of the 3D axis panes. Note that there is still no way to directly display the coordinates of plotted data points.

### 3D plots can share view angles

3D plots can now share the same view angles, so that when you rotate one plot the other plots also rotate. This can be done with the *shareview* keyword argument when adding an axes, or by using the *ax1.shareview(ax2)* method of existing 3D axes.

### Other improvements

#### macosx: New figures can be opened in either windows or tabs

There is a new *rcParams["macosx.window_mode`"]* rcParam to control how new figures are opened with the macosx backend. The default is **system** which uses the system settings, or one can specify either **tab** or **window** to explicitly choose the mode used to open new figures.

#### `matplotlib.mpl_toolkits` is now an implicit namespace package

Following the deprecation of `pkg_resources.declare_namespace` in `setuptools` 67.3.0, `matplotlib.mpl_toolkits` is now implemented as an implicit namespace, following PEP 420.

#### Plot Directive now can make responsive images with "srcset"

The plot sphinx directive (`matplotlib.sphinxext.plot_directive`, invoked in rst as `.. plot::`) can be configured to automatically make higher res figures and add these to the the built html docs. In `conf.py`:

```
extensions = [
...
    'matplotlib.sphinxext.plot_directive',
    'matplotlib.sphinxext.figmpl_directive',
...]

plot_srcset = ['2x']
```

will make png files with double the resolution for hiDPI displays. Resulting html files will have image entries like:

```
<img src="../_images/nestedpage-index-2.png" style="" srcset="../_images/
↪nestedpage-index-2.png, ../_images/nestedpage-index-2.2x.png 2.00x" alt=""␣
↪class="plot-directive "/>
```

### 9.2.2 API Changes for 3.8.1

#### Behaviour

#### Default behaviour of `hexbin` with *C* provided requires at least 1 point

The behaviour changed in 3.8.0 to be inclusive of *mincnt*. However, that resulted in errors or warnings with some reduction functions, so now the default is to require at least 1 point to call the reduction function. This effectively restores the default behaviour to match that of Matplotlib 3.7 and before.

#### Deprecations

#### Deprecations removed in `contour`

`contour.allsegs`, `contour.allkinds`, and `contour.find_nearest_contour` are no longer marked for deprecation.

#### Development

#### Minimum version of setuptools bumped to 64

To comply with requirements of `setuptools_scm`, the minimum version of `setuptools` has been increased from 42 to 64.

### 9.2.3 API Changes for 3.8.0


- *Behaviour Changes*
- *Deprecations*
- *Removals*
- *Development changes*


#### Behaviour Changes

##### Tk backend respects file format selection when saving figures

When saving a figure from a Tkinter GUI to a filename without an extension, the file format is now selected based on the value of the dropdown menu, rather than defaulting to PNG. When the filename contains an extension, or the OS automatically appends one, the behavior remains unchanged.

##### Placing of maximum and minimum minor ticks

Calculation of minor tick locations has been corrected to make the maximum and minimum minor ticks more consistent. In some cases this results in an extra minor tick on an Axis.

##### `hexbin` now defaults to `rcParams["patch.linewidth"]`

The default value of the *linewidths* argument of *Axes.hexbin* has been changed from `1.0` to *rcParams["patch.linewidth"]* (default: `1.0`). This improves the consistency with *QuadMesh* in *Axes.pcolormesh* and *Axes.hist2d*.

##### TwoSlopeNorm now auto-expands to always have two slopes

In the case where either `vmin` or `vmax` are not manually specified to *TwoSlopeNorm*, and where the data it is scaling is all less than or greater than the center point, the limits are now auto-expanded so there are two symmetrically sized slopes either side of the center point.

Previously `vmin` and `vmax` were clipped at the center point, which caused issues when displaying color bars.

This does not affect behaviour when `vmin` and `vmax` are manually specified by the user.

### Event objects emitted for `axes_leave_event`

`axes_leave_event` now emits a synthetic *`LocationEvent`*, instead of reusing the last event object associated with a `motion_notify_event`.

### Streamplot now draws streamlines as one piece if no width or no color variance

Since there is no need to draw streamlines piece by piece if there is no color change or width change, now streamplot will draw each streamline in one piece.

The behavior for varying width or varying color is not changed, same logic is used for these kinds of streamplots.

### `canvas` argument now required for `FigureFrameWx`

`FigureFrameWx` now requires a keyword-only `canvas` argument when it is constructed.

### `ContourSet` is now a single Collection

Prior to this release, *`ContourSet`* (the object returned by *`contour`*) was a custom object holding multiple *`Collection`*s (and not an *`Artist`*) -- one collection per level, each connected component of that level's contour being an entry in the corresponding collection.

*`ContourSet`* is now instead a plain *`Collection`* (and thus an *`Artist`*). The collection contains a single path per contour level; this path may be non-continuous in case there are multiple connected components.

Setting properties on the ContourSet can now usually be done using standard collection setters (`cset.set_linewidth(3)` to use the same linewidth everywhere or `cset.set_linewidth([1, 2, 3, ...])` to set different linewidths on each level) instead of having to go through the individual subcomponents (`cset.collections[0].set_linewidth(...)`). Note that during the transition period, it remains possible to access the (deprecated) `.collections` attribute; this causes the ContourSet to modify itself to use the old-style multi-Collection representation.

### `SubFigure` default facecolor is now transparent

Subfigures default facecolor changed to `"none"`. Previously the default was the value of `figure.facecolor`.

### Reject size related keyword arguments to MovieWriter *grab_frame* method

Although we pass *Figure.savefig* keyword arguments through the *AbstractMovieWriter.grab_frame* some of the arguments will result in invalid output if passed. To be successfully stitched into a movie, each frame must be exactly the same size, thus *bbox_inches* and *dpi* are excluded. Additionally, the movie writers are opinionated about the format of each frame, so the *format* argument is also excluded. Passing these arguments will now raise `TypeError` for all writers (it already did so for some arguments and some writers). The *bbox_inches* argument is already ignored (with a warning) if passed to *Animation.save*.

Additionally, if *rcParams["savefig.bbox"]* (default: `None`) is set to `'tight'`, *AbstractMovieWriter.grab_frame* will now error. Previously this rcParam would be temporarily overridden (with a warning) in *Animation.save*, it is now additionally overridden in *AbstractMovieWriter.saving*.

### Changes of API after deprecation

- *dviread.find_tex_file* now raises `FileNotFoundError` when the requested filename is not found.
- *Figure.colorbar* now raises if *cax* is not given and it is unable to determine from which Axes to steal space, i.e. if *ax* is also not given and *mappable* has not been added to an Axes.
- *pyplot.subplot* and *pyplot.subplot2grid* no longer auto-remove preexisting overlapping Axes; explicitly call `Axes.remove` as needed.

### Invalid types for Annotation xycoords now raise TypeError

Previously, a `RuntimeError` would be raised in some cases.

### Default antialiasing behavior changes for `Text` and `Annotation`

`matplotlib.pyplot.annotate()` and `matplotlib.pyplot.text()` now support parameter *antialiased* when initializing. Examples:

```
mpl.text.Text(.5, .5, "foo\nbar", antialiased=True)
plt.text(0.5, 0.5, '6 inches x 2 inches', antialiased=True)
ax.annotate('local max', xy=(2, 1), xytext=(3, 1.5), antialiased=False)
```

See "What's New" for more details on usage.

With this new feature, you may want to make sure that you are creating and saving/showing the figure under the same context:

```
# previously this was a no-op, now it is what works
with rccontext(text.antialiased=False):
    fig, ax = plt.subplots()
```

(continues on next page)

(continued from previous page)

```
    ax.annotate('local max', xy=(2, 1), xytext=(3, 1.5))
    fig.savefig('/tmp/test.png')

# previously this had an effect, now this is a no-op
fig, ax = plt.subplots()
ax.annotate('local max', xy=(2, 1), xytext=(3, 1.5))
with rccontext(text.antialiased=False):
    fig.savefig('/tmp/test.png')
```

Also note that antialiasing for tick labels will be set with *rcParams["text.antialiased"]* (default: `True`) when they are created (usually when a `Figure` is created) - This means antialiasing for them can no longer be changed by modifying *rcParams["text.antialiased"]* (default: `True`).

#### `ScalarMappable.to_rgba()` now respects the mask of RGB(A) arrays

Previously, the mask was ignored. Now the alpha channel is set to 0 if any component (R, G, B, or A) is masked.

#### `Text.get_rotation_mode` return value

Passing `None` as `rotation_mode` to *Text* (the default value) or passing it to *Text.set_rotation_mode* will make *Text.get_rotation_mode* return `"default"` instead of `None`. The behaviour otherwise is the same.

### PostScript paper type adds option to use figure size

The *rcParams["ps.papertype"]* rcParam can now be set to `'figure'`, which will use a paper size that corresponds exactly with the size of the figure that is being saved.

#### `hexbin` *mincnt* parameter made consistently inclusive

Previously, *mincnt* was inclusive with no *C* provided but exclusive when *C* is provided. It is now inclusive of *mincnt* in both cases.

## Deprecations

### Calling `paths.get_path_collection_extents` with empty *offsets*

Calling *get_path_collection_extents* with an empty *offsets* parameter has an ambiguous interpretation and is therefore deprecated. When the deprecation period expires, this will produce an error.

### `axes_grid1.axes_divider` API changes

The `AxesLocator` class is deprecated. The `new_locator` method of divider instances now instead returns an opaque callable (which can still be passed to `ax.set_axes_locator`).

`Divider.locate` is deprecated; use `Divider.new_locator(...)(ax, renderer)` instead.

### `bbox.anchored()` with no explicit container

Not passing a *container* argument to *BboxBase.anchored* is now deprecated.

### Functions in `mpl_toolkits.mplot3d.proj3d`

The function `transform` is just an alias for `proj_transform`, use the latter instead.

The following functions are either unused (so no longer required in Matplotlib) or considered private. If you rely on them, please make a copy of the code, including all functions that starts with a _ (considered private).

- `ortho_transformation`
- `persp_transformation`
- `proj_points`
- `proj_trans_points`
- `rot_x`
- `rotation_about_vector`
- `view_transformation`

### Arguments other than `renderer` to `get_tightbbox`

... are keyword-only arguments. This is for consistency and that different classes have different additional arguments.

### The object returned by `pcolor()` has changed to a `PolyQuadMesh` class

The old object was a *PolyCollection* with flattened vertices and array data. The new *PolyQuadMesh* class subclasses *PolyCollection*, but adds in better 2D coordinate and array handling in alignment with *QuadMesh*. Previously, if a masked array was input, the list of polygons within the collection would shrink to the size of valid polygons and users were required to keep track of which polygons were drawn and call `set_array()` with the smaller "compressed" array size. Passing the "compressed" and flattened array values is now deprecated and the full 2D array of values (including the mask) should be passed to *PolyQuadMesh.set_array*.

#### `LocationEvent.lastevent`

... is deprecated with no replacement.

#### `allsegs`, `allkinds`, `tcolors` and `tlinewidths` attributes of `ContourSet`

These attributes are deprecated; if required, directly retrieve the vertices and codes of the Path objects from `ContourSet.get_paths()` and the colors and the linewidths via `ContourSet.get_facecolor()`, `ContourSet.get_edgecolor()` and `ContourSet.get_linewidths()`.

#### `ContourSet.collections`

... is deprecated. *`ContourSet`* is now implemented as a single *`Collection`* of paths, each path corresponding to a contour level, possibly including multiple unconnected components.

During the deprecation period, accessing `ContourSet.collections` will revert the current ContourSet instance to the old object layout, with a separate *`PathCollection`* per contour level.

#### `INVALID_NON_AFFINE`, `INVALID_AFFINE`, `INVALID` attributes of `TransformNode`

These attributes are deprecated.

#### `Grouper.clean()`

with no replacement. The Grouper class now cleans itself up automatically.

#### `GridHelperCurveLinear.get_data_boundary`

... is deprecated. Use `grid_finder.extreme_finder(*[None] * 5)` to get the extremes of the grid.

#### *`np_load`* parameter of `cbook.get_sample_data`

This parameter is deprecated; *`get_sample_data`* now auto-loads numpy arrays. Use `get_sample_data(..., asfileobj=False)` instead to get the filename of the data file, which can then be passed to `open`, if desired.

#### `RendererAgg.tostring_rgb` and `FigureCanvasAgg.tostring_rgb`

... are deprecated with no direct replacement. Consider using `buffer_rgba` instead, which should cover most use cases.

### The parameter of `Annotation.contains` and `Legend.contains` is renamed to *mouseevent*

... consistently with *`Artist.contains`*.

### Accessing `event.guiEvent` after event handlers return

... is deprecated: for some GUI toolkits, it is unsafe to do so. In the future, `event.guiEvent` will be set to None once the event handlers return; you may separately stash the object at your own risk.

### Widgets

The *visible* attribute getter of Selector widgets has been deprecated; use `get_visible`

### Method parameters renamed to match base classes

The only parameter of `transform_affine` and `transform_non_affine` in `Transform` subclasses is renamed to *values*.

The *points* parameter of `transforms.IdentityTransform.transform` is renamed to *values*.

The *trans* parameter of `table.Cell.set_transform` is renamed to *t* consistently with *`Artist.set_transform`*.

The *clippath* parameters of `axis.Axis.set_clip_path` and `axis.Tick.set_clip_path` are renamed to *path* consistently with *`Artist.set_clip_path`*.

The *s* parameter of `images.NonUniformImage.set_filternorm` is renamed to *filternorm* consistently with `` `_ImageBase.set_filternorm ``.

The *s* parameter of `images.NonUniformImage.set_filterrad` is renamed to *filterrad* consistently with `` `_ImageBase.set_filterrad ``.

### *numdecs* parameter and attribute of `LogLocator`

... are deprecated without replacement, because they have no effect.

### `NavigationToolbar2QT.message` is deprecated

... with no replacement.

### `ft2font.FT2Image.draw_rect` and `ft2font.FT2Font.get_xys`

... are deprecated as they are unused. If you rely on these, please let us know.

### `backend_ps.psDefs`

The `psDefs` module-level variable in `backend_ps` is deprecated with no replacement.

## Callable axisartist Axes

Calling an axisartist Axes to mean *`axis`* is deprecated; explicitly call the method instead.

### `AnchoredEllipse` is deprecated

Instead, directly construct an *`AnchoredOffsetbox`*, an *`AuxTransformBox`*, and an *`Ellipse`*, as demonstrated in *Anchored Artists*.

## Automatic papersize selection in PostScript

Setting *`rcParams["ps.papersize"]`* (default: `'letter'`) to `'auto'` or passing `papersize='auto'` to *`Figure.savefig`* is deprecated. Either pass an explicit paper type name, or omit this parameter to use the default from the rcParam.

### `Tick.set_label1` and `Tick.set_label2`

... are deprecated. Calling these methods from third-party code usually has no effect, as the labels are overwritten at draw time by the tick formatter.

### Passing extra positional arguments to `Figure.add_axes`

Positional arguments passed to *`Figure.add_axes`* other than a rect or an existing `Axes` are currently ignored, and doing so is now deprecated.

#### `CbarAxesBase.toggle_label`

... is deprecated. Instead, use standard methods for manipulating colorbar labels (*`Colorbar.set_label`*) and tick labels (*`Axes.tick_params`*).

#### `TexManager.texcache`

... is considered private and deprecated. The location of the cache directory is clarified in the doc-string.

### Artists explicitly passed in will no longer be filtered by legend() based on their label

Currently, artists explicitly passed to `legend(handles=[...])` are filtered out if their label starts with an underscore. This behavior is deprecated; explicitly filter out such artists (`[art for art in artists if not art.get_label().startswith('_')]`) if necessary.

#### `FigureCanvasBase.switch_backends`

... is deprecated with no replacement.

#### `cbook.Stack` is deprecated

... with no replacement.

#### `inset_location.InsetPosition` is deprecated

Use *`inset_axes`* instead.

#### `axisartist.axes_grid` and `axisartist.axes_rgb`

These modules, which provide wrappers combining the functionality of *`axes_grid1`* and *`axisartist`*, are deprecated; directly use e.g. `AxesGrid(..., axes_class=axislines.Axes)` instead.

#### `ContourSet.antialiased`

... is deprecated; use *get_antialiased* or *set_antialiased* instead. Note that *get_antialiased* returns an array.

### Passing non-int or sequence of non-int to `Table.auto_set_column_width`

Column numbers are ints, and formerly passing any other type was effectively ignored. This will become an error in the future.

#### `PdfPages(keep_empty=True)`

A zero-page pdf is not valid, thus passing `keep_empty=True` to *backend_pdf.PdfPages* and *backend_pgf.PdfPages*, and the `keep_empty` attribute of these classes, are deprecated. Currently, these classes default to keeping empty outputs, but that behavior is deprecated too. Explicitly passing `keep_empty=False` remains supported for now to help transition to the new behavior.

Furthermore, *backend_pdf.PdfPages* no longer immediately creates the target file upon instantiation, but only when the first figure is saved. To fully control file creation, directly pass an opened file object as argument (`with open(path, "wb") as file, PdfPages(file) as pdf: ...`).

### Auto-closing of figures when switching backend

... is deprecated. Explicitly call `plt.close("all")` if necessary. In the future, allowable backend switches (i.e. those that do not swap a GUI event loop with another one) will not close existing figures.

### Support for passing the "frac" key in `annotate(..., arrowprops={"frac": ...})`

... has been removed. This key has had no effect since Matplotlib 1.5.

## Removals

### cbook removals

- `matplotlib.cbook.MatplotlibDeprecationWarning` and `matplotlib.cbook.mplDeprecation` are removed; use *matplotlib.MatplotlibDeprecationWarning* instead.
- `cbook.maxdict`; use the standard library `functools.lru_cache` instead.

### Groupers from `get_shared_x_axes` / `get_shared_y_axes` are immutable

Modifications to the Groupers returned by `get_shared_x_axes` and `get_shared_y_axes` are no longer allowed. Note that previously, calling e.g. `join()` would already fail to set up the correct structures for sharing axes; use *Axes.sharex* or *Axes.sharey* instead.

### Deprecated modules removed

The following deprecated modules are removed:

- `afm`
- `docstring`
- `fontconfig_pattern`
- `tight_bbox`
- `tight_layout`
- `type1font`

### Parameters to `plt.figure()` and the `Figure` constructor

All parameters to *pyplot.figure* and the *Figure* constructor, other than *num*, *figsize*, and *dpi*, are now keyword-only.

#### `stem(..., use_line_collection=False)`

... is no longer supported. This was a compatibility fallback to a former more inefficient representation of the stem lines.

### Positional / keyword arguments

Passing all but the very few first arguments positionally in the constructors of Artists is no longer possible. Most arguments are now keyword-only.

The *emit* and *auto* parameters of `set_xlim`, `set_ylim`, `set_zlim`, `set_rlim` are now keyword-only.

The *transOffset* parameter of *Collection.set_offset_transform* and the various `create_collection` methods of legend handlers has been renamed to *offset_transform* (consistently with the property name).

`Axes.get_window_extent` / `Figure.get_window_extent` accept only *renderer*. This aligns the API with the general *Artist.get_window_extent* API. All other parameters were ignored anyway.

### Methods to set parameters in `LogLocator` and `LogFormatter*`

In *`LogFormatter`* and derived subclasses, the methods `base` and `label_minor` for setting the respective parameter are removed and replaced by `set_base` and `set_label_minor`, respectively.

In *`LogLocator`*, the methods `base` and `subs` for setting the respective parameter are removed. Instead, use `set_params(base=..., subs=...)`.

#### `Axes.get_renderer_cache`

The canvas now takes care of the renderer and whether to cache it or not, so the `Axes.get_renderer_cache` method is removed. The alternative is to call `axes.figure.canvas.get_renderer()`.

### Unused methods in `Axis`, `Tick`, `XAxis`, and `YAxis`

`Tick.label` is now removed. Use `Tick.label1` instead.

The following methods are no longer used and removed without a replacement:

- `Axis.get_ticklabel_extents`
- `Tick.get_pad_pixels`
- `XAxis.get_text_heights`
- `YAxis.get_text_widths`

#### `mlab.stride_windows`

... is removed. Use `numpy.lib.stride_tricks.sliding_window_view` instead.

#### `Axes3D`

The `dist` attribute has been privatized. Use the *zoom* keyword argument in *`Axes3D.set_box_aspect`* instead.

The `w_xaxis`, `w_yaxis`, and `w_zaxis` attributes are now removed. Instead use `xaxis`, `yaxis`, and `zaxis`.

### 3D Axis

`mplot3d.axis3d.Axis.set_pane_pos` is removed. This is an internal method where the provided values are overwritten during drawing. Hence, it does not serve any purpose to be directly accessible.

The two helper functions `mplot3d.axis3d.move_from_center` and `mplot3d.axis3d.tick_update_position` are considered internal and deprecated. If these are required, please vendor the code from the corresponding private methods `_move_from_center` and `_tick_update_position`.

### `checkdep_usetex` removed

This method was only intended to disable tests in case no latex install was found. As such, it is considered to be private and for internal use only.

Please vendor the code from a previous version if you need this.

### `date_ticker_factory` removed

The `date_ticker_factory` method in the *`matplotlib.dates`* module is removed. Instead use *`AutoDateLocator`* and *`AutoDateFormatter`* for a more flexible and scalable locator and formatter.

If you need the exact `date_ticker_factory` behavior, please copy the code from a previous version.

### `transforms.Affine2D.identity()`

... is removed in favor of directly calling the *`Affine2D`* constructor with no arguments.

### Removals in `testing.decorators`

The unused class `CleanupTestCase` and decorator `cleanup` are removed. The function `check_freetype_version` is considered internal and removed. Vendor the code from a previous version.

### `text.get_rotation()`

... is removed with no replacement. Copy the previous implementation if needed. `Figure.callbacks` is removed ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

The Figure `callbacks` property has been removed. The only signal was "dpi_changed", which can be replaced by connecting to the "resize_event" on the canvas `figure.canvas.mpl_connect("resize_event", func)` instead.

### Passing too many positional arguments to `tripcolor`

... raises `TypeError` (extra arguments were previously ignored).

### The *filled* argument to `Colorbar` is removed

This behavior was already governed by the underlying `ScalarMappable`.

### Widgets

The *visible* attribute setter of Selector widgets has been removed; use `set_visible` The associated getter is also deprecated, but not yet expired.

### `Axes3D.set_frame_on` and `Axes3D.get_frame_on` removed

`Axes3D.set_frame_on` is documented as "Set whether the 3D axes panels are drawn.". However, it has no effect on 3D axes and is being removed in favor of `Axes3D.set_axis_on` and `Axes3D.set_axis_off`.

### Miscellaneous internals

- `axes_grid1.axes_size.AddList`; use `sum(sizes, start=Fixed(0))` (for example) to sum multiple size objects.
- `axes_size.Padded`; use `size + pad` instead
- `axes_size.SizeFromFunc`, `axes_size.GetExtentHelper`
- `AxisArtistHelper.delta1` and `AxisArtistHelper.delta2`
- `axislines.GridHelperBase.new_gridlines` and `axislines.Axes.new_gridlines`
- `_DummyAxis.dataLim` and `_DummyAxis.viewLim`; use `get_data_interval()`, `set_data_interval()`, `get_view_interval()`, and `set_view_interval()` instead.
- `ImageMagickBase.delay` and `ImageMagickBase.output_args`
- `MathtextBackend`, `MathtextBackendAgg`, `MathtextBackendPath`, `MathTextWarning`
- `TexManager.get_font_config`; it previously returned an internal hashed key for used for caching purposes.
- `TextToPath.get_texmanager`; directly construct a *`texmanager.TexManager`* instead.
- `ticker.is_close_to_int`; use `math.isclose(x, round(x))` instead.

- `ticker.is_decade`; use `y = numpy.log(x)/numpy.log(base); numpy.isclose(y, numpy.round(y))` instead.

### Backend-specific removals

- `backend_pdf.Name.hexify`
- `backend_pdf.Operator` and `backend_pdf.Op.op` are removed in favor of a single standard `enum.Enum` interface on *`backend_pdf.Op`*.
- `backend_pdf.fill`; vendor the code of the similarly named private functions if you rely on these functions.
- `backend_pgf.LatexManager.texcommand` and `backend_pgf.LatexManager.latex_header`
- `backend_pgf.NO_ESCAPE`
- `backend_pgf.common_texification`
- `backend_pgf.get_fontspec`
- `backend_pgf.get_preamble`
- `backend_pgf.re_mathsep`
- `backend_pgf.writeln`
- `backend_ps.convert_psfrags`
- `backend_ps.quote_ps_string`; vendor the code of the similarly named private functions if you rely on it.
- `backend_svg.escape_attrib`; vendor the code of the similarly named private functions if you rely on it.
- `backend_svg.escape_cdata`; vendor the code of the similarly named private functions if you rely on it.
- `backend_svg.escape_comment`; vendor the code of the similarly named private functions if you rely on it.
- `backend_svg.short_float_fmt`; vendor the code of the similarly named private functions if you rely on it.
- `backend_svg.generate_transform` and `backend_svg.generate_css`

### Removal of deprecated APIs

The following deprecated APIs have been removed. Unless a replacement is stated, please vendor the previous implementation if needed.

- The following methods of *FigureCanvasBase*: `pick` (use `Figure.pick` instead), `resize`, `draw_event`, `resize_event`, `close_event`, `key_press_event`, `key_release_event`, `pick_event`, `scroll_event`, `button_press_event`, `button_release_event`, `motion_notify_event`, `leave_notify_event`, `enter_notify_event` (for all the `foo_event` methods, construct the relevant *Event* object and call `canvas.callbacks.process(event.name, event)` instead).
- `ToolBase.destroy` (connect to `tool_removed_event` instead).
- The *cleared* parameter to *FigureCanvasAgg.get_renderer* (call `renderer.clear()` instead).
- The following methods of *RendererCairo*: `set_ctx_from_surface` and `set_width_height` (use `set_context` instead, which automatically infers the canvas size).
- The `window` or `win` parameters and/or attributes of `NavigationToolbar2Tk`, `NavigationToolbar2GTK3`, and `NavigationToolbar2GTK4`, and the `lastrect` attribute of `NavigationToolbar2Tk`
- The `error_msg_gtk` function and the `icon_filename` and `window_icon` globals in `backend_gtk3`; the `error_msg_wx` function in `backend_wx`.
- `FigureManagerGTK3Agg` and `FigureManagerGTK4Agg` (use `FigureManagerGTK3` instead); `RendererGTK3Cairo` and `RendererGTK4Cairo`.
- `NavigationToolbar2Mac.prepare_configure_subplots` (use *configure_subplots* instead).
- `FigureManagerMac.close`.
- The `qApp` global in *backend_qt* (use `QtWidgets.QApplication.instance()` instead).
- The `offset_text_height` method of `RendererWx`; the `sizer`, `figmgr`, `num`, `toolbar`, `toolmanager`, `get_canvas`, and `get_figure_manager` attributes or methods of `FigureFrameWx` (use `frame.GetSizer()`, `frame.canvas.manager`, `frame.canvas.manager.num`, `frame.GetToolBar()`, `frame.canvas.manager.toolmanager`, the *canvas_class* constructor parameter, and `frame.canvas.manager`, respectively, instead).
- `FigureFrameWxAgg` and `FigureFrameWxCairo` (use `FigureFrameWx(..., canvas_class=FigureCanvasWxAgg)` and `FigureFrameWx(..., canvas_class=FigureCanvasWxCairo)`, respectively, instead).
- The `filled` attribute and the `draw_all` method of *Colorbar* (instead of `draw_all`, use `figure.draw_without_rendering`).
- Calling *MarkerStyle* without setting the *marker* parameter or setting it to None (use `MarkerStyle("")` instead).

- Support for third-party canvas classes without a `required_interactive_framework` attribute (this can only occur if the canvas class does not inherit from *`FigureCanvasBase`*).
- The `canvas` and `background` attributes of *`MultiCursor`*; the `state_modifier_keys` attribute of selector widgets.
- Passing *useblit*, *horizOn*, or *vertOn* positionally to *`MultiCursor`*.
- Support for the `seaborn-<foo>` styles; use `seaborn-v0_8-<foo>` instead, or directly use the seaborn API.

### Development changes

#### Increase to minimum supported versions of dependencies

For Matplotlib 3.8, the *minimum supported versions* are being bumped:

| Dependency | min in mpl3.7 | min in mpl3.8 |
|---|---|---|
| Python | 3.8 | 3.9 |
| kiwisolver | 1.0.1 | 1.3.1 |
| NumPy | 1.20.0 | 1.21.0 |
| Pillow | 6.2.1 | 8.0 |

This is consistent with our *Dependency version policy* and NEP29

#### Increase to minimum supported optional dependencies

For Matplotlib 3.8, the *minimum supported versions of optional dependencies* are being bumped:

| Dependency | min in mpl3.7 | min in mpl3.8 |
|---|---|---|
| Tk | 8.4 | 8.5 |
| Qt | 5.10 | 5.12 |

- There are no wheels or conda packages that support both Qt 5.11 (or older) and Python 3.9 (or newer).

This is consistent with our *Dependency version policy*

#### Provisional support for PEP484 Type Hint Annotations

New public API should be type hinted in `.pyi` stub files (except `pyplot` and tests which are typed in-line). Tests should be type hinted minimally, essentially only when `mypy` generates errors.

CI and configuration for running `mypy` have been added.

#### Generation of `pyplot.py` requires `black`

The autogenerated portions of `pyplot.py` use `black` autoformatting to ensure syntax-correct, readable output code.

As such `black` is now a development and test requirement (for the test which regenerates `pyplot`).

#### Wheels for some systems are no longer distributed

Pre-compiled wheels for 32-bit Linux and Windows are no longer provided on PyPI since Matplotlib 3.8.

Multi-architecture `universal2` wheels for macOS are no longer provided on PyPI since Matplotlib 3.8. In general, `pip` will always prefer the architecture-specific (`amd64`- or `arm64`-only) wheels, so these provided little benefit.

#### New wheel architectures

Wheels have been added for:

- musl based systems

## 9.3 Version 3.7

### 9.3.1 What's new in Matplotlib 3.7.0 (Feb 13, 2023)

For a list of all of the issues and pull requests since the last revision, see the *GitHub statistics for 3.9.0 (May 15, 2024)*.

**Table of Contents**


- *What's new in Matplotlib 3.7.0 (Feb 13, 2023)*
  - *Plotting and Annotation improvements*
    - *`hatch` parameter for pie*
    - *Polar plot errors drawn in polar coordinates*
    - *Additional format string options in `bar_label`*

        * *ellipse boxstyle option for annotations*
        * *The* extent *of imshow can now be expressed with units*
        * *Reversed order of legend entries*
        * *pcolormesh accepts RGB(A) colors*
        * *View current appearance settings for ticks, tick labels, and gridlines*
        * *Style files can be imported from third-party packages*
    - *Improvements to 3D Plotting*
        * *3D plot pan and zoom buttons*
        * adjustable *keyword argument for setting equal aspect ratios in 3D*
        * *Poly3DCollection supports shading*
        * *rcParam for 3D pane color*
    - *Figure and Axes Layout*
        * *colorbar now has a* location *keyword argument*
        * *Figure legends can be placed outside figures using constrained_layout*
        * *Per-subplot keyword arguments in subplot_mosaic*
        * *subplot_mosaic no longer provisional*
    - *Widget Improvements*
        * *Custom styling of button widgets*
        * *Blitting in Button widgets*
    - *Other Improvements*
        * *Source links can be shown or hidden for each Sphinx plot directive*
        * *Figure hooks*
    - *New & Improved Narrative Documentation*


## Plotting and Annotation improvements

### `hatch` parameter for pie

*pie* now accepts a *hatch* keyword that takes as input a hatch or list of hatches:

```
fig, (ax1, ax2) = plt.subplots(ncols=2)
x = [10, 30, 60]

ax1.pie(x, hatch=['.', 'o', 'O'])
ax2.pie(x, hatch='.O')
```

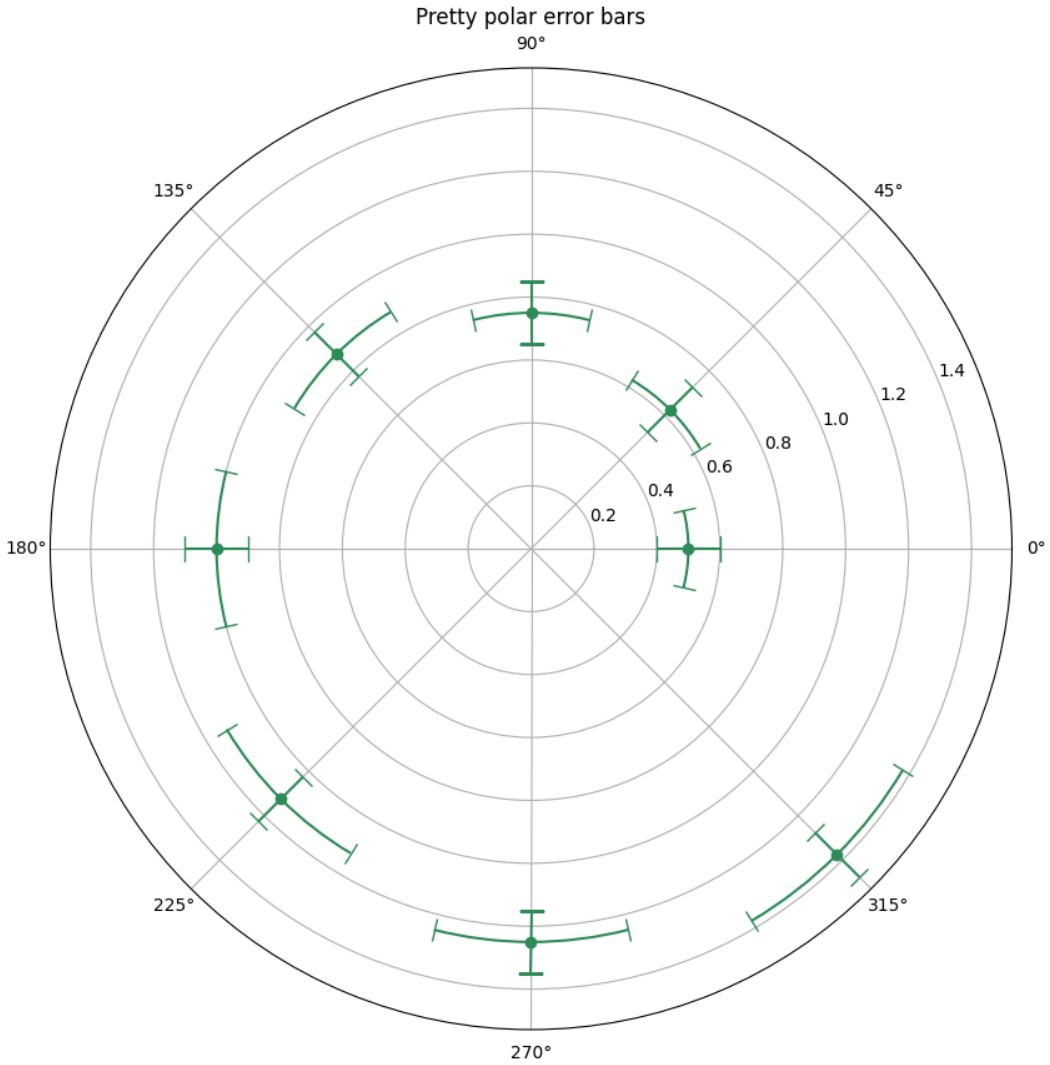
(continues on next page)

(continued from previous page)

```
ax1.set_title("hatch=['.', 'o', 'O']")
ax2.set_title("hatch='.O'")
```

### Polar plot errors drawn in polar coordinates

Caps and error lines are now drawn with respect to polar coordinates, when plotting errorbars on polar plots.



### Additional format string options in `bar_label`

The `fmt` argument of *`bar_label`* now accepts {}-style format strings:

```
import matplotlib.pyplot as plt

fruit_names = ['Coffee', 'Salted Caramel', 'Pistachio']
fruit_counts = [4000, 2000, 7000]

fig, ax = plt.subplots()
bar_container = ax.bar(fruit_names, fruit_counts)
ax.set(ylabel='pints sold', title='Gelato sales by flavor', ylim=(0, 8000))
ax.bar_label(bar_container, fmt='{:,.0f}')
```

It also accepts callables:

```
animal_names = ['Lion', 'Gazelle', 'Cheetah']
mph_speed = [50, 60, 75]

fig, ax = plt.subplots()
bar_container = ax.bar(animal_names, mph_speed)
ax.set(ylabel='speed in MPH', title='Running speeds', ylim=(0, 80))
ax.bar_label(
    bar_container, fmt=lambda x: '{:.1f} km/h'.format(x * 1.61)
)
```

### `ellipse` boxstyle option for annotations

The `'ellipse'` option for boxstyle can now be used to create annotations with an elliptical outline. It can be used as a closed curve shape for longer texts instead of the `'circle'` boxstyle which can get quite big.

```
import matplotlib.pyplot as plt
fig, ax = plt.subplots(figsize=(5, 5))
t = ax.text(0.5, 0.5, "elliptical box",
        ha="center", size=15,
        bbox=dict(boxstyle="ellipse,pad=0.3"))
```

### The *extent* of `imshow` can now be expressed with units

The *extent* parameter of *imshow* and *set_extent* can now be expressed with units.

```
import matplotlib.pyplot as plt
import numpy as np

fig, ax = plt.subplots(layout='constrained')
date_first = np.datetime64('2020-01-01', 'D')
date_last = np.datetime64('2020-01-11', 'D')

arr = [[i+j for i in range(10)] for j in range(10)]

ax.imshow(arr, origin='lower', extent=[0, 10, date_first, date_last])

plt.show()
```

### Reversed order of legend entries

The order of legend entries can now be reversed by passing `reverse=True` to *legend*.

### `pcolormesh` accepts RGB(A) colors

The *pcolormesh* method can now handle explicit colors specified with RGB(A) values. To specify colors, the array must be 3D with a shape of `(M, N, [3, 4])`.

```
import matplotlib.pyplot as plt
import numpy as np

colors = np.linspace(0, 1, 90).reshape((5, 6, 3))
plt.pcolormesh(colors)
plt.show()
```

### View current appearance settings for ticks, tick labels, and gridlines

The new *get_tick_params* method can be used to retrieve the appearance settings that will be applied to any additional ticks, tick labels, and gridlines added to the plot:

```
>>> import matplotlib.pyplot as plt

>>> fig, ax = plt.subplots()
>>> ax.yaxis.set_tick_params(labelsize=30, labelcolor='red',
...                          direction='out', which='major')
>>> ax.yaxis.get_tick_params(which='major')
```

(continues on next page)

(continued from previous page)

```
{'direction': 'out',
'left': True,
'right': False,
'labelleft': True,
'labelright': False,
'gridOn': False,
'labelsize': 30,
'labelcolor': 'red'}
>>> ax.yaxis.get_tick_params(which='minor')
{'left': True,
'right': False,
'labelleft': True,
'labelright': False,
'gridOn': False}
```

### Style files can be imported from third-party packages

Third-party packages can now distribute style files that are globally available as follows. Assume that a package is importable as `import mypackage`, with a `mypackage/__init__.py` module. Then a `mypackage/presentation.mplstyle` style sheet can be used as `plt.style.use("mypackage.presentation")`.

The implementation does not actually import `mypackage`, making this process safe against possible import-time side effects. Subpackages (e.g. `dotted.package.name`) are also supported.

## Improvements to 3D Plotting

### 3D plot pan and zoom buttons

The pan and zoom buttons in the toolbar of 3D plots are now enabled. Unselect both to rotate the plot. When the zoom button is pressed, zoom in by using the left mouse button to draw a bounding box, and out by using the right mouse button to draw the box. When zooming a 3D plot, the current view aspect ratios are kept fixed.

### *adjustable* keyword argument for setting equal aspect ratios in 3D

While setting equal aspect ratios for 3D plots, users can choose to modify either the data limits or the bounding box in parity with 2D Axes.

```
import matplotlib.pyplot as plt
import numpy as np
from itertools import combinations, product

aspects = ('auto', 'equal', 'equalxy', 'equalyz', 'equalxz')
fig, axs = plt.subplots(1, len(aspects), subplot_kw={'projection': '3d'},
```

(continues on next page)

(continued from previous page)

```
                          figsize=(12, 6))

# Draw rectangular cuboid with side lengths [4, 3, 5]
r = [0, 1]
scale = np.array([4, 3, 5])
pts = combinations(np.array(list(product(r, r, r))), 2)
for start, end in pts:
    if np.sum(np.abs(start - end)) == r[1] - r[0]:
        for ax in axs:
            ax.plot3D(*zip(start*scale, end*scale), color='C0')

# Set the aspect ratios
for i, ax in enumerate(axs):
    ax.set_aspect(aspects[i], adjustable='datalim')
    # Alternatively: ax.set_aspect(aspects[i], adjustable='box')
    # which will change the box aspect ratio instead of axis data limits.
    ax.set_title(f"set_aspect('{aspects[i]}')")

plt.show()
```

### `Poly3DCollection` supports shading

It is now possible to shade a *`Poly3DCollection`*. This is useful if the polygons are obtained from e.g. a 3D model.

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d.art3d import Poly3DCollection

# Define 3D shape
block = np.array([
    [[1, 1, 0],
     [1, 0, 0],
     [0, 1, 0]],
    [[1, 1, 0],
     [1, 1, 1],
     [1, 0, 0]],
    [[1, 1, 0],
     [1, 1, 1],
     [0, 1, 0]],
    [[1, 0, 0],
     [1, 1, 1],
     [0, 1, 0]]
])

ax = plt.subplot(projection='3d')
pc = Poly3DCollection(block, facecolors='b', shade=True)
ax.add_collection(pc)
plt.show()
```

### rcParam for 3D pane color

The rcParams *rcParams["axes3d.xaxis.panecolor"]* (default: `(0.95, 0.95, 0.95, 0.5)`), *rcParams["axes3d.yaxis.panecolor"]* (default: `(0.9, 0.9, 0.9, 0.5)`), *rcParams["axes3d.zaxis.panecolor"]* (default: `(0.925, 0.925, 0.925, 0.5)`) can be used to change the color of the background panes in 3D plots. Note that it is often beneficial to give them slightly different shades to obtain a "3D effect" and to make them slightly transparent (alpha < 1).

```
import matplotlib.pyplot as plt
with plt.rc_context({'axes3d.xaxis.panecolor': (0.9, 0.0, 0.0, 0.5),
                     'axes3d.yaxis.panecolor': (0.7, 0.0, 0.0, 0.5),
                     'axes3d.zaxis.panecolor': (0.8, 0.0, 0.0, 0.5)}):
    fig = plt.figure()
    fig.add_subplot(projection='3d')
```

## Figure and Axes Layout

### `colorbar` now has a *location* keyword argument

The `colorbar` method now supports a *location* keyword argument to more easily position the color bar. This is useful when providing your own inset axes using the *cax* keyword argument and behaves similar to the case where axes are not provided (where the *location* keyword is passed through). *orientation* and *ticklocation* are no longer required as they are determined by *location*. *ticklocation* can still be provided if the automatic setting is not preferred. (*orientation* can also be provided but must be compatible with the *location*.)

An example is:

```
import matplotlib.pyplot as plt
import numpy as np
rng = np.random.default_rng(19680801)
imdata = rng.random((10, 10))
fig, ax = plt.subplots(layout='constrained')
im = ax.imshow(imdata)
fig.colorbar(im, cax=ax.inset_axes([0, 1.05, 1, 0.05]),
             location='top')
```

### Figure legends can be placed outside figures using constrained_layout

Constrained layout will make space for Figure legends if they are specified by a *loc* keyword argument that starts with the string "outside". The codes are unique from axes codes, in that "outside upper right" will make room at the top of the figure for the legend, whereas "outside right upper" will make room on the right-hand side of the figure. See *Legend guide* for details.

### Per-subplot keyword arguments in `subplot_mosaic`

It is now possible to pass keyword arguments through to Axes creation in each specific call to `add_subplot` in *`Figure.subplot_mosaic`* and *`pyplot.subplot_mosaic`*:

```
fig, axd = plt.subplot_mosaic(
    "AB;CD",
    per_subplot_kw={
        "A": {"projection": "polar"},
        ("C", "D"): {"xscale": "log"},
        "B": {"projection": "3d"},
    },
)
```

This is particularly useful for creating mosaics with mixed projections, but any keyword arguments can be passed through.

### `subplot_mosaic` no longer provisional

The API on *`Figure.subplot_mosaic`* and *`pyplot.subplot_mosaic`* are now considered stable and will change under Matplotlib's normal deprecation process.

### Widget Improvements

### Custom styling of button widgets

Additional custom styling of button widgets may be achieved via the *label_props* and *radio_props* arguments to *`RadioButtons`*; and the *label_props*, *frame_props*, and *check_props* arguments to *`CheckButtons`*.

```
from matplotlib.widgets import CheckButtons, RadioButtons

fig, ax = plt.subplots(nrows=2, ncols=2, figsize=(5, 2), width_ratios=[1, 2])
default_rb = RadioButtons(ax[0, 0], ['Apples', 'Oranges'])
styled_rb = RadioButtons(ax[0, 1], ['Apples', 'Oranges'],
                         label_props={'color': ['red', 'orange'],
                                      'fontsize': [16, 20]},
                         radio_props={'edgecolor': ['red', 'orange'],
                                      'facecolor': ['mistyrose', 'peachpuff']}
```

(continues on next page)

(continued from previous page)

```
 ↪)

default_cb = CheckButtons(ax[1, 0], ['Apples', 'Oranges'],
                          actives=[True, True])
styled_cb = CheckButtons(ax[1, 1], ['Apples', 'Oranges'],
                         actives=[True, True],
                         label_props={'color': ['red', 'orange'],
                                      'fontsize': [16, 20]},
                         frame_props={'edgecolor': ['red', 'orange'],
                                      'facecolor': ['mistyrose', 'peachpuff']}
 ↪,
                         check_props={'color': ['darkred', 'darkorange']})

ax[0, 0].set_title('Default')
ax[0, 1].set_title('Stylized')
```

### Blitting in Button widgets

The *Button*, *CheckButtons*, and *RadioButtons* widgets now support blitting for faster rendering, on backends that support it, by passing `useblit=True` to the constructor. Blitting is enabled by default on supported backends.

### Other Improvements

### Source links can be shown or hidden for each Sphinx plot directive

The *Sphinx plot directive* (`.. plot::`) now supports a `:show-source-link:` option to show or hide the link to the source code for each plot. The default is set using the `plot_html_show_source_link` variable in `conf.py` (which defaults to True).

### Figure hooks

The new *rcParams["figure.hooks"]* (default: `[]`) provides a mechanism to register arbitrary customizations on pyplot figures; it is a list of "dotted.module.name:dotted.callable.name" strings specifying functions that are called on each figure created by *pyplot.figure*; these functions can e.g. attach callbacks or modify the toolbar. See *mplcvd -- an example of figure hook* for an example of toolbar customization.

### New & Improved Narrative Documentation

- Brand new *Animations* tutorial.
- New grouped and stacked bar chart examples.
- New section for new contributors and reorganized git instructions in the *contributing guide*.
- Restructured *Annotations* tutorial.

## 9.3.2 API Changes for 3.7.0


- *Behaviour Changes*
- *Deprecations*
- *Removals*
- *Development changes*


### Behaviour Changes

#### All Axes have `get_subplotspec` and `get_gridspec` methods now, which returns None for Axes not positioned via a gridspec

Previously, this method was only present for Axes positioned via a gridspec. Following this change, checking `hasattr(ax, "get_gridspec")` should now be replaced by `ax.get_gridspec() is not None`. For compatibility with older Matplotlib releases, one can also check `hasattr(ax, "get_gridspec") and ax.get_gridspec() is not None`.

#### `HostAxesBase.get_aux_axes` now defaults to using the same base axes class as the host axes

If using an `mpl_toolkits.axisartist`-based host Axes, the parasite Axes will also be based on `mpl_toolkits.axisartist`. This behavior is consistent with `HostAxesBase.twin`, `HostAxesBase.twinx`, and `HostAxesBase.twiny`.

### `plt.get_cmap` and `matplotlib.cm.get_cmap` return a copy

Formerly, *get_cmap* and `matplotlib.cm.get_cmap` returned a global version of a *Colormap*. This was prone to errors as modification of the colormap would propagate from one location to another without warning. Now, a new copy of the colormap is returned.

### `TrapezoidMapTriFinder` uses different random number generator

The random number generator used to determine the order of insertion of triangle edges in `TrapezoidMapTriFinder` has changed. This can result in a different triangle index being returned for a point that lies exactly on an edge between two triangles. This can also affect triangulation interpolation and refinement algorithms that use `TrapezoidMapTriFinder`.

### `FuncAnimation(save_count=None)`

Passing `save_count=None` to *FuncAnimation* no longer limits the number of frames to 100. Make sure that it either can be inferred from *frames* or provide an integer *save_count*.

### `CenteredNorm` halfrange is not modified when vcenter changes

Previously, the **halfrange** would expand in proportion to the amount that **vcenter** was moved away from either **vmin** or **vmax**. Now, the halfrange remains fixed when vcenter is changed, and **vmin** and **vmax** are updated based on the **vcenter** and **halfrange** values.

For example, this is what the values were when changing vcenter previously.

```
norm = CenteredNorm(vcenter=0, halfrange=1)
# Move vcenter up by one
norm.vcenter = 1
# updates halfrange and vmax (vmin stays the same)
# norm.halfrange == 2, vmin == -1, vmax == 3
```

and now, with that same example

```
norm = CenteredNorm(vcenter=0, halfrange=1)
norm.vcenter = 1
# updates vmin and vmax (halfrange stays the same)
# norm.halfrange == 1, vmin == 0, vmax == 2
```

The **halfrange** can be set manually or `norm.autoscale()` can be used to automatically set the limits after setting **vcenter**.

### `fig.subplot_mosaic` no longer passes the `gridspec_kw` args to nested gridspecs.

For nested *Figure.subplot_mosaic* layouts, it is almost always inappropriate for *gridspec_kw* arguments to be passed to lower nest levels, and these arguments are incompatible with the lower levels in many cases. This dictionary is no longer passed to the inner layouts. Users who need to modify *gridspec_kw* at multiple levels should use *Figure.subfigures* to get nesting, and construct the inner layouts with *Figure.subplots* or *Figure.subplot_mosaic*.

### `HPacker` alignment with bottom or top are now correct

Previously, the **bottom** and **top** alignments were swapped. This has been corrected so that the alignments correspond appropriately.

### On Windows only fonts known to the registry will be discovered

Previously, Matplotlib would recursively walk user and system font directories to discover fonts, however this lead to a number of undesirable behaviors including finding deleted fonts. Now Matplotlib will only find fonts that are known to the Windows registry.

This means that any user installed fonts must go through the Windows font installer rather than simply being copied to the correct folder.

This only impacts the set of fonts Matplotlib will consider when using *matplotlib.font_manager.findfont*. To use an arbitrary font, directly pass the path to a font as shown in *Using ttf font files*.

### `QuadMesh.set_array` now always raises `ValueError` for inputs with incorrect shapes

It could previously also raise `TypeError` in some cases.

### `contour` and `contourf` auto-select suitable levels when given boolean inputs

If the height array given to *Axes.contour* or *Axes.contourf* is of bool dtype and *levels* is not specified, *levels* now defaults to `[0.5]` for *contour* and `[0, 0.5, 1]` for *Axes.contourf*.

### `contour` no longer warns if no contour lines are drawn.

This can occur if the user explicitly passes a `levels` array with no values between `z.min()` and `z.max()`; or if `z` has the same value everywhere.

#### `AxesImage.set_extent` now raises `TypeError` for unknown keyword arguments

It previously raised a `ValueError`.

### Change of `legend(loc="best")` behavior

The algorithm of the auto-legend locator has been tweaked to better handle non rectangular patches. Additional details on this change can be found in #9580 and #9598.

## Deprecations

#### `Axes` subclasses should override `clear` instead of `cla`

For clarity, *`axes.Axes.clear`* is now preferred over *`Axes.cla`*. However, for backwards compatibility, the latter will remain as an alias for the former.

For additional compatibility with third-party libraries, Matplotlib will continue to call the `cla` method of any *`Axes`* subclasses if they define it. In the future, this will no longer occur, and Matplotlib will only call the `clear` method in *`Axes`* subclasses.

It is recommended to define only the `clear` method when on Matplotlib 3.6, and only `cla` for older versions.

### rcParams type

Relying on `rcParams` being a `dict` subclass is deprecated.

Nothing will change for regular users because `rcParams` will continue to be dict-like (technically fulfill the `MutableMapping` interface).

The *`RcParams`* class does validation checking on calls to `.RcParams.__getitem__` and `.RcParams.__setitem__`. However, there are rare cases where we want to circumvent the validation logic and directly access the underlying data values. Previously, this could be accomplished via a call to the parent methods `dict.__getitem__(rcParams, key)` and `dict.__setitem__(rcParams, key, val)`.

Matplotlib 3.7 introduces `rcParams._set(key, val)` and `rcParams._get(key)` as a replacement to calling the parent methods. They are intentionally marked private to discourage external use; However, if direct *`RcParams`* data access is needed, please switch from the dict functions to the new `_get()` and `_set()`. Even though marked private, we guarantee API stability for these methods and they are subject to Matplotlib's API and deprecation policy.

Please notify the Matplotlib developers if you rely on `rcParams` being a dict subclass in any other way, for which there is no migration path yet.

### Deprecation aliases in cbook

The module `matplotlib.cbook.deprecation` was previously deprecated in Matplotlib 3.4, along with deprecation-related API in `matplotlib.cbook`. Due to technical issues, `matplotlib.cbook.MatplotlibDeprecationWarning` and `matplotlib.cbook.mplDeprecation` did not raise deprecation warnings on use. Changes in Python have now made it possible to warn when these aliases are being used.

In order to avoid downstream breakage, these aliases will now warn, and their removal has been pushed from 3.6 to 3.8 to give time to notice said warnings. As replacement, please use *matplotlib.MatplotlibDeprecationWarning*.

#### `draw_gouraud_triangle`

... is deprecated as in most backends this is a redundant call. Use *draw_gouraud_triangles* instead. A `draw_gouraud_triangle` call in a custom *Artist* can readily be replaced as:

```
self.draw_gouraud_triangles(gc, points.reshape((1, 3, 2)),
                            colors.reshape((1, 3, 4)), trans)
```

A *draw_gouraud_triangles* method can be implemented from an existing `draw_gouraud_triangle` method as:

```
transform = transform.frozen()
for tri, col in zip(triangles_array, colors_array):
    self.draw_gouraud_triangle(gc, tri, col, transform)
```

#### `matplotlib.pyplot.get_plot_commands`

... is a pending deprecation. This is considered internal and no end-user should need it.

#### `matplotlib.tri` submodules are deprecated

The `matplotlib.tri.*` submodules are deprecated. All functionality is available in `matplotlib.tri` directly and should be imported from there.

### Passing undefined *label_mode* to `Grid`

... is deprecated. This includes *mpl_toolkits.axes_grid1.axes_grid.Grid*, *mpl_toolkits.axes_grid1.axes_grid.AxesGrid*, and *mpl_toolkits.axes_grid1.axes_grid.ImageGrid* as well as the corresponding classes imported from *mpl_toolkits.axisartist.axes_grid*.

Pass `label_mode='keep'` instead to get the previous behavior of not modifying labels.

### Colorbars for orphaned mappables are deprecated, but no longer raise

Before 3.6.0, Colorbars for mappables that do not have a parent axes would steal space from the current Axes. 3.6.0 raised an error on this, but without a deprecation cycle. For 3.6.1 this is reverted, the current axes is used, but a deprecation warning is shown instead. In this undetermined case users and libraries should explicitly specify what axes they want space to be stolen from: `fig.colorbar(mappable, ax=plt.gca())`.

### `Animation` attributes

The attributes `repeat` of *TimedAnimation* and subclasses and `save_count` of *FuncAnimation* are considered private and deprecated.

### `contour.ClabelText` and `ContourLabeler.set_label_props`

... are deprecated.

Use `Text(..., transform_rotates_text=True)` as a replacement for `contour.ClabelText(...)` and `text.set(text=text, color=color, fontproperties=labeler.labelFontProps, clip_box=labeler.axes.bbox)` as a replacement for the `ContourLabeler.set_label_props(label, text, color)`.

### `ContourLabeler` attributes

The `labelFontProps`, `labelFontSizeList`, and `labelTextsList` attributes of *ContourLabeler* have been deprecated. Use the `labelTexts` attribute and the font properties of the corresponding text objects instead.

### `backend_ps.PsBackendHelper` and `backend_ps.ps_backend_helper`

... are deprecated with no replacement.

### `backend_webagg.ServerThread` is deprecated

... with no replacement.

#### `parse_fontconfig_pattern` will no longer ignore unknown constant names

Previously, in a fontconfig pattern like `DejaVu Sans:foo`, the unknown `foo` constant name would be silently ignored. This now raises a warning, and will become an error in the future.

#### `BufferRegion.to_string` and `BufferRegion.to_string_argb`

... are deprecated. Use `np.asarray(buffer_region)` to get an array view on a buffer region without making a copy; to convert that view from RGBA (the default) to ARGB, use `np.take(..., [2, 1, 0, 3], axis=2)`.

#### `num2julian`, `julian2num` and `JULIAN_OFFSET`

... of the *dates* module are deprecated without replacements. These are undocumented and not exported. If you rely on these, please make a local copy.

#### `unit_cube`, `tunit_cube`, and `tunit_edges`

... of *Axes3D* are deprecated without replacements. If you rely on them, please copy the code of the corresponding private function (name starting with _).

### Most arguments to widgets have been made keyword-only

Passing all but the very few first arguments positionally in the constructors of Widgets is deprecated. Most arguments will become keyword-only in a future version.

#### `SimpleEvent`

The `SimpleEvent` nested class (previously accessible via the public subclasses of `ConnectionStyle._Base`, such as *ConnectionStyle.Arc*, has been deprecated.

#### `RadioButtons.circles`

... is deprecated. (RadioButtons now draws itself using *scatter*.)

#### `CheckButtons.rectangles` and `CheckButtons.lines`

`CheckButtons.rectangles` and `CheckButtons.lines` are deprecated. (`CheckButtons` now draws itself using *`scatter`*.)

#### `OffsetBox.get_extent_offsets` and `OffsetBox.get_extent`

... are deprecated; these methods are also deprecated on all subclasses of *`OffsetBox`*.

To get the offsetbox extents, instead of `get_extent`, use *`OffsetBox.get_bbox`*, which directly returns a *`Bbox`* instance.

To also get the child offsets, instead of `get_extent_offsets`, separately call *`get_offset`* on each children after triggering a draw.

#### `legend.legendHandles`

... was undocumented and has been renamed to `legend_handles`. Using `legendHandles` is deprecated.

#### `ticklabels` parameter of `Axis.set_ticklabels` renamed to `labels`

#### `offsetbox.bbox_artist`

... is deprecated. This is just a wrapper to call *`patches.bbox_artist`* if a flag is set in the file, so use that directly if you need the behavior.

#### `Quiver.quiver_doc` and `Barbs.barbs_doc`

... are deprecated. These are the doc-string and should not be accessible as a named class member.

### Deprecate unused parameter *`x`* to `TextBox.begin_typing`

This parameter was unused in the method, but was a required argument.

### Deprecation of top-level cmap registration and access functions in `mpl.cm`

As part of a multi-step process we are refactoring the global state for managing the registered colormaps.

In Matplotlib 3.5 we added a *ColormapRegistry* class and exposed an instance at the top level as `matplotlib.colormaps`. The existing top level functions in *matplotlib.cm* (`get_cmap`, `register_cmap`, `unregister_cmap`) were changed to be aliases around the same instance. In Matplotlib 3.6 we have marked those top level functions as pending deprecation.

In Matplotlib 3.7, the following functions have been marked for deprecation:

- `matplotlib.cm.get_cmap`; use `matplotlib.colormaps[name]` instead if you have a `str`.

  **Added 3.6.1** Use *matplotlib.cm.ColormapRegistry.get_cmap* if you have a string, `None` or a *matplotlib.colors.Colormap* object that you want to convert to a *matplotlib.colors.Colormap* instance.

- `matplotlib.cm.register_cmap`; use *matplotlib.colormaps.register* instead
- `matplotlib.cm.unregister_cmap`; use *matplotlib.colormaps.unregister* instead
- `matplotlib.pyplot.register_cmap`; use *matplotlib.colormaps.register* instead

The *matplotlib.pyplot.get_cmap* function will stay available for backward compatibility.

### `BrokenBarHCollection` is deprecated

It was just a thin wrapper inheriting from *PolyCollection*; *broken_barh* has now been changed to return a *PolyCollection* instead.

The `BrokenBarHCollection.span_where` helper is likewise deprecated; for the duration of the deprecation it has been moved to the parent *PolyCollection* class. Use *fill_between* as a replacement; see *Shade regions defined by a logical mask using fill_between* for an example.

### Passing inconsistent `loc` and `nth_coord` to axisartist helpers

Trying to construct for example a "top y-axis" or a "left x-axis" is now deprecated.

#### `passthru_pt`

This attribute of `AxisArtistHelpers` is deprecated.

#### `axes3d.vvec`, `axes3d.eye`, `axes3d.sx`, and `axes3d.sy`

... are deprecated without replacement.

#### `Line2D`

When creating a Line2D or using *`Line2D.set_xdata`* and *`Line2D.set_ydata`*, passing x/y data as non sequence is deprecated.

### Removals

#### `epoch2num` and `num2epoch` are removed

These methods convert from unix timestamps to matplotlib floats, but are not used internally to Matplotlib, and should not be needed by end users. To convert a unix timestamp to datetime, simply use `datetime.datetime.fromtimestamp`, or to use NumPy `datetime64` `dt = np.datetime64(e*1e6, 'us')`.

### Locator and Formatter wrapper methods

The `set_view_interval`, `set_data_interval` and `set_bounds` methods of *`Locator`*s and *`Formatter`*s (and their common base class, TickHelper) are removed. Directly manipulate the view and data intervals on the underlying axis instead.

### Interactive cursor details

Setting a mouse cursor on a window has been moved from the toolbar to the canvas. Consequently, several implementation details on toolbars and within backends have been removed.

#### `NavigationToolbar2.set_cursor` and `backend_tools.SetCursorBase.set_cursor`

Instead, use the *`FigureCanvasBase.set_cursor`* method on the canvas (available as the `canvas` attribute on the toolbar or the Figure.)

#### `backend_tools.SetCursorBase` and subclasses

`backend_tools.SetCursorBase` was subclassed to provide backend-specific implementations of `set_cursor`. As that is now removed, the subclassing is no longer necessary. Consequently, the following subclasses are also removed:

- `matplotlib.backends.backend_gtk3.SetCursorGTK3`
- `matplotlib.backends.backend_qt5.SetCursorQt`
- `matplotlib.backends._backend_tk.SetCursorTk`
- `matplotlib.backends.backend_wx.SetCursorWx`

Instead, use the *backend_tools.ToolSetCursor* class.

#### `cursord` in GTK and wx backends

The `backend_gtk3.cursord` and `backend_wx.cursord` dictionaries are removed. This makes the GTK module importable on headless environments.

#### `auto_add_to_figure=True` for `Axes3D`

... is no longer supported. Instead use `fig.add_axes(ax)`.

#### The first parameter of `Axes.grid` and `Axis.grid` has been renamed to *visible*

The parameter was previously named *b*. This name change only matters if that parameter was passed using a keyword argument, e.g. `grid(b=False)`.

### Removal of deprecations in the Selector widget API

#### RectangleSelector and EllipseSelector

The *drawtype* keyword argument to *RectangleSelector* is removed. From now on, the only behaviour will be `drawtype='box'`.

Support for `drawtype=line` is removed altogether. As a result, the *lineprops* keyword argument to *RectangleSelector* is also removed.

To retain the behaviour of `drawtype='none'`, use `rectprops={'visible': False}` to make the drawn *Rectangle* invisible.

Cleaned up attributes and arguments are:

- The `active_handle` attribute has been privatized and removed.
- The `drawtype` attribute has been privatized and removed.

- The `eventpress` attribute has been privatized and removed.
- The `eventrelease` attribute has been privatized and removed.
- The `interactive` attribute has been privatized and removed.
- The *marker_props* argument is removed, use *handle_props* instead.
- The *maxdist* argument is removed, use *grab_range* instead.
- The *rectprops* argument is removed, use *props* instead.
- The `rectprops` attribute has been privatized and removed.
- The `state` attribute has been privatized and removed.
- The `to_draw` attribute has been privatized and removed.

### PolygonSelector

- The *line* attribute is removed. If you want to change the selector artist properties, use the `set_props` or `set_handle_props` methods.
- The *lineprops* argument is removed, use *props* instead.
- The *markerprops* argument is removed, use *handle_props* instead.
- The *maxdist* argument and attribute is removed, use *grab_range* instead.
- The *vertex_select_radius* argument and attribute is removed, use *grab_range* instead.

### SpanSelector

- The `active_handle` attribute has been privatized and removed.
- The `eventpress` attribute has been privatized and removed.
- The `eventrelease` attribute has been privatized and removed.
- The `pressv` attribute has been privatized and removed.
- The `prev` attribute has been privatized and removed.
- The `rect` attribute has been privatized and removed.
- The *rectprops* parameter has been renamed to *props*.
- The `rectprops` attribute has been privatized and removed.
- The *span_stays* parameter has been renamed to *interactive*.
- The `span_stays` attribute has been privatized and removed.
- The `state` attribute has been privatized and removed.

### LassoSelector

- The *lineprops* argument is removed, use *props* instead.
- The `onpress` and `onrelease` methods are removed. They are straight aliases for `press` and `release`.
- The `matplotlib.widgets.TextBox.DIST_FROM_LEFT` attribute has been removed. It was marked as private in 3.5.

### `backend_template.show`

... has been removed, in order to better demonstrate the new backend definition API.

### Unused positional parameters to `print_<fmt>` methods

None of the `print_<fmt>` methods implemented by canvas subclasses used positional arguments other that the first (the output filename or file-like), so these extra parameters are removed.

### `QuadMesh` signature

The *QuadMesh* signature

```
def __init__(meshWidth, meshHeight, coordinates,
             antialiased=True, shading='flat', **kwargs)
```

is removed and replaced by the new signature

```
def __init__(coordinates, *, antialiased=True, shading='flat', **kwargs)
```

In particular:

- The *coordinates* argument must now be a (M, N, 2) array-like. Previously, the grid shape was separately specified as (*meshHeight* + 1, *meshWidth* + 1) and *coordinates* could be an array-like of any shape with M * N * 2 elements.
- All parameters except *coordinates* are keyword-only now.

### Expiration of `FancyBboxPatch` deprecations

The *FancyBboxPatch* constructor no longer accepts the *bbox_transmuter* parameter, nor can the *boxstyle* parameter be set to "custom" -- instead, directly set *boxstyle* to the relevant boxstyle instance. The *mutation_scale* and *mutation_aspect* parameters have also become keyword-only.

The *mutation_aspect* parameter is now handled internally and no longer passed to the boxstyle callables when mutating the patch path.

### Testing support

#### `matplotlib.test()` has been removed

Run tests using `pytest` from the commandline instead. The variable `matplotlib.default_test_modules` was only used for `matplotlib.test()` and is thus removed as well.

To test an installed copy, be sure to specify both `matplotlib` and `mpl_toolkits` with `--pyargs`:

```
pytest --pyargs matplotlib.tests mpl_toolkits.tests
```

See *Testing* for more details.

#### Auto-removal of grids by `pcolor` and `pcolormesh`

*`pcolor`* and *`pcolormesh`* previously remove any visible axes major grid. This behavior is removed; please explicitly call `ax.grid(False)` to remove the grid.

#### Modification of `Axes` children sublists

See *Axes children are no longer separated by type* for more information; modification of the following sublists is no longer supported:

- `Axes.artists`
- `Axes.collections`
- `Axes.images`
- `Axes.lines`
- `Axes.patches`
- `Axes.tables`
- `Axes.texts`

To remove an Artist, use its *`Artist.remove`* method. To add an Artist, use the corresponding `Axes.add_*` method.

### Passing incorrect types to `Axes.add_*` methods

The following `Axes.add_*` methods will now raise if passed an unexpected type. See their documentation for the types they expect.

- *Axes.add_collection*
- *Axes.add_image*
- *Axes.add_line*
- *Axes.add_patch*
- *Axes.add_table*

### `ConversionInterface.convert` no longer accepts unitless values

Previously, custom subclasses of *units.ConversionInterface* needed to implement a `convert` method that not only accepted instances of the unit, but also unitless values (which are passed through as is). This is no longer the case (`convert` is never called with a unitless value), and such support in `.StrCategoryConverter` is removed. Likewise, the `.ConversionInterface.is_numlike` helper is removed.

Consider calling *Axis.convert_units* instead, which still supports unitless values.

### Normal list of `Artist` objects now returned by `HandlerLine2D.create_artists`

For Matplotlib 3.5 and 3.6 a proxy list was returned that simulated the return of *HandlerLine2DCompound.create_artists*. Now a list containing only the single artist is return.

### rcParams will no longer cast inputs to str

rcParams that expect a (non-pathlike) str no longer cast non-str inputs using `str`. This will avoid confusing errors in subsequent code if e.g. a list input gets implicitly cast to a str.

### Case-insensitive scales

Previously, scales could be set case-insensitively (e.g., `set_xscale("LoG")`). Now all builtin scales use lowercase names.

### Support for `nx1 = None` or `ny1 = None` in `AxesLocator` and `Divider.locate`

In *axes_grid1.axes_divider*, various internal APIs no longer supports passing `nx1 = None` or `ny1 = None` to mean `nx + 1` or `ny + 1`, in preparation for a possible future API which allows indexing and slicing of dividers (possibly `divider[a:b] == divider.new_locator(a, b)`, but also `divider[a:] == divider.new_locator(a, <end>)`). The user-facing *Divider.new_locator* API is unaffected -- it correctly normalizes `nx1 = None` and `ny1 = None` as needed.

### change signature of `.FigureCanvasBase.enter_notify_event`

The *xy* parameter is now required and keyword only. This was deprecated in 3.0 and originally slated to be removed in 3.5.

### `Colorbar` tick update parameters

The *update_ticks* parameter of *Colorbar.set_ticks* and *Colorbar.set_ticklabels* was ignored since 3.5 and has been removed.

### plot directive removals

The public methods:

- `matplotlib.sphinxext.split_code_at_show`
- `matplotlib.sphinxext.unescape_doctest`
- `matplotlib.sphinxext.run_code`

have been removed.

The deprecated *encoding* option to the plot directive has been removed.

### Miscellaneous removals

- `is_url` and `URL_REGEX` are removed. (They were previously defined in the toplevel *matplotlib* module.)
- The `ArrowStyle.beginarrow` and `ArrowStyle.endarrow` attributes are removed; use the `arrow` attribute to define the desired heads and tails of the arrow.
- `backend_pgf.LatexManager.str_cache` is removed.
- `backends.qt_compat.ETS` and `backends.qt_compat.QT_RC_MAJOR_VERSION` are removed, with no replacement.
- The `blocking_input` module is removed. Instead, use `canvas.start_event_loop()` and `canvas.stop_event_loop()` while connecting event callbacks as needed.

- `cbook.report_memory` is removed; use `psutil.virtual_memory` instead.
- `cm.LUTSIZE` is removed. Use *`rcParams["image.lut"]`* (default: `256`) instead. This value only affects colormap quantization levels for default colormaps generated at module import time.
- `Colorbar.patch` is removed; this attribute was not correctly updated anymore.
- `ContourLabeler.get_label_width` is removed.
- `Dvi.baseline` is removed (with no replacement).
- The *format* parameter of `dviread.find_tex_file` is removed (with no replacement).
- `FancyArrowPatch.get_path_in_displaycoord` and `ConnectionPath.get_path_in_displaycoord` are removed. The path in display coordinates can still be obtained, as for other patches, using `patch.get_transform().transform_path(patch.get_path())`.
- The `font_manager.win32InstalledFonts` and `font_manager.get_fontconfig_fonts` helper functions are removed.
- All parameters of `imshow` starting from *aspect* are keyword-only.
- `QuadMesh.convert_mesh_to_paths` and `QuadMesh.convert_mesh_to_triangles` are removed. `QuadMesh.get_paths()` can be used as an alternative for the former; there is no replacement for the latter.
- `ScalarMappable.callbacksSM` is removed. Use `ScalarMappable.callbacks` instead.
- `streamplot.get_integrator` is removed.
- `style.core.STYLE_FILE_PATTERN`, `style.core.load_base_library`, and `style.core.iter_user_libraries` are removed.
- `SubplotParams.validate` is removed. Use *`SubplotParams.update`* to change *`SubplotParams`* while always keeping it in a valid state.
- The `grey_arrayd`, `font_family`, `font_families`, and `font_info` attributes of *`TexManager`* are removed.
- `Text.get_prop_tup` is removed with no replacements (because the *`Text`* class cannot know whether a backend needs to update cache e.g. when the text's color changes).
- `Tick.apply_tickdir` didn't actually update the tick markers on the existing Line2D objects used to draw the ticks and is removed; use *`Axis.set_tick_params`* instead.
- `tight_layout.auto_adjust_subplotpars` is removed.
- The `grid_info` attribute of `axisartist` classes has been removed.
- `axes_grid1.axes_grid.CbarAxes` and `axisartist.axes_grid.CbarAxes` are removed (they are now dynamically generated based on the owning axes class).
- The `axes_grid1.Divider.get_vsize_hsize` and `axes_grid1.Grid.get_vsize_hsize` methods are removed.
- `AxesDivider.append_axes(..., add_to_figure=False)` is removed. Use `ax.remove()` to remove the Axes from the figure if needed.

- `FixedAxisArtistHelper.change_tick_coord` is removed with no replacement.
- `floating_axes.GridHelperCurveLinear.get_boundary` is removed with no replacement.
- `ParasiteAxesBase.get_images_artists` is removed.
- The "units finalize" signal (previously emitted by Axis instances) is removed. Connect to "units" instead.
- Passing formatting parameters positionally to `stem()` is no longer possible.
- `axisartist.clip_path` is removed with no replacement.

### Development changes

#### Windows wheel runtime bundling

Wheels built for Windows now bundle the MSVC runtime DLL `msvcp140.dll`. This enables importing Matplotlib on systems that do not have the runtime installed.

#### Increase to minimum supported versions of dependencies

For Matplotlib 3.7, the *minimum supported versions* are being bumped:

| Dependency | min in mpl3.6 | min in mpl3.7 |
|---|---|---|
| NumPy | 1.19 | 1.20 |
| pyparsing | 2.2.1 | 2.3.1 |
| Qt | | 5.10 |

- There are no wheels or conda packages that support both Qt 5.9 (or older) and Python 3.8 (or newer).

This is consistent with our *Dependency version policy* and NEP29

#### New dependencies

- importlib-resources (>= 3.2.0; only required on Python < 3.10)

#### Maximum line length increased to 88 characters

The maximum line length for new contributions has been extended from 79 characters to 88 characters. This change provides an extra 9 characters to allow code which is a single idea to fit on fewer lines (often a single line). The chosen length is the same as black.

## 9.4 Version 3.6

### 9.4.1 What's new in Matplotlib 3.6.0 (Sep 15, 2022)

For a list of all of the issues and pull requests since the last revision, see the *GitHub statistics for 3.9.0 (May 15, 2024)*.

**Table of Contents**


- *What's new in Matplotlib 3.6.0 (Sep 15, 2022)*
  - *Figure and Axes creation / management*
    - `subplots`, `subplot_mosaic` *accept* height_ratios *and* width_ratios *arguments*
    - *Constrained layout is no longer considered experimental*
    - *New* `layout_engine` *module*
    - *Compressed layout for fixed-aspect ratio Axes*
    - *Layout engines may now be removed*
    - `Axes.inset_axes` *flexibility*
    - *WebP is now a supported output format*
    - *Garbage collection is no longer run on figure close*
  - *Plotting methods*
    - *Striped lines (experimental)*
    - *Custom cap widths in box and whisker plots in* `bxp` *and* `boxplot`
    - *Easier labelling of bars in bar plot*
    - *New style format string for colorbar ticks*
    - *Linestyles for negative contours may be set individually*
    - *Improved quad contour calculations via ContourPy*
    - `errorbar` *supports* markerfacecoloralt
    - `streamplot` *can disable streamline breaks*
    - *New axis scale* `asinh` *(experimental)*

    - *`stairs(..., fill=True)` hides patch edge by setting linewidth*
    - *Fix the dash offset of the Patch class*
    - *Rectangle patch rotation point*
  - *Colors and colormaps*
    - *Color sequence registry*
    - *Colormap method for creating a different lookup table size*
    - *Setting norms with strings*
  - *Titles, ticks, and labels*
    - *`plt.xticks` and `plt.yticks` support* minor *keyword argument*
  - *Legends*
    - *Legend can control alignment of title and handles*
    - ncol *keyword argument to `legend` renamed to* ncols
  - *Markers*
    - *`marker` can now be set to the string "none"*
    - *Customization of `MarkerStyle` join and cap style*
  - *Fonts and Text*
    - *Font fallback*
    - *List of available font names*
    - *`math_to_image` now has a* color *keyword argument*
    - *Active URL area rotates with link text*
  - *rcParams improvements*
    - *Allow setting figure label size and weight globally and separately from title*
    - *Mathtext parsing can be disabled globally*
    - *Double-quoted strings in matplotlibrc*
  - *3D Axes improvements*
    - *Standardized views for primary plane viewing angles*
    - *Custom focal length for 3D camera*
    - *3D plots gained a 3rd "roll" viewing angle*
    - *Equal aspect ratio for 3D plots*
  - *Interactive tool improvements*
    - *Rotation, aspect ratio correction and add/remove state*

        * *`MultiCursor` now supports Axes split over multiple figures*
        * *`PolygonSelector` bounding boxes*
        * *Setting `PolygonSelector` vertices*
        * *`SpanSelector` widget can now be snapped to specified values*
        * *More toolbar icons are styled for dark themes*
    - *Platform-specific changes*
        * *Wx backend uses standard toolbar*
        * *Improvements to macosx backend*
            · *Modifier keys handled more consistently*
            · *`savefig.directory` rcParam support*
            · *`figure.raise_window` rcParam support*
            · *Full-screen toggle support*
            · *Improved animation and blitting support*
        * *macOS application icon applied on Qt backend*
        * *New minimum macOS version*
        * *Windows on ARM support*

### Figure and Axes creation / management

#### `subplots`, `subplot_mosaic` accept *height_ratios* and *width_ratios* arguments

The relative width and height of columns and rows in *`subplots`* and *`subplot_mosaic`* can be controlled by passing *height_ratios* and *width_ratios* keyword arguments to the methods:

```
fig = plt.figure()
axs = fig.subplots(3, 1, sharex=True, height_ratios=[3, 1, 1])
```

Previously, this required passing the ratios in *gridspec_kw* arguments:

```
fig = plt.figure()
axs = fig.subplots(3, 1, sharex=True,
                   gridspec_kw=dict(height_ratios=[3, 1, 1]))
```

### Constrained layout is no longer considered experimental

The constrained layout engine and API is no longer considered experimental. Arbitrary changes to behaviour and API are no longer permitted without a deprecation period.

### New `layout_engine` module

Matplotlib ships with `tight_layout` and `constrained_layout` layout engines. A new *layout_engine* module is provided to allow downstream libraries to write their own layout engines and *Figure* objects can now take a *LayoutEngine* subclass as an argument to the *layout* parameter.

### Compressed layout for fixed-aspect ratio Axes

Simple arrangements of Axes with fixed aspect ratios can now be packed together with `fig, axs = plt.subplots(2, 3, layout='compressed')`.

With `layout='tight'` or `'constrained'`, Axes with a fixed aspect ratio can leave large gaps between each other:

Using the `layout='compressed'` layout reduces the space between the Axes, and adds the extra space to the outer margins:

See *Grids of fixed aspect-ratio Axes: "compressed" layout* for further details.

### Layout engines may now be removed

The layout engine on a Figure may now be removed by calling *Figure.set_layout_engine* with `'none'`. This may be useful after computing layout in order to reduce computations, e.g., for subsequent animation loops.

A different layout engine may be set afterwards, so long as it is compatible with the previous layout engine.

### `Axes.inset_axes` flexibility

*matplotlib.axes.Axes.inset_axes* now accepts the *projection*, *polar* and *axes_class* keyword arguments, so that subclasses of *matplotlib.axes.Axes* may be returned.

```
fig, ax = plt.subplots()

ax.plot([0, 2], [1, 2])

polar_ax = ax.inset_axes([0.75, 0.25, 0.2, 0.2], projection='polar')
polar_ax.plot([0, 2], [1, 2])
```

### WebP is now a supported output format

Figures may now be saved in WebP format by using the `.webp` file extension, or passing `format='webp'` to *savefig*. This relies on Pillow support for WebP.

### Garbage collection is no longer run on figure close

Matplotlib has a large number of circular references (between Figure and Manager, between Axes and Figure, Axes and Artist, Figure and Canvas, etc.) so when the user drops their last reference to a Figure (and clears it from pyplot's state), the objects will not immediately be deleted.

To account for this we have long (since before 2004) had a `gc.collect` (of the lowest two generations only) in the closing code in order to promptly clean up after ourselves. However this is both not doing what we want (as most of our objects will actually survive) and due to clearing out the first generation opened us up to having unbounded memory usage.

In cases with a very tight loop between creating the figure and destroying it (e.g. `plt.figure(); plt.close()`) the first generation will never grow large enough for Python to consider running the collection on the higher generations. This will lead to unbounded memory usage as the long-lived objects are never re-considered to look for reference cycles and hence are never deleted.

We now no longer do any garbage collection when a figure is closed, and rely on Python automatically deciding to run garbage collection periodically. If you have strict memory requirements, you can call `gc.collect` yourself but this may have performance impacts in a tight computation loop.

## Plotting methods

### Striped lines (experimental)

The new *gapcolor* parameter to *plot* enables the creation of striped lines.

```
x = np.linspace(1., 3., 10)
y = x**3

fig, ax = plt.subplots()
ax.plot(x, y, linestyle='--', color='orange', gapcolor='blue',
        linewidth=3, label='a striped line')
ax.legend()
```

### Custom cap widths in box and whisker plots in `bxp` and `boxplot`

The new *capwidths* parameter to *bxp* and *boxplot* allows controlling the widths of the caps in box and whisker plots.

```
x = np.linspace(-7, 7, 140)
x = np.hstack([-25, x, 25])
capwidths = [0.01, 0.2]

fig, ax = plt.subplots()
ax.boxplot([x, x], notch=True, capwidths=capwidths)
ax.set_title(f'{capwidths=}')
```

### Easier labelling of bars in bar plot

The *label* argument of *bar* and *barh* can now be passed a list of labels for the bars. The list must be the same length as *x* and labels the individual bars. Repeated labels are not de-duplicated and will cause repeated label entries, so this is best used when bars also differ in style (e.g., by passing a list to *color*, as below.)

```
x = ["a", "b", "c"]
y = [10, 20, 15]
color = ['C0', 'C1', 'C2']

fig, ax = plt.subplots()
ax.bar(x, y, color=color, label=x)
ax.legend()
```

### New style format string for colorbar ticks

The *format* argument of *colorbar* (and other colorbar methods) now accepts `{}`-style format strings.

```
fig, ax = plt.subplots()
im = ax.imshow(z)
fig.colorbar(im, format='{x:.2e}')  # Instead of '%.2e'
```

### Linestyles for negative contours may be set individually

The line style of negative contours may be set by passing the *negative_linestyles* argument to *Axes.contour*. Previously, this style could only be set globally via *rcParams["contour.negative_linestyles"]*.

```
delta = 0.025
x = np.arange(-3.0, 3.0, delta)
y = np.arange(-2.0, 2.0, delta)
X, Y = np.meshgrid(x, y)
Z1 = np.exp(-X**2 - Y**2)
Z2 = np.exp(-(X - 1)**2 - (Y - 1)**2)
Z = (Z1 - Z2) * 2

fig, axs = plt.subplots(1, 2)

CS = axs[0].contour(X, Y, Z, 6, colors='k')
axs[0].clabel(CS, fontsize=9, inline=True)
axs[0].set_title('Default negative contours')

CS = axs[1].contour(X, Y, Z, 6, colors='k', negative_linestyles='dotted')
axs[1].clabel(CS, fontsize=9, inline=True)
axs[1].set_title('Dotted negative contours')
```

### Improved quad contour calculations via ContourPy

The contouring functions *contour* and *contourf* have a new keyword argument *algorithm* to control which algorithm is used to calculate the contours. There is a choice of four algorithms to use, and the default is to use `algorithm='mpl2014'` which is the same algorithm that Matplotlib has been using since 2014.

Previously Matplotlib shipped its own C++ code for calculating the contours of quad grids. Now the external library ContourPy is used instead.

Other possible values of the *algorithm* keyword argument at this time are `'mpl2005'`, `'serial'` and `'threaded'`; see the ContourPy documentation for further details.

---

**Note:** Contour lines and polygons produced by `algorithm='mpl2014'` will be the same as those produced before this change to within floating-point tolerance. The exception is for duplicate points, i.e. contours containing adjacent (x, y) points that are identical; previously the duplicate points were removed, now they are kept. Contours affected by this will produce the same visual output, but there will be a greater number of points in the contours.

The locations of contour labels obtained by using *clabel* may also be different.

---

### `errorbar` supports *markerfacecoloralt*

The *markerfacecoloralt* parameter is now passed to the line plotter from *Axes.errorbar*. The documentation now accurately lists which properties are passed to *Line2D*, rather than claiming that all keyword arguments are passed on.

```
x = np.arange(0.1, 4, 0.5)
y = np.exp(-x)

fig, ax = plt.subplots()
ax.errorbar(x, y, xerr=0.2, yerr=0.4,
            linestyle=':', color='darkgrey',
            marker='o', markersize=20, fillstyle='left',
            markerfacecolor='tab:blue', markerfacecoloralt='tab:orange',
            markeredgecolor='tab:brown', markeredgewidth=2)
```

### `streamplot` can disable streamline breaks

It is now possible to specify that streamplots have continuous, unbroken streamlines. Previously streamlines would end to limit the number of lines within a single grid cell. See the difference between the plots below:

### New axis scale `asinh` (experimental)

The new `asinh` axis scale offers an alternative to `symlog` that smoothly transitions between the quasi-linear and asymptotically logarithmic regions of the scale. This is based on an arcsinh transformation that allows plotting both positive and negative values that span many orders of magnitude.

### `stairs(..., fill=True)` hides patch edge by setting linewidth

`stairs(..., fill=True)` would previously hide Patch edges by setting `edgecolor="none"`. Consequently, calling `set_color()` on the Patch later would make the Patch appear larger.

Now, by using `linewidth=0`, this apparent size change is prevented. Likewise calling `stairs(..., fill=True, linewidth=3)` will behave more transparently.

### Fix the dash offset of the Patch class

Formerly, when setting the line style on a *Patch* object using a dash tuple, the offset was ignored. Now the offset is applied to the Patch as expected and it can be used as it is used with *Line2D* objects.

### Rectangle patch rotation point

The rotation point of the *Rectangle* can now be set to 'xy', 'center' or a 2-tuple of numbers using the *rotation_point* argument.

## Colors and colormaps

### Color sequence registry

The color sequence registry, *ColorSequenceRegistry*, contains sequences (i.e., simple lists) of colors that are known to Matplotlib by name. This will not normally be used directly, but through the universal instance at *matplotlib.color_sequences*.

### Colormap method for creating a different lookup table size

The new method *Colormap.resampled* creates a new *Colormap* instance with the specified lookup table size. This is a replacement for manipulating the lookup table size via `get_cmap`.

Use:

```
get_cmap(name).resampled(N)
```

instead of:

```
get_cmap(name, lut=N)
```

### Setting norms with strings

Norms can now be set (e.g. on images) using the string name of the corresponding scale, e.g. `imshow(array, norm="log")`. Note that in that case, it is permissible to also pass *vmin* and *vmax*, as a new Norm instance will be created under the hood.

### Titles, ticks, and labels

#### `plt.xticks` and `plt.yticks` support *minor* keyword argument

It is now possible to set or get minor ticks using *pyplot.xticks* and *pyplot.yticks* by setting `minor=True`.

```
plt.figure()
plt.plot([1, 2, 3, 3.5], [2, 1, 0, -0.5])
plt.xticks([1, 2, 3], ["One", "Zwei", "Trois"])
plt.xticks([np.sqrt(2), 2.5, np.pi],
           [r"$\sqrt{2}$", r"$\frac{5}{2}$", r"$\pi$"], minor=True)
```

### Legends

#### Legend can control alignment of title and handles

*Legend* now supports controlling the alignment of the title and handles via the keyword argument *alignment*. You can also use *Legend.set_alignment* to control the alignment on existing Legends.

```
fig, axs = plt.subplots(3, 1)
for i, alignment in enumerate(['left', 'center', 'right']):
    axs[i].plot(range(10), label='test')
    axs[i].legend(title=f'{alignment=}', alignment=alignment)
```

#### *ncol* keyword argument to `legend` renamed to *ncols*

The *ncol* keyword argument to *legend* for controlling the number of columns is renamed to *ncols* for consistency with the *ncols* and *nrows* keywords of *subplots* and *GridSpec*. *ncol* remains supported for backwards compatibility, but is discouraged.

### Markers

#### `marker` can now be set to the string "none"

The string "none" means *no-marker*, consistent with other APIs which support the lowercase version. Using "none" is recommended over using "None", to avoid confusion with the None object.

### Customization of `MarkerStyle` join and cap style

New *MarkerStyle* parameters allow control of join style and cap style, and for the user to supply a transformation to be applied to the marker (e.g. a rotation).

```
from matplotlib.markers import CapStyle, JoinStyle, MarkerStyle
from matplotlib.transforms import Affine2D

fig, axs = plt.subplots(3, 1, layout='constrained')
for ax in axs:
    ax.axis('off')
    ax.set_xlim(-0.5, 2.5)

axs[0].set_title('Cap styles', fontsize=14)
for col, cap in enumerate(CapStyle):
    axs[0].plot(col, 0, markersize=32, markeredgewidth=8,
                marker=MarkerStyle('1', capstyle=cap))
    # Show the marker edge for comparison with the cap.
    axs[0].plot(col, 0, markersize=32, markeredgewidth=1,
                markerfacecolor='none', markeredgecolor='lightgrey',
                marker=MarkerStyle('1'))
    axs[0].annotate(cap.name, (col, 0),
                    xytext=(20, -5), textcoords='offset points')

axs[1].set_title('Join styles', fontsize=14)
for col, join in enumerate(JoinStyle):
    axs[1].plot(col, 0, markersize=32, markeredgewidth=8,
                marker=MarkerStyle('*', joinstyle=join))
    # Show the marker edge for comparison with the join.
    axs[1].plot(col, 0, markersize=32, markeredgewidth=1,
                markerfacecolor='none', markeredgecolor='lightgrey',
                marker=MarkerStyle('*'))
    axs[1].annotate(join.name, (col, 0),
                    xytext=(20, -5), textcoords='offset points')

axs[2].set_title('Arbitrary transforms', fontsize=14)
for col, (size, rot) in enumerate(zip([2, 5, 7], [0, 45, 90])):
    t = Affine2D().rotate_deg(rot).scale(size)
    axs[2].plot(col, 0, marker=MarkerStyle('*', transform=t))
```

## Fonts and Text

### Font fallback

It is now possible to specify a list of fonts families and Matplotlib will try them in order to locate a required glyph.

```
plt.rcParams["font.size"] = 20
fig = plt.figure(figsize=(4.75, 1.85))
```

(continues on next page)

(continued from previous page)

```
text = "There are □□□□ in between!"
fig.text(0.05, 0.65, text, family=["Noto Sans CJK JP", "Noto Sans TC"])
fig.text(0.05, 0.45, text, family=["DejaVu Sans", "Noto Sans CJK JP", "Noto␣
 ↪Sans TC"])
```

This currently works with the Agg (and all of the GUI embeddings), svg, pdf, ps, and inline backends.

### List of available font names

The list of available fonts are now easily accessible. To get a list of the available font names in Matplotlib use:

```
from matplotlib import font_manager
font_manager.get_font_names()
```

### `math_to_image` now has a *color* keyword argument

To easily support external libraries that rely on the MathText rendering of Matplotlib to generate equation images, a *color* keyword argument was added to *`math_to_image`*.

```
from matplotlib import mathtext
mathtext.math_to_image('$x^2$', 'filename.png', color='Maroon')
```

### Active URL area rotates with link text

When link text is rotated in a figure, the active URL area will now include the rotated link area. Previously, the active area remained in the original, non-rotated, position.

### rcParams improvements

### Allow setting figure label size and weight globally and separately from title

For figure labels, `Figure.supxlabel` and `Figure.supylabel`, the size and weight can be set separately from the figure title using *`rcParams["figure.labelsize"]`* (default: `'large'`) and *`rcParams["figure.labelweight"]`* (default: `'normal'`).

```
# Original (previously combined with below) rcParams:
plt.rcParams['figure.titlesize'] = 64
plt.rcParams['figure.titleweight'] = 'bold'

# New rcParams:
```

(continues on next page)

(continued from previous page)

```
plt.rcParams['figure.labelsize'] = 32
plt.rcParams['figure.labelweight'] = 'bold'

fig, axs = plt.subplots(2, 2, layout='constrained')
for ax in axs.flat:
    ax.set(xlabel='xlabel', ylabel='ylabel')

fig.suptitle('suptitle')
fig.supxlabel('supxlabel')
fig.supylabel('supylabel')
```

Note that if you have changed *rcParams["figure.titlesize"]* (default: `'large'`) or *rcParams["figure.titleweight"]* (default: `'normal'`), you must now also change the introduced parameters for a result consistent with past behaviour.

### Mathtext parsing can be disabled globally

The *rcParams["text.parse_math"]* (default: `True`) setting may be used to disable parsing of mathtext in all *Text* objects (most notably from the *Axes.text* method).

### Double-quoted strings in matplotlibrc

You can now use double-quotes around strings. This allows using the '#' character in strings. Without quotes, '#' is interpreted as start of a comment. In particular, you can now define hex-colors:

```
grid.color: "#b0b0b0"
```

### 3D Axes improvements

### Standardized views for primary plane viewing angles

When viewing a 3D plot in one of the primary view planes (i.e., perpendicular to the XY, XZ, or YZ planes), the Axis will be displayed in a standard location. For further information on 3D views, see *mplot3d View Angles* and *Primary 3D view planes*.

### Custom focal length for 3D camera

The 3D Axes can now better mimic real-world cameras by specifying the focal length of the virtual camera. The default focal length of 1 corresponds to a Field of View (FOV) of 90°, and is backwards-compatible with existing 3D plots. An increased focal length between 1 and infinity "flattens" the image, while a decreased focal length between 1 and 0 exaggerates the perspective and gives the image more apparent depth.

The focal length can be calculated from a desired FOV via the equation:

$$focal\_length = 1/\tan(FOV/2) \tag{9.3}$$

```
from mpl_toolkits.mplot3d import axes3d

X, Y, Z = axes3d.get_test_data(0.05)

fig, axs = plt.subplots(1, 3, figsize=(7, 4),
                        subplot_kw={'projection': '3d'})

for ax, focal_length in zip(axs, [0.2, 1, np.inf]):
    ax.plot_wireframe(X, Y, Z, rstride=10, cstride=10)
    ax.set_proj_type('persp', focal_length=focal_length)
    ax.set_title(f"{focal_length=}")
```

### 3D plots gained a 3rd "roll" viewing angle

3D plots can now be viewed from any orientation with the addition of a 3rd roll angle, which rotates the plot about the viewing axis. Interactive rotation using the mouse still only controls elevation and azimuth, meaning that this feature is relevant to users who create more complex camera angles programmatically. The default roll angle of 0 is backwards-compatible with existing 3D plots.

```
from mpl_toolkits.mplot3d import axes3d

X, Y, Z = axes3d.get_test_data(0.05)

fig, ax = plt.subplots(subplot_kw={'projection': '3d'})

ax.plot_wireframe(X, Y, Z, rstride=10, cstride=10)
ax.view_init(elev=0, azim=0, roll=30)
ax.set_title('elev=0, azim=0, roll=30')
```

### Equal aspect ratio for 3D plots

Users can set the aspect ratio for the X, Y, Z axes of a 3D plot to be 'equal', 'equalxy', 'equalxz', or 'equalyz' rather than the default of 'auto'.

```
from itertools import combinations, product

aspects = [
    ['auto', 'equal', '.'],
    ['equalxy', 'equalyz', 'equalxz'],
]
fig, axs = plt.subplot_mosaic(aspects, figsize=(7, 6),
                              subplot_kw={'projection': '3d'})

# Draw rectangular cuboid with side lengths [1, 1, 5]
r = [0, 1]
scale = np.array([1, 1, 5])
pts = combinations(np.array(list(product(r, r, r))), 2)
for start, end in pts:
    if np.sum(np.abs(start - end)) == r[1] - r[0]:
        for ax in axs.values():
            ax.plot3D(*zip(start*scale, end*scale), color='C0')

# Set the aspect ratios
for aspect, ax in axs.items():
    ax.set_box_aspect((3, 4, 5))
    ax.set_aspect(aspect)
    ax.set_title(f'set_aspect({aspect!r})')
```

### Interactive tool improvements

### Rotation, aspect ratio correction and add/remove state

The *RectangleSelector* and *EllipseSelector* can now be rotated interactively between -45° and 45°. The range limits are currently dictated by the implementation. The rotation is enabled or disabled by striking the *r* key ('r' is the default key mapped to 'rotate' in *state_modifier_keys*) or by calling `selector.add_state('rotate')`.

The aspect ratio of the axes can now be taken into account when using the "square" state. This is enabled by specifying `use_data_coordinates='True'` when the selector is initialized.

In addition to changing selector state interactively using the modifier keys defined in *state_modifier_keys*, the selector state can now be changed programmatically using the *add_state* and *remove_state* methods.

```
from matplotlib.widgets import RectangleSelector

values = np.arange(0, 100)

fig = plt.figure()
```

(continues on next page)

(continued from previous page)

```
ax = fig.add_subplot()
ax.plot(values, values)

selector = RectangleSelector(ax, print, interactive=True,
                             drag_from_anywhere=True,
                             use_data_coordinates=True)
selector.add_state('rotate')  # alternatively press 'r' key
# rotate the selector interactively

selector.remove_state('rotate')  # alternatively press 'r' key

selector.add_state('square')
```

### `MultiCursor` now supports Axes split over multiple figures

Previously, *`MultiCursor`* only worked if all target Axes belonged to the same figure.

As a consequence of this change, the first argument to the *`MultiCursor`* constructor has become unused (it was previously the joint canvas of all Axes, but the canvases are now directly inferred from the list of Axes).

### `PolygonSelector` bounding boxes

*`PolygonSelector`* now has a *draw_bounding_box* argument, which when set to `True` will draw a bounding box around the polygon once it is complete. The bounding box can be resized and moved, allowing the points of the polygon to be easily resized.

### Setting `PolygonSelector` vertices

The vertices of *`PolygonSelector`* can now be set programmatically by using the *`PolygonSelector.verts`* property. Setting the vertices this way will reset the selector, and create a new complete selector with the supplied vertices.

### `SpanSelector` widget can now be snapped to specified values

The *`SpanSelector`* widget can now be snapped to values specified by the *snap_values* argument.

#### More toolbar icons are styled for dark themes

On the macOS and Tk backends, toolbar icons will now be inverted when using a dark theme.

### Platform-specific changes

#### Wx backend uses standard toolbar

Instead of a custom sizer, the toolbar is set on Wx windows as a standard toolbar.

### Improvements to macosx backend

#### Modifier keys handled more consistently

The macosx backend now handles modifier keys in a manner more consistent with other backends. See the table in *Event connections* for further information.

#### `savefig.directory` rcParam support

The macosx backend will now obey the *rcParams["savefig.directory"]* (default: `'~'`) setting. If set to a non-empty string, then the save dialog will default to this directory, and preserve subsequent save directories as they are changed.

#### `figure.raise_window` rcParam support

The macosx backend will now obey the *rcParams["figure.raise_window"]* (default: `True`) setting. If set to False, figure windows will not be raised to the top on update.

#### Full-screen toggle support

As supported on other backends, the macosx backend now supports toggling fullscreen view. By default, this view can be toggled by pressing the `f` key.

#### Improved animation and blitting support

The macosx backend has been improved to fix blitting, animation frames with new artists, and to reduce unnecessary draw calls.

#### macOS application icon applied on Qt backend

When using the Qt-based backends on macOS, the application icon will now be set, as is done on other backends/platforms.

#### New minimum macOS version

The macosx backend now requires macOS >= 10.12.

#### Windows on ARM support

Preliminary support for Windows on arm64 target has been added. This support requires FreeType 2.11 or above.

No binary wheels are available yet but it may be built from source.

### 9.4.2 API Changes for 3.6.1

#### Deprecations

#### Colorbars for orphaned mappables are deprecated, but no longer raise

Before 3.6.0, Colorbars for mappables that do not have a parent Axes would steal space from the current Axes. 3.6.0 raised an error on this, but without a deprecation cycle. For 3.6.1 this is reverted; the current Axes is used, but a deprecation warning is shown instead. In this undetermined case, users and libraries should explicitly specify what Axes they want space to be stolen from: `fig.colorbar(mappable, ax=plt.gca())`.

### 9.4.3 API Changes for 3.6.0


- *Behaviour changes*
- *Deprecations*
- *Removals*
- *Development changes*

### Behaviour changes

#### `plt.get_cmap` and `matplotlib.cm.get_cmap` return a copy

Formerly, *get_cmap* and `matplotlib.cm.get_cmap` returned a global version of a *Colormap*. This was prone to errors as modification of the colormap would propagate from one location to another without warning. Now, a new copy of the colormap is returned.

#### Large `imshow` images are now downsampled

When showing an image using *imshow* that has more than $2^{24}$ columns or $2^{23}$ rows, the image will now be downsampled to below this resolution before being resampled for display by the AGG renderer. Previously such a large image would be shown incorrectly. To prevent this downsampling and the warning it raises, manually downsample your data before handing it to *imshow*.

#### Default date limits changed to 1970-01-01 – 1970-01-02

Previously the default limits for an empty axis set up for dates (*Axis.axis_date*) was 2000-01-01 to 2010-01-01. This has been changed to 1970-01-01 to 1970-01-02. With the default epoch, this makes the numeric limit for date axes the same as for other axes (0.0-1.0), and users are less likely to set a locator with far too many ticks.

#### *markerfmt* argument to `stem`

The behavior of the *markerfmt* parameter of *stem* has changed:

- If *markerfmt* does not contain a color, the color is taken from *linefmt*.
- If *markerfmt* does not contain a marker, the default is 'o'.

Before, *markerfmt* was passed unmodified to `plot(..., fmt)`, which had a number of unintended side-effects; e.g. only giving a color switched to a solid line without markers.

For a simple call `stem(x, y)` without parameters, the new rules still reproduce the old behavior.

#### `get_ticklabels` now always populates labels

Previously *Axis.get_ticklabels* (and *Axes.get_xticklabels*, *Axes.get_yticklabels*) would only return empty strings unless a draw had already been performed. Now the ticks and their labels are updated when the labels are requested.

### Warning when scatter plot color settings discarded

When making an animation of a scatter plot, if you don't set *c* (the color value parameter) when initializing the artist, the color settings are ignored. `Axes.scatter` now raises a warning if color-related settings are changed without setting *c*.

### 3D `contourf` polygons placed between levels

The polygons used in a 3D `contourf` plot are now placed halfway between the contour levels, as each polygon represents the location of values that lie between two levels.

### Axes title now avoids y-axis offset

Previously, Axes titles could overlap the y-axis offset text, which is often in the upper left corner of the axes. Now titles are moved above the offset text if overlapping when automatic title positioning is in effect (i.e. if *y* in `Axes.set_title` is *None* and `rcParams["axes.titley"]` (default: `None`) is also *None*).

### Dotted operators gain extra space in mathtext

In mathtext, `\doteq \doteqdot \dotminus \dotplus \dots` are now surrounded by extra space because they are correctly treated as relational or binary operators.

### *math* parameter of `mathtext.get_unicode_index` defaults to False

In math mode, ASCII hyphens (U+002D) are now replaced by Unicode minus signs (U+2212) at the parsing stage.

### `ArtistList` proxies copy contents on iteration

When iterating over the contents of the dynamically generated proxy lists for the Artist-type accessors (see *Axes children are no longer separated by type*), a copy of the contents is made. This ensure that artists can safely be added or removed from the Axes while iterating over their children.

This is a departure from the expected behavior of mutable iterable data types in Python — iterating over a list while mutating it has surprising consequences and dictionaries will error if they change size during iteration. Because all of the accessors are filtered views of the same underlying list, it is possible for seemingly unrelated changes, such as removing a Line, to affect the iteration over any of the other accessors. In this case, we have opted to make a copy of the relevant children before yielding them to the user.

This change is also consistent with our plan to make these accessors immutable in Matplotlib 3.7.

### `AxesImage` string representation

The string representation of *AxesImage* changes from stating the position in the figure `"AxesImage(80,52.8;496x369.6)"` to giving the number of pixels `"AxesImage(size=(300, 200))"`.

### Improved autoscaling for Bézier curves

Bézier curves are now autoscaled to their extents - previously they were autoscaled to their ends and control points, which in some cases led to unnecessarily large limits.

### `QuadMesh` mouseover defaults to False

New in 3.5, *QuadMesh.get_cursor_data* allows display of data values under the cursor. However, this can be very slow for large meshes, so mouseover now defaults to *False*.

### Changed pgf backend document class

The pgf backend now uses the `article` document class as basis for compilation.

### `MathtextBackendAgg.get_results` no longer returns `used_characters`

The last item (`used_characters`) in the tuple returned by `MathtextBackendAgg.get_results` has been removed. In order to unpack this tuple in a backward and forward-compatible way, use e.g. `ox, oy, width, height, descent, image, *_ = parse(...)`, which will ignore `used_characters` if it was present.

### `Type1Font` objects include more properties

The `matplotlib._type1font.Type1Font.prop` dictionary now includes more keys, such as `CharStrings` and `Subrs`. The value of the `Encoding` key is now a dictionary mapping codes to glyph names. The `matplotlib._type1font.Type1Font.transform` method now correctly removes `UniqueID` properties from the font.

### `rcParams.copy()` returns `RcParams` rather than `dict`

Returning an *RcParams* instance from *RcParams.copy* makes the copy still validate inputs, and additionally avoids emitting deprecation warnings when using a previously copied instance to update the global instance (even if some entries are deprecated).

### `rc_context` no longer resets the value of `'backend'`

*matplotlib.rc_context* incorrectly reset the value of *rcParams["backend"]* if backend resolution was triggered in the context. This affected only the value. The actual backend was not changed. Now, *matplotlib.rc_context* does not reset *rcParams["backend"]* anymore.

### Default `rcParams["animation.convert_args"]` changed

It now defaults to `["-layers", "OptimizePlus"]` to try to generate smaller GIFs. Set it back to an empty list to recover the previous behavior.

### Style file encoding now specified to be UTF-8

It has been impossible to import Matplotlib with a non UTF-8 compatible locale encoding because we read the style library at import time. This change is formalizing and documenting the status quo so there is no deprecation period.

### MacOSX backend uses sRGB instead of GenericRGB color space

MacOSX backend now display sRGB tagged image instead of GenericRGB which is an older (now deprecated) Apple color space. This is the source color space used by ColorSync to convert to the current display profile.

### Renderer optional for `get_tightbbox` and `get_window_extent`

The *Artist.get_tightbbox* and *Artist.get_window_extent* methods no longer require the *renderer* keyword argument, saving users from having to query it from `fig.canvas.get_renderer`. If the *renderer* keyword argument is not supplied, these methods first check if there is a cached renderer from a previous draw and use that. If there is no cached renderer, then the methods will use `fig.canvas.get_renderer()` as a fallback.

#### `FigureFrameWx` constructor, subclasses, and `get_canvas`

The `FigureCanvasWx` constructor gained a *canvas_class* keyword-only parameter which specifies the canvas class that should be used. This parameter will become required in the future. The `get_canvas` method, which was previously used to customize canvas creation, is deprecated. The `FigureFrameWxAgg` and `FigureFrameWxCairo` subclasses, which overrode `get_canvas`, are deprecated.

#### `FigureFrameWx.sizer`

... has been removed. The frame layout is no longer based on a sizer, as the canvas is now the sole child widget; the toolbar is now a regular toolbar added using `SetToolBar`.

### Incompatible layout engines raise

You cannot switch between `tight_layout` and `constrained_layout` if a colorbar has already been added to a figure. Invoking the incompatible layout engine used to warn, but now raises with a `RuntimeError`.

#### `CallbackRegistry` raises on unknown signals

When Matplotlib instantiates a *`CallbackRegistry`*, it now limits callbacks to the signals that the registry knows about. In practice, this means that calling *`mpl_connect`* with an invalid signal name now raises a `ValueError`.

### Changed exception type for incorrect SVG date metadata

Providing date metadata with incorrect type to the SVG backend earlier resulted in a `ValueError`. Now, a `TypeError` is raised instead.

### Specified exception types in `Grid`

In a few cases an `Exception` was thrown when an incorrect argument value was set in the *`mpl_toolkits.axes_grid1.axes_grid.Grid`* (= *`mpl_toolkits.axisartist.axes_grid.Grid`*) constructor. These are replaced as follows:

- Providing an incorrect value for *ngrids* now raises a `ValueError`
- Providing an incorrect type for *rect* now raises a `TypeError`

## Deprecations

### Parameters to `plt.figure()` and the `Figure` constructor

All parameters to *pyplot.figure* and the *Figure* constructor, other than *num*, *figsize*, and *dpi*, will become keyword-only after a deprecation period.

### Deprecation aliases in cbook

The module `matplotlib.cbook.deprecation` was previously deprecated in Matplotlib 3.4, along with deprecation-related API in `matplotlib.cbook`. Due to technical issues, `matplotlib.cbook.MatplotlibDeprecationWarning` and `matplotlib.cbook.mplDeprecation` did not raise deprecation warnings on use. Changes in Python have now made it possible to warn when these aliases are being used.

In order to avoid downstream breakage, these aliases will now warn, and their removal has been pushed from 3.6 to 3.8 to give time to notice said warnings. As replacement, please use *matplotlib.MatplotlibDeprecationWarning*.

### `Axes` subclasses should override `clear` instead of `cla`

For clarity, *axes.Axes.clear* is now preferred over *Axes.cla*. However, for backwards compatibility, the latter will remain as an alias for the former.

For additional compatibility with third-party libraries, Matplotlib will continue to call the `cla` method of any *Axes* subclasses if they define it. In the future, this will no longer occur, and Matplotlib will only call the `clear` method in *Axes* subclasses.

It is recommended to define only the `clear` method when on Matplotlib 3.6, and only `cla` for older versions.

### Pending deprecation top-level cmap registration and access functions in `mpl.cm`

As part of a multi-step process we are refactoring the global state for managing the registered colormaps.

In Matplotlib 3.5 we added a *ColormapRegistry* class and exposed an instance at the top level as `matplotlib.colormaps`. The existing top level functions in *matplotlib.cm* (`get_cmap`, `register_cmap`, `unregister_cmap`) were changed to be aliases around the same instance.

In Matplotlib 3.6 we have marked those top level functions as pending deprecation with the intention of deprecation in Matplotlib 3.7. The following functions have been marked for pending deprecation:

- `matplotlib.cm.get_cmap`; use `matplotlib.colormaps[name]` instead if you have a `str`.

  **Added 3.6.1** Use *matplotlib.cm.ColormapRegistry.get_cmap* if you have a string, `None` or a *matplotlib.colors.Colormap* object that you want to convert to a *matplotlib.colors.Colormap* instance.

- `matplotlib.cm.register_cmap`; use *matplotlib.colormaps.register* instead
- `matplotlib.cm.unregister_cmap`; use *matplotlib.colormaps.unregister* instead
- `matplotlib.pyplot.register_cmap`; use *matplotlib.colormaps.register* instead

The *matplotlib.pyplot.get_cmap* function will stay available for backward compatibility.

### Pending deprecation of layout methods

The methods *set_tight_layout*, *set_constrained_layout*, are discouraged, and now emit a `PendingDeprecationWarning` in favor of explicitly referencing the layout engine via `figure.set_layout_engine('tight')` and `figure.set_layout_engine('constrained')`. End users should not see the warning, but library authors should adjust.

The methods *set_constrained_layout_pads* and *get_constrained_layout_pads* are will be deprecated in favor of `figure.get_layout_engine().set()` and `figure.get_layout_engine().get()`, and currently emit a `PendingDeprecationWarning`.

### seaborn styles renamed

Matplotlib currently ships many style files inspired from the seaborn library ("seaborn", "seaborn-bright", "seaborn-colorblind", etc.) but they have gone out of sync with the library itself since the release of seaborn 0.9. To prevent confusion, the style files have been renamed "seaborn-v0_8", "seaborn-v0_8-bright", "seaborn-v0_8-colorblind", etc. Users are encouraged to directly use seaborn to access the up-to-date styles.

### Auto-removal of overlapping Axes by `plt.subplot` and `plt.subplot2grid`

Previously, *pyplot.subplot* and *pyplot.subplot2grid* would automatically remove preexisting Axes that overlap with the newly added Axes. This behavior was deemed confusing, and is now deprecated. Explicitly call `ax.remove()` on Axes that need to be removed.

### Passing *linefmt* positionally to `stem` is undeprecated

Positional use of all formatting parameters in *stem* has been deprecated since Matplotlib 3.5. This deprecation is relaxed so that one can still pass *linefmt* positionally, i.e. `stem(x, y, 'r')`.

#### `stem(..., use_line_collection=False)`

... is deprecated with no replacement. This was a compatibility fallback to a former more inefficient representation of the stem lines.

### Positional / keyword arguments

Passing all but the very few first arguments positionally in the constructors of Artists is deprecated. Most arguments will become keyword-only in a future version.

Passing too many positional arguments to `tripcolor` is now deprecated (extra arguments were previously silently ignored).

Passing *emit* and *auto* parameters of `set_xlim`, `set_ylim`, `set_zlim`, `set_rlim` positionally is deprecated; they will become keyword-only in a future release.

The *transOffset* parameter of *`Collection.set_offset_transform`* and the various `create_collection` methods of legend handlers has been renamed to *offset_transform* (consistently with the property name).

Calling `MarkerStyle()` with no arguments or `MarkerStyle(None)` is deprecated; use `MarkerStyle("")` to construct an empty marker style.

`Axes.get_window_extent` / `Figure.get_window_extent` accept only *renderer*. This aligns the API with the general *`Artist.get_window_extent`* API. All other parameters were ignored anyway.

The *cleared* parameter of `get_renderer`, which only existed for AGG-based backends, has been deprecated. Use `renderer.clear()` instead to explicitly clear the renderer buffer.

### Methods to set parameters in `LogLocator` and `LogFormatter*`

In *`LogFormatter`* and derived subclasses, the methods `base` and `label_minor` for setting the respective parameter are deprecated and replaced by `set_base` and `set_label_minor`, respectively.

In *`LogLocator`*, the methods `base` and `subs` for setting the respective parameter are deprecated. Instead, use `set_params(base=..., subs=...)`.

#### `Axes.get_renderer_cache`

The canvas now takes care of the renderer and whether to cache it or not. The alternative is to call `axes.figure.canvas.get_renderer()`.

### Groupers from `get_shared_x_axes` / `get_shared_y_axes` will be immutable

Modifications to the Groupers returned by `get_shared_x_axes` and `get_shared_y_axes` are deprecated. In the future, these methods will return immutable views on the grouper structures. Note that previously, calling e.g. `join()` would already fail to set up the correct structures for sharing axes; use *`Axes.sharex`* or *`Axes.sharey`* instead.

### Unused methods in `Axis`, `Tick`, `XAxis`, and `YAxis`

`Tick.label` has been pending deprecation since 3.1 and is now deprecated. Use `Tick.label1` instead.

The following methods are no longer used and deprecated without a replacement:

- `Axis.get_ticklabel_extents`
- `Tick.get_pad_pixels`
- `XAxis.get_text_heights`
- `YAxis.get_text_widths`

### `mlab.stride_windows`

... is deprecated. Use `np.lib.stride_tricks.sliding_window_view` instead (or `np.lib.stride_tricks.as_strided` on NumPy < 1.20).

### Event handlers

The `draw_event`, `resize_event`, `close_event`, `key_press_event`, `key_release_event`, `pick_event`, `scroll_event`, `button_press_event`, `button_release_event`, `motion_notify_event`, `enter_notify_event` and `leave_notify_event` methods of *`FigureCanvasBase`* are deprecated. They had inconsistent signatures across backends, and made it difficult to improve event metadata.

In order to trigger an event on a canvas, directly construct an *`Event`* object of the correct class and call `canvas.callbacks.process(event.name, event)`.

### Widgets

All parameters to `MultiCursor` starting from *useblit* are becoming keyword-only (passing them positionally is deprecated).

The `canvas` and `background` attributes of `MultiCursor` are deprecated with no replacement.

The *visible* attribute of Selector widgets has been deprecated; use `set_visible` or `get_visible` instead.

The *state_modifier_keys* attribute of Selector widgets has been privatized and the modifier keys must be set when creating the widget.

#### `Axes3D.dist`

... has been privatized. Use the *zoom* keyword argument in *`Axes3D.set_box_aspect`* instead.

### 3D Axis

The previous constructor of *`axis3d.Axis`*, with signature `(self, adir, v_intervalx, d_intervalx, axes, *args, rotate_label=None, **kwargs)` is deprecated in favor of a new signature closer to the one of 2D Axis; it is now `(self, axes, *, rotate_label=None, **kwargs)` where `kwargs` are forwarded to the 2D Axis constructor. The axis direction is now inferred from the axis class' `axis_name` attribute (as in the 2D case); the `adir` attribute is deprecated.

The `init3d` method of 3D Axis is also deprecated; all the relevant initialization is done as part of the constructor.

The `d_interval` and `v_interval` attributes of 3D Axis are deprecated; use `get_data_interval` and `get_view_interval` instead.

The `w_xaxis`, `w_yaxis`, and `w_zaxis` attributes of `Axis3D` have been pending deprecation since 3.1. They are now deprecated. Instead use `xaxis`, `yaxis`, and `zaxis`.

`mplot3d.axis3d.Axis.set_pane_pos` is deprecated. This is an internal method where the provided values are overwritten during drawing. Hence, it does not serve any purpose to be directly accessible.

The two helper functions `mplot3d.axis3d.move_from_center` and `mplot3d.axis3d.tick_update_position` are considered internal and deprecated. If these are required, please vendor the code from the corresponding private methods `_move_from_center` and `_tick_update_position`.

#### `Figure.callbacks` is deprecated

The Figure `callbacks` property is deprecated. The only signal was "dpi_changed", which can be replaced by connecting to the "resize_event" on the canvas `figure.canvas.mpl_connect("resize_event", func)` instead.

#### `FigureCanvas` without a `required_interactive_framework` attribute

Support for such canvas classes is deprecated. Note that canvas classes which inherit from `FigureCanvasBase` always have such an attribute.

### Backend-specific deprecations

- `backend_gtk3.FigureManagerGTK3Agg` and `backend_gtk4.FigureManagerGTK4Agg`; directly use `backend_gtk3.FigureManagerGTK3` and `backend_gtk4.FigureManagerGTK4` instead.
- The *window* parameter to `backend_gtk3.NavigationToolbar2GTK3` had no effect, and is now deprecated.
- `backend_gtk3.NavigationToolbar2GTK3.win`
- `backend_gtk3.RendererGTK3Cairo` and `backend_gtk4.RendererGTK4Cairo`; use *RendererCairo* instead, which has gained the `set_context` method, which also auto-infers the size of the underlying surface.
- `backend_cairo.RendererCairo.set_ctx_from_surface` and `backend_cairo.RendererCairo.set_width_height` in favor of *RendererCairo.set_context*.
- `backend_gtk3.error_msg_gtk`
- `backend_gtk3.icon_filename` and `backend_gtk3.window_icon`
- `backend_macosx.NavigationToolbar2Mac.prepare_configure_subplots` has been replaced by `configure_subplots()`.
- `backend_pdf.Name.hexify`
- `backend_pdf.Operator` and `backend_pdf.Op.op` are deprecated in favor of a single standard `enum.Enum` interface on *backend_pdf.Op*.
- `backend_pdf.fill`; vendor the code of the similarly named private functions if you rely on these functions.
- `backend_pgf.LatexManager.texcommand` and `backend_pgf.LatexManager.latex_header`
- `backend_pgf.NO_ESCAPE`
- `backend_pgf.common_texification`
- `backend_pgf.get_fontspec`
- `backend_pgf.get_preamble`
- `backend_pgf.re_mathsep`
- `backend_pgf.writeln`
- `backend_ps.convert_psfrags`
- `backend_ps.quote_ps_string`; vendor the code of the similarly named private functions if you rely on it.
- `backend_qt.qApp`; use `QtWidgets.QApplication.instance()` instead.
- `backend_svg.escape_attrib`; vendor the code of the similarly named private functions if you rely on it.

- `backend_svg.escape_cdata`; vendor the code of the similarly named private functions if you rely on it.
- `backend_svg.escape_comment`; vendor the code of the similarly named private functions if you rely on it.
- `backend_svg.short_float_fmt`; vendor the code of the similarly named private functions if you rely on it.
- `backend_svg.generate_transform` and `backend_svg.generate_css`
- `backend_tk.NavigationToolbar2Tk.lastrect` and `backend_tk.RubberbandTk.lastrect`
- `backend_tk.NavigationToolbar2Tk.window`; use `toolbar.master` instead.
- `backend_tools.ToolBase.destroy`; To run code upon tool removal, connect to the `tool_removed_event` event.
- `backend_wx.RendererWx.offset_text_height`
- `backend_wx.error_msg_wx`
- `FigureCanvasBase.pick`; directly call *`Figure.pick`*, which has taken over the responsibility of checking the canvas widget lock as well.
- `FigureCanvasBase.resize`, which has no effect; use `FigureManagerBase.resize` instead.
- `FigureManagerMac.close`
- `FigureFrameWx.sizer`; use `frame.GetSizer()` instead.
- `FigureFrameWx.figmgr` and `FigureFrameWx.get_figure_manager`; use `frame.canvas.manager` instead.
- `FigureFrameWx.num`; use `frame.canvas.manager.num` instead.
- `FigureFrameWx.toolbar`; use `frame.GetToolBar()` instead.
- `FigureFrameWx.toolmanager`; use `frame.canvas.manager.toolmanager` instead.

### Modules

The modules `matplotlib.afm`, `matplotlib.docstring`, `matplotlib.fontconfig_pattern`, `matplotlib.tight_bbox`, `matplotlib.tight_layout`, and `matplotlib.type1font` are considered internal and public access is deprecated.

### `checkdep_usetex` deprecated

This method was only intended to disable tests in case no latex install was found. As such, it is considered to be private and for internal use only.

Please vendor the code if you need this.

### `date_ticker_factory` deprecated

The `date_ticker_factory` method in the *`matplotlib.dates`* module is deprecated. Instead use *`AutoDateLocator`* and *`AutoDateFormatter`* for a more flexible and scalable locator and formatter.

If you need the exact `date_ticker_factory` behavior, please copy the code.

### `dviread.find_tex_file` will raise `FileNotFoundError`

In the future, `dviread.find_tex_file` will raise a `FileNotFoundError` for missing files. Previously, it would return an empty string in such cases. Raising an exception allows attaching a user-friendly message instead. During the transition period, a warning is raised.

### `transforms.Affine2D.identity()`

... is deprecated in favor of directly calling the *`Affine2D`* constructor with no arguments.

### Deprecations in `testing.decorators`

The unused class `CleanupTestCase` and decorator `cleanup` are deprecated and will be removed. Vendor the code, including the private function `_cleanup_cm`.

The function `check_freetype_version` is considered internal and deprecated. Vendor the code of the private function `_check_freetype_version`.

### `text.get_rotation()`

... is deprecated with no replacement. Copy the original implementation if needed.

### Miscellaneous internals

- `axes_grid1.axes_size.AddList`; use `sum(sizes, start=Fixed(0))` (for example) to sum multiple size objects.
- `axes_size.Padded`; use `size + pad` instead
- `axes_size.SizeFromFunc`, `axes_size.GetExtentHelper`
- `AxisArtistHelper.delta1` and `AxisArtistHelper.delta2`
- `axislines.GridHelperBase.new_gridlines` and `axislines.Axes.new_gridlines`
- `cbook.maxdict`; use the standard library `functools.lru_cache` instead.
- `_DummyAxis.dataLim` and `_DummyAxis.viewLim`; use `get_data_interval()`, `set_data_interval()`, `get_view_interval()`, and `set_view_interval()` instead.
- `GridSpecBase.get_grid_positions(..., raw=True)`
- `ImageMagickBase.delay` and `ImageMagickBase.output_args`
- `MathtextBackend`, `MathtextBackendAgg`, `MathtextBackendPath`, `MathTextWarning`
- `TexManager.get_font_config`; it previously returned an internal hashed key for used for caching purposes.
- `TextToPath.get_texmanager`; directly construct a *texmanager.TexManager* instead.
- `ticker.is_close_to_int`; use `math.isclose(x, round(x))` instead.
- `ticker.is_decade`; use `y = numpy.log(x)/numpy.log(base); numpy.isclose(y, numpy.round(y))` instead.

### Removals

The following deprecated APIs have been removed:

### Removed behaviour

### Stricter validation of function parameters

- Unknown keyword arguments to *Figure.savefig*, *pyplot.savefig*, and the `FigureCanvas.print_*` methods now raise a `TypeError`, instead of being ignored.
- Extra parameters to the *Axes* constructor, i.e., those other than *fig* and *rect*, are now keyword only.
- Passing arguments not specifically listed in the signatures of *Axes3D.plot_surface* and *Axes3D.plot_wireframe* is no longer supported; pass any extra arguments as keyword arguments instead.

- Passing positional arguments to *LineCollection* has been removed; use specific keyword argument names now.

### `imread` no longer accepts URLs

Passing a URL to *imread()* has been removed. Please open the URL for reading and directly use the Pillow API (e.g., `PIL.Image.open(urllib.request.urlopen(url))`, or `PIL.Image.open(io.BytesIO(requests.get(url).content))`) instead.

### MarkerStyle is immutable

The methods `MarkerStyle.set_fillstyle` and `MarkerStyle.set_marker` have been removed. Create a new *MarkerStyle* with the respective parameters instead.

### Passing bytes to `FT2Font.set_text`

... is no longer supported. Pass `str` instead.

### Support for passing tool names to `ToolManager.add_tool`

... has been removed. The second parameter to *ToolManager.add_tool* must now always be a tool class.

### `backend_tools.ToolFullScreen` now inherits from `ToolBase`, not from `ToolToggleBase`

*ToolFullScreen* can only switch between the non-fullscreen and fullscreen states, but not unconditionally put the window in a given state; hence the `enable` and `disable` methods were misleadingly named. Thus, the *ToolToggleBase*-related API (`enable`, `disable`, etc.) was removed.

### `BoxStyle._Base` and `transmute` method of box styles

... have been removed. Box styles implemented as classes no longer need to inherit from a base class.

### Loaded modules logging

The list of currently loaded modules is no longer logged at the DEBUG level at Matplotlib import time, because it can produce extensive output and make other valuable DEBUG statements difficult to find. If you were relying on this output, please arrange for your own logging (the built-in `sys.modules` can be used to get the currently loaded modules).

### Modules

- The `cbook.deprecation` module has been removed from the public API as it is considered internal.
- The `mpl_toolkits.axes_grid` module has been removed. All functionality from `mpl_toolkits.axes_grid` can be found in either *mpl_toolkits.axes_grid1* or *mpl_toolkits.axisartist*. Axes classes from `mpl_toolkits.axes_grid` based on `Axis` from *mpl_toolkits.axisartist* can be found in *mpl_toolkits.axisartist*.

### Classes, methods and attributes

The following module-level classes/variables have been removed:

- `cm.cmap_d`
- `colorbar.colorbar_doc`, `colorbar.colorbar_kw_doc`
- `ColorbarPatch`
- `mathtext.Fonts` and all its subclasses
- `mathtext.FontConstantsBase` and all its subclasses
- `mathtext.latex_to_bakoma`, `mathtext.latex_to_cmex`, `mathtext.latex_to_standard`
- `mathtext.MathtextBackendPdf`, `mathtext.MathtextBackendPs`, `mathtext.MathtextBackendSvg`, `mathtext.MathtextBackendCairo`; use `MathtextBackendPath` instead.
- `mathtext.Node` and all its subclasses
- `mathtext.NUM_SIZE_LEVELS`
- `mathtext.Parser`
- `mathtext.Ship`
- `mathtext.SHRINK_FACTOR` and `mathtext.GROW_FACTOR`
- `mathtext.stix_virtual_fonts`,
- `mathtext.tex2uni`
- `backend_pgf.TmpDirCleaner`

- `backend_ps.GraphicsContextPS`; use `GraphicsContextBase` instead.
- `backend_wx.IDLE_DELAY`
- `axes_grid1.parasite_axes.ParasiteAxesAuxTransBase`; use *ParasiteAxesBase* instead.
- `axes_grid1.parasite_axes.ParasiteAxesAuxTrans`; use *ParasiteAxes* instead.

The following class attributes have been removed:

- `Line2D.validCap` and `Line2D.validJoin`; validation is centralized in `rcsetup`.
- `Patch.validCap` and `Patch.validJoin`; validation is centralized in `rcsetup`.
- `renderer.M`, `renderer.eye`, `renderer.vvec`, `renderer.get_axis_position` placed on the Renderer during 3D Axes draw; these attributes are all available via *Axes3D*, which can be accessed via `self.axes` on all *Artist*s.
- `RendererPdf.mathtext_parser`, `RendererPS.mathtext_parser`, `RendererSVG.mathtext_parser`, `RendererCairo.mathtext_parser`
- `StandardPsFonts.pswriter`
- `Subplot.figbox`; use *Axes.get_position* instead.
- `Subplot.numRows`; `ax.get_gridspec().nrows` instead.
- `Subplot.numCols`; `ax.get_gridspec().ncols` instead.
- `SubplotDivider.figbox`
- `cids`, `cnt`, `observers`, `change_observers`, and `submit_observers` on all *Widget*s

The following class methods have been removed:

- `Axis.cla()`; use *Axis.clear* instead.
- `RadialAxis.cla()` and `ThetaAxis.cla()`; use *RadialAxis.clear* or *ThetaAxis.clear* instead.
- `Spine.cla()`; use *Spine.clear* instead.
- `ContourLabeler.get_label_coords()`; there is no replacement as it was considered an internal helper.
- `FancyArrowPatch.get_dpi_cor` and `FancyArrowPatch.set_dpi_cor`
- `FigureCanvas.get_window_title()` and `FigureCanvas.set_window_title()`; use *FigureManagerBase.get_window_title* or *FigureManagerBase.set_window_title* if using pyplot, or use GUI-specific methods if embedding.
- `FigureManager.key_press()` and `FigureManager.button_press()`; trigger the events directly on the canvas using `canvas.callbacks.process(event.name, event)` for key and button events.
- `RendererAgg.get_content_extents()` and `RendererAgg.tostring_rgba_minimized()`
- `NavigationToolbar2Wx.get_canvas()`

- `ParasiteAxesBase.update_viewlim()`; use `ParasiteAxesBase.apply_aspect` instead.
- `Subplot.get_geometry()`; use `SubplotBase.get_subplotspec` instead.
- `Subplot.change_geometry()`; use `SubplotBase.set_subplotspec` instead.
- `Subplot.update_params()`; this method did nothing.
- `Subplot.is_first_row()`; use `ax.get_subplotspec().is_first_row` instead.
- `Subplot.is_first_col()`; use `ax.get_subplotspec().is_first_col` instead.
- `Subplot.is_last_row()`; use `ax.get_subplotspec().is_last_row` instead.
- `Subplot.is_last_col()`; use `ax.get_subplotspec().is_last_col` instead.
- `SubplotDivider.change_geometry()`; use *SubplotDivider.set_subplotspec* instead.
- `SubplotDivider.get_geometry()`; use *SubplotDivider.get_subplotspec* instead.
- `SubplotDivider.update_params()`
- `get_depth`, `parse`, `to_mask`, `to_rgba`, and `to_png` of *MathTextParser*; use *mathtext.math_to_image* instead.
- `MovieWriter.cleanup()`; the cleanup logic is instead fully implemented in *MovieWriter.finish* and `cleanup` is no longer called.

### Functions

The following functions have been removed;

- `backend_template.new_figure_manager()`, `backend_template.new_figure_manager_given_figure()`, and `backend_template.draw_if_interactive()` have been removed, as part of the introduction of the simplified backend API.
- Deprecation-related re-imports `cbook.deprecated()`, and `cbook.warn_deprecated()`.
- `colorbar.colorbar_factory()`; use *Colorbar* instead. `colorbar.make_axes_kw_doc()`
- `mathtext.Error()`
- `mathtext.ship()`
- `mathtext.tex2uni()`
- `axes_grid1.parasite_axes.parasite_axes_auxtrans_class_factory()`; use *parasite_axes_class_factory* instead.
- `sphinext.plot_directive.align()`; use `docutils.parsers.rst.directives.images.Image.align` instead.

### Arguments

The following arguments have been removed:

- *dpi* from `print_ps()` in the PS backend and `print_pdf()` in the PDF backend. Instead, the methods will obtain the DPI from the `savefig` machinery.
- *dpi_cor* from *FancyArrowPatch*
- *minimum_descent* from `TextArea`; it is now effectively always True
- *origin* from `FigureCanvasWx.gui_repaint()`
- *project* from `Line3DCollection.draw()`
- *renderer* from *Line3DCollection.do_3d_projection*, *Patch3D.do_3d_projection*, *PathPatch3D.do_3d_projection*, *Path3DCollection.do_3d_projection*, *Patch3DCollection.do_3d_projection*, *Poly3DCollection.do_3d_projection*
- *resize_callback* from the Tk backend; use `get_tk_widget().bind('<Configure>', ..., True)` instead.
- *return_all* from `gridspec.get_position()`
- Keyword arguments to `gca()`; there is no replacement.

### rcParams

The setting *rcParams["ps.useafm"]* (default: `False`) no longer has any effect on *matplotlib.mathtext*.

### Development changes

#### Increase to minimum supported versions of dependencies

For Matplotlib 3.6, the *minimum supported versions* are being bumped:

| Dependency | min in mpl3.5 | min in mpl3.6 |
|---|---|---|
| Python | 3.7 | 3.8 |
| NumPy | 1.17 | 1.19 |

This is consistent with our *Dependency version policy* and NEP29

### Build setup options changes

The `gui_support.macosx` setup option has been renamed to `packages.macosx`.

### New wheel architectures

Wheels have been added for:

- Python 3.11
- PyPy 3.8 and 3.9

### Increase to required versions of documentation dependencies

sphinx >= 3.0 and numpydoc >= 1.0 are now required for building the documentation.

## 9.5 Version 3.5

### 9.5.1 What's new in Matplotlib 3.5.2 (May 02, 2022)

For a list of all of the issues and pull requests since the last revision, see the *GitHub statistics for 3.9.0 (May 15, 2024)*.

**Table of Contents**


- *What's new in Matplotlib 3.5.2 (May 02, 2022)*
  - *Windows on ARM support*


### Windows on ARM support

Preliminary support for Windows on arm64 target has been added; this requires FreeType 2.11 or above.

No binary wheels are available yet but it may be built from source.

### 9.5.2 What's new in Matplotlib 3.5.0 (Nov 15, 2021)

For a list of all of the issues and pull requests since the last revision, see the *GitHub statistics for 3.9.0 (May 15, 2024)*.

**Table of Contents**


- *What's new in Matplotlib 3.5.0 (Nov 15, 2021)*
  - *Figure and Axes creation / management*
    - *`subplot_mosaic` supports simple Axes sharing*
    - *Figure now has `draw_without_rendering` method*
    - *Figure `__init__` passes keyword arguments through to set*
  - *Plotting methods*
    - *Add `Annulus` patch*
    - *`set_data` method for `FancyArrow` patch*
    - *New arrow styles in `ArrowStyle` and `ConnectionPatch`*
    - *Setting collection offset transform after initialization*
  - *Colors and colormaps*
    - *Colormap registry (experimental)*
    - *Image interpolation now possible at RGBA stage*
    - *`imshow` supports half-float arrays*
    - *A callback registry has been added to Normalize objects*
  - *Titles, ticks, and labels*
    - *Settings tick positions and labels simultaneously in `set_ticks`*
  - *Fonts and Text*
    - *Triple and quadruple dot mathtext accents*
    - *Font properties of legend title are configurable*
    - *`Text` and `TextBox` added* parse_math *option*
    - *Text can be positioned inside TextBox widget*
    - *Simplifying the font setting for usetex mode*
    - *Type 42 subsetting is now enabled for PDF/PS backends*
  - *rcParams improvements*
    - *Allow setting default legend labelcolor globally*

- *3D Axes improvements*
  - *Axes3D now allows manual control of draw order*
  - *Allow changing the vertical axis in 3d plots*
  - `plot_surface` *supports masked arrays and NaNs*
  - *3D plotting methods support* data *keyword argument*
- *Interactive tool improvements*
  - *Colorbars now have pan and zoom functionality*
  - *Updated the appearance of Slider widgets*
  - *Removing points on a PolygonSelector*
  - *Dragging selectors*
  - *Clearing selectors*
  - *Setting artist properties of selectors*
  - *Ignore events outside selection*
  - `CallbackRegistry` *objects gain a method to temporarily block signals*
  - *Directional sizing cursors*
- *Sphinx extensions*
  - *More configuration of* `mathmpl` *sphinx extension*
- *Backend-specific improvements*
  - *GTK backend*
  - *Qt backends*
  - *HiDPI support in Cairo-based, GTK, and Tk backends*
  - *Qt figure options editor improvements*
  - *WX backends support mouse navigation buttons*
  - *WebAgg uses asyncio instead of Tornado*
- *Version information*

### Figure and Axes creation / management

#### `subplot_mosaic` supports simple Axes sharing

*Figure.subplot_mosaic*, *pyplot.subplot_mosaic* support *simple* Axes sharing (i.e., only `True`/`False` may be passed to *sharex*/*sharey*). When `True`, tick label visibility and Axis units will be shared.

```
mosaic = [
    ['A', [['B', 'C'],
           ['D', 'E']]],
    ['F', 'G'],
]
fig = plt.figure(constrained_layout=True)
ax_dict = fig.subplot_mosaic(mosaic, sharex=True, sharey=True)
# All Axes use these scales after this call.
ax_dict['A'].set(xscale='log', yscale='logit')
```

### Figure now has `draw_without_rendering` method

Some aspects of a figure are only determined at draw-time, such as the exact position of text artists or deferred computation like automatic data limits. If you need these values, you can use `figure.canvas.draw()` to force a full draw. However, this has side effects, sometimes requires an open file, and is doing more work than is needed.

The new *Figure.draw_without_rendering* method runs all the updates that `draw()` does, but skips rendering the figure. It's thus more efficient if you need the updated values to configure further aspects of the figure.

### Figure `__init__` passes keyword arguments through to set

Similar to many other sub-classes of *Artist*, the *FigureBase*, *SubFigure*, and *Figure* classes will now pass any additional keyword arguments to *set* to allow properties of the newly created object to be set at initialization time. For example:

```
from matplotlib.figure import Figure
fig = Figure(label='my figure')
```

### Plotting methods

#### Add `Annulus` patch

*Annulus* is a new class for drawing elliptical annuli.

#### `set_data` method for `FancyArrow` patch

*FancyArrow*, the patch returned by `ax.arrow`, now has a `set_data` method that allows modifying the arrow after creation, e.g., for animation.

#### New arrow styles in `ArrowStyle` and `ConnectionPatch`

The new *arrow* parameter to *ArrowStyle* substitutes the use of the *beginarrow* and *endarrow* parameters in the creation of arrows. It receives arrows strings like `'<-'`, `']-['` and `']->'` instead of individual booleans.

Two new styles `']->'` and `'<-['` are also added via this mechanism. *ConnectionPatch*, which accepts arrow styles though its *arrowstyle* parameter, also accepts these new styles.

#### Setting collection offset transform after initialization

The added *collections.Collection.set_offset_transform* may be used to set the offset transform after initialization. This can be helpful when creating a *collections.Collection* outside an Axes object, and later adding it with *Axes.add_collection()* and setting the offset transform to `Axes.transData`.

### Colors and colormaps

#### Colormap registry (experimental)

Colormaps are now managed via *matplotlib.colormaps* (or *pyplot.colormaps*), which is a *ColormapRegistry*. While we are confident that the API is final, we formally mark it as experimental for 3.5 because we want to keep the option to still modify the API for 3.6 should the need arise.

Colormaps can be obtained using item access:

```
import matplotlib.pyplot as plt
cmap = plt.colormaps['viridis']
```

To register new colormaps use:

```
plt.colormaps.register(my_colormap)
```

We recommend to use the new API instead of the `matplotlib.cm.get_cmap` and `matplotlib.cm.register_cmap` functions for new code. `matplotlib.cm.get_cmap` and `matplotlib.cm.register_cmap` will eventually be deprecated and removed. Within *pyplot*, `plt.get_cmap()` and `plt.register_cmap()` will continue to be supported for backward compatibility.

### Image interpolation now possible at RGBA stage

Images in Matplotlib created via *imshow* are resampled to match the resolution of the current canvas. It is useful to apply an auto-aliasing filter when downsampling to reduce Moiré effects. By default, interpolation is done on the data, a norm applied, and then the colormapping performed.

However, it is often desirable for the anti-aliasing interpolation to happen in RGBA space, where the colors are interpolated rather than the data. This usually leads to colors outside the colormap, but visually blends adjacent colors, and is what browsers and other image processing software do.

A new keyword argument *interpolation_stage* is provided for *imshow* to set the stage at which the anti-aliasing interpolation happens. The default is the current behaviour of "data", with the alternative being "rgba" for the newly-available behavior.

For more details see the discussion of the new keyword argument in *Image antialiasing*.

### `imshow` supports half-float arrays

The *imshow* method now supports half-float arrays, i.e., NumPy arrays with dtype `np.float16`.

### A callback registry has been added to Normalize objects

*colors.Normalize* objects now have a callback registry, `callbacks`, that can be connected to by other objects to be notified when the norm is updated. The callback emits the key `changed` when the norm is modified. *cm.ScalarMappable* is now a listener and will register a change when the norm's vmin, vmax or other attributes are changed.

### Titles, ticks, and labels

### Settings tick positions and labels simultaneously in `set_ticks`

*Axis.set_ticks* (and the corresponding *Axes.set_xticks* / *Axes.set_yticks*) has a new parameter *labels* allowing to set tick positions and labels simultaneously.

Previously, setting tick labels was done using *Axis.set_ticklabels* (or the corresponding *Axes.set_xticklabels* / *Axes.set_yticklabels*); this usually only makes sense if tick positions were previously fixed with *set_ticks*:

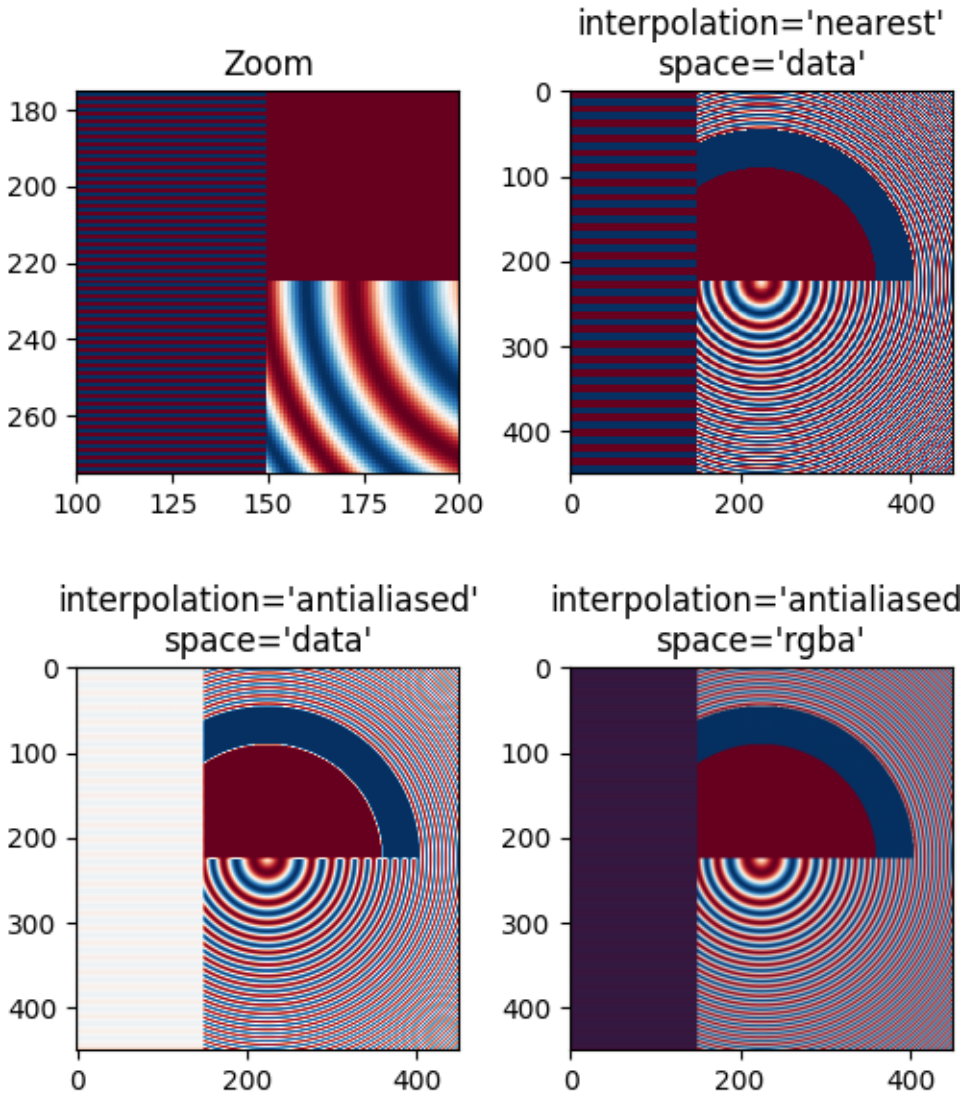



Fig. 1: Example of the interpolation stage options.

```
ax.set_xticks([1, 2, 3])
ax.set_xticklabels(['a', 'b', 'c'])
```

The combined functionality is now available in *set_ticks*:

```
ax.set_xticks([1, 2, 3], ['a', 'b', 'c'])
```

The use of *Axis.set_ticklabels* is discouraged, but it will stay available for backward compatibility.

Note: This addition makes the API of *set_ticks* also more similar to *pyplot.xticks* / *pyplot.yticks*, which already had the additional *labels* parameter.

### Fonts and Text

#### Triple and quadruple dot mathtext accents

In addition to single and double dot accents, mathtext now supports triple and quadruple dot accents.

```
fig = plt.figure(figsize=(3, 1))
fig.text(0.5, 0.5, r'$\dot{a} \ddot{b} \dddot{c} \ddddot{d}$', fontsize=40,
         horizontalalignment='center', verticalalignment='center')
```

#### Font properties of legend title are configurable

Title's font properties can be set via the *title_fontproperties* keyword argument, for example:

#### `Text` and `TextBox` added *parse_math* option

*Text* and *TextBox* objects now allow a *parse_math* keyword-only argument which controls whether math should be parsed from the displayed string. If *True*, the string will be parsed as a math text object. If *False*, the string will be considered a literal and no parsing will occur.

#### Text can be positioned inside TextBox widget

A new parameter called *textalignment* can be used to control for the position of the text inside the Axes of the *TextBox* widget.

### Simplifying the font setting for usetex mode

Now the *rcParams["font.family"]* (default: ['sans-serif']) accepts some font names as value for a more user-friendly setup.

```
plt.rcParams.update({
    "text.usetex": True,
    "font.family": "Helvetica"
})
```

### Type 42 subsetting is now enabled for PDF/PS backends

*backend_pdf* and *backend_ps* now use a unified Type 42 font subsetting interface, with the help of fontTools

Set *rcParams["pdf.fonttype"]* (default: 3) or *rcParams["ps.fonttype"]* (default: 3) to 42 to trigger this workflow:

```
# for PDF backend
plt.rcParams['pdf.fonttype'] = 42

# for PS backend
plt.rcParams['ps.fonttype'] = 42

fig, ax = plt.subplots()
ax.text(0.4, 0.5, 'subsetted document is smaller in size!')

fig.savefig("document.pdf")
fig.savefig("document.ps")
```

### rcParams improvements

### Allow setting default legend labelcolor globally

A new *rcParams["legend.labelcolor"]* (default: 'None') sets the default *labelcolor* argument for *Figure.legend*. The special values 'linecolor', 'markerfacecolor' (or 'mfc'), or 'markeredgecolor' (or 'mec') will cause the legend text to match the corresponding color of marker.

### 3D Axes improvements

#### Axes3D now allows manual control of draw order

The *Axes3D* class now has *computed_zorder* parameter. When set to False, Artists are drawn using their `zorder` attribute.

#### Allow changing the vertical axis in 3d plots

*view_init* now has the parameter *vertical_axis* which allows switching which axis is aligned vertically.

#### `plot_surface` supports masked arrays and NaNs

*axes3d.Axes3D.plot_surface* supports masked arrays and NaNs, and will now hide quads that contain masked or NaN points. The behaviour is similar to *Axes.contour* with `corner_mask=True`.

#### 3D plotting methods support *data* keyword argument

To match all 2D plotting methods, the 3D Axes now support the *data* keyword argument. This allows passing arguments indirectly from a DataFrame-like structure.

```
data = {  # A labelled data set, or e.g., Pandas DataFrame.
    'x': ...,
    'y': ...,
    'z': ...,
    'width': ...,
    'depth': ...,
    'top': ...,
}

fig, ax = plt.subplots(subplot_kw={'projection': '3d')
ax.bar3d('x', 'y', 'z', 'width', 'depth', 'top', data=data)
```

### Interactive tool improvements

#### Colorbars now have pan and zoom functionality

Interactive plots with colorbars can now be zoomed and panned on the colorbar axis. This adjusts the *vmin* and *vmax* of the `ScalarMappable` associated with the colorbar. This is currently only enabled for continuous norms. Norms used with contourf and categoricals, such as `BoundaryNorm` and `NoNorm`, have the interactive capability disabled by default. `cb.ax.set_navigate()` can be used to set whether a colorbar axes is interactive or not.

### Updated the appearance of Slider widgets

The appearance of *Slider* and *RangeSlider* widgets were updated and given new styling parameters for the added handles.

### Removing points on a PolygonSelector

After completing a *PolygonSelector*, individual points can now be removed by right-clicking on them.

### Dragging selectors

The *SpanSelector*, *RectangleSelector* and *EllipseSelector* have a new keyword argument, *drag_from_anywhere*, which when set to `True` allows you to click and drag from anywhere inside the selector to move it. Previously it was only possible to move it by either activating the move modifier button, or clicking on the central handle.

The size of the *SpanSelector* can now be changed using the edge handles.

### Clearing selectors

The selectors (*EllipseSelector*, *LassoSelector*, *PolygonSelector*, *RectangleSelector*, and *SpanSelector*) have a new method *clear*, which will clear the current selection and get the selector ready to make a new selection. This is equivalent to pressing the *escape* key.

### Setting artist properties of selectors

The artist properties of the *EllipseSelector*, *LassoSelector*, *PolygonSelector*, *RectangleSelector* and *SpanSelector* selectors can be changed using the `set_props` and `set_handle_props` methods.

### Ignore events outside selection

The *EllipseSelector*, *RectangleSelector* and *SpanSelector* selectors have a new keyword argument, *ignore_event_outside*, which when set to `True` will ignore events outside of the current selection. The handles or the new dragging functionality can instead be used to change the selection.

### `CallbackRegistry` objects gain a method to temporarily block signals

The context manager *`blocked`* can be used to block callback signals from being processed by the `CallbackRegistry`. The optional keyword, *signal*, can be used to block a specific signal from being processed and let all other signals pass.

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
ax.imshow([[0, 1], [2, 3]])

# Block all interactivity through the canvas callbacks
with fig.canvas.callbacks.blocked():
    plt.show()

fig, ax = plt.subplots()
ax.imshow([[0, 1], [2, 3]])

# Only block key press events
with fig.canvas.callbacks.blocked(signal="key_press_event"):
    plt.show()
```

### Directional sizing cursors

Canvases now support setting directional sizing cursors, i.e., horizontal and vertical double arrows. These are used in e.g., selector widgets. Try the *Mouse Cursor* example to see the cursor in your desired backend.

### Sphinx extensions

### More configuration of `mathmpl` sphinx extension

The *`matplotlib.sphinxext.mathmpl`* sphinx extension supports two new configuration options that may be specified in your `conf.py`:

- `mathmpl_fontsize` (float), which sets the font size of the math text in points;
- `mathmpl_srcset` (list of str), which provides a list of sizes to support responsive resolution images The list should contain additional x-descriptors (`'1.5x'`, `'2x'`, etc.) to generate (1x is the default and always included.)

### Backend-specific improvements

#### GTK backend

A backend supporting GTK4 has been added. Both Agg and Cairo renderers are supported. The GTK4 backends may be selected as GTK4Agg or GTK4Cairo.

#### Qt backends

Support for Qt6 (using either PyQt6 or PySide6) has been added, with either the Agg or Cairo renderers. Simultaneously, support for Qt4 has been dropped. Both Qt6 and Qt5 are supported by a combined backend (QtAgg or QtCairo), and the loaded version is determined by modules already imported, the `QT_API` environment variable, and available packages. See *Qt Bindings* for details. The versioned Qt5 backend names (Qt5Agg or Qt5Cairo) remain supported for backwards compatibility.

#### HiDPI support in Cairo-based, GTK, and Tk backends

The GTK3 backends now support HiDPI fully, including mixed monitor cases (on Wayland only). The newly added GTK4 backends also support HiDPI.

The TkAgg backend now supports HiDPI **on Windows only**, including mixed monitor cases.

All Cairo-based backends correctly support HiDPI as well as their Agg counterparts did (i.e., if the toolkit supports HiDPI, then the *Cairo backend will now support it, but not otherwise.)

#### Qt figure options editor improvements

The figure options editor in the Qt backend now also supports editing the left and right titles (plus the existing centre title). Editing Axis limits is better supported when using a date converter. The `symlog` option is now available in Axis scaling options. All entries with the same label are now shown in the Curves tab.

#### WX backends support mouse navigation buttons

The WX backends now support navigating through view states using the mouse forward/backward buttons, as in other backends.

#### WebAgg uses asyncio instead of Tornado

The WebAgg backend defaults to using `asyncio` over Tornado for timer support. This allows using the WebAgg backend in JupyterLite.

#### Version information

We switched to the release-branch-semver version scheme of setuptools-scm. This only affects the version information for development builds. Their version number now describes the targeted release, i.e. 3.5.0.dev820+g6768ef8c4c is 820 commits after the previous release and is scheduled to be officially released as 3.5.0 later.

In addition to the string `__version__`, there is now a namedtuple `__version_info__` as well, which is modelled after sys.version_info. Its primary use is safely comparing version information, e.g. `if __version_info__ >= (3, 4, 2)`.

### 9.5.3 API Changes for 3.5.3


- *Passing* linefmt *positionally is undeprecated*


#### Passing *linefmt* positionally is undeprecated

Positional use of all formatting parameters in *stem* has been deprecated since Matplotlib 3.5. This deprecation is relaxed so that one can still pass *linefmt* positionally, i.e. `stem(x, y, 'r')`.

### 9.5.4 API Changes for 3.5.2


- *QuadMesh mouseover defaults to False*


#### QuadMesh mouseover defaults to False

New in 3.5, *QuadMesh.get_cursor_data* allows display of data values under the cursor. However, this can be very slow for large meshes, so by `.QuadMesh.set_mouseover` defaults to *False*.

### 9.5.5 API Changes for 3.5.0


- *Behaviour changes*
- *Deprecations*
- *Removals*
- *Development changes*


#### Behaviour changes

##### First argument to `subplot_mosaic` renamed

Both *Figure.subplot_mosaic*, and *pyplot.subplot_mosaic* have had the first positional argument renamed from *layout* to *mosaic*. As we have consolidated the *constrained_layout* and *tight_layout* keyword arguments in the Figure creation functions of *pyplot* into a single *layout* keyword argument, the original `subplot_mosaic` argument name would collide.

As this API is provisional, we are changing this argument name with no deprecation period.

##### `Axes` children are no longer separated by type

Formerly, *axes.Axes* children were separated by *Artist* type, into sublists such as `Axes.lines`. For methods that produced multiple elements (such as *Axes.errorbar*), though individual parts would have similar *zorder*, this separation might cause them to be drawn at different times, causing inconsistent results when overlapping other Artists.

Now, the children are no longer separated by type, and the sublist properties are generated dynamically when accessed. Consequently, Artists will now always appear in the correct sublist; e.g., if *axes.Axes.add_line* is called on a *Patch*, it will appear in the `Axes.patches` sublist, *not* `Axes.lines`. The `Axes.add_*` methods will now warn if passed an unexpected type.

Modification of the following sublists is still accepted, but deprecated:

- `Axes.artists`
- `Axes.collections`
- `Axes.images`
- `Axes.lines`
- `Axes.patches`
- `Axes.tables`
- `Axes.texts`

To remove an Artist, use its *Artist.remove* method. To add an Artist, use the corresponding `Axes.add_*` method.

### `MatplotlibDeprecationWarning` now subclasses `DeprecationWarning`

Historically, it has not been possible to filter *MatplotlibDeprecationWarning*s by checking for `DeprecationWarning`, since we subclass `UserWarning` directly.

The decision to not subclass `DeprecationWarning` has to do with a decision from core Python in the 2.x days to not show `DeprecationWarning`s to users. However, there is now a more sophisticated filter in place (see https://www.python.org/dev/peps/pep-0565/).

Users will now see *MatplotlibDeprecationWarning* only during interactive sessions, and these can be silenced by the standard mechanism:

```
warnings.filterwarnings("ignore", category=DeprecationWarning)
```

Library authors must now enable `DeprecationWarning`s explicitly in order for (non-interactive) CI/CD pipelines to report back these warnings, as is standard for the rest of the Python ecosystem:

```
warnings.filterwarnings("always", DeprecationWarning)
```

### `Artist.set` applies artist properties in the order in which they are given

The change only affects the interaction between the *color*, *edgecolor*, *facecolor*, and (for *Collection*s) *alpha* properties: the *color* property now needs to be passed first in order not to override the other properties. This is consistent with e.g. *Artist.update*, which did not reorder the properties passed to it.

### `pcolor(mesh)` shading defaults to auto

The *shading* keyword argument for *Axes.pcolormesh* and *Axes.pcolor* default has been changed to 'auto'.

Passing `Z(M, N), x(N), y(M)` to `pcolormesh` with `shading='flat'` will now raise a `TypeError`. Use `shading='auto'` or `shading='nearest'` for `x` and `y` to be treated as cell centers, or drop the last column and row of `Z` to get the old behaviour with `shading='flat'`.

### Colorbars now have pan and zoom functionality

Interactive plots with colorbars can now be zoomed and panned on the colorbar axis. This adjusts the *vmin* and *vmax* of the *ScalarMappable* associated with the colorbar. This is currently only enabled for continuous norms. Norms used with `contourf` and categoricals, such as *BoundaryNorm* and *NoNorm*, have the interactive capability disabled by default. *cb.ax.set_navigate()* can be used to set whether a colorbar axes is interactive or not.

### Colorbar lines no longer clipped

If a colorbar has lines added to it (e.g. for contour lines), these will no longer be clipped. This is an improvement for lines on the edge of the colorbar, but could lead to lines off the colorbar if the limits of the colorbar are changed.

### `Figure.suppressComposite` now also controls compositing of Axes images

### The output of `NonUniformImage` and `PcolorImage` has changed

Pixel-level differences may be observed in images generated using *NonUniformImage* or *PcolorImage*, typically for pixels exactly at the boundary between two data cells (no user-facing axes method currently generates *NonUniformImage*s, and only *pcolorfast* can generate *PcolorImage*s). These artists are also now slower, normally by ~1.5x but sometimes more (in particular for `NonUniformImage(interpolation="bilinear")`. This slowdown arises from fixing occasional floating point inaccuracies.

### Change of the (default) legend handler for `Line2D` instances

The default legend handler for Line2D instances (*HandlerLine2D*) now consistently exposes all the attributes and methods related to the line marker (#11358). This makes it easy to change the marker features after instantiating a legend.

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()

ax.plot([1, 3, 2], marker="s", label="Line", color="pink", mec="red", ms=8)
leg = ax.legend()

leg.legendHandles[0].set_color("lightgray")
leg.legendHandles[0].set_mec("black")  # marker edge color
```

The former legend handler for Line2D objects has been renamed *HandlerLine2DCompound*. To revert to the previous behaviour, one can use

```
import matplotlib.legend as mlegend
from matplotlib.legend_handler import HandlerLine2DCompound
from matplotlib.lines import Line2D

mlegend.Legend.update_default_handler_map({Line2D: HandlerLine2DCompound()})
```

### Setting `Line2D` marker edge/face color to *None* use rcParams

`Line2D.set_markeredgecolor(None)` and `Line2D.set_markerfacecolor(None)` now set the line property using the corresponding rcParam (*rcParams["lines.markeredgecolor"]* (default: `'auto'`) and *rcParams["lines.markerfacecolor"]* (default: `'auto'`)). This is consistent with other *Line2D* property setters.

### Default theta tick locations for wedge polar plots have changed

For polar plots that don't cover a full circle, the default theta tick locations are now at multiples of 10°, 15°, 30°, 45°, 90°, rather than using values that mostly make sense for linear plots (20°, 25°, etc.).

### `axvspan` now plots full wedges in polar plots

... rather than triangles.

### Convenience converter from `Scale` to `Normalize` now public

Downstream libraries can take advantage of *colors.make_norm_from_scale* to create a *Normalize* subclass directly from an existing scale. Usually norms have a scale, and the advantage of having a *ScaleBase* attached to a norm is to provide a scale, and associated tick locators and formatters, for the colorbar.

### `ContourSet` always use `PathCollection`

In order to correct rendering issues with closed loops, the *ContourSet* now creates a *PathCollection* instead of a *LineCollection* for line contours. This type matches the artist used for filled contours.

This affects *ContourSet* itself and its subclasses, *QuadContourSet* (returned by *Axes.contour*), and *TriContourSet* (returned by *Axes.tricontour*).

### `hatch.SmallFilledCircles` inherits from `hatch.Circles`

The *hatch.SmallFilledCircles* class now inherits from *hatch.Circles* rather than from *hatch.SmallCircles*.

### hexbin with a log norm

*hexbin* no longer (incorrectly) adds 1 to every bin value if a log norm is being used.

### Setting invalid `rcParams["date.converter"]` now raises ValueError

Previously, invalid values passed to *rcParams["date.converter"]* (default: `'auto'`) would be ignored with a `UserWarning`, but now raise `ValueError`.

### `Text` and `TextBox` added *parse_math* option

*Text* and *TextBox* objects now allow a *parse_math* keyword-only argument which controls whether math should be parsed from the displayed string. If *True*, the string will be parsed as a math text object. If *False*, the string will be considered a literal and no parsing will occur.

For *Text*, this argument defaults to *True*. For *TextBox* this argument defaults to *False*.

### `Type1Font` objects now decrypt the encrypted part

Type 1 fonts have a large part of their code encrypted as an obsolete copy-protection measure. This part is now available decrypted as the `decrypted` attribute of `matplotlib.type1font.Type1Font`. This decrypted data is not yet parsed, but this is a prerequisite for implementing subsetting.

### 3D contourf polygons placed between levels

The polygons used in a 3D *contourf* plot are now placed halfway between the contour levels, as each polygon represents the location of values that lie between two levels.

### `AxesDivider` now defaults to rcParams-specified pads

*AxesDivider.append_axes*, `AxesDivider.new_horizontal`, and `AxesDivider.new_vertical` now default to paddings specified by *rcParams["figure.subplot.wspace"]* (default: `0.2`) and *rcParams["figure.subplot.hspace"]* (default: `0.2`) rather than zero.

### Deprecations

#### Discouraged: `Figure` parameters *tight_layout* and *constrained_layout*

The `Figure` parameters *tight_layout* and *constrained_layout* are triggering competing layout mechanisms and thus should not be used together.

To make the API clearer, we've merged them under the new parameter *layout* with values 'constrained' (equal to `constrained_layout=True`), 'tight' (equal to `tight_layout=True`). If given, *layout* takes precedence.

The use of *tight_layout* and *constrained_layout* is discouraged in favor of *layout*. However, these parameters will stay available for backward compatibility.

#### Modification of `Axes` children sublists

See *Axes children are no longer separated by type* for more information; modification of the following sublists is deprecated:

- `Axes.artists`
- `Axes.collections`
- `Axes.images`
- `Axes.lines`
- `Axes.patches`
- `Axes.tables`
- `Axes.texts`

To remove an Artist, use its *`Artist.remove`* method. To add an Artist, use the corresponding `Axes.add_*` method.

#### Passing incorrect types to `Axes.add_*` methods

The following `Axes.add_*` methods will now warn if passed an unexpected type. See their documentation for the types they expect.

- *`Axes.add_collection`*
- *`Axes.add_image`*
- *`Axes.add_line`*
- *`Axes.add_patch`*
- *`Axes.add_table`*

### Discouraged: `plot_date`

The use of *plot_date* is discouraged. This method exists for historic reasons and may be deprecated in the future.

- `datetime`-like data should directly be plotted using *plot*.
- If you need to plot plain numeric data as *Matplotlib date format* or need to set a timezone, call `ax.xaxis.axis_date` / `ax.yaxis.axis_date` before *plot*. See *Axis.axis_date*.

### `epoch2num` and `num2epoch` are deprecated

These methods convert from unix timestamps to matplotlib floats, but are not used internally to matplotlib, and should not be needed by end users. To convert a unix timestamp to datetime, simply use `datetime.datetime.utcfromtimestamp`, or to use NumPy `datetime64` `dt = np.datetime64(e*1e6, 'us')`.

### Auto-removal of grids by `pcolor` and `pcolormesh`

*pcolor* and *pcolormesh* currently remove any visible axes major grid. This behavior is deprecated; please explicitly call `ax.grid(False)` to remove the grid.

### The first parameter of `Axes.grid` and `Axis.grid` has been renamed to *visible*

The parameter was previously named *b*. This deprecation only matters if that parameter was passed using a keyword argument, e.g. `grid(b=False)`.

### Unification and cleanup of Selector widget API

The API for Selector widgets has been unified to use:

- *props* for the properties of the Artist representing the selection.
- *handle_props* for the Artists representing handles for modifying the selection.
- *grab_range* for the maximal tolerance to grab a handle with the mouse.

Additionally, several internal parameters and attribute have been deprecated with the intention of keeping them private.

### RectangleSelector and EllipseSelector

The *drawtype* keyword argument to `RectangleSelector` is deprecated. In the future the only behaviour will be the default behaviour of `drawtype='box'`.

Support for `drawtype=line` will be removed altogether as it is not clear which points are within and outside a selector that is just a line. As a result, the *lineprops* keyword argument to `RectangleSelector` is also deprecated.

To retain the behaviour of `drawtype='none'`, use `rectprops={'visible': False}` to make the drawn `Rectangle` invisible.

Cleaned up attributes and arguments are:

- The `active_handle` attribute has been privatized and deprecated.
- The `drawtype` attribute has been privatized and deprecated.
- The `eventpress` attribute has been privatized and deprecated.
- The `eventrelease` attribute has been privatized and deprecated.
- The `interactive` attribute has been privatized and deprecated.
- The *marker_props* argument is deprecated, use *handle_props* instead.
- The *maxdist* argument is deprecated, use *grab_range* instead.
- The *rectprops* argument is deprecated, use *props* instead.
- The `rectprops` attribute has been privatized and deprecated.
- The `state` attribute has been privatized and deprecated.
- The `to_draw` attribute has been privatized and deprecated.

### PolygonSelector

- The *line* attribute is deprecated. If you want to change the selector artist properties, use the `set_props` or `set_handle_props` methods.
- The *lineprops* argument is deprecated, use *props* instead.
- The *markerprops* argument is deprecated, use *handle_props* instead.
- The *maxdist* argument and attribute is deprecated, use *grab_range* instead.
- The *vertex_select_radius* argument and attribute is deprecated, use *grab_range* instead.

### SpanSelector

- The `active_handle` attribute has been privatized and deprecated.
- The `eventpress` attribute has been privatized and deprecated.
- The `eventrelease` attribute has been privatized and deprecated.
- The *maxdist* argument and attribute is deprecated, use *grab_range* instead.
- The `pressv` attribute has been privatized and deprecated.
- The `prev` attribute has been privatized and deprecated.
- The `rect` attribute has been privatized and deprecated.
- The *rectprops* argument is deprecated, use *props* instead.
- The `rectprops` attribute has been privatized and deprecated.
- The *span_stays* argument is deprecated, use the *interactive* argument instead.
- The `span_stays` attribute has been privatized and deprecated.
- The `state` attribute has been privatized and deprecated.

### LassoSelector

- The *lineprops* argument is deprecated, use *props* instead.
- The `onpress` and `onrelease` methods are deprecated. They are straight aliases for `press` and `release`.

### `ConversionInterface.convert` no longer needs to accept unitless values

Previously, custom subclasses of *`units.ConversionInterface`* needed to implement a `convert` method that not only accepted instances of the unit, but also unitless values (which are passed through as is). This is no longer the case (`convert` is never called with a unitless value), and such support in *`StrCategoryConverter`* is deprecated. Likewise, the `.ConversionInterface.is_numlike` helper is deprecated.

Consider calling *`Axis.convert_units`* instead, which still supports unitless values.

### Locator and Formatter wrapper methods

The `set_view_interval`, `set_data_interval` and `set_bounds` methods of *Locator*s and *Formatter*s (and their common base class, TickHelper) are deprecated. Directly manipulate the view and data intervals on the underlying axis instead.

### Unused positional parameters to `print_<fmt>` methods

None of the `print_<fmt>` methods implemented by canvas subclasses used positional arguments other that the first (the output filename or file-like), so these extra parameters are deprecated.

### `QuadMesh` signature

The *QuadMesh* signature

```
def __init__(meshWidth, meshHeight, coordinates,
             antialiased=True, shading='flat', **kwargs)
```

is deprecated and replaced by the new signature

```
def __init__(coordinates, *, antialiased=True, shading='flat', **kwargs)
```

In particular:

- The *coordinates* argument must now be a (M, N, 2) array-like. Previously, the grid shape was separately specified as (*meshHeight* + 1, *meshWidth* + 1) and *coordinates* could be an array-like of any shape with M * N * 2 elements.
- All parameters except *coordinates* are keyword-only now.

### rcParams will no longer cast inputs to str

After a deprecation period, rcParams that expect a (non-pathlike) str will no longer cast non-str inputs using `str`. This will avoid confusing errors in subsequent code if e.g. a list input gets implicitly cast to a str.

### Case-insensitive scales

Previously, scales could be set case-insensitively (e.g., `set_xscale("LoG")`). This is deprecated; all builtin scales use lowercase names.

### Interactive cursor details

Setting a mouse cursor on a window has been moved from the toolbar to the canvas. Consequently, several implementation details on toolbars and within backends have been deprecated.

#### `NavigationToolbar2.set_cursor` and `backend_tools.SetCursorBase.set_cursor`

Instead, use the *`FigureCanvasBase.set_cursor`* method on the canvas (available as the `canvas` attribute on the toolbar or the Figure.)

#### `backend_tools.SetCursorBase` and subclasses

`backend_tools.SetCursorBase` was subclassed to provide backend-specific implementations of `set_cursor`. As that is now deprecated, the subclassing is no longer necessary. Consequently, the following subclasses are also deprecated:

- `matplotlib.backends.backend_gtk3.SetCursorGTK3`
- `matplotlib.backends.backend_qt5.SetCursorQt`
- `matplotlib.backends._backend_tk.SetCursorTk`
- `matplotlib.backends.backend_wx.SetCursorWx`

Instead, use the *`backend_tools.ToolSetCursor`* class.

#### `cursord` in GTK, Qt, and wx backends

The `backend_gtk3.cursord`, `backend_qt.cursord`, and `backend_wx.cursord` dictionaries are deprecated. This makes the GTK module importable on headless environments.

### Miscellaneous deprecations

- `is_url` and `URL_REGEX` are deprecated. (They were previously defined in the toplevel *`matplotlib`* module.)
- The `ArrowStyle.beginarrow` and `ArrowStyle.endarrow` attributes are deprecated; use the `arrow` attribute to define the desired heads and tails of the arrow.
- `backend_pgf.LatexManager.str_cache` is deprecated.
- `backends.qt_compat.ETS` and `backends.qt_compat.QT_RC_MAJOR_VERSION` are deprecated, with no replacement.
- The `blocking_input` module has been deprecated. Instead, use `canvas.start_event_loop()` and `canvas.stop_event_loop()` while connecting event callbacks as needed.

- `cbook.report_memory` is deprecated; use `psutil.virtual_memory` instead.
- `cm.LUTSIZE` is deprecated. Use *`rcParams["image.lut"]`* (default: `256`) instead. This value only affects colormap quantization levels for default colormaps generated at module import time.
- `Collection.__init__` previously ignored *transOffset* without *offsets* also being specified. In the future, *transOffset* will begin having an effect regardless of *offsets*. In the meantime, if you wish to set *transOffset*, call *`Collection.set_offset_transform`* explicitly.
- `Colorbar.patch` is deprecated; this attribute is not correctly updated anymore.
- `ContourLabeler.get_label_width` is deprecated.
- `dviread.PsfontsMap` now raises LookupError instead of KeyError for missing fonts.
- `Dvi.baseline` is deprecated (with no replacement).
- The *format* parameter of `dviread.find_tex_file` is deprecated (with no replacement).
- `FancyArrowPatch.get_path_in_displaycoord` and `ConnectionPath.get_path_in_displaycoord` are deprecated. The path in display coordinates can still be obtained, as for other patches, using `patch.get_transform().transform_path(patch.get_path())`.
- The `font_manager.win32InstalledFonts` and `font_manager.get_fontconfig_fonts` helper functions have been deprecated.
- All parameters of `imshow` starting from *aspect* will become keyword-only.
- `QuadMesh.convert_mesh_to_paths` and `QuadMesh.convert_mesh_to_triangles` are deprecated. `QuadMesh.get_paths()` can be used as an alternative for the former; there is no replacement for the latter.
- `ScalarMappable.callbacksSM` is deprecated. Use `ScalarMappable.callbacks` instead.
- `streamplot.get_integrator` is deprecated.
- `style.core.STYLE_FILE_PATTERN`, `style.core.load_base_library`, and `style.core.iter_user_libraries` are deprecated.
- `SubplotParams.validate` is deprecated. Use *`SubplotParams.update`* to change *`SubplotParams`* while always keeping it in a valid state.
- The `grey_arrayd`, `font_family`, `font_families`, and `font_info` attributes of *`TexManager`* are deprecated.
- `Text.get_prop_tup` is deprecated with no replacements (because the *`Text`* class cannot know whether a backend needs to update cache e.g. when the text's color changes).
- `Tick.apply_tickdir` didn't actually update the tick markers on the existing Line2D objects used to draw the ticks and is deprecated; use *`Axis.set_tick_params`* instead.
- `tight_layout.auto_adjust_subplotpars` is deprecated.
- The `grid_info` attribute of `axisartist` classes has been deprecated.
- `axisartist.clip_path` is deprecated with no replacement.

- `axes_grid1.axes_grid.CbarAxes` and `axes_grid1.axisartist.CbarAxes` are deprecated (they are now dynamically generated based on the owning axes class).
- The `axes_grid1.Divider.get_vsize_hsize` and `axes_grid1.Grid.get_vsize_hsize` methods are deprecated. Copy their implementations if needed.
- `AxesDivider.append_axes(..., add_to_figure=False)` is deprecated. Use `ax.remove()` to remove the Axes from the figure if needed.
- `FixedAxisArtistHelper.change_tick_coord` is deprecated with no replacement.
- `floating_axes.GridHelperCurveLinear.get_boundary` is deprecated, with no replacement.
- `ParasiteAxesBase.get_images_artists` has been deprecated.
- The "units finalize" signal (previously emitted by Axis instances) is deprecated. Connect to "units" instead.
- Passing formatting parameters positionally to `stem()` is deprecated

#### `plot_directive` deprecations

The `:encoding:` option to `.. plot` directive has had no effect since Matplotlib 1.3.1, and is now deprecated.

The following helpers in *`matplotlib.sphinxext.plot_directive`* are deprecated:

- `unescape_doctest` (use `doctest.script_from_examples` instead),
- `split_code_at_show`,
- `run_code`.

### Testing support

#### `matplotlib.test()` is deprecated

Run tests using `pytest` from the commandline instead. The variable `matplotlib.default_test_modules` is only used for `matplotlib.test()` and is thus deprecated as well.

To test an installed copy, be sure to specify both `matplotlib` and `mpl_toolkits` with `--pyargs`:

```
pytest --pyargs matplotlib.tests mpl_toolkits.tests
```

See *Testing* for more details.

### Unused pytest fixtures and markers

The fixture `matplotlib.testing.conftest.mpl_image_comparison_parameters` is not used internally by Matplotlib. If you use this please copy it into your code base.

The `@pytest.mark.style` marker is deprecated; use `@mpl.style.context`, which has the same effect.

### Support for `nx1 = None` or `ny1 = None` in `AxesLocator` and `Divider.locate`

In *axes_grid1.axes_divider*, various internal APIs will stop supporting passing `nx1 = None` or `ny1 = None` to mean `nx + 1` or `ny + 1`, in preparation for a possible future API which allows indexing and slicing of dividers (possibly `divider[a:b] == divider.new_locator(a, b)`, but also `divider[a:] == divider.new_locator(a, <end>)`). The user-facing *Divider.new_locator* API is unaffected -- it correctly normalizes `nx1 = None` and `ny1 = None` as needed.

## Removals

The following deprecated APIs have been removed:

## Removed behaviour

### Stricter validation of function parameters

- Calling *Figure.add_axes* with no arguments will raise an error. Adding a free-floating axes needs a position rectangle. If you want a figure-filling single axes, use *Figure.add_subplot* instead.
- *Figure.add_subplot* validates its inputs; in particular, for `add_subplot(rows, cols, index)`, all parameters must be integral. Previously strings and floats were accepted and converted to int.
- Passing *None* as the *which* argument to `autofmt_xdate` is no longer supported; use its more explicit synonym, `which="major"`, instead.
- Setting the *orientation* of an `eventplot()` or *EventCollection* to "none" or *None* is no longer supported; set it to "horizontal" instead. Moreover, the two orientations ("horizontal" and "vertical") are now case-sensitive.
- Passing parameters *norm* and *vmin*/*vmax* simultaneously to functions using colormapping such as `scatter()` and `imshow()` is no longer supported. Instead of `norm=LogNorm(), vmin=min_val, vmax=max_val` pass `norm=LogNorm(min_val, max_val)`. *vmin* and *vmax* should only be used without setting *norm*.
- Passing *None* as either the *radius* or *startangle* arguments of an *Axes.pie* is no longer accepted; use the explicit defaults of 1 and 0, respectively, instead.

- Passing *None* as the *normalize* argument of *Axes.pie* (the former default) is no longer accepted, and the pie will always be normalized by default. If you wish to plot an incomplete pie, explicitly pass `normalize=False`.
- Support for passing *None* to `subplot_class_factory` has been removed. Explicitly pass in the base *Axes* class instead.
- Passing multiple keys as a single comma-separated string or multiple arguments to *ToolManager.update_keymap* is no longer supported; pass keys as a list of strings instead.
- Passing the dash offset as *None* is no longer accepted, as this was never universally implemented, e.g. for vector output. Set the offset to 0 instead.
- Setting a custom method overriding *Artist.contains* using `Artist.set_contains` has been removed, as has `Artist.get_contains`. There is no replacement, but you may still customize pick events using *Artist.set_picker*.
- *semilogx*, *semilogy*, *loglog*, *LogScale*, and *SymmetricalLogScale* used to take keyword arguments that depends on the axis orientation ("basex" vs "basey", "subsx" vs "subsy", "nonposx" vs "nonposy"); these parameter names have been removed in favor of "base", "subs", "nonpositive". This removal also affects e.g. `ax.set_yscale("log", basey=...)` which must now be spelled `ax.set_yscale("log", base=...)`.

  The change from "nonpos" to "nonpositive" also affects *LogTransform*, *InvertedLogTransform*, *SymmetricalLogTransform*, etc.

  To use *different* bases for the x-axis and y-axis of a *loglog* plot, use e.g. `ax.set_xscale("log", base=10); ax.set_yscale("log", base=2)`.
- Passing *None*, or no argument, to `parasite_axes_class_factory`, `parasite_axes_auxtrans_class_factory`, `host_axes_class_factory` is no longer accepted; pass an explicit base class instead.

### Case-sensitivity is now enforced more

- Upper or mixed-case property names are no longer normalized to lowercase in *Artist.set* and *Artist.update*. This allows one to pass names such as *patchA* or *UVC*.
- Case-insensitive capstyles and joinstyles are no longer lower-cased; please pass capstyles ("miter", "round", "bevel") and joinstyles ("butt", "round", "projecting") as lowercase.
- Saving metadata in PDF with the PGF backend no longer changes keys to lowercase. Only the canonically cased keys listed in the PDF specification (and the *PdfPages* documentation) are accepted.

### No implicit initialization of `Tick` attributes

The *Tick* constructor no longer initializes the attributes `tick1line`, `tick2line`, `gridline`, `label1`, and `label2` via `_get_tick1line`, `_get_tick2line`, `_get_gridline`, `_get_text1`, and `_get_text2`. Please directly set the attribute in the subclass' `__init__` instead.

### `NavigationToolbar2` subclass changes

Overriding the `_init_toolbar` method of *NavigationToolbar2* to initialize third-party toolbars is no longer supported. Instead, the toolbar should be initialized in the `__init__` method of the subclass (which should call the base-class' `__init__` as appropriate).

The `press` and `release` methods of *NavigationToolbar2* were called when pressing or releasing a mouse button, but *only* when an interactive pan or zoom was occurring (contrary to what the docs stated). They are no longer called; if you write a backend which needs to customize such events, please directly override `press_pan`/`press_zoom`/`release_pan`/`release_zoom` instead.

### Removal of old file mode flag

Flags containing "U" passed to *cbook.to_filehandle* and *cbook.open_file_cm* are no longer accepted. This is consistent with their removal from `open` in Python 3.9.

### Keymaps toggling `Axes.get_navigate` have been removed

This includes numeric key events and rcParams.

### The `TTFPATH` and `AFMPATH` environment variables

Support for the (undocumented) `TTFPATH` and `AFMPATH` environment variables has been removed. Register additional fonts using `matplotlib.font_manager.fontManager.addfont()`.

### Modules

- `matplotlib.backends.qt_editor.formsubplottool`; use `matplotlib.backends.backend_qt.SubplotToolQt` instead.
- `matplotlib.compat`
- `matplotlib.ttconv`
- The Qt4-based backends, `qt4agg` and `qt4cairo`, have been removed. Qt4 has reached its end-of-life in 2015 and there are no releases of either PyQt4 or PySide for recent versions of Python. Please use one of the Qt5 or Qt6 backends.

### Classes, methods and attributes

The following module-level classes/variables have been removed:

- `backend_bases.StatusbarBase` and all its subclasses, and `StatusBarWx`; messages are displayed in the toolbar
- `backend_pgf.GraphicsContextPgf`
- `MODIFIER_KEYS`, `SUPER`, `ALT`, `CTRL`, and `SHIFT` of *matplotlib.backends.backend_qt5agg* and *matplotlib.backends.backend_qt5cairo*
- `backend_wx.DEBUG_MSG`
- `dviread.Encoding`
- `Fil`, `Fill`, `Filll`, `NegFil`, `NegFill`, `NegFilll`, and `SsGlue` from *mathtext*; directly construct glue instances with `Glue("fil")`, etc.
- `mathtext.GlueSpec`
- `OldScalarFormatter`, `IndexFormatter` and `IndexDateFormatter`; use *FuncFormatter* instead
- `OldAutoLocator`
- `AVConvBase`, `AVConvWriter` and `AVConvFileWriter`. Debian 8 (2015, EOL 06/2020) and Ubuntu 14.04 (EOL 04/2019) were the last versions of Debian and Ubuntu to ship avconv. It remains possible to force the use of avconv by using the FFmpeg-based writers with *rcParams["animation.ffmpeg_path"]* (default: `'ffmpeg'`) set to "avconv".
- `matplotlib.axes._subplots._subplot_classes`
- `axes_grid1.axes_rgb.RGBAxesBase`; use `RGBAxes` instead

The following class attributes have been removed:

- `backend_pgf.LatexManager.latex_stdin_utf8`
- `backend_pgf.PdfPages.metadata`
- `ContourSet.ax` and `Quiver.ax`; use `ContourSet.axes` or `Quiver.axes` as with other artists
- `DateFormatter.illegal_s`
- `dates.YearLocator.replaced`; *YearLocator* is now a subclass of *RRuleLocator*, and the attribute `YearLocator.replaced` has been removed. For tick locations that required modifying this, a custom rrule and *RRuleLocator* can be used instead.
- `FigureManagerBase.statusbar`; messages are displayed in the toolbar
- `FileMovieWriter.clear_temp`
- `mathtext.Glue.glue_subtype`
- `MovieWriter.args_key`, `MovieWriter.exec_key`, and `HTMLWriter.args_key`

- `NavigationToolbar2QT.basedir`; the base directory to the icons is `os.path.join(mpl.get_data_path(), "images")`
- `NavigationToolbar2QT.ctx`
- `NavigationToolbar2QT.parent`; to access the parent window, use `toolbar.canvas.parent()` or `toolbar.parent()`
- `prevZoomRect`, `retinaFix`, `savedRetinaImage`, `wxoverlay`, `zoomAxes`, `zoomStartX`, and `zoomStartY` attributes of `NavigationToolbar2Wx`
- `NonUniformImage.is_grayscale`, `PcolorImage.is_grayscale`, for consistency with `AxesImage.is_grayscale`. (Note that previously, these attributes were only available *after rendering the image*).
- `RendererCairo.fontweights`, `RendererCairo.fontangles`
- `used_characters` of *RendererPdf*, *PdfFile*, and *RendererPS*
- `LogScale.LogTransform`, `LogScale.InvertedLogTransform`, `SymmetricalScale.SymmetricalTransform`, and `SymmetricalScale.InvertedSymmetricalTransform`; directly access the transform classes from *matplotlib.scale*
- `cachedir`, `rgba_arrayd`, `serif`, `sans_serif`, `cursive`, and `monospace` attributes of *TexManager*
- `axleft`, `axright`, `axbottom`, `axtop`, `axwspace`, and `axhspace` attributes of *widgets.SubplotTool*; access the `ax` attribute of the corresponding slider
- `widgets.TextBox.params_to_disable`
- `angle_helper.LocatorBase.den`; it has been renamed to *nbins*
- `axes_grid.CbarAxesBase.cbid` and `axes_grid.CbarAxesBase.locator`; use `mappable.colorbar_cid` or `colorbar.locator` instead

The following class methods have been removed:

- `Axes.update_datalim_bounds`; use `ax.dataLim.set(Bbox.union([ax.dataLim, bounds]))`
- `pan` and `zoom` methods of *Axis* and *Locator* have been removed; panning and zooming are now implemented using the `start_pan`, `drag_pan`, and `end_pan` methods of *Axes*
- `.BboxBase.inverse_transformed`; call *BboxBase.transformed* on the *inverted()* transform
- `Collection.set_offset_position` and `Collection.get_offset_position` have been removed; the `offset_position` of the *Collection* class is now "screen"
- `Colorbar.on_mappable_changed` and `Colorbar.update_bruteforce`; use `Colorbar.update_normal()` instead
- `docstring.Substitution.from_params` has been removed; directly assign to `params` of `docstring.Substitution` instead
- `DraggableBase.artist_picker`; set the artist's picker instead

- `DraggableBase.on_motion_blit`; use *`DraggableBase.on_motion`* instead
- `FigureCanvasGTK3._renderer_init`
- `Locator.refresh()` and the associated helper methods `NavigationToolbar2.draw()` and `ToolViewsPositions.refresh_locators()`
- `track_characters` and `merge_used_characters` of *`RendererPdf`*, *`PdfFile`*, and *`RendererPS`*
- `RendererWx.get_gc`
- `SubplotSpec.get_rows_columns`; use the `GridSpec.nrows`, `GridSpec.ncols`, `SubplotSpec.rowspan`, and `SubplotSpec.colspan` properties instead.
- `ScalarMappable.update_dict`, `ScalarMappable.add_checker()`, and `ScalarMappable.check_update()`; register a callback in `ScalarMappable.callbacks` to be notified of updates
- `TexManager.make_tex_preview` and `TexManager.make_dvi_preview`
- `funcleft`, `funcright`, `funcbottom`, `functop`, `funcwspace`, and `funchspace` methods of *`widgets.SubplotTool`*
- `axes_grid1.axes_rgb.RGBAxes.add_RGB_to_figure`
- `axisartist.axis_artist.AxisArtist.dpi_transform`
- `axisartist.grid_finder.MaxNLocator.set_factor` and `axisartist.grid_finder.FixedLocator.set_factor`; the factor is always 1 now

### Functions

- `bezier.make_path_regular` has been removed; use `Path.cleaned()` (or `Path.cleaned(curves=True)`, etc.) instead, but note that these methods add a `STOP` code at the end of the path.
- `bezier.concatenate_paths` has been removed; use `Path.make_compound_path()` instead.
- `cbook.local_over_kwdict` has been removed; use *`cbook.normalize_kwargs`* instead.
- `qt_compat.is_pyqt5` has been removed due to the release of PyQt6. The Qt version can be checked using `QtCore.qVersion()`.
- `testing.compare.make_external_conversion_command` has been removed.
- `axes_grid1.axes_rgb.imshow_rgb` has been removed; use `imshow(np.dstack([r, g, b]))` instead.

### Arguments

- The *s* parameter to *Axes.annotate* and *pyplot.annotate* is no longer supported; use the new name *text*.
- The *inframe* parameter to *matplotlib.axes.Axes.draw* has been removed; use *Axes.redraw_in_frame* instead.
- The *required*, *forbidden* and *allowed* parameters of *cbook.normalize_kwargs* have been removed.
- The *ismath* parameter of the `draw_tex` method of all renderer classes has been removed (as a call to `draw_tex` — not to be confused with `draw_text`! — means that the entire string should be passed to the `usetex` machinery anyways). Likewise, the text machinery will no longer pass the *ismath* parameter when calling `draw_tex` (this should only matter for backend implementers).
- The *quality*, *optimize*, and *progressive* parameters of *Figure.savefig* (which only affected JPEG output) have been removed, as well as from the corresponding `print_jpg` methods. JPEG output options can be set by directly passing the relevant parameters in *pil_kwargs*.
- The *clear_temp* parameter of *FileMovieWriter* has been removed; files placed in a temporary directory (using `frame_prefix=None`, the default) will be cleared; files placed elsewhere will not.
- The *copy* parameter of `mathtext.Glue` has been removed.
- The *quantize* parameter of *Path.cleaned()* has been removed.
- The *dummy* parameter of *RendererPgf* has been removed.
- The *props* parameter of *Shadow* has been removed; use keyword arguments instead.
- The *recursionlimit* parameter of `matplotlib.test` has been removed.
- The *label* parameter of *Tick* has no effect and has been removed.
- *MaxNLocator* no longer accepts a positional parameter and the keyword argument *nbins* simultaneously because they specify the same quantity.
- The *add_all* parameter to `axes_grid.Grid`, `axes_grid.ImageGrid`, `axes_rgb.make_rgb_axes`, and `axes_rgb.RGBAxes` have been removed; the APIs always behave as if `add_all=True`.
- The *den* parameter of `axisartist.angle_helper.LocatorBase` has been removed; use *nbins* instead.
- The *s* keyword argument to *AnnotationBbox.get_fontsize* has no effect and has been removed.
- The *offset_position* keyword argument of the *Collection* class has been removed; the `offset_position` now "screen".
- Arbitrary keyword arguments to `StreamplotSet` have no effect and have been removed.
- The *fontdict* and *minor* parameters of *Axes.set_xticklabels* / *Axes.set_yticklabels* are now keyword-only.

- All parameters of *Figure.subplots* except *nrows* and *ncols* are now keyword-only; this avoids typing e.g. `subplots(1, 1, 1)` when meaning `subplot(1, 1, 1)`, but actually getting `subplots(1, 1, sharex=1)`.
- All parameters of *pyplot.tight_layout* are now keyword-only, to be consistent with *Figure.tight_layout*.
- `ColorbarBase` only takes a single positional argument now, the `Axes` to create it in, with all other options required to be keyword arguments. The warning for keyword arguments that were overridden by the mappable is now removed.
- Omitting the *renderer* parameter to *matplotlib.axes.Axes.draw* is no longer supported; use `axes.draw_artist(axes)` instead.
- Passing `ismath="TeX!"` to *RendererAgg.get_text_width_height_descent* is no longer supported; pass `ismath="TeX"` instead,
- Changes to the signature of the *matplotlib.axes.Axes.draw* method make it consistent with all other artists; thus additional parameters to *Artist.draw* have also been removed.

### rcParams

- The `animation.avconv_path` and `animation.avconv_args` rcParams have been removed.
- The `animation.html_args` rcParam has been removed.
- The `keymap.all_axes` rcParam has been removed.
- The `mathtext.fallback_to_cm` rcParam has been removed. Use *rcParams["mathtext.fallback"]* (default: `'cm'`) instead.
- The `savefig.jpeg_quality` rcParam has been removed.
- The `text.latex.preview` rcParam has been removed.
- The following deprecated rcParams validators, defined in *rcsetup*, have been removed:
  - `validate_alignment`
  - `validate_axes_titlelocation`
  - `validate_axis_locator`
  - `validate_bool_maybe_none`
  - `validate_fontset`
  - `validate_grid_axis`
  - `validate_hinting`
  - `validate_legend_loc`
  - `validate_mathtext_default`
  - `validate_movie_frame_fmt`

  - `validate_movie_html_fmt`
  - `validate_movie_writer`
  - `validate_nseq_float`
  - `validate_nseq_int`
  - `validate_orientation`
  - `validate_pgf_texsystem`
  - `validate_ps_papersize`
  - `validate_svg_fonttype`
  - `validate_toolbar`
  - `validate_webagg_address`
- Some rcParam validation has become stricter:
  - *`rcParams["axes.axisbelow"]`* (default: `'line'`) no longer accepts strings starting with "line" (case-insensitive) as "line"; use "line" (case-sensitive) instead.
  - *`rcParams["text.latex.preamble"]`* (default: `''`) and *`rcParams["pdf.preamble"]`* no longer accept non-string values.
  - All `*.linestyle` rcParams no longer accept `offset = None`; set the offset to 0 instead.

### Development changes

#### Increase to minimum supported versions of dependencies

For Matplotlib 3.5, the *minimum supported versions* and some *optional dependencies* are being bumped:

| Dependency | min in mpl3.4 | min in mpl3.5 |
|---|---|---|
| NumPy | 1.16 | 1.17 |
| Tk (optional) | 8.3 | 8.4 |

This is consistent with our *Dependency version policy* and NEP29

#### New wheel architectures

Wheels have been added for:

- Python 3.10
- PyPy 3.7
- macOS on Apple Silicon (both arm64 and universal2)

### New build dependencies

Versioning has been switched from bundled versioneer to setuptools-scm using the `release-branch-semver` version scheme. The latter is well-maintained, but may require slight modification to packaging scripts.

The setuptools-scm-git-archive plugin is also used for consistent version export.

### Data directory is no longer optional

Historically, the `mpl-data` directory has been optional (example files were unnecessary, and fonts could be deleted if a suitable dependency on a system font were provided). Though example files are still optional, they have been substantially pared down, and we now consider the directory to be required.

Specifically, the `matplotlibrc` file found there is used for runtime verifications and must exist. Packagers may still symlink fonts to system versions if needed.

### New runtime dependencies

#### fontTools for type 42 subsetting

A new dependency fontTools is integrated into Matplotlib 3.5. It is designed to be used with PS/EPS and PDF documents; and handles Type 42 font subsetting.

#### Underscore support in LaTeX

The underscore package is now a requirement to improve support for underscores in LaTeX.

This is consistent with our *Dependency version policy*.

### Matplotlib-specific build options moved from `setup.cfg` to `mplsetup.cfg`

In order to avoid conflicting with the use of `setup.cfg` by `setuptools`, the Matplotlib-specific build options have moved from `setup.cfg` to `mplsetup.cfg`. The `setup.cfg.template` has been correspondingly been renamed to `mplsetup.cfg.template`.

Note that the path to this configuration file can still be set via the *`MPLSETUPCFG`* environment variable, which allows one to keep using the same file before and after this change.

## 9.6 Version 3.4

### 9.6.1 What's new in Matplotlib 3.4.0 (Mar 26, 2021)

For a list of all of the issues and pull requests since the last revision, see the *GitHub statistics for 3.9.0 (May 15, 2024)*.

**Table of Contents**


- *What's new in Matplotlib 3.4.0 (Mar 26, 2021)*
  - *Figure and Axes creation / management*
    - *New subfigure functionality*
    - *Single-line string notation for* `subplot_mosaic`
    - *Changes to behavior of Axes creation methods (*`gca`*,* `add_axes`*,* `add_subplot`*)*
    - `add_subplot`*/*`add_axes` *gained an* axes_class *parameter*
    - *Subplot and subplot2grid can now work with constrained layout*
  - *Plotting methods*
    - `axline` *supports* transform *parameter*
    - *New automatic labeling for bar charts*
    - *A list of hatches can be specified to* `bar` *and* `barh`
    - *Setting* `BarContainer` *orientation*
    - *Contour plots now default to using ScalarFormatter*
    - `Axes.errorbar` *cycles non-color properties correctly*
    - `errorbar` errorevery *parameter matches* markevery
    - `hexbin` *supports data reference for* C *parameter*
    - *Support callable for formatting of Sankey labels*
    - `Axes.spines` *access shortcuts*
    - *New* `stairs` *method and* `StepPatch` *artist*
    - *Added* orientation *parameter for stem plots*
    - *Angles on Bracket arrow styles*
    - `TickedStroke` *patheffect*
  - *Colors and colormaps*
    - *Collection color specification and mapping*
    - *Transparency (alpha) can be set as an array in collections*

        * *pcolormesh has improved transparency handling by enabling snapping*
        * *IPython representations for Colormap objects*
        * *`Colormap.set_extremes` and `Colormap.with_extremes`*
        * *Get under/over/bad colors of Colormap objects*
        * *New `cm.unregister_cmap` function*
        * *New `CenteredNorm` for symmetrical data around a center*
        * *New `FuncNorm` for arbitrary normalizations*
        * *GridSpec-based colorbars can now be positioned above or to the left of the main axes*
    - *Titles, ticks, and labels*
        * *supxlabel and supylabel*
        * *Shared-axes `subplots` tick label visibility is now correct for top or left labels*
        * *An iterable object with labels can be passed to `Axes.plot`*
    - *Fonts and Text*
        * *Text transform can rotate text direction*
        * *`matplotlib.mathtext` now supports* overset *and* underset *LaTeX symbols*
        * math_fontfamily *parameter to change `Text` font family*
        * *`TextArea`/`AnchoredText` support* horizontalalignment
        * *PDF supports URLs on `Text` artists*
    - *rcParams improvements*
        * *New rcParams for dates: set converter and whether to use interval_multiples*
        * *Date formatters now respect* usetex *rcParam*
        * *Setting* image.cmap *to a `Colormap`*
        * *Tick and tick label colors can be set independently using rcParams*
    - *3D Axes improvements*
        * *Errorbar method in 3D Axes*
        * *Stem plots in 3D Axes*
        * *3D Collection properties are now modifiable*
        * *Panning in 3D Axes*
    - *Interactive tool improvements*
        * *New `RangeSlider` widget*
        * *Sliders can now snap to arbitrary values*

        * *Pausing and Resuming Animations*
    - *Sphinx extensions*
        * `plot_directive` caption *option*
    - *Backend-specific improvements*
        * *Consecutive rasterized draws now merged*
        * *Support raw/rgba frame format in* `FFMpegFileWriter`
        * *nbAgg/WebAgg support middle-click and double-click*
        * *nbAgg support binary communication*
        * *Indexed color for PNG images in PDF files when possible*
        * *Improved font subsettings in PDF/PS*
        * *Kerning added to strings in PDFs*
        * *Fully-fractional HiDPI in QtAgg*
        * *wxAgg supports fullscreen toggle*


### Figure and Axes creation / management

#### New subfigure functionality

New `figure.Figure.add_subfigure` and `figure.Figure.subfigures` functionalities allow creating virtual figures within figures. Similar nesting was previously done with nested gridspecs (see *Nested Gridspecs*). However, this did not allow localized figure artists (e.g., a colorbar or suptitle) that only pertained to each subgridspec.

The new methods `figure.Figure.add_subfigure` and `figure.Figure.subfigures` are meant to rhyme with `figure.Figure.add_subplot` and `figure.Figure.subplots` and have most of the same arguments.

See *Figure subfigures* for further details.

---

**Note:** The subfigure functionality is experimental API as of v3.4.

---

### Single-line string notation for `subplot_mosaic`

*Figure.subplot_mosaic* and *pyplot.subplot_mosaic* now accept a single-line string, using semicolons to delimit rows. Namely,

```
plt.subplot_mosaic(
    """
    AB
    CC
    """)
```

may be written as the shorter:

```
plt.subplot_mosaic("AB;CC")
```

### Changes to behavior of Axes creation methods (`gca`, `add_axes`, `add_subplot`)

The behavior of the functions to create new Axes (*pyplot.axes*, *pyplot.subplot*, *figure.Figure.add_axes*, *figure.Figure.add_subplot*) has changed. In the past, these functions would detect if you were attempting to create Axes with the same keyword arguments as already-existing Axes in the current Figure, and if so, they would return the existing Axes. Now, *pyplot.axes*, *figure.Figure.add_axes*, and *figure.Figure.add_subplot* will always create new Axes. *pyplot.subplot* will continue to reuse an existing Axes with a matching subplot spec and equal *kwargs*.

Correspondingly, the behavior of the functions to get the current Axes (*pyplot.gca*, *figure.Figure.gca*) has changed. In the past, these functions accepted keyword arguments. If the keyword arguments matched an already-existing Axes, then that Axes would be returned, otherwise new Axes would be created with those keyword arguments. Now, the keyword arguments are only considered if there are no Axes at all in the current figure. In a future release, these functions will not accept keyword arguments at all.

### `add_subplot`/`add_axes` gained an *axes_class* parameter

In particular, `mpl_toolkits` Axes subclasses can now be idiomatically used using, e.g., `fig.add_subplot(axes_class=mpl_toolkits.axislines.Axes)`

### Subplot and subplot2grid can now work with constrained layout

`constrained_layout` depends on a single *GridSpec* for each logical layout on a figure. Previously, *pyplot.subplot* and *pyplot.subplot2grid* added a new `GridSpec` each time they were called and were therefore incompatible with `constrained_layout`.

Now `subplot` attempts to reuse the `GridSpec` if the number of rows and columns is the same as the top level GridSpec already in the figure, i.e., `plt.subplot(2, 1, 2)` will use the same GridSpec as `plt.subplot(2, 1, 1)` and the `constrained_layout=True` option to *Figure* will work.

In contrast, mixing *nrows* and *ncols* will *not* work with `constrained_layout`: `plt.subplot(2, 2, 1)` followed by `plt.subplots(2, 1, 2)` will still produce two GridSpecs, and `constrained_layout=True` will give bad results. In order to get the desired effect, the second call can specify the cells the second Axes is meant to cover: `plt.subplots(2, 2, (2, 4))`, or the more Pythonic `plt.subplot2grid((2, 2), (0, 1), rowspan=2)` can be used.

### Plotting methods

#### `axline` supports *transform* parameter

*axline* now supports the *transform* parameter, which applies to the points *xy1*, *xy2*. The *slope* (if given) is always in data coordinates.

For example, this can be used with `ax.transAxes` for drawing lines with a fixed slope. In the following plot, the line appears through the same point on both Axes, even though they show different data limits.

```
fig, axs = plt.subplots(1, 2)

for i, ax in enumerate(axs):
    ax.axline((0.25, 0), slope=2, transform=ax.transAxes)
    ax.set(xlim=(i, i+5), ylim=(i, i+5))
```

#### New automatic labeling for bar charts

A new *Axes.bar_label* method has been added for auto-labeling bar charts.

#### A list of hatches can be specified to `bar` and `barh`

Similar to some other rectangle properties, it is now possible to hand a list of hatch styles to *bar* and *barh* in order to create bars with different hatch styles, e.g.

#### Setting `BarContainer` orientation

*BarContainer* now accepts a new string argument *orientation*. It can be either `'vertical'` or `'horizontal'`, default is `None`.

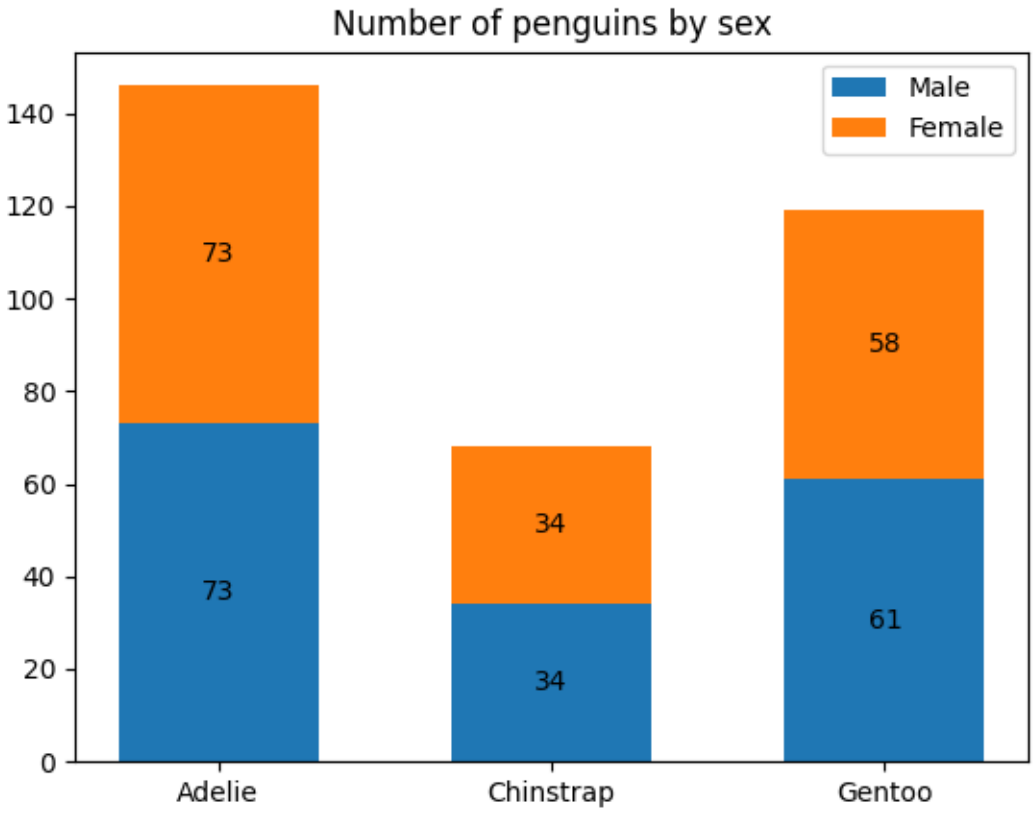



Fig. 2: Example of the new automatic labeling.

### Contour plots now default to using ScalarFormatter

Pass `fmt="%1.3f"` to the contouring call to restore the old default label format.

### `Axes.errorbar` cycles non-color properties correctly

Formerly, *Axes.errorbar* incorrectly skipped the Axes property cycle if a color was explicitly specified, even if the property cycler was for other properties (such as line style). Now, *Axes.errorbar* will advance the Axes property cycle as done for *Axes.plot*, i.e., as long as all properties in the cycler are not explicitly passed.

For example, the following will cycle through the line styles:

```
x = np.arange(0.1, 4, 0.5)
y = np.exp(-x)
offsets = [0, 1]

plt.rcParams['axes.prop_cycle'] = plt.cycler('linestyle', ['-', '--'])

fig, ax = plt.subplots()
for offset in offsets:
    ax.errorbar(x, y + offset, xerr=0.1, yerr=0.3, fmt='tab:blue')
```

### `errorbar` *errorevery* parameter matches *markevery*

Similar to the *markevery* parameter to *plot*, the *errorevery* parameter of *errorbar* now accept slices and NumPy fancy indexes (which must match the size of *x*).

### `hexbin` supports data reference for *C* parameter

As with the *x* and *y* parameters, *Axes.hexbin* now supports passing the *C* parameter using a data reference.

```
data = {
    'a': np.random.rand(1000),
    'b': np.random.rand(1000),
    'c': np.random.rand(1000),
}

fig, ax = plt.subplots()
ax.hexbin('a', 'b', C='c', data=data, gridsize=10)
```

### Support callable for formatting of Sankey labels

The `format` parameter of *`matplotlib.sankey.Sankey`* can now accept callables.

This allows the use of an arbitrary function to label flows, for example allowing the mapping of numbers to emoji.

### `Axes.spines` access shortcuts

`Axes.spines` is now a dedicated container class *`Spines`* for a set of *`Spine`*s instead of an `OrderedDict`. On top of dict-like access, `Axes.spines` now also supports some `pandas.Series`-like features.

Accessing single elements by item or by attribute:

```
ax.spines['top'].set_visible(False)
ax.spines.top.set_visible(False)
```

Accessing a subset of items:

```
ax.spines[['top', 'right']].set_visible(False)
```

Accessing all items simultaneously:

```
ax.spines[:].set_visible(False)
```

### New `stairs` method and `StepPatch` artist

*`pyplot.stairs`* and the underlying artist *`StepPatch`* provide a cleaner interface for plotting stepwise constant functions for the common case that you know the step edges. This supersedes many use cases of *`pyplot.step`*, for instance when plotting the output of `numpy.histogram`.

For both the artist and the function, the x-like edges input is one element longer than the y-like values input

See *Stairs Demo* for examples.

### Added *orientation* parameter for stem plots

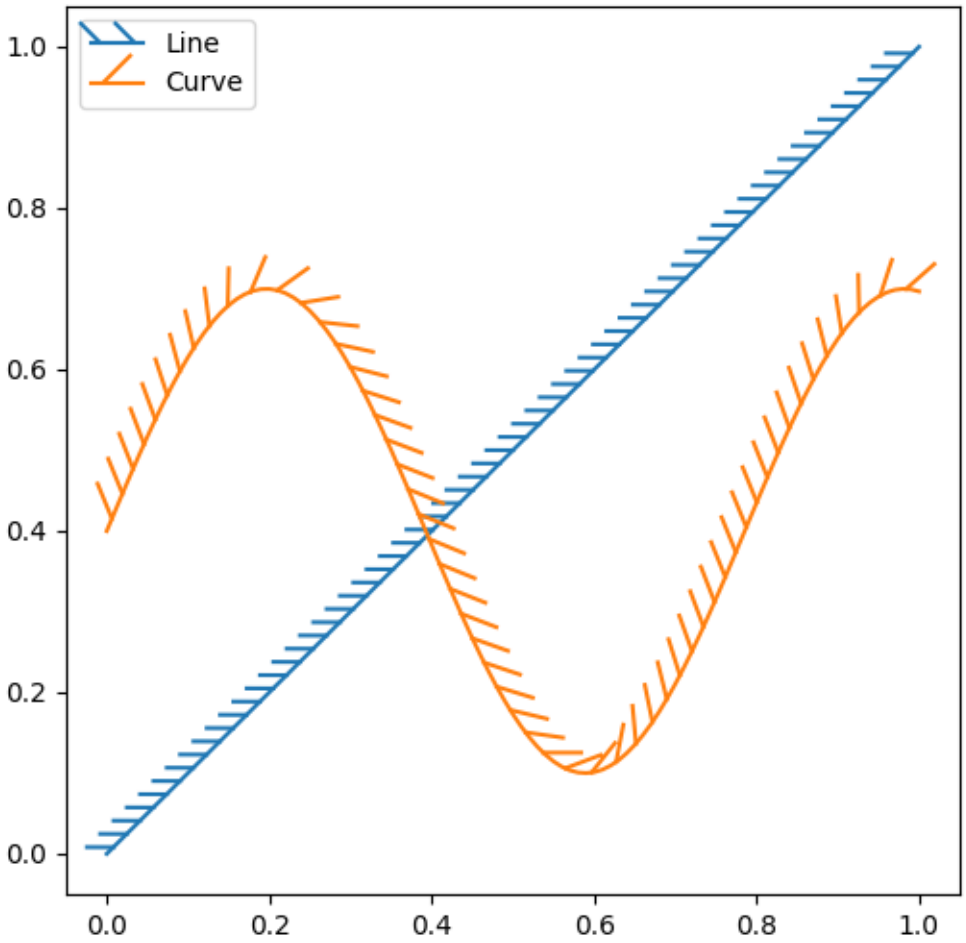
By default, stem lines are vertical. They can be changed to horizontal using the *orientation* parameter of *`Axes.stem`* or *`pyplot.stem`*:

### Angles on Bracket arrow styles

Angles specified on the *Bracket* arrow styles (`]-[`, `]-`, `-[`, or `|-|` passed to *arrowstyle* parameter of `FancyArrowPatch`) are now applied. Previously, the *angleA* and *angleB* options were allowed, but did nothing.

### `TickedStroke` patheffect

The new `TickedStroke` patheffect can be used to produce lines with a ticked style. This can be used to, e.g., distinguish the valid and invalid sides of the constraint boundaries in the solution space of optimizations.



### Colors and colormaps

#### Collection color specification and mapping

Reworking the handling of color mapping and the keyword arguments for *facecolor* and *edgecolor* has resulted in three behavior changes:

1. Color mapping can be turned off by calling `Collection.set_array(None)`. Previously, this would have no effect.
2. When a mappable array is set, with `facecolor='none'` and `edgecolor='face'`, both the faces and the edges are left uncolored. Previously the edges would be color-mapped.
3. When a mappable array is set, with `facecolor='none'` and `edgecolor='red'`, the edges are red. This addresses Issue #1302. Previously the edges would be color-mapped.

#### Transparency (alpha) can be set as an array in collections

Previously, the alpha value controlling transparency in collections could be specified only as a scalar applied to all elements in the collection. For example, all the markers in a *scatter* plot, or all the quadrilaterals in a *pcolormesh* plot, would have the same alpha value.

Now it is possible to supply alpha as an array with one value for each element (marker, quadrilateral, etc.) in a collection.

#### pcolormesh has improved transparency handling by enabling snapping

Due to how the snapping keyword argument was getting passed to the Agg backend, previous versions of Matplotlib would appear to show lines between the grid edges of a mesh with transparency. This version now applies snapping by default. To restore the old behavior (e.g., for test images), you may set *rcParams["pcolormesh.snap"]* (default: `True`) to `False`.

Note that there are lines between the grid boundaries of the main plot which are not the same transparency. The colorbar also shows these lines when a transparency is added to the colormap because internally it uses pcolormesh to draw the colorbar. With snapping on by default (below), the lines at the grid boundaries disappear.

### IPython representations for Colormap objects

The *matplotlib.colors.Colormap* object now has image representations for IPython / Jupyter backends. Cells returning a colormap on the last line will display an image of the colormap.

### `Colormap.set_extremes` and `Colormap.with_extremes`

Because the *Colormap.set_bad*, *Colormap.set_under* and *Colormap.set_over* methods modify the colormap in place, the user must be careful to first make a copy of the colormap if setting the extreme colors e.g. for a builtin colormap.

The new `Colormap.with_extremes(bad=..., under=..., over=...)` can be used to first copy the colormap and set the extreme colors on that copy.

The new *Colormap.set_extremes* method is provided for API symmetry with *Colormap.with_extremes*, but note that it suffers from the same issue as the earlier individual setters.

### Get under/over/bad colors of Colormap objects

*matplotlib.colors.Colormap* now has methods *get_under*, *get_over*, *get_bad* for the colors used for out-of-range and masked values.

### New `cm.unregister_cmap` function

`matplotlib.cm.unregister_cmap` allows users to remove a colormap that they have previously registered.

### New `CenteredNorm` for symmetrical data around a center

In cases where data is symmetrical around a center, for example, positive and negative anomalies around a center zero, *CenteredNorm* is a new norm that automatically creates a symmetrical mapping around the center. This norm is well suited to be combined with a divergent colormap which uses an unsaturated color in its center.

If the center of symmetry is different from 0, it can be set with the *vcenter* argument. To manually set the range of *CenteredNorm*, use the *halfrange* argument.

See *Colormap normalization* for an example and more details about data normalization.

### New `FuncNorm` for arbitrary normalizations

The *`FuncNorm`* allows for arbitrary normalization using functions for the forward and inverse.

See *Colormap normalization* for an example and more details about data normalization.

### GridSpec-based colorbars can now be positioned above or to the left of the main axes

... by passing `location="top"` or `location="left"` to the `colorbar()` call.

### Titles, ticks, and labels

### supxlabel and supylabel

It is possible to add x- and y-labels to a whole figure, analogous to *`Figure.suptitle`* using the new *`Figure.supxlabel`* and *`Figure.supylabel`* methods.

### Shared-axes `subplots` tick label visibility is now correct for top or left labels

When calling `subplots(..., sharex=True, sharey=True)`, Matplotlib automatically hides x tick labels for Axes not in the first column and y tick labels for Axes not in the last row. This behavior is incorrect if rcParams specify that Axes should be labeled on the top (`rcParams["xtick.labeltop"] = True`) or on the right (`rcParams["ytick.labelright"] = True`).

Cases such as the following are now handled correctly (adjusting visibility as needed on the first row and last column of Axes):

```
plt.rcParams["xtick.labelbottom"] = False
plt.rcParams["xtick.labeltop"] = True
plt.rcParams["ytick.labelleft"] = False
plt.rcParams["ytick.labelright"] = True

fig, axs = plt.subplots(2, 2, sharex=True, sharey=True)
```

### An iterable object with labels can be passed to `Axes.plot`

When plotting multiple datasets by passing 2D data as *y* value to *plot*, labels for the datasets can be passed as a list, the length matching the number of columns in *y*.

```
x = [1, 2, 3]

y = [[1, 2],
     [2, 5],
     [4, 9]]

plt.plot(x, y, label=['low', 'high'])
plt.legend()
```

## Fonts and Text

### Text transform can rotate text direction

The new *Text* parameter `transform_rotates_text` now sets whether rotations of the transform affect the text direction.

### `matplotlib.mathtext` now supports *overset* and *underset* LaTeX symbols

*mathtext* now supports *overset* and *underset*, called as `\overset{annotation}{body}` or `\underset{annotation}{body}`, where *annotation* is the text "above" or "below" the *body*.

### *math_fontfamily* parameter to change `Text` font family

The new *math_fontfamily* parameter may be used to change the family of fonts for each individual text element in a plot. If no parameter is set, the global value *rcParams["mathtext.fontset"]* (default: `'dejavusans'`) will be used.

### `TextArea`/`AnchoredText` support *horizontalalignment*

The horizontal alignment of text in a *TextArea* or *AnchoredText* may now be specified, which is mostly effective for multiline text:

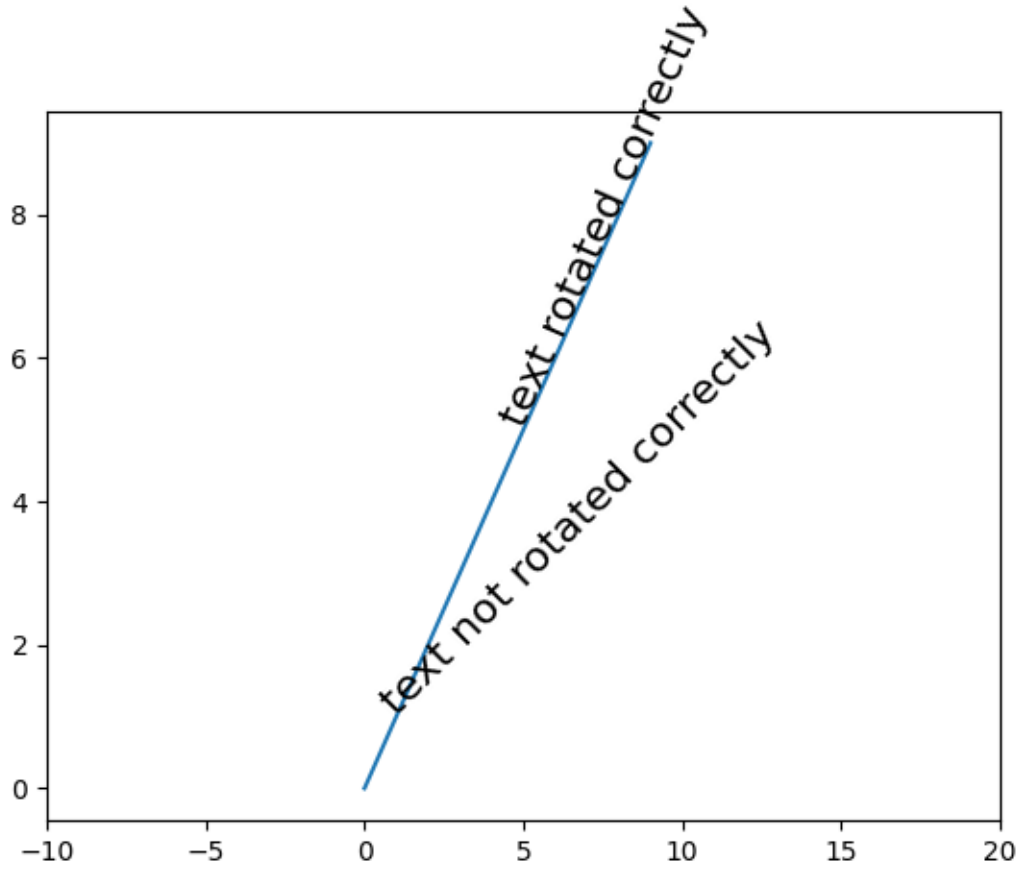
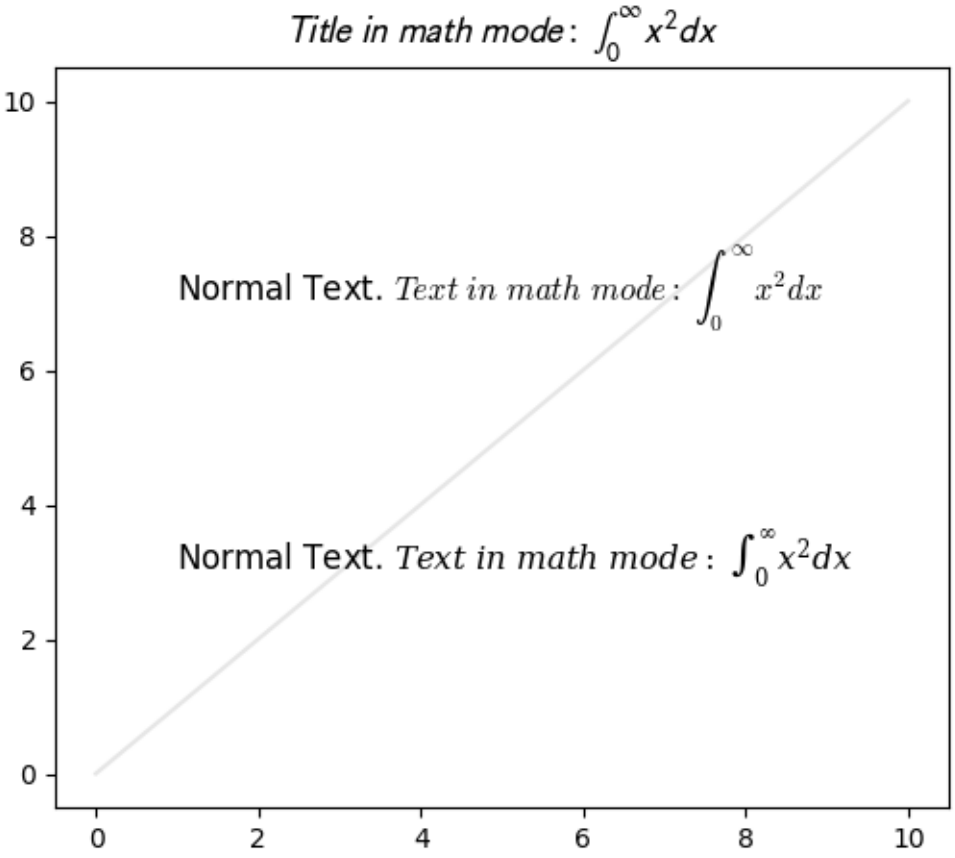



Fig. 3: Example of the new *transform_rotates_text* parameter


Title in math mode: $\int_0^\infty x^2 dx$
Normal Text. $Text\ in\ math\ mode: \int_0^\infty x^2 dx$
Normal Text. $Text\ in\ math\ mode: \int_0^\infty x^2 dx$
0
2
4
6
8
10

### PDF supports URLs on `Text` artists

URLs on *text.Text* artists (i.e., from *Artist.set_url*) will now be saved in PDF files.

### rcParams improvements

### New rcParams for dates: set converter and whether to use interval_multiples

The new *rcParams["date.converter"]* (default: `'auto'`) allows toggling between *matplotlib.dates.DateConverter* and *matplotlib.dates.ConciseDateConverter* using the strings 'auto' and 'concise' respectively.

The new *rcParams["date.interval_multiples"]* (default: `True`) allows toggling between the dates locator trying to pick ticks at set intervals (i.e., day 1 and 15 of the month), versus evenly spaced ticks that start wherever the timeseries starts:

```
dates = np.arange('2001-01-10', '2001-05-23', dtype='datetime64[D]')
y = np.sin(dates.astype(float) / 10)
fig, axs = plt.subplots(nrows=2, constrained_layout=True)

plt.rcParams['date.converter'] = 'concise'
plt.rcParams['date.interval_multiples'] = True
axs[0].plot(dates, y)

plt.rcParams['date.converter'] = 'auto'
plt.rcParams['date.interval_multiples'] = False
axs[1].plot(dates, y)
```

### Date formatters now respect *usetex* rcParam

The *AutoDateFormatter* and *ConciseDateFormatter* now respect *rcParams["text.usetex"]* (default: `False`), and will thus use fonts consistent with TeX rendering of the default (non-date) formatter. TeX rendering may also be enabled/disabled by passing the *usetex* parameter when creating the formatter instance.

In the following plot, both the x-axis (dates) and y-axis (numbers) now use the same (TeX) font:

### Setting *image.cmap* to a `Colormap`

It is now possible to set *rcParams["image.cmap"]* (default: `'viridis'`) to a *Colormap* instance, such as a colormap created with the new *set_extremes* above. (This can only be done from Python code, not from the `matplotlibrc` file.)

### Tick and tick label colors can be set independently using rcParams

Previously, *rcParams["xtick.color"]* (default: `'black'`) defined both the tick color and the label color. The label color can now be set independently using *rcParams["xtick.labelcolor"]* (default: `'inherit'`). It defaults to `'inherit'` which will take the value from *rcParams["xtick.color"]* (default: `'black'`). The same holds for `ytick.[label]color`. For instance, to set the ticks to light grey and the tick labels to black, one can use the following code in a script:

```
import matplotlib as mpl

mpl.rcParams['xtick.labelcolor'] = 'lightgrey'
mpl.rcParams['xtick.color'] = 'black'
mpl.rcParams['ytick.labelcolor'] = 'lightgrey'
mpl.rcParams['ytick.color'] = 'black'
```

Or by adding the following lines to the *matplotlibrc* file, or a Matplotlib style file:

```
xtick.labelcolor : lightgrey
xtick.color      : black
ytick.labelcolor : lightgrey
ytick.color      : black
```

### 3D Axes improvements

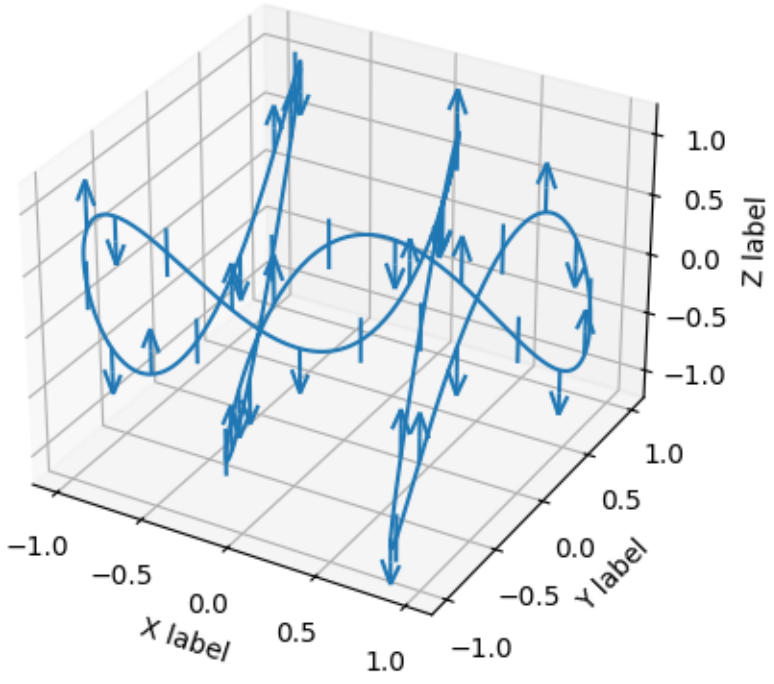
### Errorbar method in 3D Axes

The errorbar function *Axes.errorbar* is ported into the 3D Axes framework in its entirety, supporting features such as custom styling for error lines and cap marks, control over errorbar spacing, upper and lower limit marks.

### Stem plots in 3D Axes

Stem plots are now supported on 3D Axes. Much like 2D stems, *stem* supports plotting the stems in various orientations:

See also the *3D stem* demo.


1.0
0.5
0.0
−0.5
−1.0
Z label
1.0
0.5
0.0
−0.5
−1.0
Y label
−1.0
−0.5
0.0
0.5
1.0
X label

### 3D Collection properties are now modifiable

Previously, properties of a 3D Collection that were used for 3D effects (e.g., colors were modified to produce depth shading) could not be changed after it was created.

Now it is possible to modify all properties of 3D Collections at any time.

### Panning in 3D Axes

Click and drag with the middle mouse button to pan 3D Axes.

### Interactive tool improvements

#### New `RangeSlider` widget

*widgets.RangeSlider* allows for creating a slider that defines a range rather than a single value.

### Sliders can now snap to arbitrary values

The *Slider* UI widget now accepts arrays for *valstep*. This generalizes the previous behavior by allowing the slider to snap to arbitrary values.

### Pausing and Resuming Animations

The *animation.Animation.pause* and *animation.Animation.resume* methods allow you to pause and resume animations. These methods can be used as callbacks for event listeners on UI elements so that your plots can have some playback control UI.

### Sphinx extensions

#### `plot_directive` *caption* option

Captions were previously supported when using the `plot_directive` directive with an external source file by specifying content:

```
.. plot:: path/to/plot.py

    This is the caption for the plot.
```

The `:caption:` option allows specifying the caption for both external:

```
.. plot:: path/to/plot.py
    :caption: This is the caption for the plot.
```

and inline plots:

```
.. plot::
    :caption: This is a caption for the plot.

    plt.plot([1, 2, 3])
```

### Backend-specific improvements

### Consecutive rasterized draws now merged

Elements of a vector output can be individually set to rasterized, using the *rasterized* keyword argument, or *`set_rasterized()`*. This can be useful to reduce file sizes. For figures with multiple raster elements they are now automatically merged into a smaller number of bitmaps where this will not effect the visual output. For cases with many elements this can result in significantly smaller file sizes.

To ensure this happens do not place vector elements between raster ones.

To inhibit this merging set `Figure.suppressComposite` to True.

### Support raw/rgba frame format in `FFMpegFileWriter`

When using *`FFMpegFileWriter`*, the *frame_format* may now be set to `"raw"` or `"rgba"`, which may be slightly faster than an image format, as no encoding/decoding need take place between Matplotlib and FFmpeg.

### nbAgg/WebAgg support middle-click and double-click

Double click events are now supported by the nbAgg and WebAgg backends. Formerly, WebAgg would report middle-click events as right clicks, but now reports the correct button type.

### nbAgg support binary communication

If the web browser and notebook support binary websockets, nbAgg will now use them for slightly improved transfer of figure display.

#### Indexed color for PNG images in PDF files when possible

When PNG images have 256 colors or fewer, they are converted to indexed color before saving them in a PDF. This can result in a significant reduction in file size in some cases. This is particularly true for raster data that uses a colormap but no interpolation, such as Healpy mollview plots. Currently, this is only done for RGB images.

#### Improved font subsettings in PDF/PS

Font subsetting in PDF and PostScript has been re-written from the embedded `ttconv` C code to Python. Some composite characters and outlines may have changed slightly. This fixes ttc subsetting in PDF, and adds support for subsetting of type 3 OTF fonts, resulting in smaller files (much smaller when using CJK fonts), and avoids running into issues with type 42 embedding and certain PDF readers such as Acrobat Reader.

#### Kerning added to strings in PDFs

As with text produced in the Agg backend (see *the previous what's new entry* for examples), PDFs now include kerning in text strings.

#### Fully-fractional HiDPI in QtAgg

Fully-fractional HiDPI (that is, HiDPI ratios that are not whole integers) was added in Qt 5.14, and is now supported by the QtAgg backend when using this version of Qt or newer.

#### wxAgg supports fullscreen toggle

The wxAgg backend supports toggling fullscreen using the `f` shortcut, or the manager function *`FigureManagerBase.full_screen_toggle`*.

### 9.6.2 API Changes for 3.4.2

#### Behaviour changes

#### Rename first argument to `subplot_mosaic`

Both *`Figure.subplot_mosaic`*, and *`pyplot.subplot_mosaic`* have had the first position argument renamed from *layout* to *mosaic*. This is because we are considering to consolidate *constrained_layout* and *tight_layout* keyword arguments in the Figure creation functions of *`pyplot`* into a single *layout* keyword argument which would collide.

As this API is provisional, we are changing this with no deprecation period.

### 9.6.3 API Changes for 3.4.0


- *Behaviour changes*
- *Deprecations*
- *Removals*
- *Development changes*


#### Behaviour changes

#### Constrained layout rewrite

The layout manager `constrained_layout` was re-written with different outer constraints that should be more robust to complicated subplot layouts. User-facing changes are:

- some poorly constrained layouts will have different width/height plots than before.
- colorbars now respect the `anchor` keyword argument of *`matplotlib.colorbar.make_axes`*
- colorbars are wider.
- colorbars in different rows or columns line up more robustly.
- *hspace* and *wspace* options to *`Figure.set_constrained_layout_pads`* were twice as wide as the docs said they should be. So these now follow the docs.

This feature will remain "experimental" until the new changes have been used enough by users, so we anticipate version 3.5 or 3.6. On the other hand, `constrained_layout` is extensively tested and used in examples in the library, so using it should be safe, but layouts may not be exactly the same as more development takes place.

Details of using `constrained_layout`, and its algorithm are available at *Constrained layout guide*

#### `plt.subplot` re-selection without keyword arguments

The purpose of *`pyplot.subplot`* is to facilitate creating and re-selecting Axes in a Figure when working strictly in the implicit pyplot API. When creating new Axes it is possible to select the projection (e.g. polar, 3D, or various cartographic projections) as well as to pass additional keyword arguments through to the Axes-subclass that is created.

The first time *`pyplot.subplot`* is called for a given position in the Axes grid it always creates and returns a new Axes with the passed arguments and projection (defaulting to rectilinear). On subsequent calls to *`pyplot.subplot`* we have to determine if an existing Axes has a) equivalent parameters, in which case it should be selected as the current Axes and returned, or b) different parameters, in which case a new Axes is created and the existing Axes is removed. This leaves the question of what is "equivalent parameters".

Previously it was the case that an existing Axes subclass, except for Axes3D, would be considered equivalent to a 2D rectilinear Axes, despite having different projections, if the keyword arguments (other than *projection*) matched. Thus:

```
ax1 = plt.subplot(1, 1, 1, projection='polar')
ax2 =  plt.subplots(1, 1, 1)
ax1 is ax2
```

We are embracing this long standing behavior to ensure that in the case when no keyword arguments (of any sort) are passed to *pyplot.subplot* any existing Axes is returned, without consideration for keywords or projection used to initially create it. This will cause a change in behavior when additional keywords were passed to the original Axes:

```
ax1 = plt.subplot(111, projection='polar', theta_offset=.75)
ax2 = plt.subplots(1, 1, 1)
ax1 is ax2          # new behavior
# ax1 is not ax2    # old behavior, made a new axes

ax1 = plt.subplot(111, label='test')
ax2 = plt.subplots(1, 1, 1)
ax1 is ax2          # new behavior
# ax1 is not ax2    # old behavior, made a new axes
```

For the same reason, if there was an existing Axes that was not rectilinear, passing `projection='rectilinear'` would reuse the existing Axes

```
ax1 = plt.subplot(projection='polar')
ax2 = plt.subplot(projection='rectilinear')
ax1 is not ax2      # new behavior, makes new Axes
# ax1 is ax2        # old behavior
```

contrary to the user's request.

Previously Axes3D could not be re-selected with *pyplot.subplot* due to an unrelated bug (also fixed in Matplotlib 3.4). While Axes3D are now consistent with all other projections there is a change in behavior for

```
plt.subplot(projection='3d')   # create a 3D Axes

plt.subplot()                  # now returns existing 3D Axes, but
                               # previously created new 2D Axes

plt.subplot(projection='rectilinear')  # to get a new 2D Axes
```

### `ioff` and `ion` can be used as context managers

*pyplot.ion* and *pyplot.ioff* may now be used as context managers to create a context with interactive mode on or off, respectively. The old behavior of calling these functions is maintained. To use the new functionality call as:

```
with plt.ioff():
   # non-interactive code
```

### Locators and formatters must be in the class hierarchy

Axis locators and formatters must now be subclasses of *Locator* and *Formatter* respectively.

### Date locator for DAILY interval now returns middle of month

The *matplotlib.dates.AutoDateLocator* has a default of `interval_multiples=True` that attempts to align ticks with the start of meaningful intervals like the start of the month, or start of the day, etc. That lead to approximately 140-day intervals being mapped to the first and 22nd of the month. This has now been changed so that it chooses the first and 15th of the month, which is probably what most people want.

### `ScalarFormatter` *useLocale* option obeys grouping

When the *ScalarFormatter* option *useLocale* is enabled (or *rcParams["axes.formatter.use_locale"]* (default: `False`) is *True*) and the configured locale uses grouping, a separator will be added as described in `locale.format_string`.

### `Axes.errorbar` cycles non-color properties correctly

Formerly, *Axes.errorbar* incorrectly skipped the Axes property cycle if a color was explicitly specified, even if the property cycler was for other properties (such as line style). Now, *Axes.errorbar* will advance the Axes property cycle as done for *Axes.plot*, i.e., as long as all properties in the cycler are not explicitly passed.

### pyplot.specgram always uses origin='upper'

Previously if *rcParams["image.origin"]* (default: `'upper'`) was set to something other than `'upper'` or if the *origin* keyword argument was passed with a value other than `'upper'`, the spectrogram itself would flip, but the Axes would remain oriented for an origin value of `'upper'`, so that the resulting plot was incorrectly labelled.

Now, the *origin* keyword argument is not supported and the `image.origin` rcParam is ignored. The function *matplotlib.pyplot.specgram* is forced to use `origin='upper'`, so that the Axes are correct for the plotted spectrogram.

### xunits=None and yunits=None passed as keyword arguments are treated as "no action"

Many (but not all) of the methods on *Axes* take the (undocumented) keyword arguments *xunits* and *yunits* that will update the units on the given Axis by calling *Axis.set_units* and *Axis.update_units*.

Previously if *None* was passed it would clear the value stored in `.Axis.units` which will in turn break converters which rely on the value in `.Axis.units` to work properly (notably *StrCategoryConverter*).

This changes the semantics of `ax.meth(..., xunits=None, yunits=None)` from "please clear the units" to "do the default thing as if they had not been passed" which is consistent with the standard behavior of Matplotlib keyword arguments.

If you were relying on passing `xunits=None` to plotting methods to clear the `.Axes.units` attribute, directly call *Axis.set_units* (and *Axis.update_units* if you also require the converter to be updated).

### Annotations with `annotation_clip` no longer affect `tight_layout`

Previously, *text.Annotation.get_tightbbox* always returned the full *text.Annotation.get_window_extent* of the object, independent of the value of `annotation_clip`. *text.Annotation.get_tightbbox* now correctly takes this extra clipping box into account, meaning that *Annotation*s that are not drawn because of `annotation_clip` will not count towards the Axes bounding box calculations, such as those done by *tight_layout*.

This is now consistent with the API described in *Artist*, which specifies that `get_window_extent` should return the full extents and `get_tightbbox` should "account for any clipping".

### Parasite Axes pcolor and pcolormesh now defaults to placing grid edges at integers, not half-integers

This is consistent with *pcolor* and *pcolormesh*.

### `Colorbar` outline is now a `Spine`

The outline of *Colorbar* is now a *Spine* and drawn as one, instead of a *Polygon* drawn as an artist. This ensures it will always be drawn after (i.e., on top of) all artists, consistent with Spines on normal Axes.

### `Colorbar.dividers` changes

This attribute is now always a *LineCollection* -- an empty one if `drawedges` is *False*. Its default colors and linewidth (*rcParams["axes.edgecolor"]* (default: `'black'`), *rcParams["axes.linewidth"]* (default: `0.8`)) are now resolved at instantiation time, not at draw time.

### Raise or warn on registering a colormap twice

When using `matplotlib.cm.register_cmap` to register a user provided or third-party colormap it will now raise a `ValueError` if trying to over-write one of the built in colormaps and warn if trying to over write a user registered colormap. This may raise for user-registered colormaps in the future.

### Consecutive rasterized draws now merged

Tracking of depth of raster draws has moved from *backend_mixed.MixedModeRenderer.start_rasterizing* and *backend_mixed.MixedModeRenderer.stop_rasterizing* into *artist.allow_rasterization*. This means the start and stop functions are only called when the rasterization actually needs to be started and stopped.

The output of vector backends will change in the case that rasterized elements are merged. This should not change the appearance of outputs.

The renders in 3rd party backends are now expected to have `self._raster_depth` and `self._rasterizing` initialized to `0` and *False* respectively.

### Consistent behavior of `draw_if_interactive()` across backends

*pyplot.draw_if_interactive* no longer shows the window (if it was previously unshown) on the Tk and nbAgg backends, consistently with all other backends.

### The Artist property *rasterized* cannot be *None* anymore

It is now a boolean only. Before the default was *None* and *Artist.set_rasterized* was documented to accept *None*. However, *None* did not have a special meaning and was treated as *False*.

### Canvas's callback registry now stored on Figure

The canonical location of the *CallbackRegistry* used to handle Figure/Canvas events has been moved from the Canvas to the Figure. This change should be transparent to almost all users, however if you are swapping switching the Figure out from on top of a Canvas or visa versa you may see a change in behavior.

### Harmonized key event data across backends

The different backends with key translation support, now handle "Shift" as a sometimes modifier, where the `'shift+'` prefix won't be added if a key translation was made.

In the Qt5 backend, the `matplotlib.backends.backend_qt5.SPECIAL_KEYS` dictionary contains keys that do *not* return their unicode name instead they have manually specified names. The name for `QtCore.Qt.Key_Meta` has changed to `'meta'` to be consistent with the other GUI backends.

The WebAgg backend now handles key translations correctly on non-US keyboard layouts.

In the GTK and Tk backends, the handling of non-ASCII keypresses (as reported in the KeyEvent passed to `key_press_event`-handlers) now correctly reports Unicode characters (e.g., €), and better respects NumLock on the numpad.

In the GTK and Tk backends, the following key names have changed; the new names are consistent with those reported by the Qt backends:

- The "Break/Pause" key (keysym 0xff13) is now reported as `"pause"` instead of `"break"` (this is also consistent with the X key name).
- The numpad "delete" key is now reported as `"delete"` instead of `"dec"`.

### WebAgg backend no longer reports a middle click as a right click

Previously when using the WebAgg backend the event passed to a callback by `fig.canvas.mpl_connect('mouse_button_event', callback)` on a middle click would report *`MouseButton.RIGHT`* instead of *`MouseButton.MIDDLE`*.

### ID attribute of XML tags in SVG files now based on SHA256 rather than MD5

Matplotlib generates unique ID attributes for various tags in SVG files. Matplotlib previously generated these unique IDs using the first 10 characters of an MD5 hash. The MD5 hashing algorithm is not available in Python on systems with Federal Information Processing Standards (FIPS) enabled. Matplotlib now uses the first 10 characters of an SHA256 hash instead. SVG files that would otherwise match those saved with earlier versions of matplotlib, will have different ID attributes.

### `RendererPS.set_font` is no longer a no-op in AFM mode

*`RendererPS.set_font`* now sets the current PostScript font in all cases.

### Autoscaling in Axes3D

In Matplotlib 3.2.0, autoscaling was made lazier for 2D Axes, i.e., limits would only be recomputed when actually rendering the canvas, or when the user queries the Axes limits. This performance improvement is now extended to *Axes3D*. This also fixes some issues with autoscaling being triggered unexpectedly in Axes3D.

Please see *the API change for 2D Axes* for further details.

### Axes3D automatically adding itself to Figure is deprecated

New *Axes3D* objects previously added themselves to figures when they were created, unlike all other Axes classes, which lead to them being added twice if `fig.add_subplot(111, projection='3d')` was called.

This behavior is now deprecated and will warn. The new keyword argument *auto_add_to_figure* controls the behavior and can be used to suppress the warning. The default value will change to *False* in Matplotlib 3.5, and any non-*False* value will be an error in Matplotlib 3.6.

In the future, *Axes3D* will need to be explicitly added to the figure

```
fig = Figure()
# create Axes3D
ax = Axes3d(fig)
# add to Figure
fig.add_axes(ax)
```

as needs to be done for other *axes.Axes* sub-classes. Or, a 3D projection can be made via:

```
fig.add_subplot(projection='3d')
```

### `mplot3d.art3d.get_dir_vector` always returns NumPy arrays

For consistency, *get_dir_vector* now always returns NumPy arrays, even if the input is a 3-element iterable.

### Changed cursive and fantasy font definitions

The Comic Sans and Comic Neue fonts were moved from the default *rcParams["font.fantasy"]* (default: `['Chicago', 'Charcoal', 'Impact', 'Western', 'xkcd script', 'fantasy']`) list to the default *rcParams["font.cursive"]* (default: `['Apple Chancery', 'Textile', 'Zapf Chancery', 'Sand', 'Script MT', 'Felipa', 'Comic Neue', 'Comic Sans MS', 'cursive']`) setting, in accordance with the CSS font families example and in order to provide a cursive font present in Microsoft's Core Fonts set.

### docstring.Substitution now always dedents docstrings before string interpolation

### Deprecations

### Extra parameters to Axes constructor

Parameters of the Axes constructor other than *fig* and *rect* will become keyword-only in a future version.

### `pyplot.gca` and `Figure.gca` keyword arguments

Passing keyword arguments to *`pyplot.gca`* or *`figure.Figure.gca`* will not be supported in a future release.

### `Axis.cla`, `RadialAxis.cla`, `ThetaAxis.cla` and `Spine.cla`

These methods are deprecated in favor of the respective `clear()` methods.

### Invalid hatch pattern characters are no longer ignored

When specifying hatching patterns, characters that are not recognized will raise a deprecation warning. In the future, this will become a hard error.

### `imread` reading from URLs

Passing a URL to *`imread()`* is deprecated. Please open the URL for reading and directly use the Pillow API (`PIL.Image.open(urllib.request.urlopen(url))`, or `PIL.Image.open(io.BytesIO(requests.get(url).content))`) instead.

### Subplot-related attributes and methods

Some `SubplotBase` methods and attributes have been deprecated and/or moved to *`SubplotSpec`*:

- `get_geometry` (use `SubplotBase.get_subplotspec` instead),
- `change_geometry` (use `SubplotBase.set_subplotspec` instead),
- `is_first_row`, `is_last_row`, `is_first_col`, `is_last_col` (use the corresponding methods on the *`SubplotSpec`* instance instead),
- `update_params` (now a no-op),
- `figbox` (use `ax.get_subplotspec().get_geometry(ax.figure)` instead to recompute the geometry, or `ax.get_position()` to read its current value),
- `numRows`, `numCols` (use the `nrows` and `ncols` attribute on the *`GridSpec`* instead).

Likewise, the `get_geometry`, `change_geometry`, `update_params`, and `figbox` methods/attributes of *`SubplotDivider`* have been deprecated, with similar replacements.

#### `is_url` and `URL_REGEX`

... are deprecated. (They were previously defined in the toplevel *`matplotlib`* module.)

#### `matplotlib.style.core` deprecations

`STYLE_FILE_PATTERN`, `load_base_library`, and `iter_user_libraries` are deprecated.

#### `dpi_cor` property of `FancyArrowPatch`

This parameter is considered internal and deprecated.

#### Passing `boxstyle="custom", bbox_transmuter=...` to `FancyBboxPatch`

In order to use a custom boxstyle, directly pass it as the *boxstyle* argument to *`FancyBboxPatch`*. This was previously already possible, and is consistent with custom arrow styles and connection styles.

#### BoxStyles are now called without passing the *mutation_aspect* parameter

Mutation aspect is now handled by the artist itself. Hence the *mutation_aspect* parameter of `BoxStyle._Base.__call__` is deprecated, and custom boxstyles should be implemented to not require this parameter (it can be left as a parameter defaulting to 1 for back-compatibility).

#### `ContourLabeler.get_label_coords` is deprecated

It is considered an internal helper.

#### Line2D and Patch no longer duplicate `validJoin` and `validCap`

Validation of joinstyle and capstyles is now centralized in `rcsetup`.

### Setting a Line2D's pickradius via `set_picker` is undeprecated

This cancels the deprecation introduced in Matplotlib 3.3.0.

### `MarkerStyle` is considered immutable

`MarkerStyle.set_fillstyle()` and `MarkerStyle.set_marker()` are deprecated. Create a new `MarkerStyle` with the respective parameters instead.

### `MovieWriter.cleanup` is deprecated

Cleanup logic is now fully implemented in *`MovieWriter.finish`*. Third-party movie writers should likewise move the relevant cleanup logic there, as overridden `cleanup`s will no longer be called in the future.

### *minimumdescent* parameter/property of `TextArea`

*`offsetbox.TextArea`* has behaved as if *minimumdescent* was always True (regardless of the value to which it was set) since Matplotlib 1.3, so the parameter/property is deprecated.

### `colorbar` now warns when the mappable's Axes is different from the current Axes

Currently, *`Figure.colorbar`* and *`pyplot.colorbar`* steal space by default from the current Axes to place the colorbar. In a future version, they will steal space from the mappable's Axes instead. In preparation for this change, *`Figure.colorbar`* and *`pyplot.colorbar`* now emits a warning when the current Axes is not the same as the mappable's Axes.

### Colorbar docstrings

The following globals in *`matplotlib.colorbar`* are deprecated: `colorbar_doc`, `colormap_kw_doc`, `make_axes_kw_doc`.

### `ColorbarPatch` and `colorbar_factory` are deprecated

All the relevant functionality has been moved to the *`Colorbar`* class.

### Backend deprecations

- `FigureCanvasBase.get_window_title` and `FigureCanvasBase.set_window_title` are deprecated. Use the corresponding methods on the FigureManager if using pyplot, or GUI-specific methods if embedding.
- The *resize_callback* parameter to `FigureCanvasTk` was never used internally and is deprecated. Tk-level custom event handlers for resize events can be added to a `FigureCanvasTk` using e.g. `get_tk_widget().bind('<Configure>', ..., True)`.
- The `key_press` and `button_press` methods of *FigureManagerBase*, which incorrectly did nothing when using `toolmanager`, are deprecated in favor of directly passing the event to the *CallbackRegistry* via `self.canvas.callbacks.process(event.name, event)`.
- `RendererAgg.get_content_extents` and `RendererAgg.tostring_rgba_minimized` are deprecated.
- `backend_pgf.TmpDirCleaner` is deprecated, with no replacement.
- `GraphicsContextPS` is deprecated. The PostScript backend now uses *GraphicsContextBase*.

### wx backend cleanups

The *origin* parameter to `_FigureCanvasWxBase.gui_repaint` is deprecated with no replacement; `gui_repaint` now automatically detects the case where it is used with the wx renderer.

The `NavigationToolbar2Wx.get_canvas` method is deprecated; directly instantiate a canvas (`FigureCanvasWxAgg(frame, -1, figure)`) if needed.

### Unused positional parameters to `print_<fmt>` methods are deprecated

None of the `print_<fmt>` methods implemented by canvas subclasses used positional arguments other that the first (the output filename or file-like), so these extra parameters are deprecated.

### The *dpi* parameter of `FigureCanvas.print_foo` printers is deprecated

The *savefig* machinery already took care of setting the figure DPI to the desired value, so `print_foo` can directly read it from there. Not passing *dpi* to `print_foo` allows clearer detection of unused parameters passed to *savefig*.

### Passing `bytes` to `FT2Font.set_text`

... is deprecated, pass `str` instead.

### `ps.useafm` deprecated for mathtext

Outputting mathtext using only standard PostScript fonts has likely been broken for a while (issue #18722). In Matplotlib 3.5, the setting *rcParams["ps.useafm"]* (default: `False`) will have no effect on mathtext.

### `MathTextParser("bitmap")` is deprecated

The associated APIs `MathtextBackendBitmap`, `MathTextParser.to_mask`, `MathTextParser.to_rgba`, `MathTextParser.to_png`, and `MathTextParser.get_depth` are likewise deprecated.

To convert a text string to an image, either directly draw the text to an empty *Figure* and save the figure using a tight bbox, as demonstrated in *Convert texts to images*, or use *mathtext.math_to_image*.

When using *math_to_image*, text color can be set with e.g.:

```
with plt.rc_context({"text.color": "tab:blue"}):
    mathtext.math_to_image(text, filename)
```

and an RGBA array can be obtained with e.g.:

```
from io import BytesIO
buf = BytesIO()
mathtext.math_to_image(text, buf, format="png")
buf.seek(0)
rgba = plt.imread(buf)
```

### Deprecation of mathtext internals

The following API elements previously exposed by the *mathtext* module are considered to be implementation details and public access to them is deprecated:

- `Fonts` and all its subclasses,
- `FontConstantsBase` and all its subclasses,
- `Node` and all its subclasses,
- `Ship`, `ship`,
- `Error`,
- `Parser`,
- `SHRINK_FACTOR`, `GROW_FACTOR`,

- NUM_SIZE_LEVELS,
- latex_to_bakoma, latex_to_cmex, latex_to_standard,
- stix_virtual_fonts,
- tex2uni.

### Deprecation of various mathtext helpers

The MathtextBackendPdf, MathtextBackendPs, MathtextBackendSvg, and MathtextBackendCairo classes from the *mathtext* module, as well as the corresponding .mathtext_parser attributes on RendererPdf, RendererPS, RendererSVG, and RendererCairo, are deprecated. The MathtextBackendPath class can be used to obtain a list of glyphs and rectangles in a mathtext expression, and renderer-specific logic should be directly implemented in the renderer.

StandardPsFonts.pswriter is unused and deprecated.

### Widget class internals

Several *widgets.Widget* class internals have been privatized and deprecated:

- AxesWidget.cids
- Button.cnt and Button.observers
- CheckButtons.cnt and CheckButtons.observers
- RadioButtons.cnt and RadioButtons.observers
- Slider.cnt and Slider.observers
- TextBox.cnt, TextBox.change_observers and TextBox.submit_observers

### 3D properties on renderers

The properties of the 3D Axes that were placed on the Renderer during draw are now deprecated:

- renderer.M
- renderer.eye
- renderer.vvec
- renderer.get_axis_position

These attributes are all available via *Axes3D*, which can be accessed via self.axes on all *Artist*s.

#### *renderer* argument of `do_3d_projection` method for `Collection3D`/`Patch3D`

The *renderer* argument for the `do_3d_projection` method on `Collection3D` and `Patch3D` is no longer necessary, and passing it during draw is deprecated.

#### *project* argument of `draw` method for `Line3DCollection`

The *project* argument for the `draw` method on `Line3DCollection` is deprecated. Call *`Line3DCollection.do_3d_projection`* explicitly instead.

### Extra positional parameters to `plot_surface` and `plot_wireframe`

Positional parameters to *`plot_surface`* and *`plot_wireframe`* other than `X`, `Y`, and `Z` are deprecated. Pass additional artist properties as keyword arguments instead.

#### `ParasiteAxesAuxTransBase` class

The functionality of that mixin class has been moved to the base `ParasiteAxesBase` class. Thus, `ParasiteAxesAuxTransBase`, `ParasiteAxesAuxTrans`, and `parasite_axes_auxtrans_class_factory` are deprecated.

In general, it is suggested to use `HostAxes.get_aux_axes` to create parasite Axes, as this saves the need of manually appending the parasite to `host.parasites` and makes sure that their `remove()` method works properly.

#### `AxisArtist.ZORDER` attribute

Use `AxisArtist.zorder` instead.

#### `GridHelperBase` invalidation

The `GridHelperBase.invalidate`, `GridHelperBase.valid`, and `axislines.Axes.invalidate_grid_helper` methods are considered internal and deprecated.

**sphinxext.plot_directive.align**

... is deprecated. Use `docutils.parsers.rst.directives.images.Image.align` instead.

### Deprecation-related functionality is considered internal

The module `matplotlib.cbook.deprecation` is considered internal and will be removed from the public API. This also holds for deprecation-related re-imports in `matplotlib.cbook`, i.e. `matplotlib.cbook.deprecated()`, `matplotlib.cbook.warn_deprecated()`, `matplotlib.cbook.MatplotlibDeprecationWarning` and `matplotlib.cbook.mplDeprecation`.

If needed, external users may import `MatplotlibDeprecationWarning` directly from the `matplotlib` namespace. `mplDeprecation` is only an alias of `MatplotlibDeprecationWarning` and should not be used anymore.

### Removals

The following deprecated APIs have been removed:

### Removed behaviour

- The "smart bounds" functionality on *`Axis`* and *`Spine`* has been deleted, and the related methods have been removed.
- Converting a string with single color characters (e.g. `'cymk'`) in *`to_rgba_array`* is no longer supported. Instead, the colors can be passed individually in a list (e.g. `['c', 'y', 'm', 'k']`).
- Returning a factor equal to `None` from `mpl_toolkits.axisartist` Locators (which are **not** the same as "standard" tick Locators), or passing a factor equal to `None` to axisartist Formatters (which are **not** the same as "standard" tick Formatters) is no longer supported. Pass a factor equal to 1 instead.

### Modules

- The entire `matplotlib.testing.disable_internet` module has been removed. The pytest-remotedata package can be used instead.
- The `mpl_toolkits.axes_grid1.colorbar` module and its colorbar implementation have been removed in favor of *`matplotlib.colorbar`*.

### Classes, methods and attributes

- The *animation.MovieWriterRegistry* methods `.set_dirty()`, `.ensure_not_dirty()`, and `.reset_available_writers()` do nothing and have been removed. The `.avail()` method has been removed; use `.list()` instead to get a list of available writers.
- The `matplotlib.artist.Artist.eventson` and `matplotlib.container.Container.eventson` attributes have no effect and have been removed.
- `matplotlib.axes.Axes.get_data_ratio_log` has been removed.
- `matplotlib.axes.SubplotBase.rowNum`; use `ax.get_subplotspec().rowspan.start` instead.
- `matplotlib.axes.SubplotBase.colNum`; use `ax.get_subplotspec().colspan.start` instead.
- `matplotlib.axis.Axis.set_smart_bounds` and `matplotlib.axis.Axis.get_smart_bounds` have been removed.
- `matplotlib.colors.DivergingNorm` has been renamed to *TwoSlopeNorm*.
- `matplotlib.figure.AxesStack` has been removed.
- `matplotlib.font_manager.JSONEncoder` has been removed; use *font_manager.json_dump* to dump a *FontManager* instance.
- The `matplotlib.ft2font.FT2Image` methods `.as_array()`, `.as_rgba_str()`, `.as_str()`, `.get_height()` and `.get_width()` have been removed. Convert the `FT2Image` to a NumPy array with `np.asarray` before processing it.
- `matplotlib.quiver.QuiverKey.quiverkey_doc` has been removed; use `matplotlib.quiver.QuiverKey.__init__.__doc__` instead.
- `matplotlib.spines.Spine.set_smart_bounds` and `matplotlib.spines.Spine.get_smart_bounds` have been removed.
- `matplotlib.testing.jpl_units.UnitDbl.checkUnits` has been removed; use `units not in self.allowed` instead.
- The unused `matplotlib.ticker.Locator.autoscale` method has been removed (pass the axis limits to *Locator.view_limits* instead). The derived methods `Locator.autoscale`, `AutoDateLocator.autoscale`, `RRuleLocator.autoscale`, `RadialLocator.autoscale`, `ThetaLocator.autoscale`, and `YearLocator.autoscale` have also been removed.
- `matplotlib.transforms.BboxBase.is_unit` has been removed; check the *Bbox* extents if needed.
- `matplotlib.transforms.Affine2DBase.matrix_from_values(...)` has been removed; use (for example) `Affine2D.from_values(...).get_matrix()` instead.
- `matplotlib.backend_bases.FigureCanvasBase.draw_cursor` has been removed.

- `matplotlib.backends.backend_gtk.ConfigureSubplotsGTK3.destroy` and `matplotlib.backends.backend_gtk.ConfigureSubplotsGTK3.init_window` methods have been removed.
- `matplotlib.backends.backend_gtk.ConfigureSubplotsGTK3.window` property has been removed.
- `matplotlib.backends.backend_macosx.FigureCanvasMac.invalidate` has been removed.
- `matplotlib.backends.backend_pgf.RendererPgf.latexManager` has been removed.
- `matplotlib.backends.backend_wx.FigureFrameWx.statusbar`, `matplotlib.backends.backend_wx.NavigationToolbar2Wx.set_status_bar`, and `matplotlib.backends.backend_wx.NavigationToolbar2Wx.statbar` have been removed. The status bar can be retrieved by calling standard wx methods (`frame.GetStatusBar()` and `toolbar.GetTopLevelParent().GetStatusBar()`).
- `matplotlib.backends.backend_wx.ConfigureSubplotsWx.configure_subplots` and `matplotlib.backends.backend_wx.ConfigureSubplotsWx.get_canvas` have been removed.
- `mpl_toolkits.axisartist.grid_finder.GridFinderBase` has been removed; use *`GridFinder`* instead.
- `mpl_toolkits.axisartist.axis_artist.BezierPath` has been removed; use *`patches.PathPatch`* instead.

### Functions

- `matplotlib.backends.backend_pgf.repl_escapetext` and `matplotlib.backends.backend_pgf.repl_mathdefault` have been removed.
- `matplotlib.checkdep_ps_distiller` has been removed.
- `matplotlib.cm.revcmap` has been removed; use *`Colormap.reversed`* instead.
- `matplotlib.colors.makeMappingArray` has been removed.
- `matplotlib.compare_versions` has been removed; use comparison of `distutils.version.LooseVersions` instead.
- `matplotlib.dates.mx2num` has been removed.
- `matplotlib.font_manager.createFontList` has been removed; *`font_manager.FontManager.addfont`* is now available to register a font at a given path.
- `matplotlib.get_home` has been removed; use standard library instead.
- `matplotlib.mlab.apply_window` and `matplotlib.mlab.stride_repeat` have been removed.
- `matplotlib.rcsetup.update_savefig_format` has been removed; this just replaced `'auto'` with `'png'`, so do the same.

- `matplotlib.rcsetup.validate_animation_writer_path` has been removed.
- `matplotlib.rcsetup.validate_path_exists` has been removed; use `os.path.exists` or `pathlib.Path.exists` instead.
- `matplotlib.style.core.is_style_file` and `matplotlib.style.core.iter_style_files` have been removed.
- `matplotlib.testing.is_called_from_pytest` has been removed.
- `mpl_toolkits.mplot3d.axes3d.unit_bbox` has been removed; use *`Bbox.unit`* instead.

### Arguments

- Passing more than one positional argument to *`axes.Axes.axis`* will now raise an error.
- Passing `"range"` to the *whis* parameter of *`Axes.boxplot`* and *`cbook.boxplot_stats`* to mean "the whole data range" is no longer supported.
- Passing scalars to the *where* parameter in *`axes.Axes.fill_between`* and *`axes.Axes.fill_betweenx`* is no longer accepted and non-matching sizes now raise a `ValueError`.
- The *verts* parameter to *`Axes.scatter`* has been removed; use *marker* instead.
- The *minor* parameter in *`Axis.set_ticks`* and `SecondaryAxis.set_ticks` is now keyword-only.
- *`scale.ScaleBase`*, *`scale.LinearScale`* and *`scale.SymmetricalLogScale`* now error if any unexpected keyword arguments are passed to their constructors.
- The *renderer* parameter to *`Figure.tight_layout`* has been removed; this method now always uses the renderer instance cached on the *`Figure`*.
- The *locator* parameter to *`mpl_toolkits.axes_grid1.axes_grid.CbarAxesBase.colorbar`* has been removed in favor of its synonym *ticks* (which already existed previously, and is consistent with *`matplotlib.colorbar`*).
- The *switch_backend_warn* parameter to `matplotlib.test` has no effect and has been removed.
- The *dryrun* parameter to the various `FigureCanvas*.print_*` methods has been removed.

### rcParams

- The `datapath` rcParam has been removed. Use *`matplotlib.get_data_path`* instead.
- The `mpl_toolkits.legacy_colorbar` rcParam has no effect and has been removed.
- Setting *`rcParams["boxplot.whiskers"]`* (default: `1.5`) to `"range"` is no longer valid; set it to `0, 100` instead.
- Setting *`rcParams["savefig.format"]`* (default: `'png'`) to `"auto"` is no longer valid; use `"png"` instead.

- Setting *rcParams["text.hinting"]* (default: `'force_autohint'`) to `False` or `True` is no longer valid; set it to `"auto"` or `"none"` respectively.

### sample_data removals

The sample datasets listed below have been removed. Suggested replacements for demonstration purposes are listed in parentheses.

- `None_vs_nearest-pdf.png`,
- `aapl.npz` (use `goog.npz`),
- `ada.png`, `grace_hopper.png` (use `grace_hopper.jpg`),
- `ct.raw.gz` (use `s1045.ima.gz`),
- `damodata.csv` (use `msft.csv`).

## Development changes

### Increase to minimum supported versions of Python and dependencies

For Matplotlib 3.4, the *minimum supported versions* are being bumped:

| Dependency | min in mpl3.3 | min in mpl3.4 |
|---|---|---|
| Python | 3.6 | 3.7 |
| dateutil | 2.1 | 2.7 |
| numpy | 1.15 | 1.16 |
| pyparsing | 2.0.3 | 2.2.1 |

This is consistent with our *Dependency version policy* and NEP29

### Qhull downloaded at build-or-sdist time

Much like FreeType, Qhull is now downloaded at build time, or upon creation of the sdist. To link against system Qhull, set the `system_qhull` option to `True` in the `setup.cfg` file. Note that Matplotlib now requires the re-entrant version of Qhull (`qhull_r`).

#### `FigureBase` class added, and `Figure` class made a child

The new subfigure feature motivated some re-organization of the *figure.Figure* class, so that the new *figure.SubFigure* class could have all the capabilities of a figure.

The *figure.Figure* class is now a subclass of *figure.FigureBase*, where *figure.FigureBase* contains figure-level artist addition routines, and the *figure.Figure* subclass just contains features that are unique to the outer figure.

Note that there is a new *transSubfigure* transform associated with the subfigure. This transform also exists for a *Figure* instance, and is equal to *transFigure* in that case, so code that uses the transform stack that wants to place objects on either the parent figure or one of the subfigures should use *transSubfigure*.

## 9.7 Version 3.3

### 9.7.1 What's new in Matplotlib 3.3.0 (Jul 16, 2020)

For a list of all of the issues and pull requests since the last revision, see the *GitHub statistics for 3.9.0 (May 15, 2024)*.

**Table of Contents**


- *What's new in Matplotlib 3.3.0 (Jul 16, 2020)*
  - *Figure and Axes creation / management*
    - *Provisional API for composing semantic axes layouts from text or nested lists*
    - *GridSpec.subplots()*
    - *New Axes.sharex, Axes.sharey methods*
    - *tight_layout now supports suptitle*
    - *Setting axes box aspect*
  - *Colors and colormaps*
    - *Turbo colormap*
    - *colors.BoundaryNorm supports* extend *keyword argument*
    - *Text color for legend labels*
    - *Pcolor and Pcolormesh now accept shading='nearest' and 'auto'*
  - *Titles, ticks, and labels*
    - *Align labels to Axes edges*
    - *Allow tick formatters to be set with str or function inputs*
    - *Axes.set_title gains a* y *keyword argument to control auto positioning*

        * *Offset text is now set to the top when using* `axis.tick_top()`
        * *Set zorder of contour labels*
    - *Other changes*
        * *New* `Axes.axline` *method*
        * `imshow` *now coerces 3D arrays with depth 1 to 2D*
        * *Better control of* `Axes.pie` *normalization*
        * *Dates use a modern epoch*
        * *Lines now accept* `MarkerStyle` *instances as input*
    - *Fonts*
        * *Simple syntax to select fonts by absolute path*
        * *Improved font weight detection*
    - *rcParams improvements*
        * `matplotlib.rc_context` *can be used as a decorator*
        * *rcParams for controlling default "raise window" behavior*
        * *Add generalized* `mathtext.fallback` *to rcParams*
        * *Add* `contour.linewidth` *to rcParams*
    - *3D Axes improvements*
        * `Axes3D` *no longer distorts the 3D plot to match the 2D aspect ratio*
        * *3D axes now support minor ticks*
        * *Home/Forward/Backward buttons now work with 3D axes*
    - *Interactive tool improvements*
        * *More consistent toolbar behavior across backends*
        * *Toolbar icons are now styled for dark themes*
        * *Cursor text now uses a number of significant digits matching pointing precision*
        * *GTK / Qt zoom rectangle now black and white*
        * *Event handler simplifications*
    - *Functions to compute a Path's size*
        * *Better interface for Path segment iteration*
        * *Fixed bug that computed a Path's Bbox incorrectly*
    - *Backend-specific improvements*
        * `savefig()` *gained a* backend *keyword argument*

* *The SVG backend can now render hatches with transparency*
* *SVG supports URLs on more artists*
* *Images in SVG will no longer be blurred in some viewers*
* *Saving SVG now supports adding metadata*
* *Saving PDF metadata via PGF now consistent with PDF backend*
* *NbAgg and WebAgg no longer use jQuery & jQuery UI*

### Figure and Axes creation / management

#### Provisional API for composing semantic axes layouts from text or nested lists

The *Figure* class has a provisional method to generate complex grids of named *axes.Axes* based on nested list input or ASCII art:

```
axd = plt.figure(constrained_layout=True).subplot_mosaic(
    [['.', 'histx'],
     ['histy', 'scat']]
)
for k, ax in axd.items():
    ax.text(0.5, 0.5, k,
            ha='center', va='center', fontsize=36,
            color='darkgrey')
```

or as a string (with single-character Axes labels):

```
axd = plt.figure(constrained_layout=True).subplot_mosaic(
    """
    TTE
    L.E
    """)
for k, ax in axd.items():
    ax.text(0.5, 0.5, k,
            ha='center', va='center', fontsize=36,
            color='darkgrey')
```

See *Complex and semantic figure composition (subplot_mosaic)* for more details and examples.

### `GridSpec.subplots()`

The *GridSpec* class gained a *subplots* method, so that one can write

```
fig.add_gridspec(2, 2, height_ratios=[3, 1]).subplots()
```

as an alternative to

```
fig.subplots(2, 2, gridspec_kw={"height_ratios": [3, 1]})
```

### New `Axes.sharex`, `Axes.sharey` methods

These new methods allow sharing axes *immediately* after creating them. Note that behavior is indeterminate if axes are not shared immediately after creation.

For example, they can be used to selectively link some axes created all together using *subplot_mosaic*:

```
fig = plt.figure(constrained_layout=True)
axd = fig.subplot_mosaic([['.', 'histx'], ['histy', 'scat']],
                         gridspec_kw={'width_ratios': [1, 7],
                                      'height_ratios': [2, 7]})

axd['histx'].sharex(axd['scat'])
axd['histy'].sharey(axd['scat'])
```

### tight_layout now supports suptitle

Previous versions did not consider *Figure.suptitle*, so it may overlap with other artists after calling *tight_layout*:

From now on, the `suptitle` will be considered:

### Setting axes box aspect

It is now possible to set the aspect of an axes box directly via *set_box_aspect*. The box aspect is the ratio between axes height and axes width in physical units, independent of the data limits. This is useful to, e.g., produce a square plot, independent of the data it contains, or to have a non-image plot with the same axes dimensions next to an image plot with fixed (data-)aspect.

For use cases check out the *Axes box aspect* example.

### Colors and colormaps

### Turbo colormap

Turbo is an improved rainbow colormap for visualization, created by the Google AI team for computer vision and machine learning. Its purpose is to display depth and disparity data. Please see the Google AI Blog for further details.

### `colors.BoundaryNorm` supports *extend* keyword argument

*BoundaryNorm* now has an *extend* keyword argument, analogous to *extend* in *contourf*. When set to 'both', 'min', or 'max', it maps the corresponding out-of-range values to *Colormap* lookup-table indices near the appropriate ends of their range so that the colors for out-of range values are adjacent to, but distinct from, their in-range neighbors. The colorbar inherits the *extend* argument from the norm, so with `extend='both'`, for example, the colorbar will have triangular extensions for out-of-range values with colors that differ from adjacent in-range colors.

### Text color for legend labels

The text color of legend labels can now be set by passing a parameter `labelcolor` to *legend*. The `labelcolor` keyword can be:

- A single color (either a string or RGBA tuple), which adjusts the text color of all the labels.
- A list or tuple, allowing the text color of each label to be set individually.
- `linecolor`, which sets the text color of each label to match the corresponding line color.
- `markerfacecolor`, which sets the text color of each label to match the corresponding marker face color.
- `markeredgecolor`, which sets the text color of each label to match the corresponding marker edge color.

### Pcolor and Pcolormesh now accept `shading='nearest'` and `'auto'`

Previously *axes.Axes.pcolor* and *axes.Axes.pcolormesh* handled the situation where $x$ and $y$ have the same (respective) size as $C$ by dropping the last row and column of $C$, and $x$ and $y$ are regarded as the edges of the remaining rows and columns in $C$. However, many users want $x$ and $y$ centered on the rows and columns of $C$.

To accommodate this, `shading='nearest'` and `shading='auto'` are new allowed strings for the *shading* keyword argument. `'nearest'` will center the color on $x$ and $y$ if $x$ and $y$ have the same dimensions

as *C* (otherwise an error will be thrown). `shading='auto'` will choose 'flat' or 'nearest' based on the size of *X*, *Y*, *C*.

If `shading='flat'` then *X*, and *Y* should have dimensions one larger than *C*. If *X* and *Y* have the same dimensions as *C*, then the previous behavior is used and the last row and column of *C* are dropped, and a DeprecationWarning is emitted.

Users can also specify this by the new *rcParams["pcolor.shading"]* (default: `'auto'`) in their `.matplotlibrc` or via *rcParams*.

See *pcolormesh* for examples.

### Titles, ticks, and labels

### Align labels to Axes edges

*set_xlabel*, *set_ylabel* and `ColorbarBase.set_label` support a parameter `loc` for simplified positioning. For the xlabel, the supported values are 'left', 'center', or 'right'. For the ylabel, the supported values are 'bottom', 'center', or 'top'.

The default is controlled via *rcParams["xaxis.labelposition"]* and *rcParams["yaxis.labelposition"]*; the Colorbar label takes the rcParam based on its orientation.

### Allow tick formatters to be set with str or function inputs

*set_major_formatter* and *set_minor_formatter* now accept `str` or function inputs in addition to *Formatter* instances. For a `str` a *StrMethodFormatter* is automatically generated and used. For a function a *FuncFormatter* is automatically generated and used. In other words,

```
ax.xaxis.set_major_formatter('{x} km')
ax.xaxis.set_minor_formatter(lambda x, pos: str(x-5))
```

are shortcuts for:

```
import matplotlib.ticker as mticker

ax.xaxis.set_major_formatter(mticker.StrMethodFormatter('{x} km'))
ax.xaxis.set_minor_formatter(
    mticker.FuncFormatter(lambda x, pos: str(x-5))
```

### `Axes.set_title` gains a *y* keyword argument to control auto positioning

*set_title* tries to auto-position the title to avoid any decorators on the top x-axis. This is not always desirable so now *y* is an explicit keyword argument of *set_title*. It defaults to *None* which means to use auto-positioning. If a value is supplied (i.e. the pre-3.0 default was `y=1.0`) then auto-positioning is turned off. This can also be set with the new rcParameter *rcParams["axes.titley"]* (default: `None`).

### Offset text is now set to the top when using `axis.tick_top()`

Solves the issue that the power indicator (e.g., 1e4) stayed on the bottom, even if the ticks were on the top.

### Set zorder of contour labels

*clabel* now accepts a *zorder* keyword argument making it easier to set the *zorder* of contour labels. If not specified, the default *zorder* of clabels used to always be 3 (i.e. the default *zorder* of *Text*) irrespective of the *zorder* passed to *contour*/*contourf*. The new default *zorder* for clabels has been changed to (`2 + zorder` passed to *contour* / *contourf*).

### Other changes

### New `Axes.axline` method

A new *axline* method has been added to draw infinitely long lines that pass through two points.

```
fig, ax = plt.subplots()

ax.axline((.1, .1), slope=5, color='C0', label='by slope')
ax.axline((.1, .2), (.8, .7), color='C3', label='by points')

ax.legend()
```

### `imshow` now coerces 3D arrays with depth 1 to 2D

Starting from this version arrays of size MxNx1 will be coerced into MxN for displaying. This means commands like `plt.imshow(np.random.rand(3, 3, 1))` will no longer return an error message that the image shape is invalid.

### Better control of `Axes.pie` normalization

Previously, *Axes.pie* would normalize its input *x* if `sum(x) > 1`, but would do nothing if the sum were less than 1. This can be confusing, so an explicit keyword argument *normalize* has been added. By default, the old behavior is preserved.

By passing *normalize*, one can explicitly control whether any rescaling takes place or whether partial pies should be created. If normalization is disabled, and `sum(x) > 1`, then an error is raised.

### Dates use a modern epoch

Matplotlib converts dates to days since an epoch using *dates.date2num* (via *matplotlib.units*). Previously, an epoch of `0000-12-31T00:00:00` was used so that `0001-01-01` was converted to 1.0. An epoch so distant in the past meant that a modern date was not able to preserve microseconds because 2000 years times the 2^(-52) resolution of a 64-bit float gives 14 microseconds.

Here we change the default epoch to the more reasonable UNIX default of `1970-01-01T00:00:00` which for a modern date has 0.35 microsecond resolution. (Finer resolution is not possible because we rely on `datetime.datetime` for the date locators). Access to the epoch is provided by *get_epoch*, and there is a new *rcParams["date.epoch"]* (default: `'1970-01-01T00:00:00'`) rcParam. The user may also call *set_epoch*, but it must be set *before* any date conversion or plotting is used.

If you have data stored as ordinal floats in the old epoch, you can convert them to the new ordinal using the following formula:

```
new_ordinal = old_ordinal + mdates.date2num(np.datetime64('0000-12-31'))
```

### Lines now accept `MarkerStyle` instances as input

Similar to *scatter*, *plot* and *Line2D* now accept *MarkerStyle* instances as input for the *marker* parameter:

```
plt.plot(..., marker=matplotlib.markers.MarkerStyle("D"))
```

### Fonts

### Simple syntax to select fonts by absolute path

Fonts can now be selected by passing an absolute `pathlib.Path` to the *font* keyword argument of *Text*.

### Improved font weight detection

Matplotlib is now better able to determine the weight of fonts from their metadata, allowing to differentiate between fonts within the same family more accurately.

### rcParams improvements

#### `matplotlib.rc_context` can be used as a decorator

*matplotlib.rc_context* can now be used as a decorator (technically, it is now implemented as a `contextlib.contextmanager`), e.g.,

```
@rc_context({"lines.linewidth": 2})
def some_function(...):
    ...
```

### rcParams for controlling default "raise window" behavior

The new config option *rcParams["figure.raise_window"]* (default: `True`) allows disabling of the raising of the plot window when calling *show* or *pause*. The `MacOSX` backend is currently not supported.

### Add generalized `mathtext.fallback` to rcParams

New *rcParams["mathtext.fallback"]* (default: `'cm'`) rcParam. Takes "cm", "stix", "stixsans" or "none" to turn fallback off. The rcParam *mathtext.fallback_to_cm* is deprecated, but if used, will override new fallback.

### Add `contour.linewidth` to rcParams

The new config option *rcParams["contour.linewidth"]* (default: `None`) allows to control the default line width of contours as a float. When set to `None`, the line widths fall back to *rcParams["lines.linewidth"]* (default: `1.5`). The config value is overridden as usual by the *linewidths* argument passed to *contour* when it is not set to `None`.

### 3D Axes improvements

#### `Axes3D` no longer distorts the 3D plot to match the 2D aspect ratio

Plots made with *Axes3D* were previously stretched to fit a square bounding box. As this stretching was done after the projection from 3D to 2D, it resulted in distorted images if non-square bounding boxes were used. As of 3.3, this no longer occurs.

Currently, modes of setting the aspect (via *set_aspect*) in data space are not supported for Axes3D but may be in the future. If you want to simulate having equal aspect in data space, set the ratio of your data limits to match the value of *get_box_aspect*. To control these ratios use the *set_box_aspect* method which accepts the ratios as a 3-tuple of X:Y:Z. The default aspect ratio is 4:4:3.

#### 3D axes now support minor ticks

```
ax = plt.figure().add_subplot(projection='3d')

ax.scatter([0, 1, 2], [1, 3, 5], [30, 50, 70])

ax.set_xticks([0.25, 0.75, 1.25, 1.75], minor=True)
ax.set_xticklabels(['a', 'b', 'c', 'd'], minor=True)

ax.set_yticks([1.5, 2.5, 3.5, 4.5], minor=True)
ax.set_yticklabels(['A', 'B', 'C', 'D'], minor=True)

ax.set_zticks([35, 45, 55, 65], minor=True)
ax.set_zticklabels([r'$\alpha$', r'$\beta$', r'$\delta$', r'$\gamma$'],
                   minor=True)

ax.tick_params(which='major', color='C0', labelcolor='C0', width=5)
ax.tick_params(which='minor', color='C1', labelcolor='C1', width=3)
```

#### Home/Forward/Backward buttons now work with 3D axes

#### Interactive tool improvements

#### More consistent toolbar behavior across backends

Toolbar features are now more consistent across backends. The history buttons will auto-disable when there is no further action in a direction. The pan and zoom buttons will be marked active when they are in use.

In NbAgg and WebAgg, the toolbar buttons are now grouped similarly to other backends. The WebAgg toolbar now uses the same icons as other backends.

### Toolbar icons are now styled for dark themes

On dark themes, toolbar icons will now be inverted. When using the GTK3Agg backend, toolbar icons are now symbolic, and both foreground and background colors will follow the theme. Tooltips should also behave correctly.

### Cursor text now uses a number of significant digits matching pointing precision

Previously, the x/y position displayed by the cursor text would usually include far more significant digits than the mouse pointing precision (typically one pixel). This is now fixed for linear scales.

### GTK / Qt zoom rectangle now black and white

This makes it visible even over a dark background.

### Event handler simplifications

The *backend_bases.key_press_handler* and *backend_bases.button_press_handler* event handlers can now be directly connected to a canvas with `canvas.mpl_connect("key_press_event", key_press_handler)` and `canvas.mpl_connect("button_press_event", button_press_handler)`, rather than having to write wrapper functions that fill in the (now optional) *canvas* and *toolbar* parameters.

### Functions to compute a Path's size

Various functions were added to *BezierSegment* and *Path* to allow computation of the shape/size of a *Path* and its composite Bezier curves.

In addition to the fixes below, *BezierSegment* has gained more documentation and usability improvements, including properties that contain its dimension, degree, control_points, and more.

### Better interface for Path segment iteration

*iter_bezier* iterates through the *BezierSegment*'s that make up the Path. This is much more useful typically than the existing *iter_segments* function, which returns the absolute minimum amount of information possible to reconstruct the Path.

### Fixed bug that computed a Path's Bbox incorrectly

Historically, *get_extents* has always simply returned the Bbox of a curve's control points, instead of the Bbox of the curve itself. While this is a correct upper bound for the path's extents, it can differ dramatically from the Path's actual extents for non-linear Bezier curves.

### Backend-specific improvements

#### `savefig()` gained a *backend* keyword argument

The *backend* keyword argument to `savefig` can now be used to pick the rendering backend without having to globally set the backend; e.g., one can save PDFs using the pgf backend with `savefig("file.pdf", backend="pgf")`.

### The SVG backend can now render hatches with transparency

The SVG backend now respects the hatch stroke alpha. Useful applications are, among others, semi-transparent hatches as a subtle way to differentiate columns in bar plots.

### SVG supports URLs on more artists

URLs on more artists (i.e., from *Artist.set_url*) will now be saved in SVG files, namely, `Ticks` and `Line2Ds` are now supported.

### Images in SVG will no longer be blurred in some viewers

A style is now supplied to images without interpolation (`imshow(..., interpolation='none')` so that SVG image viewers will no longer perform interpolation when rendering themselves.

### Saving SVG now supports adding metadata

When saving SVG files, metadata can now be passed which will be saved in the file using Dublin Core and RDF. A list of valid metadata can be found in the documentation for *FigureCanvasSVG.print_svg*.

#### Saving PDF metadata via PGF now consistent with PDF backend

When saving PDF files using the PGF backend, passed metadata will be interpreted in the same way as with the PDF backend. Previously, this metadata was only accepted by the PGF backend when saving a multi-page PDF with *backend_pgf.PdfPages*, but is now allowed when saving a single figure, as well.

#### NbAgg and WebAgg no longer use jQuery & jQuery UI

Instead, they are implemented using vanilla JavaScript. Please report any issues with browsers.

### 9.7.2 API Changes for 3.3.1

#### Deprecations

#### Reverted deprecation of `num2epoch` and `epoch2num`

These two functions were deprecated in 3.3.0, and did not return an accurate Matplotlib datenum relative to the new Matplotlib epoch handling (*get_epoch* and *rcParams["date.epoch"]* (default: `'1970-01-01T00:00:00'`)). This version reverts the deprecation.

#### Functions `epoch2num` and `dates.julian2num` use `date.epoch` rcParam

Now `epoch2num` and (undocumented) `julian2num` return floating point days since *get_epoch* as set by *rcParams["date.epoch"]* (default: `'1970-01-01T00:00:00'`), instead of floating point days since the old epoch of "0000-12-31T00:00:00". If needed, you can translate from the new to old values as `old = new + mdates.date2num(np.datetime64('0000-12-31'))`

### 9.7.3 API Changes for 3.3.0


- *Behaviour changes*
- *Deprecations*
- *Removals*
- *Development changes*

### Behaviour changes

#### `Formatter.fix_minus`

*Formatter.fix_minus* now performs hyphen-to-unicode-minus replacement whenever *rcParams["axes.unicode_minus"]* (default: `True`) is True; i.e. its behavior matches the one of `ScalarFormatter.fix_minus` (*ScalarFormatter* now just inherits that implementation).

This replacement is now used by the `format_data_short` method of the various builtin formatter classes, which affects the cursor value in the GUI toolbars.

#### `FigureCanvasBase` now always has a `manager` attribute, which may be None

Previously, it did not necessarily have such an attribute. A check for `hasattr(figure.canvas, "manager")` should now be replaced by `figure.canvas.manager is not None` (or `getattr(figure.canvas, "manager", None) is not None` for back-compatibility).

#### `cbook.CallbackRegistry` now propagates exceptions when no GUI event loop is running

*cbook.CallbackRegistry* now defaults to propagating exceptions thrown by callbacks when no interactive GUI event loop is running. If a GUI event loop *is* running, *cbook.CallbackRegistry* still defaults to just printing a traceback, as unhandled exceptions can make the program completely `abort()` in that case.

#### `Axes.locator_params()` validates `axis` parameter

*axes.Axes.locator_params* used to accept any value for `axis` and silently did nothing, when passed an unsupported value. It now raises a `ValueError`.

#### `Axis.set_tick_params()` validates `which` parameter

*Axis.set_tick_params* (and the higher level *axes.Axes.tick_params* and *pyplot.tick_params*) used to accept any value for `which` and silently did nothing, when passed an unsupported value. It now raises a `ValueError`.

#### `Axis.set_ticklabels()` must match `FixedLocator.locs`

If an axis is using a *ticker.FixedLocator*, typically set by a call to *Axis.set_ticks*, then the number of ticklabels supplied must match the number of locations available (`FixedFormattor.locs`). If not, a `ValueError` is raised.

#### `backend_pgf.LatexManager.latex`

`backend_pgf.LatexManager.latex` is now created with `encoding="utf-8"`, so its `stdin`, `stdout`, and `stderr` attributes are utf8-encoded.

#### `pyplot.xticks()` and `pyplot.yticks()`

Previously, passing labels without passing the ticks to either *pyplot.xticks* and *pyplot.yticks* would result in:

```
TypeError: object of type 'NoneType' has no len()
```

It now raises a `TypeError` with a proper description of the error.

#### Setting the same property under multiple aliases now raises a TypeError

Previously, calling e.g. `plot(..., color=somecolor, c=othercolor)` would emit a warning because `color` and `c` actually map to the same Artist property. This now raises a TypeError.

#### `FileMovieWriter` temporary frames directory

*FileMovieWriter* now defaults to writing temporary frames in a temporary directory, which is always cleared at exit. In order to keep the individual frames saved on the filesystem, pass an explicit *frame_prefix*.

#### `Axes.plot` no longer accepts *x* and *y* being both 2D and with different numbers of columns

Previously, calling *Axes.plot* e.g. with *x* of shape `(n, 3)` and *y* of shape `(n, 2)` would plot the first column of *x* against the first column of *y*, the second column of *x* against the second column of *y*, **and** the first column of *x* against the third column of *y*. This now raises an error instead.

### `Text.update_from` now copies usetex state from the source Text

### `stem` now defaults to `use_line_collection=True`

This creates the stem plot as a *`LineCollection`* rather than individual *`Line2D`* objects, greatly improving performance.

### rcParams color validator is now stricter

Previously, rcParams entries whose values were color-like accepted "spurious" extra letters or characters in the "middle" of the string, e.g. `"(0, 1a, '0.5')"` would be interpreted as `(0, 1, 0.5)`. These extra characters (including the internal quotes) now cause a ValueError to be raised.

### `SymLogNorm` now has a *base* parameter

Previously, *`SymLogNorm`* had no *base* keyword argument, and defaulted to `base=np.e` whereas the documentation said it was `base=10`. In preparation to make the default 10, calling *`SymLogNorm`* without the new *base* keyword argument emits a deprecation warning.

### `errorbar` now color cycles when only errorbar color is set

Previously setting the *ecolor* would turn off automatic color cycling for the plot, leading to the the lines and markers defaulting to whatever the first color in the color cycle was in the case of multiple plot calls.

### `rcsetup.validate_color_for_prop_cycle` now always raises TypeError for bytes input

It previously raised `TypeError`, **except** when the input was of the form `b"C[number]"` in which case it raised a ValueError.

### `FigureCanvasPS.print_ps` and `FigureCanvasPS.print_eps` no longer apply edgecolor and facecolor

These methods now assume that the figure edge and facecolor have been correctly applied by *`FigureCanvasBase.print_figure`*, as they are normally called through it.

This behavior is consistent with other figure saving methods (*`FigureCanvasAgg.print_png`*, *`FigureCanvasPdf.print_pdf`*, *`FigureCanvasSVG.print_svg`*).

### `pyplot.subplot()` now raises TypeError when given an incorrect number of arguments

This is consistent with other signature mismatch errors. Previously a ValueError was raised.

### Shortcut for closing all figures

Shortcuts for closing all figures now also work for the classic toolbar. There is no default shortcut any more because unintentionally closing all figures by a key press might happen too easily. You can configure the shortcut yourself using *rcParams["keymap.quit_all"]* (default: `[]`).

### Autoscale for arrow

Calling ax.arrow() will now autoscale the axes.

### `set_tick_params(label1On=False)` now also makes the offset text (if any) invisible

... because the offset text can rarely be interpreted without tick labels anyways.

### `Axes.annotate` and `pyplot.annotate` parameter name changed

The parameter `s` to *Axes.annotate* and *pyplot.annotate* is renamed to `text`, matching *Annotation*.

The old parameter name remains supported, but support for it will be dropped in a future Matplotlib release.

### `font_manager.json_dump` now locks the font manager dump file

... to prevent multiple processes from writing to it at the same time.

### `pyplot.rgrids` and `pyplot.thetagrids` now act as setters also when called with only kwargs

Previously, keyword arguments were silently ignored when no positional arguments were given.

### `Axis.get_minorticklabels` and `Axis.get_majorticklabels` now returns plain list

Previously, *Axis.get_minorticklabels* and *Axis.get_majorticklabels* returns silent_list. Their return type is now changed to normal list. *get_xminorticklabels*, *get_yminorticklabels*, *get_zminorticklabels*, *Axis.get_ticklabels*, *get_xmajorticklabels*, *get_ymajorticklabels* and *get_zmajorticklabels* methods will be affected by this change.

### Default slider formatter

The default method used to format *Slider* values has been changed to use a *ScalarFormatter* adapted the slider values limits. This should ensure that values are displayed with an appropriate number of significant digits even if they are much smaller or much bigger than 1. To restore the old behavior, explicitly pass a "%1.2f" as the *valfmt* parameter to *Slider*.

### Add *normalize* keyword argument to `Axes.pie`

`pie()` used to draw a partial pie if the sum of the values was $< 1$. This behavior is deprecated and will change to always normalizing the values to a full pie by default. If you want to draw a partial pie, please pass `normalize=False` explicitly.

### `table.CustomCell` is now an alias for `table.Cell`

All the functionality of `CustomCell` has been moved to its base class *Cell*.

### wx Timer interval

Setting the timer interval on a not-yet-started `TimerWx` won't start it anymore.

### "step"-type histograms default to the zorder of `Line2D`

This ensures that they go above gridlines by default. The old `zorder` can be kept by passing it as a keyword argument to *Axes.hist*.

### `Legend` and `OffsetBox` visibility

*Legend* and *OffsetBox* subclasses (*PaddedBox*, *AnchoredOffsetbox*, and *AnnotationBbox*) no longer directly keep track of the visibility of their underlying *Patch* artist, but instead pass that flag down to the *Patch*.

### `Legend` and `Table` no longer allow invalid locations

This affects legends produced on an Axes (*Axes.legend* and *pyplot.legend*) and on a Figure (*Figure.legend* and *pyplot.figlegend*). Figure legends also no longer accept the unsupported `'best'` location. Previously, invalid Axes locations would use `'best'` and invalid Figure locations would used `'upper right'`.

### Passing Line2D's *drawstyle* together with *linestyle* is removed

Instead of `plt.plot(..., linestyle="steps--")`, use `plt.plot(..., linestyle="--", drawstyle="steps")`. `ds` is also an alias for `drawstyle`.

### Upper case color strings

Support for passing single-letter colors (one of "rgbcmykw") as UPPERCASE characters is removed; these colors are now case-sensitive (lowercase).

### tight/constrained_layout no longer worry about titles that are too wide

*tight_layout* and *constrained_layout* shrink axes to accommodate "decorations" on the axes. However, if an xlabel or title is too long in the x direction, making the axes smaller in the x-direction doesn't help. The behavior of both has been changed to ignore the width of the title and xlabel and the height of the ylabel in the layout logic.

This also means there is a new keyword argument for *axes.Axes.get_tightbbox* and *axis.Axis.get_tightbbox*: `for_layout_only`, which defaults to *False*, but if *True* returns a bounding box using the rules above.

### `rcParams["savefig.facecolor"]` (default: `'auto'`) and `rcParams["savefig.edgecolor"]` (default: `'auto'`) now default to "auto"

This newly allowed value for *rcParams["savefig.facecolor"]* (default: `'auto'`) and *rcParams["savefig.edgecolor"]* (default: `'auto'`), as well as the *facecolor* and *edgecolor* parameters to *Figure.savefig*, means "use whatever facecolor and edgecolor the figure current has".

### When using a single dataset, `Axes.hist` no longer wraps the added artist in a `silent_list`

When *Axes.hist* is called with a single dataset, it adds to the axes either a *BarContainer* object (when `histtype="bar"` or `"barstacked"`), or a *Polygon* object (when `histype="step"` or `"stepfilled"`) -- the latter being wrapped in a list-of-one-element. Previously, either artist would be wrapped in a *silent_list*. This is no longer the case: the *BarContainer* is now returned as is (this is an API breaking change if you were directly relying on the concrete `list` API; however, *BarContainer* inherits from `tuple` so most common operations remain available), and the list-of-one *Polygon* is returned as is. This makes the `repr` of the returned artist more accurate: it is now

```
<BarContainer object of 10 artists>  # "bar", "barstacked"
[<matplotlib.patches.Polygon object at 0xdeadbeef>]  # "step", "stepfilled"
```

instead of

```
<a list of 10 Patch objects>  # "bar", "barstacked"
<a list of 1 Patch objects>  # "step", "stepfilled"
```

When *Axes.hist* is called with multiple artists, it still wraps its return value in a *silent_list*, but uses more accurate type information

```
<a list of 3 BarContainer objects>  # "bar", "barstacked"
<a list of 3 List[Polygon] objects>  # "step", "stepfilled"
```

instead of

```
<a list of 3 Lists of Patches objects>  # "bar", "barstacked"
<a list of 3 Lists of Patches objects>  # "step", "stepfilled"
```

### Qt and wx backends no longer create a status bar by default

The coordinates information is now displayed in the toolbar, consistently with the other backends. This is intended to simplify embedding of Matplotlib in larger GUIs, where Matplotlib may control the toolbar but not the status bar.

### `rcParams["text.hinting"]` (default: `'force_autohint'`) now supports names mapping to FreeType flags

*rcParams["text.hinting"]* (default: `'force_autohint'`) now supports the values "default", "no_autohint", "force_autohint", and "no_hinting", which directly map to the FreeType flags FT_LOAD_DEFAULT, etc. The old synonyms (respectively "either", "native", "auto", and "none") are still supported, but their use is discouraged. To get normalized values, use *backend_agg.get_hinting_flag*, which returns integer flag values.

### `cbook.get_sample_data` auto-loads numpy arrays

When *`cbook.get_sample_data`* is used to load a npy or npz file and the keyword-only parameter `np_load` is True, the file is automatically loaded using `numpy.load`. `np_load` defaults to False for backwards compatibility, but will become True in a later release.

### `get_text_width_height_descent` now checks `ismath` rather than `rcParams["text.usetex"]` (default: `False`)

... to determine whether a string should be passed to the usetex machinery or not. This allows single strings to be marked as not-usetex even when the rcParam is True.

### `Axes.vlines`, `Axes.hlines`, `pyplot.vlines` and `pyplot.hlines` *colors* parameter default change

The *colors* parameter will now default to *`rcParams["lines.color"]`* (default: `'C0'`), while previously it defaulted to 'k'.

### Aggressively autoscale clim in `ScalerMappable` classes

Previously some plotting methods would defer autoscaling until the first draw if only one of the *vmin* or *vmax* keyword arguments were passed (*`Axes.scatter`*, *`Axes.hexbin`*, *`Axes.imshow`*, *`Axes.pcolorfast`*) but would scale based on the passed data if neither was passed (independent of the *norm* keyword arguments). Other methods (*`Axes.pcolor`*, *`Axes.pcolormesh`*) always autoscaled base on the initial data.

All of the plotting methods now resolve the unset *vmin* or *vmax* at the initial call time using the data passed in.

If you were relying on exactly one of the *vmin* or *vmax* remaining unset between the time when the method is called and the first time the figure is rendered you get back the old behavior by manually setting the relevant limit back to `None`

```
cm_obj.norm.vmin = None
# or
cm_obj.norm.vmax = None
```

which will be resolved during the draw process.

### Deprecations

#### `figure.add_axes()` without arguments

Calling `fig.add_axes()` with no arguments currently does nothing. This call will raise an error in the future. Adding a free-floating axes needs a position rectangle. If you want a figure-filling single axes, use `add_subplot()` instead.

#### `backend_wx.DEBUG_MSG`

`backend_wx.DEBUG_MSG` is deprecated. The wx backends now use regular logging.

#### `Colorbar.config_axis()`

`Colorbar.config_axis()` is considered internal. Its use is deprecated.

#### `NonUniformImage.is_grayscale` and `PcolorImage.is_grayscale`

These attributes are deprecated, for consistency with `AxesImage.is_grayscale`, which was removed back in Matplotlib 2.0.0. (Note that previously, these attributes were only available *after rendering the image*).

#### `den` parameter and attribute to `mpl_toolkits.axisartist.angle_helper`

For all locator classes defined in *`mpl_toolkits.axisartist.angle_helper`*, the `den` parameter has been renamed to `nbins`, and the `den` attribute deprecated in favor of its (preexisting) synonym `nbins`, for consistency with locator classes defined in *`matplotlib.ticker`*.

#### `backend_pgf.LatexManager.latex_stdin_utf8`

`backend_pgf.LatexManager.latex` is now created with `encoding="utf-8"`, so its `stdin` attribute is already utf8-encoded; the `latex_stdin_utf8` attribute is thus deprecated.

#### Flags containing "U" passed to `cbook.to_filehandle` and `cbook.open_file_cm`

Please remove "U" from flags passed to *`cbook.to_filehandle`* and *`cbook.open_file_cm`*. This is consistent with their removal from `open` in Python 3.9.

### PDF and PS character tracking internals

The `used_characters` attribute and `track_characters` and `merge_used_characters` methods of *`RendererPdf`*, *`PdfFile`*, and *`RendererPS`* are deprecated.

### Case-insensitive capstyles and joinstyles

Please pass capstyles ("miter", "round", "bevel") and joinstyles ("butt", "round", "projecting") as lowercase.

### Passing raw data to `register_cmap()`

Passing raw data via parameters *data* and *lut* to `matplotlib.cm.register_cmap()` is deprecated. Instead, explicitly create a *`LinearSegmentedColormap`* and pass it via the *cmap* parameter: `register_cmap(cmap=LinearSegmentedColormap(name, data, lut))`.

#### `DateFormatter.illegal_s`

This attribute is unused and deprecated.

#### `widgets.TextBox.params_to_disable`

This attribute is deprecated.

### Revert deprecation *min, *max keyword arguments to `set_x/y/zlim_3d()`

These keyword arguments were deprecated in 3.0, alongside with the respective parameters in `set_xlim()` / `set_ylim()`. The deprecations of the 2D versions were already reverted in 3.1.

#### `cbook.local_over_kwdict`

This function is deprecated. Use *`cbook.normalize_kwargs`* instead.

### Passing both singular and plural *colors*, *linewidths*, *linestyles* to `Axes.eventplot`

Passing e.g. both *linewidth* and *linewidths* will raise a TypeError in the future.

### Setting `rcParams["text.latex.preamble"]` (default: `''`) or `rcParams["pdf.preamble"]` to non-strings

These rcParams should be set to string values. Support for None (meaning the empty string) and lists of strings (implicitly joined with newlines) is deprecated.

### Parameters *norm* and *vmin*/*vmax* should not be used simultaneously

Passing parameters *norm* and *vmin*/*vmax* simultaneously to functions using colormapping such as `scatter()` and `imshow()` is deprecated. Instead of `norm=LogNorm(), vmin=min_val, vmax=max_val` pass `norm=LogNorm(min_val, max_val)`. *vmin* and *vmax* should only be used without setting *norm*.

### Effectless parameters of `Figure.colorbar` and `matplotlib.colorbar.Colorbar`

The *cmap* and *norm* parameters of *Figure.colorbar* and *matplotlib.colorbar.Colorbar* have no effect because they are always overridden by the mappable's colormap and norm; they are thus deprecated. Likewise, passing the *alpha*, *boundaries*, *values*, *extend*, or *filled* parameters with a *ContourSet* mappable, or the *alpha* parameter with an *Artist* mappable, is deprecated, as the mappable would likewise override them.

### `args_key` and `exec_key` attributes of builtin `MovieWriter`s

These attributes are deprecated.

### Unused parameters

The following parameters do not have any effect and are deprecated:

- arbitrary keyword arguments to `StreamplotSet`
- parameter *quantize* of *Path.cleaned()*
- parameter *s* of *AnnotationBbox.get_fontsize()*
- parameter *label* of *Tick*

### Passing *props* to `Shadow`

The parameter *props* of *Shadow* is deprecated. Use keyword arguments instead.

#### `Axes.update_datalim_bounds`

This method is deprecated. Use `ax.dataLim.set(Bbox.union([ax.dataLim, bounds]))` instead.

#### `{,Symmetrical}LogScale.{,Inverted}LogTransform`

`LogScale.LogTransform`, `LogScale.InvertedLogTransform`, `SymmetricalScale.SymmetricalTransform` and `SymmetricalScale.InvertedSymmetricalTransform` are deprecated. Directly access the transform classes from the *scale* module.

#### `TexManager.cachedir`, `TexManager.rgba_arrayd`

Use *matplotlib.get_cachedir()* instead for the former; there is no replacement for the latter.

### Setting `Line2D`'s pickradius via `Line2D.set_picker`

Setting a *Line2D*'s pickradius (i.e. the tolerance for pick events and containment checks) via *Line2D.set_picker* is deprecated. Use *Line2D.set_pickradius* instead.

*Line2D.set_picker* no longer sets the artist's custom-contain() check.

#### `Artist.set_contains`, `Artist.get_contains`

Setting a custom method overriding *Artist.contains* is deprecated. There is no replacement, but you may still customize pick events using *Artist.set_picker*.

### `Colorbar` methods

The `on_mappable_changed` and `update_bruteforce` methods of *Colorbar* are deprecated; both can be replaced by calls to *update_normal*.

#### `OldScalarFormatter`, `IndexFormatter` and `IndexDateFormatter`

These formatters are deprecated. Their functionality can be implemented using e.g. *`FuncFormatter`*.

#### `OldAutoLocator`

This ticker is deprecated.

### *required*, *forbidden* and *allowed* parameters of `cbook.normalize_kwargs`

These parameters are deprecated.

### The `TTFPATH` and `AFMPATH` environment variables

Support for the (undocumented) `TTFPATH` and `AFMPATH` environment variables is deprecated. Additional fonts may be registered using `matplotlib.font_manager.fontManager.addfont()`.

#### `matplotlib.compat`

This module is deprecated.

#### `matplotlib.backends.qt_editor.formsubplottool`

This module is deprecated. Use `matplotlib.backends.backend_qt5.SubplotToolQt` instead.

### AVConv animation writer deprecated

The `AVConvBase`, `AVConvWriter` and `AVConvFileWriter` classes, and the associated `animation.avconv_path` and `animation.avconv_args` rcParams are deprecated.

Debian 8 (2015, EOL 06/2020) and Ubuntu 14.04 (EOL 04/2019) were the last versions of Debian and Ubuntu to ship avconv. It remains possible to force the use of avconv by using the ffmpeg-based writers with *`rcParams["animation.ffmpeg_path"]`* (default: `'ffmpeg'`) set to "avconv".

### log/symlog scale base, ticks, and nonpos specification

*semilogx*, *semilogy*, *loglog*, *LogScale*, and *SymmetricalLogScale* used to take keyword arguments that depends on the axis orientation ("basex" vs "basey", "subsx" vs "subsy", "nonposx" vs "nonposy"); these parameter names are now deprecated in favor of "base", "subs", "nonpositive". This deprecation also affects e.g. `ax.set_yscale("log", basey=...)` which must now be spelled `ax.set_yscale("log", base=...)`.

The change from "nonpos" to "nonpositive" also affects *LogTransform*, *InvertedLogTransform*, *SymmetricalLogTransform*, etc.

To use *different* bases for the x-axis and y-axis of a *loglog* plot, use e.g. `ax.set_xscale("log", base=10); ax.set_yscale("log", base=2)`.

### `DraggableBase.artist_picker`

This method is deprecated. If you previously reimplemented it in a subclass, set the artist's picker instead with *Artist.set_picker*.

### *clear_temp* parameter and attribute of `FileMovieWriter`

The *clear_temp* parameter and attribute of *FileMovieWriter* is deprecated. In the future, files placed in a temporary directory (using `frame_prefix=None`, the default) will be cleared; files placed elsewhere will not.

### Deprecated rcParams validators

The following validators, defined in *rcsetup*, are deprecated: `validate_fontset`, `validate_mathtext_default`, `validate_alignment`, `validate_svg_fonttype`, `validate_pgf_texsystem`, `validate_movie_frame_fmt`, `validate_axis_locator`, `validate_movie_html_fmt`, `validate_grid_axis`, `validate_axes_titlelocation`, `validate_toolbar`, `validate_ps_papersize`, `validate_legend_loc`, `validate_bool_maybe_none`, `validate_hinting`, `validate_movie_writer`, `validate_webagg_address`, `validate_nseq_float`, `validate_nseq_int`. To test whether an rcParam value would be acceptable, one can test e.g. `rc = RcParams(); rc[k] = v` raises an exception.

### Stricter rcParam validation

*rcParams["axes.axisbelow"]* (default: `'line'`) currently normalizes all strings starting with "line" (case-insensitive) to the option "line". This is deprecated; in a future version only the exact string "line" (case-sensitive) will be supported.

### `add_subplot()` validates its inputs

In particular, for `add_subplot(rows, cols, index)`, all parameters must be integral. Previously strings and floats were accepted and converted to int. This will now emit a deprecation warning.

### Toggling axes navigation from the keyboard using "a" and digit keys

Axes navigation can still be toggled programmatically using *Axes.set_navigate*.

The following related APIs are also deprecated: `backend_tools.ToolEnableAllNavigation`, `backend_tools.ToolEnableNavigation`, and `rcParams["keymap.all_axes"]`.

### `matplotlib.test(recursionlimit=...)`

The *recursionlimit* parameter of `matplotlib.test` is deprecated.

### mathtext glues

The *copy* parameter of `mathtext.Glue` is deprecated (the underlying glue spec is now immutable). `mathtext.GlueSpec` is deprecated.

### Signatures of `Artist.draw` and `matplotlib.axes.Axes.draw`

The *inframe* parameter to *matplotlib.axes.Axes.draw* is deprecated. Use *Axes.redraw_in_frame* instead.

Not passing the *renderer* parameter to *matplotlib.axes.Axes.draw* is deprecated. Use `axes.draw_artist(axes)` instead.

These changes make the signature of the `draw` (`artist.draw(renderer)`) method consistent across all artists; thus, additional parameters to *Artist.draw* are deprecated.

#### `DraggableBase.on_motion_blit`

This method is deprecated. *`DraggableBase.on_motion`* now handles both the blitting and the non-blitting cases.

### Passing the dash offset as None

Fine control of dash patterns can be achieved by passing an `(offset, (on-length, off-length, on-length, off-length, ...))` pair as the linestyle property of *`Line2D`* and *`LineCollection`*. Previously, certain APIs would accept `offset = None` as a synonym for `offset = 0`, but this was never universally implemented, e.g. for vector output. Support for `offset = None` is deprecated, set the offset to 0 instead.

#### `RendererCairo.fontweights`, `RendererCairo.fontangles`

... are deprecated.

#### `autofmt_xdate(which=None)`

This is deprecated, use its more explicit synonym, `which="major"`, instead.

### JPEG options

The *quality*, *optimize*, and *progressive* keyword arguments to *`savefig`*, which were only used when saving to JPEG, are deprecated. *`rcParams["savefig.jpeg_quality"]`* is likewise deprecated.

Such options should now be directly passed to Pillow using `savefig(..., pil_kwargs={"quality": ..., "optimize": ..., "progressive": ...})`.

#### `dviread.Encoding`

This class was (mostly) broken and is deprecated.

### Axis and Locator `pan` and `zoom`

The unused `pan` and `zoom` methods of *`Axis`* and *`Locator`* are deprecated. Panning and zooming are now implemented using the `start_pan`, `drag_pan`, and `end_pan` methods of *`Axes`*.

### Passing None to various Axes subclass factories

Support for passing `None` as base class to `axes.subplot_class_factory`, `axes_grid1.parasite_axes.host_axes_class_factory`, `axes_grid1.parasite_axes.host_subplot_class_factory`, `axes_grid1.parasite_axes.parasite_axes_class_factory`, and `axes_grid1.parasite_axes.parasite_axes_auxtrans_class_factory` is deprecated. Explicitly pass the correct base `Axes` class instead.

#### `axes_rgb`

In *`mpl_toolkits.axes_grid1.axes_rgb`*, `imshow_rgb` is deprecated (use `ax.imshow(np.dstack([r, g, b]))` instead); `RGBAxesBase` is deprecated (use `RGBAxes` instead); `RGBAxes.add_RGB_to_figure` is deprecated (it was an internal helper).

#### `Substitution.from_params`

This method is deprecated. If needed, directly assign to the `params` attribute of the Substitution object.

### PGF backend cleanups

The *dummy* parameter of *`RendererPgf`* is deprecated.

`GraphicsContextPgf` is deprecated (use *`GraphicsContextBase`* instead).

### `set_factor` method of `mpl_toolkits.axisartist` locators

The `set_factor` method of *`mpl_toolkits.axisartist`* locators (which are different from "standard" Matplotlib tick locators) is deprecated.

### `widgets.SubplotTool` callbacks and axes

The `funcleft`, `funcright`, `funcbottom`, `functop`, `funcwspace`, and `funchspace` methods of *`widgets.SubplotTool`* are deprecated.

The `axleft`, `axright`, `axbottom`, `axtop`, `axwspace`, and `axhspace` attributes of *`widgets.SubplotTool`* are deprecated. Access the `ax` attribute of the corresponding slider, if needed.

### mathtext `Glue` helper classes

The `Fil`, `Fill`, `Filll`, `NegFil`, `NegFill`, `NegFilll`, and `SsGlue` classes in the *`matplotlib.mathtext`* module are deprecated. As an alternative, directly construct glue instances with `Glue("fil")`, etc.

### NavigationToolbar2._init_toolbar

Overriding this method to initialize third-party toolbars is deprecated. Instead, the toolbar should be initialized in the `__init__` method of the subclass (which should call the base-class' `__init__` as appropriate). To keep back-compatibility with earlier versions of Matplotlib (which *required* `_init_toolbar` to be overridden), a fully empty implementation (`def _init_toolbar(self): pass`) may be kept and will not trigger the deprecation warning.

### NavigationToolbar2QT.parent and .basedir

These attributes are deprecated. In order to access the parent window, use `toolbar.canvas.parent()`. Once the deprecation period is elapsed, it will also be accessible as `toolbar.parent()`. The base directory to the icons is `os.path.join(mpl.get_data_path(), "images")`.

### NavigationToolbar2QT.ctx

This attribute is deprecated.

### NavigationToolbar2Wx attributes

The `prevZoomRect`, `retinaFix`, `savedRetinaImage`, `wxoverlay`, `zoomAxes`, `zoomStartX`, and `zoomStartY` attributes are deprecated.

### NavigationToolbar2.press and .release

These methods were called when pressing or releasing a mouse button, but *only* when an interactive pan or zoom was occurring (contrary to what the docs stated). They are deprecated; if you write a backend which needs to customize such events, please directly override `press_pan`/`press_zoom`/`release_pan`/`release_zoom` instead.

### FigureCanvasGTK3._renderer_init

Overriding this method to initialize renderers for GTK3 canvases is deprecated. Instead, the renderer should be initialized in the `__init__` method of the subclass (which should call the base-class' `__init__` as appropriate). To keep back-compatibility with earlier versions of Matplotlib (which *required* `_renderer_init` to be overridden), a fully empty implementation (`def _renderer_init(self): pass`) may be kept and will not trigger the deprecation warning.

### Path helpers in `bezier`

`bezier.make_path_regular` is deprecated. Use `Path.cleaned()` (or `Path.cleaned(curves=True)`, etc.) instead (but note that these methods add a `STOP` code at the end of the path).

`bezier.concatenate_paths` is deprecated. Use `Path.make_compound_path()` instead.

### `animation.html_args` rcParam

The unused `animation.html_args` rcParam and `animation.HTMLWriter.args_key` attribute are deprecated.

### `text.latex.preview` rcParam

This rcParam, which controlled the use of the preview.sty LaTeX package to align TeX string baselines, is deprecated, as Matplotlib's own dvi parser now computes baselines just as well as preview.sty.

### `SubplotSpec.get_rows_columns`

This method is deprecated. Use the `GridSpec.nrows`, `GridSpec.ncols`, `SubplotSpec.rowspan`, and `SubplotSpec.colspan` properties instead.

### Qt4-based backends

The qt4agg and qt4cairo backends are deprecated. Qt4 has reached its end-of-life in 2015 and there are no releases for recent versions of Python. Please consider switching to Qt5.

### *fontdict* and *minor* parameters of `Axes.set_xticklabels` and `Axes.set_yticklabels` will become keyword-only

### All parameters of `Figure.subplots` except *nrows* and *ncols* will become keyword-only

This avoids typing e.g. `subplots(1, 1, 1)` when meaning `subplot(1, 1, 1)`, but actually getting `subplots(1, 1, sharex=1)`.

#### `RendererWx.get_gc`

This method is deprecated. Access the `gc` attribute directly instead.

### *add_all* parameter in `axes_grid`

The *add_all* parameter of *axes_grid1.axes_grid.Grid*, *axes_grid1.axes_grid.ImageGrid*, *axes_grid1.axes_rgb.make_rgb_axes* and *axes_grid1.axes_rgb.RGBAxes* is deprecated. Axes are now always added to the parent figure, though they can be later removed with `ax.remove()`.

#### `BboxBase.inverse_transformed`

`.BboxBase.inverse_transformed` is deprecated (call *BboxBase.transformed* on the *inverted()* transform instead).

### *orientation* of `eventplot()` and `EventCollection`

Setting the *orientation* of an `eventplot()` or *EventCollection* to "none" or None is deprecated; set it to "horizontal" instead. Moreover, the two orientations ("horizontal" and "vertical") will become case-sensitive in the future.

### *minor* kwarg to `Axis.get_ticklocs` will become keyword-only

Passing this argument positionally is deprecated.

### Case-insensitive properties

Normalization of upper or mixed-case property names to lowercase in *Artist.set* and *Artist.update* is deprecated. In the future, property names will be passed as is, allowing one to pass names such as *patchA* or *UVC*.

#### `ContourSet.ax`, `Quiver.ax`

These attributes are deprecated in favor of `ContourSet.axes` and `Quiver.axes`, for consistency with other artists.

#### `Locator.refresh()` and associated methods

`Locator.refresh()` is deprecated. This method was called at certain places to let locators update their internal state, typically based on the axis limits. Locators should now always consult the axis limits when called, if needed.

The associated helper methods `NavigationToolbar2.draw()` and `ToolViewsPositions.refresh_locators()` are deprecated, and should be replaced by calls to `draw_idle()` on the corresponding canvas.

#### `ScalarMappable` checkers

The `add_checker` and `check_update` methods and `update_dict` attribute of *ScalarMappable* are deprecated.

#### `pyplot.tight_layout` and `ColorbarBase` parameters will become keyword-only

All parameters of *pyplot.tight_layout* and all parameters of `ColorbarBase` except for the first (*ax*) will become keyword-only, consistently with *Figure.tight_layout* and `Colorbar`, respectively.

### `Axes.pie` radius and startangle

Passing `None` as either the `radius` or `startangle` of an *`Axes.pie`* is deprecated; use the explicit defaults of 1 and 0, respectively, instead.

### `AxisArtist.dpi_transform`

... is deprecated. Scale `Figure.dpi_scale_trans` by 1/72 to achieve the same effect.

### `offset_position` property of `Collection`

The `offset_position` property of *`Collection`* is deprecated. In the future, *`Collection`*s will always behave as if `offset_position` is set to "screen" (the default).

Support for passing `offset_position="data"` to the `draw_path_collection` of all renderer classes is deprecated.

*`transforms.AffineDeltaTransform`* can be used as a replacement. This API is experimental and may change in the future.

### `testing.compare.make_external_conversion_command`

... is deprecated.

### `epoch2num` and `num2epoch` are deprecated

These are unused and can be easily reproduced by other date tools. *`get_epoch`* will return Matplotlib's epoch.

### `axes_grid1.CbarAxes` attributes

The `cbid` and `locator` attribute are deprecated. Use `mappable.colorbar_cid` and `colorbar.locator`, as for standard colorbars.

### `qt_compat.is_pyqt5`

This function is deprecated in prevision of the future release of PyQt6. The Qt version can be checked using `QtCore.qVersion()`.

### Reordering of parameters by `Artist.set`

In a future version, `Artist.set` will apply artist properties in the order in which they are given. This only affects the interaction between the *color*, *edgecolor*, *facecolor*, and, for *Collection*s, *alpha* properties: the *color* property now needs to be passed first in order not to override the other properties. This is consistent with e.g. *Artist.update*, which did not reorder the properties passed to it.

### Passing multiple keys as a single comma-separated string or multiple arguments to `ToolManager.update_keymap`

This is deprecated; pass keys as a list of strings instead.

### Statusbar classes and attributes

The `statusbar` attribute of *FigureManagerBase*, `StatusbarBase` and all its subclasses, and `StatusBarWx`, are deprecated, as messages are now displayed in the toolbar instead.

### `ismath` parameter of `draw_tex`

The `ismath` parameter of the `draw_tex` method of all renderer classes is deprecated (as a call to `draw_tex` -- not to be confused with `draw_text`! -- means that the entire string should be passed to the `usetex` machinery anyways). Likewise, the text machinery will no longer pass the `ismath` parameter when calling `draw_tex` (this should only matter for backend implementers).

Passing `ismath="TeX!"` to *RendererAgg.get_text_width_height_descent* is deprecated. Pass `ismath="TeX"` instead, consistently with other low-level APIs which support the values True, False, and "TeX" for `ismath`.

### `matplotlib.ttconv`

This module is deprecated.

### Stricter PDF metadata keys in PGF

Saving metadata in PDF with the PGF backend currently normalizes all keys to lowercase, unlike the PDF backend, which only accepts the canonical case. This is deprecated; in a future version, only the canonically cased keys listed in the PDF specification (and the *PdfPages* documentation) will be accepted.

### Qt modifier keys

The `MODIFIER_KEYS`, `SUPER`, `ALT`, `CTRL`, and `SHIFT` global variables of the `matplotlib.backends.backend_qt4agg`, `matplotlib.backends.backend_qt4cairo`, *`matplotlib.backends.backend_qt5agg`* and *`matplotlib.backends.backend_qt5cairo`* modules are deprecated.

#### `TexManager`

The `TexManager.serif`, `TexManager.sans_serif`, `TexManager.cursive` and `TexManager.monospace` attributes are deprecated.

### Removals

The following deprecated APIs have been removed:

### Modules

- `backends.qt_editor.formlayout` (use the formlayout module available on PyPI instead).

### Classes, methods and attributes

- `artist.Artist.aname` property (no replacement)
- `axis.Axis.iter_ticks` (no replacement)
- Support for custom backends that do not provide a `backend_bases.GraphicsContextBase.set_hatch_color` method
- `backend_bases.RendererBase.strip_math()` (use `cbook.strip_math()` instead)
- `backend_wx.debug_on_error()` (no replacement)
- `backend_wx.raise_msg_to_str()` (no replacement)
- `backend_wx.fake_stderr` (no replacement)
- `backend_wx.MenuButtonWx` (no replacement)
- `backend_wx.PrintoutWx` (no replacement)
- `_backend_tk.NavigationToolbar2Tk.set_active()` (no replacement)
- `backend_ps.PsBackendHelper.gs_exe` property (no replacement)
- `backend_ps.PsBackendHelper.gs_version` property (no replacement)
- `backend_ps.PsBackendHelper.supports_ps2write` property (no replacement)
- `backend_ps.RendererPS.afmfontd` property (no replacement)

- `backend_ps.GraphicsContextPS.shouldstroke` property (no replacement)
- `backend_gtk3.FileChooserDialog` (no replacement)
- `backend_gtk3.SaveFigureGTK3.get_filechooser()` (no replacement)
- `backend_gtk3.NavigationToolbar2GTK3.get_filechooser()` (no replacement)
- `backend_gtk3cairo.FigureManagerGTK3Cairo` (use `backend_gtk3.FigureManagerGTK3` instead)
- `backend_pdf.RendererPdf.afm_font_cache` property (no replacement)
- `backend_pgf.LatexManagerFactory` (no replacement)
- `backend_qt5.NavigationToolbar2QT.buttons` property (no replacement)
- `backend_qt5.NavigationToolbar2QT.adj_window` property (no replacement)
- `bezier.find_r_to_boundary_of_closedpath()` (no replacement)
- `cbook.dedent()` (use `inspect.cleandoc` instead)
- `cbook.get_label()` (no replacement)
- `cbook.is_hashable()` (use `isinstance(..., collections.abc.Hashable)` instead)
- `cbook.iterable()` (use `numpy.iterable()` instead)
- `cbook.safezip()` (no replacement)
- `colorbar.ColorbarBase.get_cmap` (use `ScalarMappable.get_cmap` instead)
- `colorbar.ColorbarBase.set_cmap` (use `ScalarMappable.set_cmap` instead)
- `colorbar.ColorbarBase.get_clim` (use `ScalarMappable.get_clim` instead)
- `colorbar.ColorbarBase.set_clim` (use `ScalarMappable.set_clim` instead)
- `colorbar.ColorbarBase.set_norm` (use `ScalarMappable.set_norm` instead)
- `dates.seconds()` (no replacement)
- `dates.minutes()` (no replacement)
- `dates.hours()` (no replacement)
- `dates.weeks()` (no replacement)
- `dates.strpdate2num` and `dates.bytespdate2num` (use `time.strptime` or `dateutil.parser.parse` or *`dates.datestr2num`* instead)
- `docstring.Appender` (no replacement)
- `docstring.dedent()` (use `inspect.getdoc` instead)
- `docstring.copy_dedent()` (use `docstring.copy()` and `inspect.getdoc` instead)
- `font_manager.OSXInstalledFonts()` (no replacement)
- `image.BboxImage.interp_at_native` property (no replacement)

- `lines.Line2D.verticalOffset` property (no replacement)
- `matplotlib.checkdep_dvipng` (no replacement)
- `matplotlib.checkdep_ghostscript` (no replacement)
- `matplotlib.checkdep_pdftops` (no replacement)
- `matplotlib.checkdep_inkscape` (no replacement)
- `matplotlib.get_py2exe_datafiles` (no replacement)
- `matplotlib.tk_window_focus` (use `rcParams['tk.window_focus']` instead)
- `mlab.demean()` (use `mlab.detrend_mean()` instead)
- `path.get_paths_extents()` (use `path.get_path_collection_extents()` instead)
- `path.Path.has_nonfinite()` (use `not np.isfinite(self.vertices).all()` instead)
- `projections.process_projection_requirements()` (no replacement)
- `pyplot.plotfile()` (Instead, load the data using `pandas.read_csv` or `numpy.loadtxt` or similar and use regular pyplot functions to plot the loaded data.)
- `quiver.Quiver.color()` (use `Quiver.get_facecolor()` instead)
- `quiver.Quiver.keyvec` property (no replacement)
- `quiver.Quiver.keytext` property (no replacement)
- `rcsetup.validate_qt4()` (no replacement)
- `rcsetup.validate_qt5()` (no replacement)
- `rcsetup.validate_verbose()` (no replacement)
- `rcsetup.ValidateInterval` (no replacement)
- `scale.LogTransformBase` (use `scale.LogTransform` instead)
- `scale.InvertedLogTransformBase` (use `scale.InvertedLogTransform` instead)
- `scale.Log10Transform` (use `scale.LogTransform` instead)
- `scale.InvertedLog10Transform` (use `scale.InvertedLogTransform` instead)
- `scale.Log2Transform` (use `scale.LogTransform` instead)
- `scale.InvertedLog2Transform` (use `scale.InvertedLogTransform` instead)
- `scale.NaturalLogTransform` (use `scale.LogTransform` instead)
- `scale.InvertedNaturalLogTransform` (use `scale.InvertedLogTransform` instead)
- `scale.get_scale_docs()` (no replacement)
- `sphinxext.plot_directive.plot_directive()` (use the class `PlotDirective` instead)

- `sphinxext.mathmpl.math_directive()` (use the class `MathDirective` instead)
- `spines.Spine.is_frame_like()` (no replacement)
- `testing.decorators.switch_backend()` (use `@pytest.mark.backend` decorator instead)
- `text.Text.is_math_text()` (use `cbook.is_math_text()` instead)
- `text.TextWithDash()` (use `text.Annotation` instead)
- `textpath.TextPath.is_math_text()` (use `cbook.is_math_text()` instead)
- `textpath.TextPath.text_get_vertices_codes()` (use `textpath.text_to_path.get_text_path()` instead)
- `textpath.TextToPath.glyph_to_path()` (use `font.get_path()` and manual translation of the vertices instead)
- `ticker.OldScalarFormatter.pprint_val()` (no replacement)
- `ticker.ScalarFormatter.pprint_val()` (no replacement)
- `ticker.LogFormatter.pprint_val()` (no replacement)
- `ticker.decade_down()` (no replacement)
- `ticker.decade_up()` (no replacement)
- `Tick` properties `gridOn`, `tick1On`, `tick2On`, `label1On`, `label2On` (use `set_visible()` / `get_visible()` on `Tick.gridline`, `Tick.tick1line`, `Tick.tick2line`, `Tick.label1`, `Tick.label2` instead)
- `widgets.SpanSelector.buttonDown` property (no replacement)
- `mplot3d.proj3d.line2d()` (no replacement)
- `mplot3d.proj3d.line2d_dist()` (no replacement)
- `mplot3d.proj3d.line2d_seg_dist()` (no replacement)
- `mplot3d.proj3d.mod()` (use `numpy.linalg.norm` instead)
- `mplot3d.proj3d.proj_transform_vec()` (no replacement)
- `mplot3d.proj3d.proj_transform_vec_clip()` (no replacement)
- `mplot3d.proj3d.vec_pad_ones()` (no replacement)
- `mplot3d.proj3d.proj_trans_clip_points()` (no replacement)
- `mplot3d.art3d.norm_angle()` (no replacement)
- `mplot3d.art3d.norm_text_angle()` (no replacement)
- `mplot3d.art3d.path_to_3d_segment()` (no replacement)
- `mplot3d.art3d.paths_to_3d_segments()` (no replacement)
- `mplot3d.art3d.path_to_3d_segment_with_codes()` (no replacement)
- `mplot3d.art3d.paths_to_3d_segments_with_codes()` (no replacement)

- `mplot3d.art3d.get_patch_verts()` (no replacement)
- `mplot3d.art3d.get_colors()` (no replacement)
- `mplot3d.art3d.zalpha()` (no replacement)
- `mplot3d.axis3d.get_flip_min_max()` (no replacement)
- `mplot3d.axis3d.Axis.get_tick_positions()` (no replacement)
- `axisartist.axis_artist.UnimplementedException` (no replacement)
- `axisartist.axislines.SimpleChainedObjects` (use `axis_grid1.mpl_axes.SimpleChainedObjects` instead)
- `axisartist.axislines.Axes.AxisDict` (use `axis_grid1.mpl_axes.Axes.AxisDict` instead)

### Arguments

- `Axes.text()` / `pyplot.text()` do not support the parameter `withdash` anymore. Use `Axes.annotate()` and `pyplot.annotate()` instead.
- The first parameter of *matplotlib.use* has been renamed from `arg` to `backend` (only relevant if you pass by keyword).
- The parameter `warn` of *matplotlib.use* has been removed. A failure to switch the backend will now always raise an `ImportError` if `force` is set; catch that error if necessary.
- All parameters of *matplotlib.use* except the first one are now keyword-only.
- The unused parameters `shape` and `imlim` of *imshow()* are now removed. All parameters beyond `extent` are now keyword-only.
- The unused parameter `interp_at_native` of *BboxImage* has been removed.
- The parameter `usetex` of *TextToPath.get_text_path* has been removed. Use `is_math='TeX'` instead.
- The parameter `block` of `show()` is now keyword-only, and arbitrary arguments or keyword arguments are no longer accepted.
- The parameter `frameon` of *Figure.savefig* has been removed. Use `facecolor="none"` to get a transparent background.
- Passing a `wx.EvtHandler` as the first argument to `backend_wx.TimerWx` is not supported anymore; the signature of `TimerWx` is now consistent with *TimerBase*.
- The `manage_xticks` parameter of *boxplot* and *bxp* has been renamed to `manage_ticks`.
- The `normed` parameter of *hist2d* has been renamed to `density`.
- The `s` parameter of *Annotation* has been renamed to `text`.
- For all functions in *bezier* that supported a `tolerance` parameter, this parameter has been renamed to `tolerance`.

- `axis("normal")` is not supported anymore. Use the equivalent `axis("auto")` instead.
- `axis()` does not accept arbitrary keyword arguments anymore.
- `Axis.set_ticklabels()` does not accept arbitrary positional arguments other than `ticklabels`.
- `mpl_toolkits.mplot3d.art3d.Poly3DCollection.set_zsort` does not accept the value `True` anymore. Pass the equivalent value 'average' instead.
- *AnchoredText* no longer accepts `horizontalalignment` or `verticalalignment` keyword arguments.
- *ConnectionPatch* no longer accepts the `arrow_transmuter` and `connector` keyword arguments, which did nothing since 3.0.
- *FancyArrowPatch* no longer accepts the `arrow_transmuter` and `connector` keyword arguments, which did nothing since 3.0.
- *TextPath* no longer accepts arbitrary positional or keyword arguments.
- *MaxNLocator.set_params()* no longer accepts arbitrary keyword arguments.
- *pie* no longer accepts and squeezes non-1D inputs; pass 1D input to the `x` argument.
- Passing (n, 1)-shaped error arrays to *Axes.errorbar()* is no longer supported; pass a 1D array instead.

### rcParams

- The `text.latex.unicode` rcParam has been removed, with no replacement. Matplotlib now always supports unicode in usetex.
- The `savefig.frameon` rcParam has been removed. Set *rcParams["savefig.facecolor"]* (default: `'auto'`) to "none" to get a transparent background.
- The `pgf.debug`, `verbose.fileo` and `verbose.verbose.level` rcParams, which had no effect, have been removed.
- Support for setting *rcParams["mathtext.default"]* (default: `'it'`) to "circled" has been removed.

### Environment variables

- `MATPLOTLIBDATA` (no replacement).

#### mathtext

- The `\stackrel` command (which behaved differently from its LaTeX version) has been removed. Use `\genfrac` instead.
- The `\mathcircled` command has been removed. Directly use Unicode characters, such as `'\N{CIRCLED LATIN CAPITAL LETTER A}'`, instead.

#### Development changes

#### Matplotlib now requires numpy>=1.15

#### Matplotlib now uses Pillow to save and read pngs

The builtin png encoder and decoder has been removed, and Pillow is now a dependency. Note that when reading 16-bit RGB(A) images, Pillow truncates them to 8-bit precision, whereas the old builtin decoder kept the full precision.

The deprecated wx backend (not wxagg!) now always uses wx's builtin jpeg and tiff support rather than relying on Pillow for writing these formats; this behavior is consistent with wx's png output.

## 9.8 Version 3.2

### 9.8.1 What's new in Matplotlib 3.2 (Mar 04, 2020)

For a list of all of the issues and pull requests since the last revision, see the *GitHub statistics for 3.9.0 (May 15, 2024)*.

**Table of Contents**


- *What's new in Matplotlib 3.2 (Mar 04, 2020)*
  - *Unit converters recognize subclasses*
  - `imsave` *accepts metadata and PIL options*
  - `cbook.normalize_kwargs`
  - `FontProperties` *accepts* `os.PathLike`
  - *Gouraud-shading alpha channel in PDF backend*
  - *Kerning adjustments now use correct values*
  - *bar3d lightsource shading*
  - *Shifting errorbars*
  - *Improvements in Logit scale ticker and formatter*

- *rcParams for axes title location and color*
- *3-digit and 4-digit hex colors*
- *Added support for RGB(A) images in pcolorfast*

### Unit converters recognize subclasses

Unit converters now also handle instances of subclasses of the class they have been registered for.

### `imsave` accepts metadata and PIL options

*imsave* has gained support for the `metadata` and `pil_kwargs` parameters. These parameters behave similarly as for the *Figure.savefig()* method.

### `cbook.normalize_kwargs`

*cbook.normalize_kwargs* now presents a convenient interface to normalize artist properties (e.g., from "lw" to "linewidth"):

```
>>> cbook.normalize_kwargs({"lw": 1}, Line2D)
{"linewidth": 1}
```

The first argument is the mapping to be normalized, and the second argument can be an artist class or an artist instance (it can also be a mapping in a specific format; see the function's docstring for details).

### `FontProperties` accepts `os.PathLike`

The *fname* argument to *FontProperties* can now be an `os.PathLike`, e.g.

```
>>> FontProperties(fname=pathlib.Path("/path/to/font.ttf"))
```

### Gouraud-shading alpha channel in PDF backend

The pdf backend now supports an alpha channel in Gouraud-shaded triangle meshes.

### Kerning adjustments now use correct values

Due to an error in how kerning adjustments were applied, previous versions of Matplotlib would under-correct kerning. This version will now correctly apply kerning (for fonts supported by FreeType). To restore the old behavior (e.g., for test images), you may set *rcParams["text.kerning_factor"]* (default: `0`) to 6 (instead of 0). Other values have undefined behavior.

Note how the spacing between characters is uniform between their bounding boxes (above). With corrected kerning (below), slanted characters (e.g., AV or VA) will be spaced closer together, as well as various other character pairs, depending on font support (e.g., T and e, or the period after the W).

### bar3d lightsource shading

*bar3d()* now supports lighting from different angles when the *shade* parameter is `True`, which can be configured using the `lightsource` parameter.

### Shifting errorbars

Previously, *errorbar()* accepted a keyword argument *errorevery* such that the command `plt.errorbar(x, y, yerr, errorevery=6)` would add error bars to datapoints `x[::6], y[::6]`.

*errorbar()* now also accepts a tuple for *errorevery* such that `plt.errorbar(x, y, yerr, errorevery=(start, N))` adds error bars to points `x[start::N], y[start::N]`.

### Improvements in Logit scale ticker and formatter

Introduced in version 1.5, the logit scale didn't have an appropriate ticker and formatter. Previously, the location of ticks was not zoom dependent, too many labels were displayed causing overlapping which broke readability, and label formatting did not adapt to precision.

Starting from this version, the logit locator has nearly the same behavior as the locator for the log scale or the linear scale, depending on used zoom. The number of ticks is controlled. Some minor labels are displayed adaptively as sublabels in log scale. Formatting is adapted for probabilities and the precision adapts to the scale.

### rcParams for axes title location and color

Two new rcParams have been added: *rcParams["axes.titlelocation"]* (default: `'center'`) denotes the default axes title alignment, and *rcParams["axes.titlecolor"]* (default: `'auto'`) the default axes title color.

Valid values for `axes.titlelocation` are: left, center, and right. Valid values for `axes.titlecolor` are: auto or a color. Setting it to auto will fall back to previous behaviour, which is using the color in `text.color`.

#### 3-digit and 4-digit hex colors

Colors can now be specified using 3-digit or 4-digit hex colors, shorthand for the colors obtained by duplicating each character, e.g. `#123` is equivalent to `#112233` and `#123a` is equivalent to `#112233aa`.

#### Added support for RGB(A) images in pcolorfast

*Axes.pcolorfast* now accepts 3D images (RGB or RGBA) arrays.

### 9.8.2 API Changes for 3.2.0


- *Behavior changes*
- *Deprecations*
- *Removals*
- *Development changes*


#### Behavior changes

#### Reduced default value of `rcParams["axes.formatter.limits"]` (default: `[-5, 6]`)

Changed the default value of *rcParams["axes.formatter.limits"]* (default: `[-5, 6]`) from -7, 7 to -5, 6 for better readability.

#### `matplotlib.colorbar.Colorbar` uses un-normalized axes for all mappables

Before 3.0, *matplotlib.colorbar.Colorbar* (*colorbar*) normalized all axes limits between 0 and 1 and had custom tickers to handle the labelling of the colorbar ticks. After 3.0, colorbars constructed from mappables that were *not* contours were constructed with axes that had limits between `vmin` and `vmax` of the mappable's norm, and the tickers were made children of the normal axes tickers.

This version of Matplotlib extends that to mappables made by contours, and allows the axes to run between the lowest boundary in the contour and the highest.

Code that worked around the normalization between 0 and 1 will need to be modified.

#### MovieWriterRegistry

*MovieWriterRegistry* now always checks the availability of the writer classes before returning them. If one wishes, for example, to get the first available writer, without performing the availability check on subsequent writers, it is now possible to iterate over the registry, which will yield the names of the available classes.

### Autoscaling

Matplotlib used to recompute autoscaled limits after every plotting (`plot()`, `bar()`, etc.) call. It now only does so when actually rendering the canvas, or when the user queries the Axes limits. This is a major performance improvement for plots with a large number of artists.

In particular, this means that artists added manually with *Axes.add_line*, *Axes.add_patch*, etc. will be taken into account by the autoscale, even without an explicit call to *Axes.autoscale_view*.

In some cases, this can result in different limits being reported. If this is an issue, consider triggering a draw with `fig.canvas.draw()`.

Autoscaling has also changed for artists that are based on the *Collection* class. Previously, the method that calculates the automatic limits *Collection.get_datalim* tried to take into account the size of objects in the collection and make the limits large enough to not clip any of the object, i.e., for *Axes.scatter* it would make the limits large enough to not clip any markers in the scatter. This is problematic when the object size is specified in physical space, or figure-relative space, because the transform from physical units to data limits requires knowing the data limits, and becomes invalid when the new limits are applied. This is an inverse problem that is theoretically solvable (if the object is physically smaller than the axes), but the extra complexity was not deemed worth it, particularly as the most common use case is for markers in scatter that are usually small enough to be accommodated by the default data limit margins.

While the new behavior is algorithmically simpler, it is conditional on properties of the *Collection* object:

1. `offsets = None`, `transform` is a child of `Axes.transData`: use the paths for the automatic limits (i.e. for *LineCollection* in *Axes.streamplot*).
2. `offsets != None`, and `offset_transform` is child of `Axes.transData`:
   a) `transform` is child of `Axes.transData`: use the `path + offset` for limits (i.e., for *Axes.bar*).
   b) `transform` is not a child of `Axes.transData`: just use the offsets for the limits (i.e. for scatter)
3. otherwise return a null *Bbox*.

While this seems complicated, the logic is simply to use the information from the object that are in data space for the limits, but not information that is in physical units.

### log-scale bar() / hist() autolimits

The autolimits computation in *bar* and *hist* when the axes already uses log-scale has changed to match the computation when the axes is switched to log-scale after the call to *bar* and *hist*, and when calling `bar(..., log=True)` / `hist(..., log=True)`: if there are at least two different bar heights, add the normal axes margins to them (in log-scale); if there is only a single bar height, expand the axes limits by one order of magnitude around it and then apply axes margins.

### Axes labels spanning multiple rows/columns

`Axes.label_outer` now correctly keep the x labels and tick labels visible for Axes spanning multiple rows, as long as they cover the last row of the Axes grid. (This is consistent with keeping the y labels and tick labels visible for Axes spanning multiple columns as long as they cover the first column of the Axes grid.)

The `Axes.is_last_row` and `Axes.is_last_col` methods now correctly return True for Axes spanning multiple rows, as long as they cover the last row or column respectively. Again this is consistent with the behavior for axes covering the first row or column.

The `Axes.rowNum` and `Axes.colNum` attributes are deprecated, as they only refer to the first grid cell covered by the Axes. Instead, use the new `ax.get_subplotspec().rowspan` and `ax.get_subplotspec().colspan` properties, which are `range` objects indicating the whole span of rows and columns covered by the subplot.

(Note that all methods and attributes mentioned here actually only exist on the `Subplot` subclass of *Axes*, which is used for grid-positioned Axes but not for Axes positioned directly in absolute coordinates.)

The *GridSpec* class gained the `nrows` and `ncols` properties as more explicit synonyms for the parameters returned by `GridSpec.get_geometry`.

### Locators

When more than *Locator.MAXTICKS* ticks are generated, the behavior of *Locator.raise_if_exceeds* changed from raising a RuntimeError to emitting a log at WARNING level.

### nonsingular Locators

`Locator.nonsingular` (introduced in mpl 3.1), `DateLocator.nonsingular`, and `AutoDateLocator.nonsingular` now returns a range `v0, v1` with `v0 <= v1`. This behavior is consistent with the implementation of `nonsingular` by the `LogLocator` and `LogitLocator` subclasses.

#### `get_data_ratio`

`Axes.get_data_ratio` now takes the axes scale into account (linear, log, logit, etc.) before computing the y-to-x ratio. This change allows fixed aspects to be applied to any combination of x and y scales.

### Artist sticky edges

Previously, the `sticky_edges` attribute of artists was a list of values such that if an axis limit coincides with a sticky edge, it would not be expanded by the axes margins (this is the mechanism that e.g. prevents margins from being added around images).

`sticky_edges` now have an additional effect on margins application: even if an axis limit did not coincide with a sticky edge, it cannot *cross* a sticky edge through margin application -- instead, the margins will only expand the axis limit until it bumps against the sticky edge.

This change improves the margins of axes displaying a *`streamplot`*:

- if the streamplot goes all the way to the edges of the vector field, then the axis limits are set to match exactly the vector field limits (whereas they would sometimes be off by a small floating point error previously).
- if the streamplot does not reach the edges of the vector field (e.g., due to the use of `start_points` and `maxlength`), then margins expansion will not cross the vector field limits anymore.

This change is also used internally to ensure that polar plots don't display negative *r* values unless the user really passes in a negative value.

### `gid` in svg output

Previously, if a figure, axis, legend or some other artists had a custom `gid` set (e.g. via `.set_gid()`), this would not be reflected in the svg output. Instead a default gid, like `figure_1` would be shown. This is now fixed, such that e.g. `fig.set_gid("myfigure")` correctly shows up as `<g id="myfigure">` in the svg file. If you relied on the gid having the default format, you now need to make sure not to set the `gid` parameter of the artists.

### Fonts

Font weight guessing now first checks for the presence of the FT_STYLE_BOLD_FLAG before trying to match substrings in the font name. In particular, this means that Times New Roman Bold is now correctly detected as bold, not normal weight.

### Color-like checking

*matplotlib.colors.is_color_like* used to return True for all string representations of floats. However, only those with values in 0-1 are valid colors (representing grayscale values). *is_color_like* now returns False for string representations of floats outside 0-1.

### Default image interpolation

Images displayed in Matplotlib previously used nearest-neighbor interpolation, leading to aliasing effects for downscaling and non-integer upscaling.

New default for *rcParams["image.interpolation"]* (default: `'antialiased'`) is the new option "antialiased". `imshow(A, interpolation='antialiased')` will apply a Hanning filter when resampling the data in A for display (or saving to file) *if* the upsample rate is less than a factor of three, and not an integer; downsampled data is always smoothed at resampling.

To get the old behavior, set *rcParams["image.interpolation"]* (default: `'antialiased'`) to the old default "nearest" (or specify the `interpolation` kwarg of *Axes.imshow*)

To always get the anti-aliasing behavior, no matter what the up/down sample rate, set *rcParams["image.interpolation"]* (default: `'antialiased'`) to "hanning" (or one of the other filters available).

Note that the "hanning" filter was chosen because it has only a modest performance penalty. Anti-aliasing can be improved with other filters.

### rcParams

When using *RendererSVG* with `rcParams["svg.image_inline"] == True`, externally written images now use a single counter even if the `renderer.basename` attribute is overwritten, rather than a counter per basename.

This change will only affect you if you used `rcParams["svg.image_inline"] = True` (the default is False) *and* manually modified `renderer.basename`.

Changed the default value of *rcParams["axes.formatter.limits"]* (default: `[-5, 6]`) from -7, 7 to -5, 6 for better readability.

#### add_subplot()

*Figure.add_subplot()* and *pyplot.subplot()* do not accept a *figure* keyword argument anymore. It only used to work anyway if the passed figure was `self` or the current figure, respectively.

#### `indicate_inset()`

In <= 3.1.0, *indicate_inset* and *indicate_inset_zoom* were documented as returning a 4-tuple of *ConnectionPatch*, where in fact they returned a 4-length list.

They now correctly return a 4-tuple. *indicate_inset* would previously raise an error if the optional *inset_ax* was not supplied; it now completes successfully, and returns *None* instead of the tuple of `ConnectionPatch`.

### PGF backend

The pgf backend's get_canvas_width_height now returns the canvas size in display units rather than in inches, which it previously did. The new behavior is the correct one given the uses of `get_canvas_width_height` in the rest of the codebase.

The pgf backend now includes images using `\includegraphics` instead of `\pgfimage` if the version of `graphicx` is recent enough to support the `interpolate` option (this is detected automatically).

#### `cbook`

The default value of the "obj_type" parameter to `cbook.warn_deprecated` has been changed from "attribute" (a default that was never used internally) to the empty string.

### Testing

The test suite no longer turns on the Python fault handler by default. Set the standard `PYTHONFAULTHANDLER` environment variable to do so.

### Backend `supports_blit`

Backends do not need to explicitly define the flag `supports_blit` anymore. This is only relevant for backend developers. Backends had to define the flag `supports_blit`. This is not needed anymore because the blitting capability is now automatically detected.

### Exception changes

Various APIs that raised a `ValueError` for incorrectly typed inputs now raise `TypeError` instead: *backend_bases.GraphicsContextBase.set_clip_path*, `blocking_input.BlockingInput.__call__`, `matplotlib.cm.register_cmap`, *dviread.DviFont*, *rcsetup.validate_hatch*, `.rcsetup.validate_animation_writer_path`, *spines.Spine*, many classes in the *matplotlib.transforms* module and *matplotlib.tri* package, and Axes methods that take a `norm` parameter.

If extra kwargs are passed to *LogScale*, `TypeError` will now be raised instead of `ValueError`.

### mplot3d auto-registration

*mpl_toolkits.mplot3d* is always registered by default now. It is no longer necessary to import mplot3d to create 3d axes with

```
ax = fig.add_subplot(111, projection="3d")
```

### `SymLogNorm` now has a *base* parameter

Previously, *SymLogNorm* had no *base* keyword argument and the base was hard-coded to `base=np.e`. This was inconsistent with the default behavior of *SymmetricalLogScale* (which defaults to `base=10`) and the use of the word "decade" in the documentation.

In preparation for changing the default base to 10, calling *SymLogNorm* without the new *base* keyword argument emits a deprecation warning.

## Deprecations

### `matplotlib.use`

The `warn` parameter to *matplotlib.use()* is deprecated (catch the `ImportError` emitted on backend switch failure and reemit a warning yourself if so desired).

### plotfile

`.pyplot.plotfile` is deprecated in favor of separately loading and plotting the data. Use pandas or NumPy to load data, and pandas or matplotlib to plot the resulting data.

### axes and axis

Setting `Axis.major.locator`, `Axis.minor.locator`, `Axis.major.formatter` or `Axis.minor.formatter` to an object that is not a subclass of *Locator* or *Formatter* (respectively) is deprecated. Note that these attributes should usually be set using *Axis.set_major_locator*, *Axis.set_minor_locator*, etc. which already raise an exception when an object of the wrong class is passed.

Passing more than one positional argument or unsupported keyword arguments to *axis()* is deprecated (such arguments used to be silently ignored).

#### `minor` argument will become keyword-only

Using the parameter `minor` to `get_*ticks()` / `set_*ticks()` as a positional parameter is deprecated. It will become keyword-only in future versions.

### `axes_grid1`

The `mpl_toolkits.axes_grid1.colorbar` module and its colorbar implementation are deprecated in favor of *matplotlib.colorbar*, as the former is essentially abandoned and the latter is a more featureful replacement with a nearly compatible API (for example, the following additional keywords are supported: `panchor`, `extendfrac`, `extendrect`).

The main differences are:

- Setting the ticks on the colorbar is done by calling `colorbar.set_ticks` rather than `colorbar.cbar_axis.set_xticks` or `colorbar.cbar_axis.set_yticks`; the `locator` parameter to `colorbar()` is deprecated in favor of its synonym `ticks` (which already existed previously, and is consistent with *matplotlib.colorbar*).
- The colorbar's long axis is accessed with `colorbar.xaxis` or `colorbar.yaxis` depending on the orientation, rather than `colorbar.cbar_axis`.
- The default ticker is no longer `MaxNLocator(5)`, but a `_ColorbarAutoLocator`.
- Overdrawing multiple colorbars on top of one another in a single Axes (e.g. when using the `cax` attribute of *ImageGrid* elements) is not supported; if you previously relied on the second colorbar being drawn over the first, you can call `cax.cla()` to clear the axes before drawing the second colorbar.

During the deprecation period, the `mpl_toolkits.legacy_colorbar` rcParam can be set to True to use `mpl_toolkits.axes_grid1.colorbar` in *mpl_toolkits.axes_grid1* code with a deprecation warning (the default), or to False to use `matplotlib.colorbar`.

Passing a `pad` size of `None` (the default) as a synonym for zero to the `append_axes`, `new_horizontal` and `new_vertical` methods of *axes_grid1.axes_divider.AxesDivider* is deprecated. In a future release, the default value of `None` will mean "use *rcParams["figure.subplot.wspace"]* (default: `0.2`) or *rcParams["figure.subplot.hspace"]* (default: `0.2`)" (depending on the orientation). Explicitly pass `pad=0` to keep the old behavior.

### Axes3D

`mplot3d.axis3d.get_flip_min_max` is deprecated.

`axes3d.unit_bbox` is deprecated (use `Bbox.unit` instead).

`axes3d.Axes3D.w_xaxis`, `.w_yaxis`, and `.w_zaxis` are deprecated (use `.xaxis`, `.yaxis`, and `.zaxis` instead).

#### matplotlib.cm

`cm.revcmap` is deprecated. Use *Colormap.reversed* to reverse a colormap.

`cm.datad` no longer contains entries for reversed colormaps in their "unconverted" form.

### axisartist

`mpl_toolkits.axisartist.grid_finder.GridFinderBase` is deprecated (its only use is to be inherited by the *GridFinder* class which just provides more defaults in the constructor and directly sets the transforms, so `GridFinderBase`'s methods were just moved to *GridFinder*).

`axisartist.axis_artist.BezierPath` is deprecated (use *patches.PathPatch* to draw arbitrary Paths).

`AxisArtist.line` is now a *patches.PathPatch* instance instead of a `BezierPath` instance.

Returning a factor equal to None from axisartist Locators (which are **not** the same as "standard" tick Locators), or passing a factor equal to None to axisartist Formatters (which are **not** the same as "standard" tick Formatters) is deprecated. Pass a factor equal to 1 instead.

For the *mpl_toolkits.axisartist.axis_artist.AttributeCopier* class, the constructor and the `set_ref_artist` method, and the *default_value* parameter of `get_attribute_from_ref_artist`, are deprecated.

Deprecation of the constructor means that classes inheriting from *AttributeCopier* should no longer call its constructor.

### Locators

The unused `Locator.autoscale` method is deprecated (pass the axis limits to *Locator.view_limits* instead).

### Animation

The following methods and attributes of the *MovieWriterRegistry* class are deprecated: `set_dirty`, `ensure_not_dirty`, `reset_available_writers`, `avail`.

#### smart_bounds()

The "smart_bounds" functionality is deprecated. This includes `Axis.set_smart_bounds()`, `Axis.get_smart_bounds()`, `Spine.set_smart_bounds()`, and `Spine.get_smart_bounds()`.

#### boxplot()

Setting the `whis` parameter of *`Axes.boxplot`* and *`cbook.boxplot_stats`* to "range" to mean "the whole data range" is deprecated; set it to (0, 100) (which gets interpreted as percentiles) to achieve the same effect.

#### fill_between()

Passing scalars to parameter *where* in `fill_between()` and `fill_betweenx()` is deprecated. While the documentation already states that *where* must be of the same size as *x* (or *y*), scalars were accepted and broadcasted to the size of *x*. Non-matching sizes will raise a `ValueError` in the future.

#### scatter()

Passing the *verts* parameter to *`axes.Axes.scatter`* is deprecated; use the *marker* parameter instead.

#### tight_layout()

The `renderer` parameter to *`Figure.tight_layout`* is deprecated; this method now always uses the renderer instance cached on the *`Figure`*.

### rcParams

The `rcsetup.validate_animation_writer_path` function is deprecated.

Setting *`rcParams["savefig.format"]`* (default: `'png'`) to "auto" is deprecated; use its synonym "png" instead.

Setting *`rcParams["text.hinting"]`* (default: `'force_autohint'`) to True or False is deprecated; use their synonyms "auto" or "none" instead.

`rcsetup.update_savefig_format` is deprecated.

`rcsetup.validate_path_exists` is deprecated (use `os.path.exists` to check whether a path exists).

`rcsetup.ValidateInterval` is deprecated.

### Dates

`dates.mx2num` is deprecated.

### TK

`NavigationToolbar2Tk.set_active` is deprecated, as it has no (observable) effect.

### WX

`FigureFrameWx.statusbar` and `NavigationToolbar2Wx.statbar` are deprecated. The status bar can be retrieved by calling standard wx methods (`frame.GetStatusBar()` and `toolbar.GetTopLevelParent().GetStatusBar()`).

`backend_wx.ConfigureSubplotsWx.configure_subplots` and `backend_wx.ConfigureSubplotsWx.get_canvas` are deprecated.

### PGF

`backend_pgf.repl_escapetext` and `backend_pgf.repl_mathdefault` are deprecated.

`RendererPgf.latexManager` is deprecated.

### FigureCanvas

`FigureCanvasBase.draw_cursor` (which has never done anything and has never been overridden in any backend) is deprecated.

`FigureCanvasMac.invalidate` is deprecated in favor of its synonym, `FigureCanvasMac.draw_idle`.

The `dryrun` parameter to the various `FigureCanvasFoo.print_foo` methods is deprecated.

### QuiverKey doc

`quiver.QuiverKey.quiverkey_doc` is deprecated; use `quiver.QuiverKey.__init__.__doc__` instead.

### matplotlib.mlab

`mlab.apply_window` and `mlab.stride_repeat` are deprecated.

### Fonts

`font_manager.JSONEncoder` is deprecated. Use *font_manager.json_dump* to dump a *FontManager* instance.

`font_manager.createFontList` is deprecated. *font_manager.FontManager.addfont* is now available to register a font at a given path.

The `as_str`, `as_rgba_str`, `as_array`, `get_width` and `get_height` methods of `matplotlib.ft2font.FT2Image` are deprecated. Convert the `FT2Image` to a NumPy array with `np.asarray` before processing it.

### Colors

The function `matplotlib.colors.makeMappingArray` is not considered part of the public API any longer. Thus, it's deprecated.

Using a string of single-character colors as a color sequence (e.g. "rgb") is deprecated. Use an explicit list instead.

### Scales

Passing unsupported keyword arguments to *ScaleBase*, and its subclasses *LinearScale* and *SymmetricalLogScale*, is deprecated and will raise a `TypeError` in 3.3.

If extra keyword arguments are passed to *LogScale*, `TypeError` will now be raised instead of `ValueError`.

### Testing

The `matplotlib.testing.disable_internet` module is deprecated. Use (for example) pytest-remotedata instead.

Support in *matplotlib.testing* for nose-based tests is deprecated (a deprecation is emitted if using e.g. the decorators from that module while both 1) matplotlib's conftests have not been called and 2) nose is in `sys.modules`).

`testing.is_called_from_pytest` is deprecated.

During the deprecation period, to force the generation of nose base tests, import nose first.

The *switch_backend_warn* parameter to `matplotlib.test` has no effect and is deprecated.

`testing.jpl_units.UnitDbl.UnitDbl.checkUnits` is deprecated.

### `DivergingNorm` renamed to `TwoSlopeNorm`

`DivergingNorm` was a misleading name; although the norm was developed with the idea that it would likely be used with diverging colormaps, the word 'diverging' does not describe or evoke the norm's mapping function. Since that function is monotonic, continuous, and piece-wise linear with two segments, the norm has been renamed to *`TwoSlopeNorm`*

### Misc

`matplotlib.get_home` is deprecated (use e.g. `os.path.expanduser("~")`) instead.

`matplotlib.compare_versions` is deprecated (use comparison of `distutils.version.LooseVersion`s instead).

`matplotlib.checkdep_ps_distiller` is deprecated.

`matplotlib.figure.AxesStack` is considered private API and will be removed from the public API in future versions.

`BboxBase.is_unit` is deprecated (check the Bbox extents if needed).

`Affine2DBase.matrix_from_values(...)` is deprecated. Use (for example) `Affine2D.from_values(...).get_matrix()` instead.

`style.core.is_style_file` and `style.core.iter_style_files` are deprecated.

### The `datapath` rcParam

Use *`get_data_path`* instead. (The rcParam is deprecated because it cannot be meaningfully set by an end user.) The rcParam had no effect from 3.2.0, but was deprecated only in 3.2.1. In 3.2.1+ if `'datapath'` is set in a `matplotlibrc` file it will be respected, but this behavior will be removed in 3.3.

### Removals

The `matplotlib.testing.determinism` module, which exposes no public API, has been deleted.

The following API elements have been removed:

- `backend_gtk3.PIXELS_PER_INCH`
- `backend_pgf.re_escapetext`, `backend_pgf.re_mathdefault`.
- the `matplotlib.backends.tkagg`, `matplotlib.backends.windowing`, `matplotlib.backends.wx_compat`, and `matplotlib.compat.subprocess` modules
- `RcParams.msg_depr`, `RcParams.msg_depr_ignore`, `RcParams.msg_depr_set`, `RcParams.msg_obsolete`, `RcParams.msg_backend_obsolete`
- `afm.parse_afm` (use `afm.AFM` instead)
- `axes.Axes.mouseover_set`

- `backend_cairo.ArrayWrapper`, `backend_cairo.RendererCairo.convert_path`
- `backend_gtk3.FileChooserDialog.sorted_filetypes` (use `sorted(self.filetypes.items())` instead)
- `backend_pgf.get_texcommand`
- `backend_pdf.PdfFile.texFontMap`
- `backend_ps.get_bbox`
- `backend_qt.FigureCanvasQt.keyAutoRepeat` (use `event.guiEvent.isAutoRepeat` instead), `backend_qt.error_msg_qt`, `backend_qt.exception_handler`
- `backend_wx.FigureCanvasWx.macros`
- `backends.pylab_setup`
- `cbook.Bunch` (use `types.SimpleNamespace` instead), `cbook.Locked`, `cbook.unicode_safe`, `cbook.is_numlike` (use `isinstance(..., numbers.Number)` instead), `cbook.mkdirs` (use `os.makedirs(..., exist_ok=True)` instead), `cbook.GetRealpathAndStat` (use `cbook.get_realpath_and_stat` instead), `cbook.listFiles`
- `container.Container.set_remove_method`
- `contour.ContourLabeler.cl`, `contour.ContourLabeler.cl_xy`, `contour.ContourLabeler.cl_cvalues` (use `labelTexts`, `labelXYs`, `labelCValues` instead)
- `dates.DateFormatter.strftime`, `dates.DateFormatter.strftime_pre_1900`
- `font_manager.TempCache`, `font_manager.FontManager.ttffiles`, `font_manager.FontManager.afmfiles`
- `mathtext.unichr_safe` (use `chr` instead)
- `patches.YAArrow` (use `patches.FancyArrowPatch` instead)
- `sphinxext.plot_directive.remove_coding`
- `table.Table.get_child_artists`
- `testing.compare.compare_float`, `testing.decorators.CleanupTest`, `testing.decorators.ImageComparisonTest`, `testing.decorators.skip_if_command_unavailable`, support for nose-based tests
- `text.Annotation.arrow` (use `text.Annotation.arrow_patch` instead)
- `textpath.TextToPath.tex_font_map`
- `ticker.Base`, `ticker.closeto`, `ticker.nearest_long`
- `axes_grid1.axes_divider.LocatableAxesBase`, `axes_grid1.axes_divider.locatable_axes_factory`, `axes_grid1.axes_divider.Axes` (use `axes_grid1.mpl_axes.Axes` instead), `axes_grid1.axes_divider.LocatableAxes` (use `axes_grid1.mpl_axes.Axes` instead)

- `axisartist.axes_divider.Axes`, `axisartist.axes_divider.LocatableAxes` (use `axisartist.axislines.Axes` instead)
- the *normed* keyword argument to `hist` (use *density* instead)
- passing `(verts, 0)` or `(..., 3)` when specifying a marker to specify a path or a circle, respectively (instead, use `verts` or `"o"`, respectively)
- *`rcParams["examples.directory"]`*

The following members of `matplotlib.backends.backend_pdf.PdfFile` were removed:

- `nextObject`
- `nextFont`
- `nextAlphaState`
- `nextHatch`
- `nextImage`
- `alphaStateObject`

The `required_interactive_framework` attribute of backend modules introduced in Matplotlib 3.0 has been moved to the `FigureCanvas` class, in order to let it be inherited by third-party canvas subclasses and to make it easier to know what interactive framework is required by a canvas class.

`backend_qt4.FigureCanvasQT5`, which is an alias for `backend_qt5.FigureCanvasQT` (but only exists under that name in `backend_qt4`), has been removed.

### Development changes

#### Windows build

Previously, when building the `matplotlib._png` extension, the build script would add "png" and "z" to the extensions `.libraries` attribute (if pkg-config information is not available, which is in particular the case on Windows).

In particular, this implies that the Windows build would look up files named `png.lib` and `z.lib`; but neither libpng upstream nor zlib upstream provides these files by default. (On Linux, this would look up `libpng.so` and `libz.so`, which are indeed standard names.)

Instead, on Windows, we now look up `libpng16.lib` and `zlib.lib`, which *are* the upstream names for the shared libraries (as of libpng 1.6.x).

For a statically-linked build, the upstream names are `libpng16_static.lib` and `zlibstatic.lib`; one still needs to manually rename them if such a build is desired.

### Packaging DLLs

Previously, it was possible to package Windows DLLs into the Matplotlib wheel (or sdist) by copying them into the source tree and setting the `package_data.dlls` entry in `setup.cfg`.

DLLs copied in the source tree are now always packaged; the `package_data.dlls` entry has no effect anymore. If you do not want to include the DLLs, don't copy them into the source tree.

## 9.9 Version 3.1

### 9.9.1 What's new in Matplotlib 3.1 (May 18, 2019)

For a list of all of the issues and pull requests since the last revision, see the *GitHub statistics for 3.9.0 (May 15, 2024)*.

**Table of Contents**


- *What's new in Matplotlib 3.1 (May 18, 2019)*
  - *New Features*
    - `ConciseDateFormatter`
    - *Secondary x/y Axis support*
    - `FuncScale` *for arbitrary axes scales*
    - *Legend for scatter*
    - *Matplotlib no longer requires framework app build on MacOSX backend*
  - *Figure, FigureCanvas, and Backends*
    - *Figure.frameon is now a direct proxy for the Figure patch visibility state*
    - pil_kwargs *argument added to savefig*
    - *Add* `inaxes` *method to* `FigureCanvasBase`
    - *cairo backend defaults to pycairo instead of cairocffi*
  - *Axes and Artists*
    - *axes_grid1 and axisartist Axes no longer draw spines twice*
    - *Return type of ArtistInspector.get_aliases changed*
    - `ConnectionPatch` *accepts arbitrary transforms*
    - *mplot3d Line3D now allows {set,get}_data_3d*
    - `Axes3D.voxels` *now shades the resulting voxels*
  - *Axis and Ticks*

        * *Added* `Axis.get_inverted` *and* `Axis.set_inverted`
        * *Adjust default minor tick spacing*
        * `EngFormatter` *now accepts* usetex, useMathText *as keyword only arguments*
    - *Animation and Interactivity*
        * *Support for forward/backward mouse buttons*
        * progress_callback *argument to* `save()`
        * *Add* `cache_frame_data` *keyword-only argument into* `animation.FuncAnimation`
        * *Endless Looping GIFs with PillowWriter*
        * *Adjusted* `matplotlib.widgets.Slider` *to have vertical orientation*
        * *Improved formatting of image values under cursor when a colorbar is present*
        * *MouseEvent button attribute is now an IntEnum*
    - *Configuration, Install, and Development*
        * *The MATPLOTLIBRC environment variable can now point to any "file" path*
        * *Allow LaTeX code* `pgf.preamble` *and* `text.latex.preamble` *in MATPLOTLIBRC file*
        * *New logging API*

## New Features

### `ConciseDateFormatter`

The automatic date formatter used by default can be quite verbose. A new formatter can be accessed that tries to make the tick labels appropriately concise.

## Secondary x/y Axis support

A new method provides the ability to add a second axis to an existing axes via `Axes.secondary_xaxis` and `Axes.secondary_yaxis`. See *Secondary Axis* for examples.

### `FuncScale` for arbitrary axes scales

A new *`FuncScale`* class was added (and *`FuncTransform`*) to allow the user to have arbitrary scale transformations without having to write a new subclass of *`ScaleBase`*. This can be accessed by:

```
ax.set_yscale('function', functions=(forward, inverse))
```

where `forward` and `inverse` are callables that return the scale transform and its inverse. See the last example in *Scales*.

### Legend for scatter

A new method for creating legends for scatter plots has been introduced. Previously, in order to obtain a legend for a *`scatter()`* plot, one could either plot several scatters, each with an individual label, or create proxy artists to show in the legend manually. Now, *`PathCollection`* provides a method *`legend_elements()`* to obtain the handles and labels for a scatter plot in an automated way. This makes creating a legend for a scatter plot as easy as

An example can be found in *Automated legend creation*.

### Matplotlib no longer requires framework app build on MacOSX backend

Previous versions of matplotlib required a Framework build of python to work. The app type was updated to no longer require this, so the MacOSX backend should work with non-framework python.

This also adds support for the MacOSX backend for PyPy3.

### Figure, FigureCanvas, and Backends

### Figure.frameon is now a direct proxy for the Figure patch visibility state

Accessing `Figure.frameon` (including via `get_frameon` and `set_frameon` now directly forwards to the visibility of the underlying Rectangle artist (`Figure.patch.get_frameon`, `Figure.patch.set_frameon`).

### *pil_kwargs* argument added to savefig

Matplotlib uses Pillow to handle saving to the JPEG and TIFF formats. The *`savefig()`* function gained a *pil_kwargs* keyword argument, which can be used to forward arguments to Pillow's `PIL.Image.Image.save`.

The *pil_kwargs* argument can also be used when saving to PNG. In that case, Matplotlib also uses Pillow's `PIL.Image.Image.save` instead of going through its own builtin PNG support.

### Add `inaxes` method to `FigureCanvasBase`

The *`FigureCanvasBase`* class has now an *`inaxes`* method to check whether a point is in an axes and returns the topmost axes, else None.

### cairo backend defaults to pycairo instead of cairocffi

This leads to faster import/runtime performance in some cases. The backend will fall back to cairocffi in case pycairo isn't available.

## Axes and Artists

### axes_grid1 and axisartist Axes no longer draw spines twice

Previously, spines of *`axes_grid1`* and *`axisartist`* Axes would be drawn twice, leading to a "bold" appearance. This is no longer the case.

### Return type of ArtistInspector.get_aliases changed

*`ArtistInspector.get_aliases`* previously returned the set of aliases as `{fullname: {alias1: None, alias2: None, ...}}`. The dict-to-None mapping was used to simulate a set in earlier versions of Python. It has now been replaced by a set, i.e. `{fullname: {alias1, alias2, ...}}`.

This value is also stored in `ArtistInspector.aliasd`, which has likewise changed.

### `ConnectionPatch` accepts arbitrary transforms

Alternatively to strings like `"data"` or `"axes fraction"`, *`ConnectionPatch`* now accepts any *`Transform`* as input for the *coordsA* and *coordsB* arguments. This allows to draw lines between points defined in different user defined coordinate systems. Also see *Using ConnectionPatch*.

### mplot3d Line3D now allows {set,get}_data_3d

Lines created with the 3d projection in mplot3d can now access the data using *`get_data_3d()`* which returns a tuple of array_likes containing the (x, y, z) data. The equivalent *`set_data_3d`* can be used to modify the data of an existing Line3D.

#### `Axes3D.voxels` now shades the resulting voxels

The *Axes3D.voxels* method now takes a *shade* parameter that defaults to `True`. This shades faces based on their orientation, behaving just like the matching parameters to *plot_trisurf()* and *bar3d()*. The plot below shows how this affects the output.

### Axis and Ticks

#### Added `Axis.get_inverted` and `Axis.set_inverted`

The *Axis.get_inverted* and *Axis.set_inverted* methods query and set whether the axis uses "inverted" orientation (i.e. increasing to the left for the x-axis and to the bottom for the y-axis).

They perform tasks similar to *Axes.xaxis_inverted*, *Axes.yaxis_inverted*, *Axes.invert_xaxis*, and *Axes.invert_yaxis*, with the specific difference that *Axis.set_inverted* makes it easier to set the inversion of an axis regardless of whether it had previously been inverted before.

#### Adjust default minor tick spacing

Default minor tick spacing was changed from 0.625 to 0.5 for major ticks spaced 2.5 units apart.

#### `EngFormatter` now accepts *usetex*, *useMathText* as keyword only arguments

A public API has been added to *EngFormatter* to control how the numbers in the ticklabels will be rendered. By default, *useMathText* evaluates to *rcParams["axes.formatter.use_mathtext"]* and *usetex* evaluates to *rcParams["'text.usetex'"]*.

If either is `True` then the numbers will be encapsulated by `$` signs. When using `TeX` this implies that the numbers will be shown in TeX's math font. When using mathtext, the `$` signs around numbers will ensure Unicode rendering (as implied by mathtext). This will make sure that the minus signs in the ticks are rendered as the Unicode minus (U+2212) when using mathtext (without relying on the *fix_minus* method).

### Animation and Interactivity

#### Support for forward/backward mouse buttons

Figure managers now support a `button_press` event for mouse buttons, similar to the `key_press` events. This allows binding actions to mouse buttons (see *MouseButton*) The first application of this mechanism is support of forward/backward mouse buttons in figures created with the Qt5 backend.

### *progress_callback* argument to `save()`

The method *Animation.save* gained an optional *progress_callback* argument to notify the saving progress.

### Add `cache_frame_data` keyword-only argument into `animation.FuncAnimation`

*matplotlib.animation.FuncAnimation* has been caching frame data by default; however, this caching is not ideal in certain cases e.g. When *FuncAnimation* needs to be only drawn(not saved) interactively and memory required by frame data is quite large. By adding *cache_frame_data* keyword-only argument, users can now disable this caching; thereby, this new argument provides a fix for issue #8528.

### Endless Looping GIFs with PillowWriter

We acknowledge that most people want to watch a GIF more than once. Saving an animation as a GIF with PillowWriter now produces an endless looping GIF.

### Adjusted `matplotlib.widgets.Slider` to have vertical orientation

The *matplotlib.widgets.Slider* widget now takes an optional argument *orientation* which indicates the direction (`'horizontal'` or `'vertical'`) that the slider should take.

### Improved formatting of image values under cursor when a colorbar is present

When a colorbar is present, its formatter is now used to format the image values under the mouse cursor in the status bar. For example, for an image displaying the values 10,000 and 10,001, the statusbar will now (using default settings) display the values as `10000` and `10001`), whereas both values were previously displayed as `1e+04`.

### MouseEvent button attribute is now an IntEnum

The `button` attribute of *MouseEvent* instances can take the values None, 1 (left button), 2 (middle button), 3 (right button), "up" (scroll), and "down" (scroll). For better legibility, the 1, 2, and 3 values are now represented using the `enum.IntEnum` class *matplotlib.backend_bases.MouseButton*, with the values *MouseButton.LEFT* (`== 1`), *MouseButton.MIDDLE* (`== 2`), and *MouseButton.RIGHT* (`== 3`).

### Configuration, Install, and Development

#### The MATPLOTLIBRC environment variable can now point to any "file" path

This includes device files; in particular, on Unix systems, one can set `MATPLOTLIBRC` to `/dev/null` to ignore the user's matplotlibrc file and fall back to Matplotlib's defaults.

As a reminder, if `MATPLOTLIBRC` points to a directory, Matplotlib will try to load the matplotlibrc file from `$MATPLOTLIBRC/matplotlibrc`.

#### Allow LaTeX code `pgf.preamble` and `text.latex.preamble` in MATPLOTLIBRC file

Previously, the rc file keys *rcParams["pgf.preamble"]* (default: `''`) and *rcParams["text.latex.preamble"]* (default: `''`) were parsed using commas as separators. This would break valid LaTeX code, such as:

```
\usepackage[protrusion=true, expansion=false]{microtype}
```

The parsing has been modified to pass the complete line to the LaTeX system, keeping all commas. Passing a list of strings from within a Python script still works as it used to.

#### New logging API

*matplotlib.set_loglevel* / *pyplot.set_loglevel* can be called to display more (or less) detailed logging output.

## 9.9.2 API Changes for 3.1.1


- *Behavior changes*


### Behavior changes

#### Locator.nonsingular return order

*Locator.nonsingular* (introduced in mpl 3.1) now returns a range `v0, v1` with `v0 <= v1`. This behavior is consistent with the implementation of `nonsingular` by the *LogLocator* and *LogitLocator* subclasses.

### 9.9.3 API Changes for 3.1.0


- *Behavior changes*
- `pgi` *support dropped*
- *rcParam changes*
- *Exception changes*
- *Removals*
- `matplotlib.mlab` *removals*
- `pylab` *removals*
- *mplot3d changes*
- *Testing*
- *Dependency changes*
- *Mathtext changes*
- *Signature deprecations*
- *Changes in parameter names*
- *Class/method/attribute deprecations*
- *Undeprecations*
- *New features*
- *Invalid inputs*


#### Behavior changes

#### Matplotlib.use

Switching backends via `matplotlib.use` is now allowed by default, regardless of whether `matplotlib.pyplot` has been imported. If the user tries to switch from an already-started interactive backend to a different interactive backend, an `ImportError` will be raised.

### Invalid points in PathCollections

PathCollections created with *scatter* now keep track of invalid points. Previously, points with nonfinite (infinite or nan) coordinates would not be included in the offsets (as returned by *PathCollection.get_offsets*) of a *PathCollection* created by *scatter*, and points with nonfinite values (as specified by the *c* kwarg) would not be included in the array (as returned by *PathCollection.get_array*)

Such points are now included, but masked out by returning a masked array.

If the *plotnonfinite* kwarg to *scatter* is set, then points with nonfinite values are plotted using the bad color of the *collections.PathCollection*'s colormap (as set by *colors.Colormap.set_bad()*).

### Alpha blending in imshow of RBGA input

The alpha-channel of RBGA images is now re-sampled independently of RGB channels. While this is a bug fix, it does change the output and may result in some down-stream image comparison tests to fail.

### Autoscaling

On log-axes where a single value is plotted at a "full" decade (1, 10, 100, etc.), the autoscaling now expands the axis symmetrically around that point, instead of adding a decade only to the right.

### Log-scaled axes

When the default *LogLocator* would generate no ticks for an axis (e.g., an axis with limits from 0.31 to 0.39) or only a single tick, it now instead falls back on the linear *AutoLocator* to pick reasonable tick positions.

### `Figure.add_subplot` with no arguments

Calling *Figure.add_subplot()* with no positional arguments used to do nothing; this now is equivalent to calling `add_subplot(111)` instead.

### `bxp` and rcparams

*bxp* now respects *rcParams["boxplot.boxprops.linewidth"]* (default: `1.0`) even when *patch_artist* is set. Previously, when the *patch_artist* parameter was set, *bxp* would ignore *rcParams["boxplot.boxprops.linewidth"]* (default: `1.0`). This was an oversight -- in particular, *boxplot* did not ignore it.

### Major/minor tick collisions

Minor ticks that collide with major ticks are now hidden by default. Previously, certain locator classes (*LogLocator*, *AutoMinorLocator*) contained custom logic to avoid emitting tick locations that collided with major ticks when they were used as minor locators. This logic has now moved to the *Axis* class, and is used regardless of the locator class. You can control this behavior via the *remove_overlapping_locs* attribute on *Axis*.

If you were relying on both the major and minor tick labels to appear on the same tick, you may need to update your code. For example, the following snippet

```
import numpy as np
import matplotlib.dates as mdates
import matplotlib.pyplot as plt

t = np.arange("2018-11-03", "2018-11-06", dtype="datetime64")
x = np.random.rand(len(t))

fig, ax = plt.subplots()
ax.plot(t, x)
ax.xaxis.set(
    major_locator=mdates.DayLocator(),
    major_formatter=mdates.DateFormatter("\n%a"),
    minor_locator=mdates.HourLocator((0, 6, 12, 18)),
    minor_formatter=mdates.DateFormatter("%H:%M"),
)
# disable removing overlapping locations
ax.xaxis.remove_overlapping_locs = False
plt.show()
```

labeled days using major ticks, and hours and minutes using minor ticks and added a newline to the major ticks labels to avoid them crashing into the minor tick labels. Setting the *remove_overlapping_locs* property (also accessible via *set_remove_overlapping_locs* / *get_remove_overlapping_locs* and *setp*) disables removing overlapping tick locations.

The major tick labels could also be adjusted include hours and minutes, as the minor ticks are gone, so the `major_formatter` would be:

```
mdates.DateFormatter("%H:%M\n%a")
```

### usetex support

Previously, if *rcParams["text.usetex"]* (default: `False`) was True, then constructing a *TextPath* on a non-mathtext string with `usetex=False` would rely on the mathtext parser (but not on usetex support!) to parse the string. The mathtext parser is not invoked anymore, which may cause slight changes in glyph positioning.

### get_window_extents

*matplotlib.axes.Axes.get_window_extent* used to return a bounding box that was slightly larger than the axes, presumably to take into account the ticks that may be on a spine. However, it was not scaling the tick sizes according to the dpi of the canvas, and it did not check if the ticks were visible, or on the spine.

Now *matplotlib.axes.Axes.get_window_extent* just returns the axes extent with no padding for ticks.

This affects *matplotlib.axes.Axes.get_tightbbox* in cases where there are outward ticks with no tick labels, and it also removes the (small) pad around axes in that case.

*spines.Spine.get_window_extent* now takes into account ticks that are on the spine.

### Sankey

Previously, *Sankey.add* would only accept a single string as the *labels* argument if its length is equal to the number of flows, in which case it would use one character of the string for each flow.

The behavior has been changed to match the documented one: when a single string is passed, it is used to label all the flows.

#### `FontManager` scores

*font_manager.FontManager.score_weight* is now more strict with its inputs. Previously, when a weight string was passed to *font_manager.FontManager.score_weight*,

- if the weight was the string representation of an integer, it would be converted to that integer,
- otherwise, if the weight was not a standard weight name, it would be silently replaced by a value of 500 ("normal" weight).

*font_manager.FontManager.score_weight* now raises an exception on such inputs.

### Text alignment

Text alignment was previously incorrect, in particular for multiline text objects with large descenders (i.e. subscripts) and rotated text. These have been fixed and made more consistent, but could make old code that has compensated for this no longer have the correct alignment.

### Upper case color strings

Support for passing single-letter colors (one of "rgbcmykw") as UPPERCASE characters is deprecated; these colors will become case-sensitive (lowercase) after the deprecation period has passed.

The goal is to decrease the number of ambiguous cases when using the `data` keyword to plotting methods; e.g. `plot("X", "Y", data={"X": ..., "Y": ...})` will not warn about "Y" possibly being a color anymore after the deprecation period has passed.

### Degenerate limits

When bounds passed to *set_xlim* are degenerate (i.e. the lower and upper value are equal), the method used to "expand" the bounds now matches the expansion behavior of autoscaling when the plot contains a single x-value, and should in particular produce nicer limits for non-linear scales.

### `plot` format string parsing

In certain cases, *plot* would previously accept format strings specifying more than one linestyle (e.g. `"---."` which specifies both `"--"` and `"-."`); only use one of them would be used. This now raises a `ValueError` instead.

### HTMLWriter

The HTMLWriter constructor is more strict: it no longer normalizes unknown values of *default_mode* to 'loop', but errors out instead.

### AFM parsing

In accordance with the AFM spec, the AFM parser no longer truncates the `UnderlinePosition` and `UnderlineThickness` fields to integers.

The `Notice` field (which can only be publicly accessed by the deprecated `afm.parse_afm` API) is no longer decoded to a `str`, but instead kept as `bytes`, to support non-conformant AFM files that use non-ASCII characters in that field.

### `Artist.set` keyword normalisation

*Artist.set* now normalizes keywords before sorting them. Previously it sorted its keyword arguments in reverse alphabetical order (with a special-case to put `color` at the end) before applying them.

It now normalizes aliases (and, as above, emits a warning on duplicate properties) before doing the sorting (so `c` goes to the end too).

### `Axes.tick_params` argument checking

Previously *`Axes.tick_params`* silently did nothing when an invalid *axis* parameter was supplied. This behavior has been changed to raise a `ValueError` instead.

### `Axes.hist` output

Input that consists of multiple empty lists will now return a list of histogram values for each one of the lists. For example, an input of `[[],[]]` will return 2 lists of histogram values. Previously, a single list was returned.

### `backend_bases.TimerBase.remove_callback` future signature change

Currently, `backend_bases.TimerBase.remove_callback(func, *args, **kwargs)` removes a callback previously added by `backend_bases.Timer.add_callback(func, *args, **kwargs)`, but if `*args, **kwargs` is not passed in (i.e., `TimerBase.remove_callback(func)`), then the first callback with a matching `func` is removed, regardless of whether it was added with or without `*args, **kwargs`.

In a future version, *`TimerBase.remove_callback`* will always use the latter behavior (not consider `*args, **kwargs`); to specifically consider them, add the callback as a `functools.partial` object

```
cb = timer.add_callback(functools.partial(func, *args, **kwargs))
# ...
# later
timer.remove_callback(cb)
```

*`TimerBase.add_callback`* was modified to return *func* to simplify the above usage (previously it returned None); this also allows using it as a decorator.

The new API is modelled after `atexit.register` / `atexit.unregister`.

### `StemContainer` performance increase

*`StemContainer`* objects can now store a *`LineCollection`* object instead of a list of *`Line2D`* objects for stem lines plotted using *`stem`*. This gives a very large performance boost to displaying and moving *`stem`* plots.

This will become the default behaviour in Matplotlib 3.3. To use it now, the *use_line_collection* keyword argument to *`stem`* can be set to `True`

```
ax.stem(..., use_line_collection=True)
```

Individual line segments can be extracted from the *`LineCollection`* using *`get_segments()`*. See the *`LineCollection`* documentation for other methods to retrieve the collection properties.

### `ColorbarBase` inheritance

*matplotlib.colorbar.ColorbarBase* is no longer a subclass of *cm.ScalarMappable*. This inheritance lead to a confusing situation where the *cm.ScalarMappable* passed to *matplotlib.colorbar.Colorbar* (*colorbar*) had a `set_norm` method, as did the colorbar. The colorbar is now purely a follower to the *ScalarMappable* norm and colormap, and the old inherited methods `matplotlib.colorbar.ColorbarBase.set_norm`, `matplotlib.colorbar.ColorbarBase.set_cmap`, `matplotlib.colorbar.ColorbarBase.set_clim` are deprecated, as are the getter versions of those calls. To set the norm associated with a colorbar do `colorbar.mappable.set_norm()` etc.

### FreeType and libpng search paths

The `MPLBASEDIRLIST` environment variables and `basedirlist` entry in `setup.cfg` have no effect anymore. Instead, if building in situations where FreeType or libpng are not in the compiler or linker's default path, set the standard environment variables `CFLAGS`/`LDFLAGS` on Linux or OSX, or `CL`/`LINK` on Windows, to indicate the relevant paths.

See details in *Installation*.

### Setting artist properties twice or more in the same call

Setting the same artist property multiple time via aliases is deprecated. Previously, code such as

```
plt.plot([0, 1], c="red", color="blue")
```

would emit a warning indicating that `c` and `color` are aliases of one another, and only keep the `color` kwarg. This behavior has been deprecated; in a future version, this will raise a TypeError, similar to Python's behavior when a keyword argument is passed twice

```
plt.plot([0, 1], c="red", c="blue")
```

This warning is raised by *normalize_kwargs*.

### Path code types

Path code types like `Path.MOVETO` are now `np.uint8` instead of `int` `Path.STOP`, `Path.MOVETO`, `Path.LINETO`, `Path.CURVE3`, `Path.CURVE4` and `Path.CLOSEPOLY` are now of the type `Path.code_type` (`np.uint8` by default) instead of plain `int`. This makes their type match the array value type of the `Path.codes` array.

### LaTeX code in matplotlibrc file

Previously, the rc file keys `pgf.preamble` and `text.latex.preamble` were parsed using commas as separators. This would break valid LaTeX code, such as:

```
\usepackage[protrusion=true, expansion=false]{microtype}
```

The parsing has been modified to pass the complete line to the LaTeX system, keeping all commas. Passing a list of strings from within a Python script still works as it used to. Passing a list containing non-strings now fails, instead of coercing the results to strings.

#### `Axes.spy`

The method *`Axes.spy`* now raises a `TypeError` for the keyword arguments *interpolation* and *linestyle* instead of silently ignoring them.

Furthermore, *`Axes.spy`* spy does now allow for an *extent* argument (was silently ignored so far).

A bug with `Axes.spy(..., origin='lower')` is fixed. Previously this flipped the data but not the y-axis resulting in a mismatch between axes labels and actual data indices. Now, *origin='lower'* flips both the data and the y-axis labels.

### Boxplot tick methods

The *manage_xticks* parameter of *`boxplot`* and *`bxp`* has been renamed (with a deprecation period) to *manage_ticks*, to take into account the fact that it manages either x or y ticks depending on the *vert* parameter.

When `manage_ticks=True` (the default), these methods now attempt to take previously drawn boxplots into account when setting the axis limits, ticks, and tick labels.

### MouseEvents

MouseEvents now include the event name in their `str()`. Previously they contained the prefix "MPL MouseEvent".

### RGBA buffer return type

*`FigureCanvasAgg.buffer_rgba`* and *`RendererAgg.buffer_rgba`* now return a memoryview The `buffer_rgba` method now allows direct access to the renderer's underlying buffer (as a `(m, n, 4)`-shape memoryview) rather than copying the data to a new bytestring. This is consistent with the behavior on Py2, where a buffer object was returned.

#### `matplotlib.font_manager.win32InstalledFonts` return type

`matplotlib.font_manager.win32InstalledFonts` returns an empty list instead of None if no fonts are found.

#### `Axes.fmt_xdata` and `Axes.fmt_ydata` error handling

Previously, if the user provided a `Axes.fmt_xdata` or `Axes.fmt_ydata` function that raised a `TypeError` (or set them to a non-callable), the exception would be silently ignored and the default formatter be used instead. This is no longer the case; the exception is now propagated out.

### Deprecation of redundant `Tick` attributes

The `gridOn`, `tick1On`, `tick2On`, `label1On`, and `label2On` *Tick* attributes have been deprecated. Directly get and set the visibility on the underlying artists, available as the `gridline`, `tick1line`, `tick2line`, `label1`, and `label2` attributes.

The `label` attribute, which was an alias for `label1`, has been deprecated.

Subclasses that relied on setting the above visibility attributes needs to be updated; see e.g. `examples/api/skewt.py`.

### Passing a Line2D's drawstyle together with the linestyle is deprecated

Instead of `plt.plot(..., linestyle="steps--")`, use `plt.plot(..., linestyle="--", drawstyle="steps")`. `ds` is now an alias for `drawstyle`.

### `pgi` support dropped

Support for `pgi` in the GTK3 backends has been dropped. `pgi` is an alternative implementation to `PyGObject`. `PyGObject` should be used instead.

### rcParam changes

### Removed

The following deprecated rcParams have been removed:

- `text.dvipnghack`
- `nbagg.transparent` (use *rcParams["figure.facecolor"]* (default: `'white'`) instead)
- `plugins.directory`
- `axes.hold`

- `backend.qt4` and `backend.qt5` (set the *`QT_API`* environment variable instead)

### Deprecated

The associated validator functions `rcsetup.validate_qt4` and `validate_qt5` are deprecated.

The `verbose.fileo` and `verbose.level` rcParams have been deprecated. These have had no effect since the switch from Matplotlib's old custom Verbose logging to the stdlib's `logging` module. In addition the `rcsetup.validate_verbose` function is deprecated.

The `text.latex.unicode` rcParam now defaults to `True` and is deprecated (i.e., in future versions of Matplotlib, unicode input will always be supported). Moreover, the underlying implementation now uses `\usepackage[utf8]{inputenc}` instead of `\usepackage{ucs}\usepackage[utf8x]{inputenc}`.

### Exception changes

- `mpl_toolkits.axes_grid1.axes_size.GetExtentHelper` now raises `ValueError` for invalid directions instead of `KeyError`.
- Previously, subprocess failures in the animation framework would raise either in a `RuntimeError` or a `ValueError` depending on when the error occurred. They now raise a `subprocess.CalledProcessError` with attributes set as documented by the exception class.
- In certain cases, Axes methods (and pyplot functions) used to raise a `RuntimeError` if they were called with a `data` kwarg and otherwise mismatched arguments. They now raise a `TypeError` instead.
- *`Axes.streamplot`* does not support irregularly gridded `x` and `y` values. So far, it used to silently plot an incorrect result. This has been changed to raise a `ValueError` instead.
- The `streamplot.Grid` class, which is internally used by streamplot code, also throws a `ValueError` when irregularly gridded values are passed in.

### Removals

The following deprecated APIs have been removed:

### Classes and methods

- `Verbose` (replaced by python logging library)
- `artist.Artist.hitlist` (no replacement)
- `artist.Artist.is_figure_set` (use `artist.figure is not None` instead)
- `axis.Axis.unit_data` (use `axis.Axis.units` instead)

- `backend_bases.FigureCanvasBase.onRemove` (no replacement) `backend_bases.FigureManagerBase.show_popup` (this never did anything)
- `backend_wx.SubplotToolWx` (no replacement)
- `backend_wx.Toolbar` (use `backend_wx.NavigationToolbar2Wx` instead)
- `cbook.align_iterators` (no replacement)
- `contour.ContourLabeler.get_real_label_width` (no replacement)
- `legend.Legend.draggable` (use *legend.Legend.set_draggable()* instead)
- `texmanager.TexManager.postscriptd`, `texmanager.TexManager.pscnt`, `texmanager.TexManager.make_ps`, `texmanager.TexManager.get_ps_bbox` (no replacements)

### Arguments

- The *fig* kwarg to *GridSpec.get_subplot_params* and *GridSpecFromSubplotSpec.get_subplot_params* (use the argument *figure* instead)
- Passing 'box-forced' to *Axes.set_adjustable* (use 'box' instead)
- Support for the strings 'on'/'true'/'off'/'false' to mean `True` / `False` (directly use `True` / `False` instead). The following functions are affected:
  - *axes.Axes.grid*
  - *Axes3D.grid*
  - *Axis.set_tick_params*
  - *pyplot.box*
- Using *pyplot.axes* with an *axes.Axes* type argument (use *pyplot.sca* instead)

### Other

The following miscellaneous API elements have been removed

- svgfont support (in *rcParams["svg.fonttype"]* (default: `'path'`))
- Logging is now done with the standard python `logging` library. `matplotlib.verbose` and the command line switches `--verbose-LEVEL` have been removed.

  To control the logging output use:

```
import logging
logger = logging.getLogger('matplotlib')
logger.setLevel(logging.INFO)
# configure log handling: Either include it into your ``logging``␣
 ↪hierarchy,
# e.g. by configuring a root looger using ``logging.basicConfig()``,
```

(continues on next page)

(continued from previous page)

```
# or add a standalone handler to the matplotlib logger:
logger.addHandler(logging.StreamHandler())
```

- `__version__numpy__`
- `collections.CIRCLE_AREA_FACTOR`
- `font_manager.USE_FONTCONFIG`
- `font_manager.cachedir`

### `matplotlib.mlab` removals

Lots of code inside the *`matplotlib.mlab`* module which was deprecated in Matplotlib 2.2 has been removed. See below for a list:

- `mlab.exp_safe` (use `numpy.exp` instead)
- `mlab.amap`
- `mlab.logspace` (use `numpy.logspace` instead)
- `mlab.rms_flat`
- `mlab.l1norm` (use `numpy.linalg.norm(a, ord=1)` instead)
- `mlab.l2norm` (use `numpy.linalg.norm(a, ord=2)` instead)
- `mlab.norm_flat` (use `numpy.linalg.norm(a.flat, ord=2)` instead)
- `mlab.frange` (use `numpy.arange` instead)
- `mlab.identity` (use `numpy.identity` instead)
- `mlab.base_repr`
- `mlab.binary_repr`
- `mlab.ispower2`
- `mlab.log2` (use `numpy.log2` instead)
- `mlab.isvector`
- `mlab.movavg`
- `mlab.safe_isinf` (use `numpy.isinf` instead)
- `mlab.safe_isnan` (use `numpy.isnan` instead)
- `mlab.cohere_pairs` (use `scipy.signal.coherence` instead)
- `mlab.entropy` (use `scipy.stats.entropy` instead)
- `mlab.normpdf` (use `scipy.stats.norm.pdf` instead)
- `mlab.find` (use `np.nonzero(np.ravel(condition))` instead)

- mlab.longest_contiguous_ones
- mlab.longest_ones
- mlab.PCA
- mlab.prctile (use numpy.percentile instead)
- mlab.prctile_rank
- mlab.center_matrix
- mlab.rk4 (use scipy.integrate.ode instead)
- mlab.bivariate_normal
- mlab.get_xyz_where
- mlab.get_sparse_matrix
- mlab.dist (use numpy.hypot instead)
- mlab.dist_point_to_segment
- mlab.griddata (use scipy.interpolate.griddata)
- mlab.less_simple_linear_interpolation (use numpy.interp)
- mlab.slopes
- mlab.stineman_interp
- mlab.segments_intersect
- mlab.fftsurr
- mlab.offset_line
- mlab.quad2cubic
- mlab.vector_lengths
- mlab.distances_along_curve
- mlab.path_length
- mlab.cross_from_above
- mlab.cross_from_below
- mlab.contiguous_regions (use *cbook.contiguous_regions* instead)
- mlab.is_closed_polygon
- mlab.poly_between
- mlab.poly_below
- mlab.inside_poly
- mlab.csv2rec
- mlab.rec2csv (use numpy.recarray.tofile instead)

- `mlab.rec2text` (use `numpy.recarray.tofile` instead)
- `mlab.rec_summarize`
- `mlab.rec_join`
- `mlab.recs_join`
- `mlab.rec_groupby`
- `mlab.rec_keep_fields`
- `mlab.rec_drop_fields`
- `mlab.rec_append_fields`
- `mlab.csvformat_factory`
- `mlab.get_formatd`
- `mlab.FormatDatetime` (use `datetime.datetime.strftime` instead)
- `mlab.FormatDate` (use `datetime.date.strftime` instead)
- `mlab.FormatMillions`, `mlab.FormatThousands`, `mlab.FormatPercent`, `mlab.FormatBool`, `mlab.FormatInt`, `mlab.FormatFloat`, `mlab.FormatFormatStr`, `mlab.FormatString`, `mlab.FormatObj`
- `mlab.donothing_callback`

### `pylab` removals

Lots of code inside the *`matplotlib.mlab`* module which was deprecated in Matplotlib 2.2 has been removed. This means the following functions are no longer available in the *`pylab`* module:

- `amap`
- `base_repr`
- `binary_repr`
- `bivariate_normal`
- `center_matrix`
- `csv2rec` (use `numpy.recarray.tofile` instead)
- `dist` (use `numpy.hypot` instead)
- `dist_point_to_segment`
- `distances_along_curve`
- `entropy` (use `scipy.stats.entropy` instead)
- `exp_safe` (use `numpy.exp` instead)
- `fftsurr`
- `find` (use `np.nonzero(np.ravel(condition))` instead)

- `frange` (use `numpy.arange` instead)
- `get_sparse_matrix`
- `get_xyz_where`
- `griddata` (use `scipy.interpolate.griddata` instead)
- `identity` (use `numpy.identity` instead)
- `inside_poly`
- `is_closed_polygon`
- `ispower2`
- `isvector`
- `l1norm` (use `numpy.linalg.norm(a, ord=1)` instead)
- `l2norm` (use `numpy.linalg.norm(a, ord=2)` instead)
- `log2` (use `numpy.log2` instead)
- `longest_contiguous_ones`
- `longest_ones`
- `movavg`
- `norm_flat` (use `numpy.linalg.norm(a.flat, ord=2)` instead)
- `normpdf` (use `scipy.stats.norm.pdf` instead)
- `path_length`
- `poly_below`
- `poly_between`
- `prctile` (use `numpy.percentile` instead)
- `prctile_rank`
- `rec2csv` (use `numpy.recarray.tofile` instead)
- `rec_append_fields`
- `rec_drop_fields`
- `rec_join`
- `rk4` (use `scipy.integrate.ode` instead)
- `rms_flat`
- `segments_intersect`
- `slopes`
- `stineman_interp`
- `vector_lengths`

### mplot3d changes

### Voxel shading

*Axes3D.voxels* now shades the resulting voxels; for more details see What's new. The previous behavior can be achieved by passing

```
ax.voxels(.., shade=False)
```

### Equal aspect axes disabled

Setting the aspect on 3D axes previously returned non-sensical results (e.g. see #1077). Calling `ax.set_aspect('equal')` or `ax.set_aspect(num)` on a 3D axes now raises a `NotImplementedError`.

#### Poly3DCollection.set_zsort

*Poly3DCollection.set_zsort* no longer silently ignores invalid inputs, or `False` (which was always broken). Passing `True` to mean `"average"` is deprecated.

### Testing

The `--no-network` flag to `tests.py` has been removed (no test requires internet access anymore). If it is desired to disable internet access both for old and new versions of Matplotlib, use `tests.py -m 'not network'` (which is now a no-op).

The image comparison test decorators now skip (rather than xfail) the test for uncomparable formats. The affected decorators are *image_comparison* and *check_figures_equal*. The deprecated `ImageComparisonTest` class is likewise changed.

### Dependency changes

### NumPy

Matplotlib 3.1 now requires NumPy>=1.11.

### ghostscript

Support for ghostscript 8.60 (released in 2007) has been removed. The oldest supported version of ghostscript is now 9.0 (released in 2010).

### Mathtext changes

- In constructs such as `"$1~2$"`, mathtext now interprets the tilde as a space, consistently with TeX (this was previously a parse error).

### Deprecations

- The `\stackrel` mathtext command has been deprecated (it behaved differently from LaTeX's `\stackrel`. To stack two mathtext expressions, use `\genfrac{left-delim}{right-delim}{fraction-bar-thickness}{}{top}{bottom}`.
- The `\mathcircled` mathtext command (which is not a real TeX command) is deprecated. Directly use unicode characters (e.g. `"\N{CIRCLED LATIN CAPITAL LETTER A}"` or `"\u24b6"`) instead.
- Support for setting *rcParams["mathtext.default"]* (default: `'it'`) to circled is deprecated.

### Signature deprecations

The following signature related behaviours are deprecated:

- The *withdash* keyword argument to *Axes.text()*. Consider using *Axes.annotate()* instead.
- Passing (n, 1)-shaped error arrays to *Axes.errorbar()*, which was not documented and did not work for `n = 2`. Pass a 1D array instead.
- The *frameon* kwarg to *savefig* and the *rcParams["savefig.frameon"]* rcParam. To emulate `frameon = False`, set *facecolor* to fully transparent (`"none"`, or `(0, 0, 0, 0)`).
- Passing a non-1D (typically, (n, 1)-shaped) input to *Axes.pie*. Pass a 1D array instead.
- The *TextPath* constructor used to silently drop ignored arguments; this behavior is deprecated.
- The *usetex* parameter of *TextToPath.get_text_path* is deprecated and folded into the *ismath* parameter, which can now take the values `False`, `True`, and `"TeX"`, consistently with other low-level text processing functions.
- Passing `'normal'` to *axes.Axes.axis()* is deprecated, use `ax.axis('auto')` instead.
- Passing the *block* argument of *pyplot.show* positionally is deprecated; it should be passed by keyword.
- When using the nbagg backend, *pyplot.show* used to silently accept and ignore all combinations of positional and keyword arguments. This behavior is deprecated.

- The unused *shape* and *imlim* parameters to `Axes.imshow` are deprecated. To avoid triggering the deprecation warning, the *filternorm*, *filterrad*, *resample*, and *url* arguments should be passed by keyword.
- The *interp_at_native* parameter to `BboxImage`, which has had no effect since Matplotlib 2.0, is deprecated.
- All arguments to the `matplotlib.cbook.deprecation.deprecated` decorator and `matplotlib.cbook.deprecation.warn_deprecated` function, except the first one (the version where the deprecation occurred), are now keyword-only. The goal is to avoid accidentally setting the "message" argument when the "name" (or "alternative") argument was intended, as this has repeatedly occurred in the past.
- The arguments of `matplotlib.testing.compare.calculate_rms` have been renamed from `expectedImage, actualImage`, to `expected_image, actual_image`.
- Passing positional arguments to `Axis.set_ticklabels` beyond *ticklabels* itself has no effect, and support for them is deprecated.
- Passing `shade=None` to `plot_surface` is deprecated. This was an unintended implementation detail with the same semantics as `shade=False`. Please use the latter code instead.
- `matplotlib.ticker.MaxNLocator` and its *set_params* method will issue a warning on unknown keyword arguments instead of silently ignoring them. Future versions will raise an error.

### Changes in parameter names

- The *arg* parameter to `matplotlib.use` has been renamed to *backend*.

  This will only affect cases where that parameter has been set as a keyword argument. The common usage pattern as a positional argument `matplotlib.use('Qt5Agg')` is not affected.
- The *normed* parameter to `Axes.hist2d` has been renamed to *density*.
- The *s* parameter to `Annotation` (and indirectly `Axes.annotate`) has been renamed to *text*.
- The *tolerence* parameter to `bezier.find_bezier_t_intersecting_with_closedpath`, `bezier.split_bezier_intersecting_with_closedpath`, `bezier.find_r_to_boundary_of_closedpath`, `bezier.split_path_inout` and `bezier.check_if_parallel` has been renamed to *tolerance*.

In each case, the old parameter name remains supported (it cannot be used simultaneously with the new name), but support for it will be dropped in Matplotlib 3.3.

### Class/method/attribute deprecations

Support for custom backends that do not provide a *GraphicsContextBase.set_hatch_color* method is deprecated. We suggest that custom backends let their `GraphicsContext` class inherit from *GraphicsContextBase*, to at least provide stubs for all required methods.

- `spine.Spine.is_frame_like`

This has not been used in the codebase since its addition in 2009.

- `axis3d.Axis.get_tick_positions`

  This has never been used internally, there is no equivalent method exists on the 2D Axis classes, and despite the similar name, it has a completely different behavior from the 2D Axis' `axis.Axis.get_ticks_position` method.

- `.backend_pgf.LatexManagerFactory`
- `mpl_toolkits.axisartist.axislines.SimpleChainedObjects`
- `mpl_toolkits.Axes.AxisDict`

### Internal Helper Functions

- `checkdep_dvipng`
- `checkdep_ghostscript`
- `checkdep_pdftops`
- `checkdep_inkscape`
- `ticker.decade_up`
- `ticker.decade_down`
- `cbook.dedent`
- `docstring.Appender`
- `docstring.dedent`
- `docstring.copy_dedent`

Use the standard library's docstring manipulation tools instead, such as `inspect.cleandoc` and `inspect.getdoc`.

- `matplotlib.scale.get_scale_docs()`
- `matplotlib.pyplot.get_scale_docs()`

These are considered internal and will be removed from the public API in a future version.

- `projections.process_projection_requirements`
- `backend_ps.PsBackendHelper`
- `backend_ps.ps_backend_helper`,

- `cbook.iterable`
- `cbook.get_label`
- `cbook.safezip` Manually check the lengths of the inputs instead, or rely on NumPy to do it.
- `cbook.is_hashable` Use `isinstance(..., collections.abc.Hashable)` instead.
- The `.backend_bases.RendererBase.strip_math`. Use *cbook.strip_math* instead.

Multiple internal functions that were exposed as part of the public API of *mpl_toolkits.mplot3d* are deprecated,

**mpl_toolkits.mplot3d.art3d**

- `mpl_toolkits.mplot3d.art3d.norm_angle`
- `mpl_toolkits.mplot3d.art3d.norm_text_angle`
- `mpl_toolkits.mplot3d.art3d.path_to_3d_segment`
- `mpl_toolkits.mplot3d.art3d.paths_to_3d_segments`
- `mpl_toolkits.mplot3d.art3d.path_to_3d_segment_with_codes`
- `mpl_toolkits.mplot3d.art3d.paths_to_3d_segments_with_codes`
- `mpl_toolkits.mplot3d.art3d.get_patch_verts`
- `mpl_toolkits.mplot3d.art3d.get_colors`
- `mpl_toolkits.mplot3d.art3d.zalpha`

**mpl_toolkits.mplot3d.proj3d**

- `mpl_toolkits.mplot3d.proj3d.line2d`
- `mpl_toolkits.mplot3d.proj3d.line2d_dist`
- `mpl_toolkits.mplot3d.proj3d.line2d_seg_dist`
- `mpl_toolkits.mplot3d.proj3d.mod`
- `mpl_toolkits.mplot3d.proj3d.proj_transform_vec`
- `mpl_toolkits.mplot3d.proj3d.proj_transform_vec_clip`
- `mpl_toolkits.mplot3d.proj3d.vec_pad_ones`
- `mpl_toolkits.mplot3d.proj3d.proj_trans_clip_points`

If your project relies on these functions, consider vendoring them.

### Font Handling

- `backend_pdf.RendererPdf.afm_font_cache`
- `backend_ps.RendererPS.afmfontd`
- `font_manager.OSXInstalledFonts`
- `.TextToPath.glyph_to_path` (Instead call `font.get_path()` and manually transform the path.)

### Date related functions

- `dates.seconds()`
- `dates.minutes()`
- `dates.hours()`
- `dates.weeks()`
- `dates.strpdate2num`
- `dates.bytespdate2num`

These are brittle in the presence of locale changes. Use standard datetime parsers such as `time.strptime` or `dateutil.parser.parse`, and additionally call *`matplotlib.dates.date2num`* if you need to convert to Matplotlib's internal datetime representation; or use `dates.datestr2num`.

### Axes3D

- `.axes3d.Axes3D.w_xaxis`
- `.axes3d.Axes3D.w_yaxis`
- `.axes3d.Axes3D.w_zaxis`

Use `axes3d.Axes3D.xaxis`, `axes3d.Axes3D.yaxis` and `axes3d.Axes3D.zaxis` instead.

### Testing

- `matplotlib.testing.decorators.switch_backend` decorator

Test functions should use `pytest.mark.backend`, and the mark will be picked up by the `matplotlib.testing.conftest.mpl_test_settings` fixture.

### Quiver

- `.color` attribute of *`Quiver`* objects

Instead, use (as for any *`Collection`*) the `get_facecolor` method. Note that setting to the `.color` attribute did not update the quiver artist, whereas calling `set_facecolor` does.

### GUI / backend details

- `.get_py2exe_datafiles`
- `.tk_window_focus`
- `.backend_gtk3.FileChooserDialog`
- `.backend_gtk3.NavigationToolbar2GTK3.get_filechooser`
- `.backend_gtk3.SaveFigureGTK3.get_filechooser`
- `.NavigationToolbar2QT.adj_window` attribute. This is unused and always `None`.
- `.backend_wx.IDLE_DELAY` global variable This is unused and only relevant to the now removed wx "idling" code (note that as it is a module-level global, no deprecation warning is emitted when accessing it).
- `mlab.demean`
- `backend_gtk3cairo.FigureCanvasGTK3Cairo`,
- `backend_wx.debug_on_error`, `backend_wx.fake_stderr`, `backend_wx.raise_msg_to_str`, `backend_wx.MenuButtonWx`, `backend_wx.PrintoutWx`,
- `matplotlib.backends.qt_editor.formlayout` module

This module is a vendored, modified version of the official formlayout module available on PyPI. Install that module separately if you need it.

- `GraphicsContextPS.shouldstroke`

### Transforms / scales

- `LogTransformBase`
- `Log10Transform`
- `Log2Transform`,
- `NaturalLogTransformLog`
- `InvertedLogTransformBase`
- `InvertedLog10Transform`
- `InvertedLog2Transform`

- `InvertedNaturalLogTransform`

These classes defined in *matplotlib.scale* are deprecated. As a replacement, use the general *LogTransform* and *InvertedLogTransform* classes, whose constructors take a *base* argument.

### Locators / Formatters

- `OldScalarFormatter.pprint_val`
- `ScalarFormatter.pprint_val`
- `LogFormatter.pprint_val`

These are helper methods that do not have a consistent signature across formatter classes.

### Path tools

- `path.get_paths_extents`

Use *get_path_collection_extents* instead.

- `.Path.has_nonfinite` attribute

Use `not np.isfinite(path.vertices).all()` instead.

- `.bezier.find_r_to_boundary_of_closedpath` function is deprecated

This has always returned None instead of the requested radius.

### Text

- `text.TextWithDash`
- `Text.is_math_text`
- `TextPath.is_math_text`
- `TextPath.text_get_vertices_codes` (As an alternative, construct a new `TextPath` object.)

### Unused attributes

- `NavigationToolbar2QT.buttons`
- `Line2D.verticalOffset`
- `Quiver.keytext`
- `Quiver.keyvec`
- `SpanSelector.buttonDown`

These are unused and never updated.

### Sphinx extensions

- `matplotlib.sphinxext.mathmpl.math_directive`
- `matplotlib.sphinxext.plot_directive.plot_directive`

This is because the `matplotlib.sphinxext.mathmpl` and `matplotlib.sphinxext.plot_directive` interfaces have changed from the (Sphinx-)deprecated function-based interface to a class-based interface; this should not affect end users.

- `mpl_toolkits.axisartist.axis_artist.UnimplementedException`

### Environmental Variables

- The `MATPLOTLIBDATA` environment variable

### Axis

- `Axis.iter_ticks`

This only served as a helper to the private `Axis._update_ticks`

### Undeprecations

The following API elements have been un-deprecated:

- The *obj_type* keyword argument to the `matplotlib.cbook.deprecation.deprecated` decorator.
- *xmin*, *xmax* keyword arguments to *Axes.set_xlim* and *ymin*, *ymax* keyword arguments to *Axes.set_ylim*

### New features

#### `Text` now has a `c` alias for the `color` property

For consistency with *Line2D*, the *Text* class has gained the `c` alias for the `color` property. For example, one can now write

```
ax.text(.5, .5, "foo", c="red")
```

### `Cn` colors now support `n>=10`

It is now possible to go beyond the tenth color in the property cycle using `Cn` syntax, e.g.

```
plt.plot([1, 2], color="C11")
```

now uses the 12th color in the cycle.

Note that previously, a construct such as:

```
plt.plot([1, 2], "C11")
```

would be interpreted as a request to use color `C1` and marker `1` (an "inverted Y"). To obtain such a plot, one should now use

```
plt.plot([1, 2], "1C1")
```

(so that the first "1" gets correctly interpreted as a marker specification), or, more explicitly:

```
plt.plot([1, 2], marker="1", color="C1")
```

### New `Formatter.format_ticks` method

The *`Formatter`* class gained a new *`format_ticks`* method, which takes the list of all tick locations as a single argument and returns the list of all formatted values. It is called by the axis tick handling code and, by default, first calls *`set_locs`* with all locations, then repeatedly calls `Formatter.__call__` for each location.

Tick-handling code in the codebase that previously performed this sequence (*`set_locs`* followed by repeated `Formatter.__call__`) have been updated to use *`format_ticks`*.

*`format_ticks`* is intended to be overridden by *`Formatter`* subclasses for which the formatting of a tick value depends on other tick values, such as *`ConciseDateFormatter`*.

### Added support for RGB(A) images in pcolorfast

pcolorfast now accepts 3D images (RGB or RGBA) arrays if the X and Y specifications allow image or pcolorimage rendering; they remain unsupported by the more general quadmesh rendering

#### Invalid inputs

Passing invalid locations to *legend* and *table* used to fallback on a default location. This behavior is deprecated and will throw an exception in a future version.

*offsetbox.AnchoredText* is unable to handle the *horizontalalignment* or *verticalalignment* kwargs, and used to ignore them with a warning. This behavior is deprecated and will throw an exception in a future version.

Passing steps less than 1 or greater than 10 to *MaxNLocator* used to result in undefined behavior. It now throws a `ValueError`.

The signature of the (private) `Axis._update_ticks` has been changed to not take the renderer as argument anymore (that argument is unused).

## 9.10 Version 3.0

### 9.10.1 What's new in Matplotlib 3.0 (Sep 18, 2018)

#### Improved default backend selection

The default backend no longer must be set as part of the build process. Instead, at run time, the builtin backends are tried in sequence until one of them imports.

Headless Linux servers (identified by the DISPLAY environment variable not being defined) will not select a GUI backend.

#### Cyclic colormaps

Two new colormaps named 'twilight' and 'twilight_shifted' have been added. These colormaps start and end on the same color, and have two symmetric halves with equal lightness, but diverging color. Since they wrap around, they are a good choice for cyclic data such as phase angles, compass directions, or time of day. Like *viridis* and *cividis*, *twilight* is perceptually uniform and colorblind friendly.

#### Ability to scale axis by a fixed order of magnitude

To scale an axis by a fixed order of magnitude, set the *scilimits* argument of *Axes.ticklabel_format* to the same (non-zero) lower and upper limits. Say to scale the y axis by a million (1e6), use

```
ax.ticklabel_format(style='sci', scilimits=(6, 6), axis='y')
```

The behavior of `scilimits=(0, 0)` is unchanged. With this setting, Matplotlib will adjust the order of magnitude depending on the axis values, rather than keeping it fixed. Previously, setting `scilimits=(m, m)` was equivalent to setting `scilimits=(0, 0)`.

### Add `AnchoredDirectionArrows` feature to mpl_toolkits

A new mpl_toolkits class *AnchoredDirectionArrows* draws a pair of orthogonal arrows to indicate directions on a 2D plot. A minimal working example takes in the transformation object for the coordinate system (typically ax.transAxes), and arrow labels. There are several optional parameters that can be used to alter layout. For example, the arrow pairs can be rotated and the color can be changed. By default the labels and arrows have the same color, but the class may also pass arguments for customizing arrow and text layout, these are passed to *matplotlib.textpath.TextPath* and *matplotlib.patches.FancyArrowPatch*. Location, length and width for both arrow tail and head can be adjusted, the direction arrows and labels can have a frame. Padding and separation parameters can be adjusted.

### Add `minorticks_on()/off()` methods for colorbar

A new method `ColorbarBase.minorticks_on` has been added to correctly display minor ticks on a colorbar. This method doesn't allow the minor ticks to extend into the regions beyond vmin and vmax when the *extend* keyword argument (used while creating the colorbar) is set to 'both', 'max' or 'min'. A complementary method `ColorbarBase.minorticks_off` has also been added to remove the minor ticks on the colorbar.

### Colorbar ticks can now be automatic

The number of ticks placed on colorbars was previously appropriate for a large colorbar, but looked bad if the colorbar was made smaller (i.e. via the *shrink* keyword argument). This has been changed so that the number of ticks is now responsive to how large the colorbar is.

### Don't automatically rename duplicate file names

Previously, when saving a figure to a file using the GUI's save dialog box, if the default filename (based on the figure window title) already existed on disk, Matplotlib would append a suffix (e.g. `Figure_1-1.png`), preventing the dialog from prompting to overwrite the file. This behaviour has been removed. Now if the file name exists on disk, the user is prompted whether or not to overwrite it. This eliminates guesswork, and allows intentional overwriting, especially when the figure name has been manually set using `figure.canvas.set_window_title()`.

### Legend now has a *title_fontsize* keyword argument (and rcParam)

The title for a *Figure.legend* and *Axes.legend* can now have its font size set via the *title_fontsize* keyword argument. There is also a new *rcParams["legend.title_fontsize"]* (default: `None`). Both default to `None`, which means the legend title will have the same font size as the axes default font size (*not* the legend font size, set by the *fontsize* keyword argument or *rcParams["legend.fontsize"]* (default: `'medium'`)).

### Support for axes.prop_cycle property *markevery* in rcParams

The Matplotlib `rcParams` settings object now supports configuration of the attribute *rcParams["axes.prop_cycle"]* (default: `cycler('color', ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728', '#9467bd', '#8c564b', '#e377c2', '#7f7f7f', '#bcbd22', '#17becf'])`) with cyclers using the *markevery* Line2D object property.

### Multi-page PDF support for pgf backend

The pgf backend now also supports multi-page PDF files.

```
from matplotlib.backends.backend_pgf import PdfPages
import matplotlib.pyplot as plt

with PdfPages('multipage.pdf') as pdf:
    # page 1
    plt.plot([2, 1, 3])
    pdf.savefig()

    # page 2
    plt.cla()
    plt.plot([3, 1, 2])
    pdf.savefig()
```

### Pie charts are now circular by default

We acknowledge that the majority of people do not like egg-shaped pies. Therefore, an axes to which a pie chart is plotted will be set to have equal aspect ratio by default. This ensures that the pie appears circular independent on the axes size or units. To revert to the previous behaviour set the axes' aspect ratio to automatic by using `ax.set_aspect("auto")` or `plt.axis("auto")`.

### Add `ax.get_gridspec` to `SubplotBase`

New method `SubplotBase.get_gridspec` is added so that users can easily get the gridspec that went into making an axes:

```
import matplotlib.pyplot as plt

fig, axs = plt.subplots(3, 2)
gs = axs[0, -1].get_gridspec()

# remove the last column
for ax in axs[:,-1].flatten():
  ax.remove()

# make a subplot in last column that spans rows.
ax = fig.add_subplot(gs[:, -1])
plt.show()
```

### Axes titles will no longer overlap xaxis

Previously an axes title had to be moved manually if an xaxis overlapped (usually when the xaxis was put on the top of the axes). Now, the title will be automatically moved above the xaxis and its decorators (including the xlabel) if they are at the top.

If desired, the title can still be placed manually. There is a slight kludge; the algorithm checks if the y-position of the title is 1.0 (the default), and moves if it is. If the user places the title in the default location (i.e. `ax.title.set_position(0.5, 1.0)`), the title will still be moved above the xaxis. If the user wants to avoid this, they can specify a number that is close (i.e. `ax.title.set_position(0.5, 1.01)`) and the title will not be moved via this algorithm.

### New convenience methods for GridSpec

There are new convenience methods for *gridspec.GridSpec* and *gridspec.GridSpecFromSubplotSpec*. Instead of the former we can now call *Figure.add_gridspec* and for the latter *SubplotSpec.subgridspec*.

```
import matplotlib.pyplot as plt

fig = plt.figure()
gs0 = fig.add_gridspec(3, 1)
ax1 = fig.add_subplot(gs0[0])
ax2 = fig.add_subplot(gs0[1])
gssub = gs0[2].subgridspec(1, 3)
for i in range(3):
    fig.add_subplot(gssub[0, i])
```

### Figure has an `add_artist` method

A method *add_artist* has been added to the *Figure* class, which allows artists to be added directly to a figure. E.g.

```
circ = plt.Circle((.7, .5), .05)
fig.add_artist(circ)
```

In case the added artist has no transform set previously, it will be set to the figure transform (`fig.transFigure`). This new method may be useful for adding artists to figures without axes or to easily position static elements in figure coordinates.

#### `:math:` directive renamed to `:mathmpl:`

The `:math:` rst role provided by *matplotlib.sphinxext.mathmpl* has been renamed to `:mathmpl:` to avoid conflicting with the `:math:` role that Sphinx 1.8 provides by default. (`:mathmpl:` uses Matplotlib to render math expressions to images embedded in html, whereas Sphinx uses MathJax.)

When using Sphinx<1.8, both names (`:math:` and `:mathmpl:`) remain available for backwards-compatibility.

### 9.10.2 API Changes for 3.0.1

`matplotlib.tight_layout.auto_adjust_subplotpars` can return `None` now if the new subplotparams will collapse axes to zero width or height. This prevents `tight_layout` from being executed. Similarly `matplotlib.tight_layout.get_tight_layout_figure` will return None.

To improve import (startup) time, private modules are now imported lazily. These modules are no longer available at these locations:

- `matplotlib.backends.backend_agg._png`
- `matplotlib.contour._contour`
- `matplotlib.image._png`
- `matplotlib.mathtext._png`
- `matplotlib.testing.compare._png`
- `matplotlib.texmanager._png`
- `matplotlib.tri.triangulation._tri`
- `matplotlib.tri.triangulation._qhull`
- `matplotlib.tri.tricontour._tri`
- `matplotlib.tri.trifinder._tri`

### 9.10.3 API Changes for 3.0.0

#### Drop support for python 2

Matplotlib 3 only supports python 3.5 and higher.

### Changes to backend loading

Failure to load backend modules (`macosx` on non-framework builds and `gtk3` when running headless) now raises `ImportError` (instead of `RuntimeError` and `TypeError`, respectively).

Third-party backends that integrate with an interactive framework are now encouraged to define the `required_interactive_framework` global value to one of the following values: "qt5", "qt4", "gtk3", "wx", "tk", or "macosx". This information will be used to determine whether it is possible to switch from a backend to another (specifically, whether they use the same interactive framework).

### `Axes.hist2d` now uses `pcolormesh` instead of `pcolorfast`

*`Axes.hist2d`* now uses *`pcolormesh`* instead of *`pcolorfast`*, which will improve the handling of log-axes. Note that the returned *image* now is of type *`QuadMesh`* instead of *`AxesImage`*.

### `matplotlib.axes.Axes.get_tightbbox` now includes all artists

For Matplotlib 3.0, *all* artists are now included in the bounding box returned by *`matplotlib.axes.Axes.get_tightbbox`*.

*`matplotlib.axes.Axes.get_tightbbox`* adds a new kwarg `bbox_extra_artists` to manually specify the list of artists on the axes to include in the tight bounding box calculation.

Layout tools like *`Figure.tight_layout`*, `constrained_layout`, and `fig.savefig('fname.png', bbox_inches="tight")` use *`matplotlib.axes.Axes.get_tightbbox`* to determine the bounds of each axes on a figure and adjust spacing between axes.

In Matplotlib 2.2 `get_tightbbox` started to include legends made on the axes, but still excluded some other artists, like text that may overspill an axes. This has been expanded to include *all* artists.

This new default may be overridden in either of three ways:

1. Make the artist to be excluded a child of the figure, not the axes. E.g., call `fig.legend()` instead of `ax.legend()` (perhaps using *`get_legend_handles_labels`* to gather handles and labels from the parent axes).
2. If the artist is a child of the axes, set the artist property `artist.set_in_layout(False)`.
3. Manually specify a list of artists in the new kwarg `bbox_extra_artists`.

### `Text.set_text` with string argument `None` sets string to empty

*`Text.set_text`* when passed a string value of `None` would set the string to `"None"`, so subsequent calls to *`Text.get_text`* would return the ambiguous `"None"` string.

This change sets text objects passed `None` to have empty strings, so that *`Text.get_text`* returns an empty string.

#### `Axes3D.get_xlim`, `get_ylim` and `get_zlim` now return a tuple

They previously returned an array. Returning a tuple is consistent with the behavior for 2D axes.

#### `font_manager.list_fonts` now follows the platform's casefolding semantics

i.e., it behaves case-insensitively on Windows only.

#### `bar` / `barh` no longer accepts `left` / `bottom` as first named argument

These arguments were renamed in 2.0 to `x` / `y` following the change of the default alignment from `edge` to `center`.

### Different exception types for undocumented options

- Passing `style='comma'` to *`ticklabel_format()`* was never supported. It now raises `ValueError` like all other unsupported styles, rather than `NotImplementedError`.
- Passing the undocumented `xmin` or `xmax` arguments to *`set_xlim()`* would silently override the `left` and `right` arguments. *`set_ylim()`* and the 3D equivalents (e.g. *`set_zlim`*) had a corresponding problem. A `TypeError` will be raised if they would override the earlier limit arguments. In 3.0 these were kwargs were deprecated, but in 3.1 the deprecation was undone.

### Improved call signature for `Axes.margins`

*`Axes.margins`* and *`Axes3D.margins`* no longer accept arbitrary keywords. `TypeError` will therefore be raised if unknown kwargs are passed; previously they would be silently ignored.

If too many positional arguments are passed, `TypeError` will be raised instead of `ValueError`, for consistency with other call-signature violations.

*`Axes3D.margins`* now raises `TypeError` instead of emitting a deprecation warning if only two positional arguments are passed. To supply only `x` and `y` margins, use keyword arguments.

### Explicit arguments instead of *args, **kwargs

**PEP 3102** describes keyword-only arguments, which allow Matplotlib to provide explicit call signatures - where we previously used `*args, **kwargs` and `kwargs.pop`, we can now expose named arguments. In some places, unknown kwargs were previously ignored but now raise `TypeError` because `**kwargs` has been removed.

- *`matplotlib.axes.Axes.stem()`* no longer accepts unknown keywords, and raises `TypeError` instead of emitting a deprecation.
- *`matplotlib.axes.Axes.stem()`* now raises TypeError when passed unhandled positional arguments. If two or more arguments are passed (ie X, Y, [linefmt], ...) and Y cannot be cast to an array, an error will be raised instead of treating X as Y and Y as linefmt.

- *mpl_toolkits.axes_grid1.axes_divider.SubplotDivider* raises `TypeError` instead of `Exception` when passed unknown kwargs.

### Cleanup decorators and test classes no longer destroy warnings filter on exit

The decorators and classes in matplotlib.testing.decorators no longer destroy the warnings filter on exit. Instead, they restore the warnings filter that existed before the test started using `warnings.catch_warnings`.

### Non-interactive FigureManager classes are now aliases of FigureManagerBase

The `FigureManagerPdf`, `FigureManagerPS`, and `FigureManagerSVG` classes, which were previously empty subclasses of *FigureManagerBase* (i.e., not adding or overriding any attribute or method), are now direct aliases for *FigureManagerBase*.

### Change to the output of `image.thumbnail`

When called with `preview=False`, *image.thumbnail* previously returned an figure whose canvas class was set according to the output file extension. It now returns a figure whose canvas class is the base *FigureCanvasBase* (and relies on *FigureCanvasBase.print_figure*) to handle the canvas switching properly).

As a side effect of this change, *image.thumbnail* now also supports .ps, .eps, and .svgz output.

### `FuncAnimation` now draws artists according to their zorder when blitting

*FuncAnimation* now draws artists returned by the user- function according to their zorder when using blitting, instead of using the order in which they are being passed. However, note that only zorder of passed artists will be respected, as they are drawn on top of any existing artists (see #11369).

### Contour color autoscaling improvements

Selection of contour levels is now the same for contour and contourf; previously, for contour, levels outside the data range were deleted. (Exception: if no contour levels are found within the data range, the `levels` attribute is replaced with a list holding only the minimum of the data range.)

When contour is called with levels specified as a target number rather than a list, and the 'extend' kwarg is used, the levels are now chosen such that some data typically will fall in the extended range.

When contour is called with a *LogNorm* or a *LogLocator*, it will now select colors using the geometric mean rather than the arithmetic mean of the contour levels.

### Streamplot last row and column fixed

A bug was fixed where the last row and column of data in *streamplot* were being dropped.

### Changed default `AutoDateLocator` kwarg *interval_multiples* to `True`

The default value of the tick locator for dates, *dates.AutoDateLocator* kwarg *interval_multiples* was set to `False` which leads to not-nice looking automatic ticks in many instances. The much nicer `interval_multiples=True` is the new default. See below to get the old behavior back:

### `Axes.get_position` now returns actual position if aspect changed

*Axes.get_position* used to return the original position unless a draw had been triggered or *Axes.apply_aspect* had been called, even if the kwarg *original* was set to `False`. Now *Axes.apply_aspect* is called so `ax.get_position()` will return the new modified position. To get the old behavior use `ax.get_position(original=True)`.

### The ticks for colorbar now adjust for the size of the colorbar

Colorbar ticks now adjust for the size of the colorbar if the colorbar is made from a mappable that is not a contour or doesn't have a BoundaryNorm, or boundaries are not specified. If boundaries, etc are specified, the colorbar maintains the original behavior.

### Colorbar for log-scaled hexbin

When using *hexbin* and plotting with a logarithmic color scale, the colorbar ticks are now correctly log scaled. Previously the tick values were linear scaled log(number of counts).

### PGF backend now explicitly makes black text black

Previous behavior with the pgf backend was for text specified as black to actually be the default color of whatever was rendering the pgf file (which was of course usually black). The new behavior is that black text is black, regardless of the default color. However, this means that there is no way to fall back on the default color of the renderer.

### Blacklisted rcparams no longer updated by `rcdefaults`, `rc_file_defaults`, `rc_file`

The rc modifier functions *rcdefaults*, *rc_file_defaults* and *rc_file* now ignore rcParams in the `matplotlib.style.core.STYLE_BLACKLIST` set. In particular, this prevents the `backend` and `interactive` rcParams from being incorrectly modified by these functions.

### `CallbackRegistry` now stores callbacks using stdlib's `weakref.WeakMethod`s

In particular, this implies that `CallbackRegistry.callbacks[signal]` is now a mapping of callback ids to `weakref.WeakMethod`s (i.e., they need to be first called with no arguments to retrieve the method itself).

### Changes regarding the text.latex.unicode rcParam

The rcParam now defaults to True and is deprecated (i.e., in future versions of Matplotlib, unicode input will always be supported).

Moreover, the underlying implementation now uses `\usepackage[utf8]{inputenc}` instead of `\usepackage{ucs}\usepackage[utf8x]{inputenc}`.

### Return type of ArtistInspector.get_aliases changed

`ArtistInspector.get_aliases` previously returned the set of aliases as `{fullname: {alias1: None, alias2: None, ...}}`. The dict-to-None mapping was used to simulate a set in earlier versions of Python. It has now been replaced by a set, i.e. `{fullname: {alias1, alias2, ...}}`.

This value is also stored in `ArtistInspector.aliasd`, which has likewise changed.

### Removed `pytz` as a dependency

Since `dateutil` and `pytz` both provide time zones, and matplotlib already depends on `dateutil`, matplotlib will now use `dateutil` time zones internally and drop the redundant dependency on `pytz`. While `dateutil` time zones are preferred (and currently recommended in the Python documentation), the explicit use of `pytz` zones is still supported.

### Deprecations

### Modules

The following modules are deprecated:

- `matplotlib.compat.subprocess`. This was a python 2 workaround, but all the functionality can now be found in the python 3 standard library `subprocess`.

- `matplotlib.backends.wx_compat`. Python 3 is only compatible with wxPython 4, so support for wxPython 3 or earlier can be dropped.

### Classes, methods, functions, and attributes

The following classes, methods, functions, and attributes are deprecated:

- `RcParams.msg_depr`, `RcParams.msg_depr_ignore`, `RcParams.msg_depr_set`, `RcParams.msg_obsolete`, `RcParams.msg_backend_obsolete`
- `afm.parse_afm`
- `backend_pdf.PdfFile.texFontMap`
- `backend_pgf.get_texcommand`
- `backend_ps.get_bbox`
- `backend_qt5.FigureCanvasQT.keyAutoRepeat` (directly check `event.guiEvent.isAutoRepeat()` in the event handler to decide whether to handle autorepeated key presses).
- `backend_qt5.error_msg_qt`, `backend_qt5.exception_handler`
- `backend_wx.FigureCanvasWx.macros`
- `backends.pylab_setup`
- `cbook.GetRealpathAndStat`, `cbook.Locked`
- `cbook.is_numlike` (use `isinstance(..., numbers.Number)` instead), `cbook.listFiles`, `cbook.unicode_safe`
- `container.Container.set_remove_method`,
- `contour.ContourLabeler.cl`, `.cl_xy`, and `.cl_cvalues`
- `dates.DateFormatter.strftime_pre_1900`, `dates.DateFormatter.strftime`
- `font_manager.TempCache`
- `image._ImageBase.iterpnames`, use the `interpolation_names` property instead. (this affects classes that inherit from `_ImageBase` including *FigureImage*, *BboxImage*, and *AxesImage*)
- `mathtext.unichr_safe` (use `chr` instead)
- `patches.Polygon.xy`
- `table.Table.get_child_artists` (use `get_children` instead)
- `testing.compare.ImageComparisonTest`, `testing.compare.compare_float`
- `testing.decorators.CleanupTest`, `testing.decorators.skip_if_command_unavailable`
- `FigureCanvasQT.keyAutoRepeat` (directly check `event.guiEvent.isAutoRepeat()` in the event handler to decide whether to handle autorepeated key presses)

- `FigureCanvasWx.macros`
- `_ImageBase.iterpnames`, use the `interpolation_names` property instead. (this affects classes that inherit from `_ImageBase` including *`FigureImage`*, *`BboxImage`*, and *`AxesImage`*)
- `patches.Polygon.xy`
- `texmanager.dvipng_hack_alpha`
- `text.Annotation.arrow`
- **`Legend.draggable()`, in favor of *`Legend.set_draggable()`***

  (`Legend.draggable` may be reintroduced as a property in future releases)
- `textpath.TextToPath.tex_font_map`
- `matplotlib.cbook.deprecation.mplDeprecation` will be removed in future versions. It is just an alias for `matplotlib.cbook.deprecation.MatplotlibDeprecationWarning`. Please use `matplotlib.cbook.MatplotlibDeprecationWarning` directly if necessary.
- The `matplotlib.cbook.Bunch` class has been deprecated. Instead, use `types.SimpleNamespace` from the standard library which provides the same functionality.
- `Axes.mouseover_set` is now a frozenset, and deprecated. Directly manipulate the artist's `.mouseover` attribute to change their mouseover status.

The following keyword arguments are deprecated:

- passing `verts` to `Axes.scatter` (use `marker` instead)
- passing `obj_type` to `cbook.deprecated`

The following call signatures are deprecated:

- passing a `wx.EvtHandler` as first argument to `backend_wx.TimerWx`

### rcParams

The following rcParams are deprecated:

- `examples.directory` (use `datapath` instead)
- `pgf.debug` (the pgf backend relies on logging)
- `text.latex.unicode` (always True now)

### marker styles

- Using `(n, 3)` as marker style to specify a circle marker is deprecated. Use `"o"` instead.
- Using `([(x0, y0), (x1, y1), ...], 0)` as marker style to specify a custom marker path is deprecated. Use `[(x0, y0), (x1, y1), ...]` instead.

### Deprecation of `LocatableAxes` in toolkits

The `LocatableAxes` classes in toolkits have been deprecated. The base *`Axes`* classes provide the same functionality to all subclasses, thus these mixins are no longer necessary. Related functions have also been deprecated. Specifically:

- `mpl_toolkits.axes_grid1.axes_divider.LocatableAxesBase`: no specific replacement; use any other `Axes`-derived class directly instead.
- `mpl_toolkits.axes_grid1.axes_divider.locatable_axes_factory`: no specific replacement; use any other `Axes`-derived class directly instead.
- `mpl_toolkits.axes_grid1.axes_divider.Axes`: use *`mpl_toolkits.axes_grid1.mpl_axes.Axes`* directly.
- `mpl_toolkits.axes_grid1.axes_divider.LocatableAxes`: use *`mpl_toolkits.axes_grid1.mpl_axes.Axes`* directly.
- `mpl_toolkits.axisartist.axes_divider.Axes`: use *`mpl_toolkits.axisartist.axislines.Axes`* directly.
- `mpl_toolkits.axisartist.axes_divider.LocatableAxes`: use *`mpl_toolkits.axisartist.axislines.Axes`* directly.

## Removals

### Hold machinery

Setting or unsetting `hold` (*deprecated in version 2.0*) has now been completely removed. Matplotlib now always behaves as if `hold=True`. To clear an axes you can manually use *`cla()`*, or to clear an entire figure use *`clear()`*.

### Removal of deprecated backends

Deprecated backends have been removed:

- GTKAgg
- GTKCairo
- GTK
- GDK

### Deprecated APIs

The following deprecated API elements have been removed:

- The deprecated methods `knownfailureif` and `remove_text` have been removed from *matplotlib.testing.decorators*.
- The entire contents of `testing.noseclasses` have also been removed.
- `matplotlib.checkdep_tex`, `matplotlib.checkdep_xmllint`
- `backend_bases.IdleEvent`
- `cbook.converter`, `cbook.tostr`, `cbook.todatetime`, `cbook.todate`, `cbook.tofloat`, `cbook.toint`, `cbook.unique`, `cbook.is_string_like`, `cbook.is_sequence_of_strings`, `cbook.is_scalar`, `cbook.soundex`, `cbook.dict_delall`, `cbook.get_split_ind`, `cbook.wrap`, `cbook.get_recursive_filelist`, `cbook.pieces`, `cbook.exception_to_str`, `cbook.allequal`, `cbook.alltrue`, `cbook.onetrue`, `cbook.allpairs`, `cbook.finddir`, `cbook.reverse_dict`, `cbook.restrict_dict`, `cbook.issubclass_safe`, `cbook.recursive_remove`, `cbook.unmasked_index_ranges`, `cbook.Null`, `cbook.RingBuffer`, `cbook.Sorter`, `cbook.Xlator`,
- `font_manager.weight_as_number`, `font_manager.ttfdict_to_fnames`
- `pyplot.colors`, `pyplot.spectral`
- `rcsetup.validate_negative_linestyle`, `rcsetup.validate_negative_linestyle_legacy`,
- `testing.compare.verifiers`, `testing.compare.verify`
- `testing.decorators.knownfailureif`, `testing.decorators.ImageComparisonTest.remove_text`
- `tests.assert_str_equal`, `tests.test_tinypages.file_same`
- `texmanager.dvipng_hack_alpha`,
- `_AxesBase.axesPatch`, `_AxesBase.set_color_cycle`, `_AxesBase.get_cursor_props`, `_AxesBase.set_cursor_props`
- `_ImageBase.iterpnames`
- `FigureCanvasBase.start_event_loop_default`;
- `FigureCanvasBase.stop_event_loop_default`;
- `Figure.figurePatch`,
- `FigureCanvasBase.dynamic_update`, `FigureCanvasBase.idle_event`, `FigureCanvasBase.get_linestyle`, `FigureCanvasBase.set_linestyle`
- `FigureCanvasQTAggBase`
- `FigureCanvasQTAgg.blitbox`
- `FigureCanvasTk.show` (alternative: `FigureCanvasTk.draw`)

- `FigureManagerTkAgg` (alternative: `FigureManagerTk`)
- `NavigationToolbar2TkAgg` (alternative: `NavigationToolbar2Tk`)
- `backend_wxagg.Toolbar` (alternative: `backend_wxagg.NavigationToolbar2WxAgg`)
- `RendererAgg.debug()`
- passing non-numbers to `EngFormatter.format_eng`
- passing `frac` to `PolarAxes.set_theta_grids`
- any mention of idle events

The following API elements have been removed:

- `backend_cairo.HAS_CAIRO_CFFI`
- `sphinxext.sphinx_version`

#### Proprietary sphinx directives

The matplotlib documentation used the proprietary sphinx directives `.. htmlonly::`, and `.. latexonly::`. These have been replaced with the standard sphinx directives `.. only:: html` and `.. only:: latex`. This change will not affect any users. Only downstream package maintainers, who have used the proprietary directives in their docs, will have to switch to the sphinx directives.

#### lib/mpl_examples symlink

The symlink from lib/mpl_examples to ../examples has been removed. This is not installed as an importable package and should not affect end users, however this may require down-stream packagers to adjust. The content is still available top-level examples directory.

## 9.11 Version 2.2

### 9.11.1 What's new in Matplotlib 2.2 (Mar 06, 2018)

#### Constrained Layout Manager

> **Warning:** Constrained Layout is **experimental**. The behaviour and API are subject to change, or the whole functionality may be removed without a deprecation period.

A new method to automatically decide spacing between subplots and their organizing `GridSpec` instances has been added. It is meant to replace the venerable `tight_layout` method. It is invoked via a new `constrained_layout=True` kwarg to *`Figure`* or *`subplots`*.

There are new *rcParams* for this package, and spacing can be more finely tuned with the new *set_constrained_layout_pads*.

Features include:

- Automatic spacing for subplots with a fixed-size padding in inches around subplots and all their decorators, and space between as a fraction of subplot size between subplots.
- Spacing for *suptitle*, and colorbars that are attached to more than one axes.
- Nested *GridSpec* layouts using *GridSpecFromSubplotSpec*.

For more details and capabilities please see the new tutorial: *Constrained layout guide*

Note the new API to access this:

### New `plt.figure` and `plt.subplots` kwarg: `constrained_layout`

*figure()* and *subplots()* can now be called with `constrained_layout=True` kwarg to enable constrained_layout.

### New `ax.set_position` behaviour

*Axes.set_position* now makes the specified axis no longer responsive to `constrained_layout`, consistent with the idea that the user wants to place an axis manually.

Internally, this means that old `ax.set_position` calls *inside* the library are changed to private `ax._set_position` calls so that `constrained_layout` will still work with these axes.

### New `figure` kwarg for `GridSpec`

In order to facilitate `constrained_layout`, `GridSpec` now accepts a `figure` keyword. This is backwards compatible, in that not supplying this will simply cause `constrained_layout` to not operate on the subplots organized by this `GridSpec` instance. Routines that use `GridSpec` (e.g. `fig.subplots`) have been modified to pass the figure to `GridSpec`.

### xlabels and ylabels can now be automatically aligned

Subplot axes `ylabels` can be misaligned horizontally if the tick labels are very different widths. The same can happen to `xlabels` if the ticklabels are rotated on one subplot (for instance). The new methods on the *Figure* class: *Figure.align_xlabels* and *Figure.align_ylabels* will now align these labels horizontally or vertically. If the user only wants to align some axes, a list of axes can be passed. If no list is passed, the algorithm looks at all the labels on the figure.

Only labels that have the same subplot locations are aligned. i.e. the ylabels are aligned only if the subplots are in the same column of the subplot layout.

Alignment is persistent and automatic after these are called.

A convenience wrapper *Figure.align_labels* calls both functions at once.

### Axes legends now included in tight_bbox

Legends created via `ax.legend` can sometimes overspill the limits of the axis. Tools like `fig.tight_layout()` and `fig.savefig(bbox_inches='tight')` would clip these legends. A change was made to include them in the `tight` calculations.

### Cividis colormap

A new dark blue/yellow colormap named 'cividis' was added. Like viridis, cividis is perceptually uniform and colorblind friendly. However, cividis also goes a step further: not only is it usable by colorblind users, it should actually look effectively identical to colorblind and non-colorblind users. For more details see Nuñez J, Anderton C, and Renslow R: "Optimizing colormaps with consideration for color vision deficiency to enable accurate interpretation of scientific data".

### New style colorblind-friendly color cycle

A new style defining a color cycle has been added, tableau-colorblind10, to provide another option for colorblind-friendly plots. A demonstration of this new style can be found in the reference of style sheets. To load this color cycle in place of the default one:

```
import matplotlib.pyplot as plt
plt.style.use('tableau-colorblind10')
```

### Support for numpy.datetime64

Matplotlib has supported `datetime.datetime` dates for a long time in *matplotlib.dates*. We now support `numpy.datetime64` dates as well. Anywhere that `datetime.datetime` could be used, `numpy.datetime64` can be used. eg:

```
time = np.arange('2005-02-01', '2005-02-02', dtype='datetime64[h]')
plt.plot(time)
```

### Writing animations with Pillow

It is now possible to use Pillow as an animation writer. Supported output formats are currently gif (Pillow>=3.4) and webp (Pillow>=5.0). Use e.g. as

```
from __future__ import division

from matplotlib import pyplot as plt
from matplotlib.animation import FuncAnimation, PillowWriter

fig, ax = plt.subplots()
line, = plt.plot([0, 1])

def animate(i):
   line.set_ydata([0, i / 20])
   return [line]

anim = FuncAnimation(fig, animate, 20, blit=True)
anim.save("movie.gif", writer=PillowWriter(fps=24))
plt.show()
```

### Slider UI widget can snap to discrete values

The slider UI widget can take the optional argument *valstep*. Doing so forces the slider to take on only discrete values, starting from *valmin* and counting up to *valmax* with steps of size *valstep*.

If *closedmax*==*True*, then the slider will snap to *valmax* as well.

### `capstyle` and `joinstyle` attributes added to `Collection`

The *`Collection`* class now has customizable `capstyle` and `joinstyle` attributes. This allows the user for example to set the `capstyle` of errorbars.

### *pad* kwarg added to ax.set_title

The method *`Axes.set_title`* now has a *pad* kwarg, that specifies the distance from the top of an axes to where the title is drawn. The units of *pad* is points, and the default is the value of the (already-existing) *`rcParams["axes.titlepad"]`* (default: `6.0`).

### Comparison of 2 colors in Matplotlib

As the colors in Matplotlib can be specified with a wide variety of ways, the *matplotlib.colors.same_color* method has been added which checks if two *colors* are the same.

### Autoscaling a polar plot snaps to the origin

Setting the limits automatically in a polar plot now snaps the radial limit to zero if the automatic limit is nearby. This means plotting from zero doesn't automatically scale to include small negative values on the radial axis.

The limits can still be set manually in the usual way using *set_ylim*.

### PathLike support

On Python 3.6+, *savefig*, *imsave*, *imread*, and animation writers now accept `os.PathLike`s as input.

### `Axes.tick_params` can set gridline properties

*Tick* objects hold gridlines as well as the tick mark and its label. *Axis.set_tick_params*, *Axes.tick_params* and *pyplot.tick_params* now have keyword arguments 'grid_color', 'grid_alpha', 'grid_linewidth', and 'grid_linestyle' for overriding the defaults in *rcParams*: 'grid.color', etc.

### `Axes.imshow` clips RGB values to the valid range

When *Axes.imshow* is passed an RGB or RGBA value with out-of-range values, it now logs a warning and clips them to the valid range. The old behaviour, wrapping back in to the range, often hid outliers and made interpreting RGB images unreliable.

### Properties in `matplotlibrc` to place xaxis and yaxis tick labels

Introducing four new boolean properties in `matplotlibrc` for default positions of xaxis and yaxis tick labels, namely, *rcParams["xtick.labeltop"]* (default: `False`), *rcParams["xtick.labelbottom"]* (default: `True`), *rcParams["ytick.labelright"]* (default: `False`) and *rcParams["ytick.labelleft"]* (default: `True`). These can also be changed in rcParams.

### PGI bindings for gtk3

The GTK3 backends can now use PGI instead of PyGObject. PGI is a fairly incomplete binding for GObject, thus its use is not recommended; its main benefit is its availability on Travis (thus allowing CI testing for the gtk3agg and gtk3cairo backends).

The binding selection rules are as follows: - if `gi` has already been imported, use it; else - if `pgi` has already been imported, use it; else - if `gi` can be imported, use it; else - if `pgi` can be imported, use it; else - error out.

Thus, to force usage of PGI when both bindings are installed, import it first.

### Cairo rendering for Qt, WX, and Tk canvases

The new `Qt4Cairo`, `Qt5Cairo`, `WXCairo`, and `TkCairo` backends allow Qt, Wx, and Tk canvases to use Cairo rendering instead of Agg.

### Added support for QT in new ToolManager

Now it is possible to use the ToolManager with Qt5 For example:

```
import matplotlib

matplotlib.use('QT5AGG')
matplotlib.rcParams['toolbar'] = 'toolmanager'
import matplotlib.pyplot as plt

plt.plot([1,2,3])
plt.show()
```

Treat the new Tool classes experimental for now, the API will likely change and perhaps the rcParam as well

The main example *Tool Manager* shows more details, just adjust the header to use QT instead of GTK3

### TkAgg backend reworked to support PyPy

PyPy can now plot using the TkAgg backend, supported on PyPy 5.9 and greater (both PyPy for python 2.7 and PyPy for python 3.5).

### Python logging library used for debug output

Matplotlib has in the past (sporadically) used an internal verbose-output reporter. This version converts those calls to using the standard python `logging` library.

Support for the old *rcParams* `verbose.level` and `verbose.fileo` is dropped.

The command-line options `--verbose-helpful` and `--verbose-debug` are still accepted, but deprecated. They are now equivalent to setting `logging.INFO` and `logging.DEBUG`.

The logger's root name is `matplotlib` and can be accessed from programs as:

```
import logging
mlog = logging.getLogger('matplotlib')
```

Instructions for basic usage are in *Troubleshooting* and for developers in *Contribute*.

### Improved `repr` for `Transform`s

*Transform*s now indent their `repr`s in a more legible manner:

```
In [1]: l, = plt.plot([]); l.get_transform()
Out[1]:
CompositeGenericTransform(
   TransformWrapper(
      BlendedAffine2D(
            IdentityTransform(),
            IdentityTransform())),
   CompositeGenericTransform(
      BboxTransformFrom(
            TransformedBbox(
               Bbox(x0=-0.05500000000000001, y0=-0.05500000000000001, x1=0.
↪05500000000000001, y1=0.05500000000000001),
               TransformWrapper(
                  BlendedAffine2D(
                        IdentityTransform(),
                        IdentityTransform())))),
      BboxTransformTo(
            TransformedBbox(
               Bbox(x0=0.125, y0=0.10999999999999999, x1=0.9, y1=0.88),
               BboxTransformTo(
                  TransformedBbox(
                        Bbox(x0=0.0, y0=0.0, x1=6.4, y1=4.8),
                        Affine2D(
                           [[ 100.    0.    0.]
                           [   0.  100.    0.]
                           [   0.    0.    1.]]))))))))
```

### 9.11.2 API Changes in 2.2.0

#### New dependency

kiwisolver is now a required dependency to support the new constrained_layout, see *Constrained layout guide* for more details.

#### Deprecations

#### Classes, functions, and methods

The unused and untested `Artist.onRemove` and `Artist.hitlist` methods have been deprecated.

The now unused `mlab.less_simple_linear_interpolation` function is deprecated.

The unused `ContourLabeler.get_real_label_width` method is deprecated.

The unused `FigureManagerBase.show_popup` method is deprecated. This introduced in e945059b327d42a99938b939a1be867fa023e7ba in 2005 but never built out into any of the backends.

`backend_tkagg.AxisMenu` is deprecated, as it has become unused since the removal of "classic" toolbars.

#### Changed function signatures

kwarg `fig` to *`GridSpec.get_subplot_params`* is deprecated, use `figure` instead.

Using *`pyplot.axes`* with an *`Axes`* as argument is deprecated. This sets the current axes, i.e. it has the same effect as *`pyplot.sca`*. For clarity `plt.sca(ax)` should be preferred over `plt.axes(ax)`.

Using strings instead of booleans to control grid and tick visibility is deprecated. Using `"on"`, `"off"`, `"true"`, or `"false"` to control grid and tick visibility has been deprecated. Instead, use normal booleans (`True`/`False`) or boolean-likes. In the future, all non-empty strings may be interpreted as `True`.

When given 2D inputs with non-matching numbers of columns, *`plot`* currently cycles through the columns of the narrower input, until all the columns of the wider input have been plotted. This behavior is deprecated; in the future, only broadcasting (1 column to *n* columns) will be performed.

#### rcparams

The *`rcParams["backend.qt4"]`* and *`rcParams["backend.qt5"]`* rcParams were deprecated in version 2.2. In order to force the use of a specific Qt binding, either import that binding first, or set the `QT_API` environment variable.

Deprecation of the `nbagg.transparent` rcParam. To control transparency of figure patches in the nbagg (or any other) backend, directly set `figure.patch.facecolor`, or the `figure.facecolor` rcParam.

### Deprecated `Axis.unit_data`

Use `Axis.units` (which has long existed) instead.

### Removals

### Function Signatures

Contouring no longer supports `legacy` corner masking. The deprecated `ContourSet.vmin` and `ContourSet.vmax` properties have been removed.

Passing `None` instead of `"none"` as format to *`errorbar`* is no longer supported.

The `bgcolor` keyword argument to `Axes` has been removed.

### Modules, methods, and functions

The `matplotlib.finance`, `mpl_toolkits.exceltools` and `mpl_toolkits.gtktools` modules have been removed. `matplotlib.finance` remains available at https://github.com/matplotlib/mpl_finance.

The `mpl_toolkits.mplot3d.art3d.iscolor` function has been removed.

The `Axes.get_axis_bgcolor`, `Axes.set_axis_bgcolor`, `Bbox.update_from_data`, `Bbox.update_datalim_numerix`, `MaxNLocator.bin_boundaries` methods have been removed.

`mencoder` can no longer be used to encode animations.

The unused `FONT_SCALE` and `fontd` attributes of the *`RendererSVG`* class have been removed.

### colormaps

The `spectral` colormap has been removed. The `Vega*` colormaps, which were aliases for the `tab*` colormaps, have been removed.

### rcparams

The following deprecated rcParams have been removed:

- `axes.color_cycle` (see `axes.prop_cycle`),
- `legend.isaxes`,
- `svg.embed_char_paths` (see `svg.fonttype`),
- `text.fontstyle`, `text.fontangle`, `text.fontvariant`, `text.fontweight`, `text.fontsize` (renamed to `text.style`, etc.),

- `tick.size` (renamed to `tick.major.size`).

### Only accept string-like for Categorical input

Do not accept mixed string / float / int input, only strings are valid categoricals.

### Removal of unused imports

Many unused imports were removed from the codebase. As a result, trying to import certain classes or functions from the "wrong" module (e.g. *Figure* from *matplotlib.backends.backend_agg* instead of *matplotlib.figure*) will now raise an `ImportError`.

### `Axes3D.get_xlim`, `get_ylim` and `get_zlim` now return a tuple

They previously returned an array. Returning a tuple is consistent with the behavior for 2D axes.

### Exception type changes

If *MovieWriterRegistry* can't find the requested *MovieWriter*, a more helpful `RuntimeError` message is now raised instead of the previously raised `KeyError`.

`matplotlib.tight_layout.auto_adjust_subplotpars` now raises `ValueError` instead of `RuntimeError` when sizes of input lists don't match

### `Figure.set_figwidth` and `Figure.set_figheight` default *forward* to True

*matplotlib.figure.Figure.set_figwidth* and *matplotlib.figure.Figure.set_figheight* had the keyword argument `forward=False` by default, but *figure.Figure.set_size_inches* now defaults to `forward=True`. This makes these functions consistent.

### Do not truncate svg sizes to nearest point

There is no reason to size the SVG out put in integer points, change to out putting floats for the *height*, *width*, and *viewBox* attributes of the *svg* element.

### Fontsizes less than 1 pt are clipped to be 1 pt.

FreeType doesn't allow fonts to get smaller than 1 pt, so all Agg backends were silently rounding up to 1 pt. PDF (other vector backends?) were letting us write fonts that were less than 1 pt, but they could not be placed properly because position information comes from FreeType. This change makes it so no backends can use fonts smaller than 1 pt, consistent with FreeType and ensuring more consistent results across backends.

### Changes to Qt backend class MRO

To support both Agg and cairo rendering for Qt backends all of the non-Agg specific code previously in `backend_qt5agg.FigureCanvasQTAggBase` has been moved to `backend_qt5.FigureCanvasQT` so it can be shared with the cairo implementation. The `FigureCanvasQTAggBase.paintEvent`, `FigureCanvasQTAggBase.blit`, and `FigureCanvasQTAggBase.print_figure` methods have moved to `FigureCanvasQTAgg.paintEvent`, `FigureCanvasQTAgg.blit`, and `FigureCanvasQTAgg.print_figure`. The first two methods assume that the instance is also a `QWidget` so to use `FigureCanvasQTAggBase` it was required to multiple inherit from a `QWidget` sub-class.

Having moved all of its methods either up or down the class hierarchy `FigureCanvasQTAggBase` has been deprecated. To do this without warning and to preserve as much API as possible, `.backend_qt5agg.FigureCanvasQTAggBase` now inherits from `backend_qt5.FigureCanvasQTAgg`.

The MRO for `FigureCanvasQTAgg` and `FigureCanvasQTAggBase` used to be

```
[matplotlib.backends.backend_qt5agg.FigureCanvasQTAgg,
 matplotlib.backends.backend_qt5agg.FigureCanvasQTAggBase,
 matplotlib.backends.backend_agg.FigureCanvasAgg,
 matplotlib.backends.backend_qt5.FigureCanvasQT,
 PyQt5.QtWidgets.QWidget,
 PyQt5.QtCore.QObject,
 sip.wrapper,
 PyQt5.QtGui.QPaintDevice,
 sip.simplewrapper,
 matplotlib.backend_bases.FigureCanvasBase,
 object]
```

and

```
[matplotlib.backends.backend_qt5agg.FigureCanvasQTAggBase,
 matplotlib.backends.backend_agg.FigureCanvasAgg,
 matplotlib.backend_bases.FigureCanvasBase,
 object]
```

respectively. They are now

```
[matplotlib.backends.backend_qt5agg.FigureCanvasQTAgg,
 matplotlib.backends.backend_agg.FigureCanvasAgg,
 matplotlib.backends.backend_qt5.FigureCanvasQT,
 PyQt5.QtWidgets.QWidget,
```

(continues on next page)

(continued from previous page)

```
 PyQt5.QtCore.QObject,
 sip.wrapper,
 PyQt5.QtGui.QPaintDevice,
 sip.simplewrapper,
 matplotlib.backend_bases.FigureCanvasBase,
 object]
```

and

```
[matplotlib.backends.backend_qt5agg.FigureCanvasQTAggBase,
 matplotlib.backends.backend_qt5agg.FigureCanvasQTAgg,
 matplotlib.backends.backend_agg.FigureCanvasAgg,
 matplotlib.backends.backend_qt5.FigureCanvasQT,
 PyQt5.QtWidgets.QWidget,
 PyQt5.QtCore.QObject,
 sip.wrapper,
 PyQt5.QtGui.QPaintDevice,
 sip.simplewrapper,
 matplotlib.backend_bases.FigureCanvasBase,
 object]
```

#### `axes.Axes.imshow` clips RGB values to the valid range

When *axes.Axes.imshow* is passed an RGB or RGBA value with out-of-range values, it now logs a warning and clips them to the valid range. The old behaviour, wrapping back in to the range, often hid outliers and made interpreting RGB images unreliable.

#### GTKAgg and GTKCairo backends deprecated

The GTKAgg and GTKCairo backends have been deprecated. These obsolete backends allow figures to be rendered via the GTK+ 2 toolkit. They are untested, known to be broken, will not work with Python 3, and their use has been discouraged for some time. Instead, use the `GTK3Agg` and `GTK3Cairo` backends for rendering to GTK+ 3 windows.

## 9.12 Version 2.1

### 9.12.1 What's new in Matplotlib 2.1.0 (Oct 7, 2017)

#### Documentation

The examples have been migrated to use sphinx gallery. This allows better mixing of prose and code in the examples, provides links to download the examples as both a Python script and a Jupyter notebook, and improves the thumbnail galleries. The examples have been re-organized into *Tutorials* and a *Examples*.

Many docstrings and examples have been clarified and improved.

## New features

### String categorical values

All plotting functions now support string categorical values as input. For example:

```
data = {'apples': 10, 'oranges': 15, 'lemons': 5, 'limes': 20}
fig, ax = plt.subplots()
ax.bar(data.keys(), data.values(), color='lightgray')
```

### Interactive JS widgets for animation

Jake Vanderplas' JSAnimation package has been merged into Matplotlib. This adds to Matplotlib the *HTMLWriter* class for generating a JavaScript HTML animation, suitable for the IPython notebook. This can be activated by default by setting the `animation.html` rc parameter to `jshtml`. One can also call the *to_jshtml* method to manually convert an animation. This can be displayed using IPython's `HTML` display class:

```
from IPython.display import HTML
HTML(animation.to_jshtml())
```

The *HTMLWriter* class can also be used to generate an HTML file by asking for the `html` writer.

### Enhancements to polar plot

The polar axes transforms have been greatly re-factored to allow for more customization of view limits and tick labelling. Additional options for view limits allow for creating an annulus, a sector, or some combination of the two.

The *set_rorigin()* method may be used to provide an offset to the minimum plotting radius, producing an annulus.

The *set_theta_zero_location()* method now has an optional `offset` argument. This argument may be used to further specify the zero location based on the given anchor point.

The *set_thetamin()* and *set_thetamax()* methods may be used to limit the range of angles plotted, producing sectors of a circle.

Previous releases allowed plots containing negative radii for which the negative values are simply used as labels, and the real radius is shifted by the configured minimum. This release also allows negative radii to be used for grids and ticks, which were previously silently ignored.

Radial ticks have been modified to be parallel to the circular grid line, and angular ticks have been modified to be parallel to the grid line. It may also be useful to rotate tick *labels* to match the boundary. Calling `ax.tick_params(rotation='auto')` will enable the new behavior: radial tick labels will be parallel to the circular grid line, and angular tick labels will be perpendicular to the grid line (i.e., parallel to the outer boundary). Additionally, tick labels now obey the padding settings that previously only

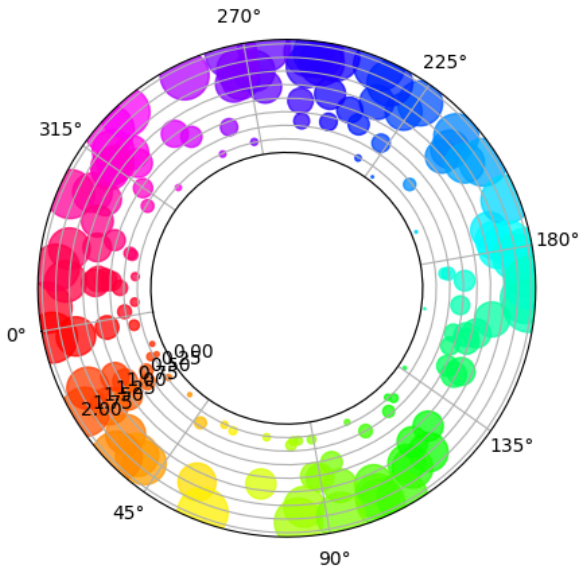



Fig. 4: Polar Offset Demo

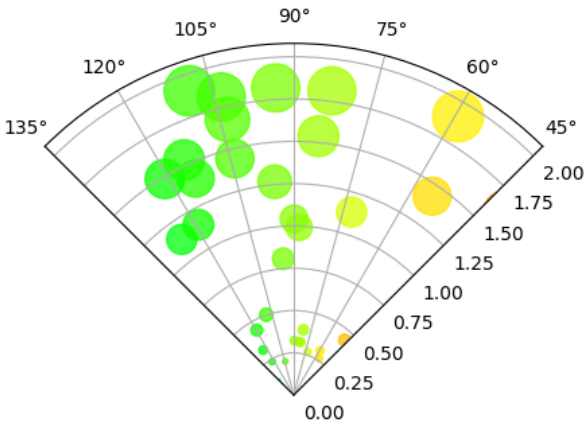



Fig. 5: Polar Sector Demo

worked on Cartesian plots. Consequently, the `frac` argument to *`PolarAxes.set_thetagrids`* is no longer applied. Tick padding can be modified with the `pad` argument to *`Axes.tick_params`* or *`Axis.set_tick_params`*.

### `Figure` class now has `subplots` method

The *`Figure`* class now has a *`subplots()`* method which behaves the same as *`pyplot.subplots()`* but on an existing figure.

### Metadata savefig keyword argument

*`savefig()`* now accepts `metadata` as a keyword argument. It can be used to store key/value pairs in the image metadata.

- 'png' with Agg backend
- 'pdf' with PDF backend (see *`writeInfoDict()`* for a list of supported keywords)
- 'eps' and 'ps' with PS backend (only 'Creator' key is accepted)

```
plt.savefig('test.png', metadata={'Software': 'My awesome software'})
```

### Busy Cursor

The interactive GUI backends will now change the cursor to busy when Matplotlib is rendering the canvas.

### PolygonSelector

A *`PolygonSelector`* class has been added to *`matplotlib.widgets`*. See *Select indices from a collection using polygon selector* for details.

### Added `matplotlib.ticker.PercentFormatter`

The new *`PercentFormatter`* formatter has some nice features like being able to convert from arbitrary data scales to percents, a customizable percent symbol and either automatic or manual control over the decimal points.

### Reproducible PS, PDF and SVG output

The `SOURCE_DATE_EPOCH` environment variable can now be used to set the timestamp value in the PS and PDF outputs. See source date epoch.

Alternatively, calling `savefig` with `metadata={'CreationDate': None}` will omit the timestamp altogether for the PDF backend.

The reproducibility of the output from the PS and PDF backends has so far been tested using various plot elements but only default values of options such as `{ps,pdf}.fonttype` that can affect the output at a low level, and not with the mathtext or usetex features. When Matplotlib calls external tools (such as PS distillers or LaTeX) their versions need to be kept constant for reproducibility, and they may add sources of nondeterminism outside the control of Matplotlib.

For SVG output, the `svg.hashsalt` rc parameter has been added in an earlier release. This parameter changes some random identifiers in the SVG file to be deterministic. The downside of this setting is that if more than one file is generated using deterministic identifiers and they end up as parts of one larger document, the identifiers can collide and cause the different parts to affect each other.

These features are now enabled in the tests for the PDF and SVG backends, so most test output files (but not all of them) are now deterministic.

### Orthographic projection for mplot3d

*`Axes3D`* now accepts `proj_type` keyword argument and has a method *`set_proj_type()`*. The default option is `'persp'` as before, and supplying `'ortho'` enables orthographic view.

Compare the z-axis which is vertical in orthographic view, but slightly skewed in the perspective view.

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure(figsize=(4, 6))
ax1 = fig.add_subplot(2, 1, 1, projection='3d')
ax1.set_proj_type('persp')
ax1.set_title('Perspective (default)')

ax2 = fig.add_subplot(2, 1, 2, projection='3d')
ax2.set_proj_type('ortho')
ax2.set_title('Orthographic')

plt.show()
```

### `voxels` function for mplot3d

*Axes3D* now has a *voxels* method, for visualizing boolean 3D data. Uses could include plotting a sparse 3D heat map, or visualizing a volumetric model.

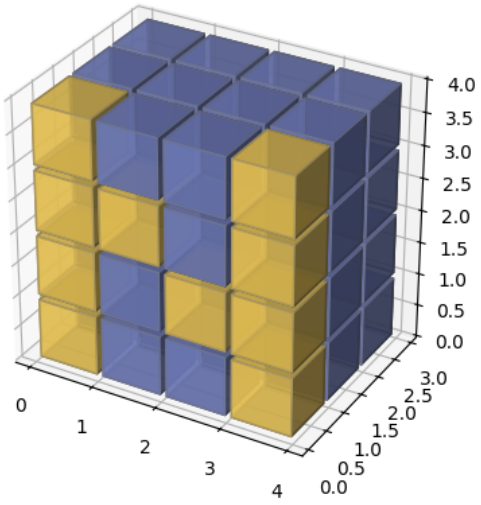



Fig. 6: Voxel Demo

## Improvements

### CheckButtons widget `get_status` function

A *get_status()* method has been added to the *matplotlib.widgets.CheckButtons* class. This `get_status` method allows user to query the status (True/False) of all of the buttons in the `CheckButtons` object.

### Add `fill_bar` argument to `AnchoredSizeBar`

The `mpl_toolkits` class *AnchoredSizeBar* now has an additional `fill_bar` argument, which makes the size bar a solid rectangle instead of just drawing the border of the rectangle. The default is `None`, and whether or not the bar will be filled by default depends on the value of `size_vertical`. If `size_vertical` is nonzero, `fill_bar` will be set to `True`. If `size_vertical` is zero then `fill_bar` will be set to `False`. If you wish to override this default behavior, set `fill_bar` to `True` or `False` to unconditionally always or never use a filled patch rectangle for the size bar.

```
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1.anchored_artists import AnchoredSizeBar
```

(continues on next page)

(continued from previous page)

```
fig, ax = plt.subplots(figsize=(3, 3))

bar0 = AnchoredSizeBar(ax.transData, 0.3, 'unfilled', loc='lower left',
                       frameon=False, size_vertical=0.05, fill_bar=False)
ax.add_artist(bar0)
bar1 = AnchoredSizeBar(ax.transData, 0.3, 'filled', loc='lower right',
                       frameon=False, size_vertical=0.05, fill_bar=True)
ax.add_artist(bar1)

plt.show()
```

### Annotation can use a default arrow style

Annotations now use the default arrow style when setting `arrowprops={}`, rather than no arrow (the new behavior actually matches the documentation).

### Barbs and Quiver Support Dates

When using the *quiver()* and *barbs()* plotting methods, it is now possible to pass dates, just like for other methods like *plot()*. This also allows these functions to handle values that need unit-conversion applied.

### Hexbin default line color

The default `linecolor` keyword argument for *hexbin()* is now `'face'`, and supplying `'none'` now prevents lines from being drawn around the hexagons.

### Figure.legend() can be called without arguments

Calling *Figure.legend()* can now be done with no arguments. In this case a legend will be created that contains all the artists on all the axes contained within the figure.

### Multiple legend keys for legend entries

A legend entry can now contain more than one legend key. The extended *HandlerTuple* class now accepts two parameters: `ndivide` divides the legend area in the specified number of sections; `pad` changes the padding between the legend keys.

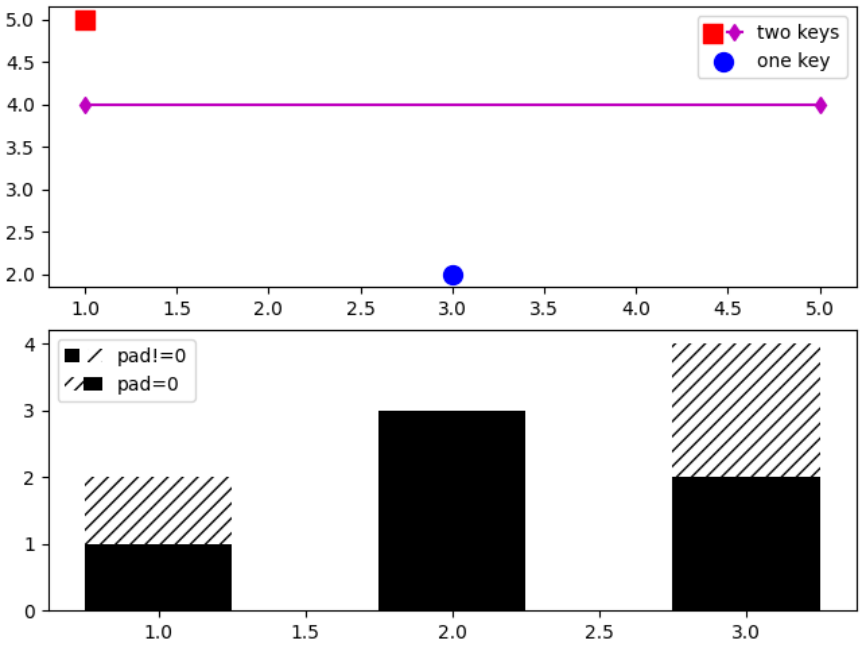



Fig. 7: Multiple Legend Keys

### New parameter *clear* for `figure()`

When the pyplot's function *figure()* is called with a `num` parameter, a new window is only created if no existing window with the same value exists. A new bool parameter *clear* was added for explicitly clearing its existing contents. This is particularly useful when utilized in interactive sessions. Since *subplots()* also accepts keyword arguments from *figure()*, it can also be used there:

```
import matplotlib.pyplot as plt

fig0 = plt.figure(num=1)
fig0.suptitle("A fancy plot")
print("fig0.texts: ", [t.get_text() for t in fig0.texts])

fig1 = plt.figure(num=1, clear=False)  # do not clear contents of window
fig1.text(0.5, 0.5, "Really fancy!")
print("fig0 is fig1: ",  fig0 is fig1)
print("fig1.texts: ", [t.get_text() for t in fig1.texts])

fig2, ax2 = plt.subplots(2, 1, num=1, clear=True)  # clear contents
print("fig0 is fig2: ",  fig0 is fig2)
print("fig2.texts: ", [t.get_text() for t in fig2.texts])

# The output:
# fig0.texts:  ['A fancy plot']
# fig0 is fig1:  True
# fig1.texts:  ['A fancy plot', 'Really fancy!']
# fig0 is fig2:  True
# fig2.texts:  []
```

### Specify minimum value to format as scalar for `LogFormatterMathtext`

*LogFormatterMathtext* now includes the option to specify a minimum value exponent to format as a scalar (i.e., 0.001 instead of $10^{-3}$).

### New quiverkey angle keyword argument

Plotting a *quiverkey()* now admits the `angle` keyword argument, which sets the angle at which to draw the key arrow.

### Colormap reversed method

The methods *matplotlib.colors.LinearSegmentedColormap.reversed()* and *matplotlib.colors.ListedColormap.reversed()* return a reversed instance of the Colormap. This implements a way for any Colormap to be reversed.

### `artist.setp` (and `pyplot.setp`) accept a *file* argument

The argument is keyword-only. It allows an output file other than `sys.stdout` to be specified. It works exactly like the *file* argument to `print`.

### `streamplot` streamline generation more configurable

The starting point, direction, and length of the stream lines can now be configured. This allows to follow the vector field for a longer time and can enhance the visibility of the flow pattern in some use cases.

### `Axis.set_tick_params` now responds to *rotation*

Bulk setting of tick label rotation is now possible via *tick_params()* using the *rotation* keyword.

```
ax.tick_params(which='both', rotation=90)
```

### Ticklabels are turned off instead of being invisible

Internally, the *Tick*'s `matplotlib.axis.Tick.label1On` attribute is now used to hide tick labels instead of setting the visibility on the tick label objects. This improves overall performance and fixes some issues. As a consequence, in case those labels ought to be shown, *tick_params()* needs to be used, e.g.

```
ax.tick_params(labelbottom=True)
```

### Shading in 3D bar plots

A new `shade` parameter has been added the 3D *`bar`* plotting method. The default behavior remains to shade the bars, but now users have the option of setting `shade` to `False`.

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

x = np.arange(2)
y = np.arange(3)
x2d, y2d = np.meshgrid(x, y)
x, y = x2d.ravel(), y2d.ravel()
z = np.zeros_like(x)
dz = x + y

fig = plt.figure(figsize=(4, 6))
ax1 = fig.add_subplot(2, 1, 1, projection='3d')
ax1.bar3d(x, y, z, 1, 1, dz, shade=True)
ax1.set_title('Shading On')

ax2 = fig.add_subplot(2, 1, 2, projection='3d')
ax2.bar3d(x, y, z, 1, 1, dz, shade=False)
ax2.set_title('Shading Off')

plt.show()
```

### New `which` Parameter for `autofmt_xdate`

A `which` parameter now exists for the method *`autofmt_xdate()`*. This allows a user to format `major`, `minor` or `both` tick labels selectively. The default behavior will rotate and align the `major` tick labels.

```
fig.autofmt_xdate(bottom=0.2, rotation=30, ha='right', which='minor')
```

### New Figure Parameter for `subplot2grid`

A `fig` parameter now exists for the function *`subplot2grid()`*. This allows a user to specify the figure where the subplots will be created. If `fig` is `None` (default) then the method will use the current figure retrieved by *`gcf()`*.

```
subplot2grid(shape, loc, rowspan=1, colspan=1, fig=myfig)
```

### Interpolation in `fill_betweenx`

The `interpolate` parameter now exists for the method *`fill_betweenx()`*. This allows a user to interpolate the data and fill the areas in the crossover points, similarly to *`fill_between()`*.

### New keyword argument `sep` for EngFormatter

A new `sep` keyword argument has been added to *`EngFormatter`* and provides a means to define the string that will be used between the value and its unit. The default string is `" "`, which preserves the former behavior. Additionally, the separator is now present between the value and its unit even in the absence of SI prefix. There was formerly a bug that was causing strings like `"3.14V"` to be returned instead of the expected `"3.14 V"` (with the default behavior).

### Extend `MATPLOTLIBRC` behavior

The environmental variable can now specify the full file path or the path to a directory containing a `matplotlibrc` file.

### `density` kwarg to hist

The *`hist()`* method now prefers `density` to `normed` to control if the histogram should be normalized, following a change upstream to NumPy. This will reduce confusion as the behavior has always been that the integral of the histogram is 1 (rather than sum or maximum value).

### Internals

### New TransformedPatchPath caching object

A newly added *`TransformedPatchPath`* provides a means to transform a *`Patch`* into a *`Path`* via a *`Transform`* while caching the resulting path. If neither the patch nor the transform have changed, a cached copy of the path is returned.

This class differs from the older *`TransformedPath`* in that it is able to refresh itself based on the underlying patch while the older class uses an immutable path.

### Abstract base class for movie writers

The new *AbstractMovieWriter* class defines the API required by a class that is to be used as the `writer` in the *matplotlib.animation.Animation.save()* method. The existing *MovieWriter* class now derives from the new abstract base class.

### Stricter validation of line style rcParams

The validation of rcParams that are related to line styles (`lines.linestyle`, `boxplot.*.linestyle`, `grid.linestyle` and `contour.negative_linestyle`) now effectively checks that the values are valid line styles. Strings like `'dashed'` or `'--'` are accepted, as well as even-length sequences of on-off ink like `[1, 1.65]`. In this latter case, the offset value is handled internally and should *not* be provided by the user.

The new validation scheme replaces the former one used for the `contour.negative_linestyle` rcParams, that was limited to `'solid'` and `'dashed'` line styles.

The validation is case-insensitive. The following are now valid:

```
grid.linestyle             : (1, 3)    # loosely dotted grid lines
contour.negative_linestyle : dashdot   # previously only solid or dashed
```

### pytest

The automated tests have been switched from nose to pytest.

### Performance

### Path simplification updates

Line simplification controlled by the `path.simplify` and `path.simplify_threshold` parameters has been improved. You should notice better rendering performance when plotting large amounts of data (as long as the above parameters are set accordingly). Only the line segment portion of paths will be simplified -- if you are also drawing markers and experiencing problems with rendering speed, you should consider using the `markevery` option to *plot*. See the *Performance* section in the usage tutorial for more information.

The simplification works by iteratively merging line segments into a single vector until the next line segment's perpendicular distance to the vector (measured in display-coordinate space) is greater than the `path.simplify_threshold` parameter. Thus, higher values of `path.simplify_threshold` result in quicker rendering times. If you are plotting just to explore data and not for publication quality, pixel perfect plots, then a value of `1.0` can be safely used. If you want to make sure your plot reflects your data *exactly*, then you should set `path.simplify` to false and/or `path.simplify_threshold` to `0`. Matplotlib currently defaults to a conservative value of `1/9`, smaller values are unlikely to cause any visible differences in your plots.

#### Implement intersects_bbox in c++

*intersects_bbox()* has been implemented in c++ which improves the performance of automatically placing the legend.

### 9.12.2 API Changes in 2.1.2

#### `Figure.legend` no longer checks for repeated lines to ignore

*matplotlib.figure.Figure.legend* used to check if a line had the same label as an existing legend entry. If it also had the same line color or marker color legend didn't add a new entry for that line. However, the list of conditions was incomplete, didn't handle RGB tuples, didn't handle linewidths or linestyles etc.

This logic did not exist in *axes.Axes.legend*. It was included (erroneously) in Matplotlib 2.1.1 when the legend argument parsing was unified PR #9324. This change removes that check in *axes.Axes.legend* again to restore the old behavior.

This logic has also been dropped from *Figure.legend*, where it was previously undocumented. Repeated lines with the same label will now each have an entry in the legend. If you do not want the duplicate entries, don't add a label to the line, or prepend the label with an underscore.

### 9.12.3 API Changes in 2.1.1

#### Default behavior of log scales reverted to clip <= 0 values

The change it 2.1.0 to mask in logscale by default had more disruptive changes than anticipated and has been reverted, however the clipping is now done in a way that fixes the issues that motivated changing the default behavior to `'mask'`.

As a side effect of this change, error bars which go negative now work as expected on log scales.

### 9.12.4 API Changes in 2.1.0

#### Default behavior of log scales changed to mask <= 0 values

Calling *matplotlib.axes.Axes.set_xscale* or *matplotlib.axes.Axes.set_yscale* now uses 'mask' as the default method to handle invalid values (as opposed to 'clip'). This means that any values <= 0 on a log scale will not be shown.

Previously they were clipped to a very small number and shown.

### `matplotlib.cbook.CallbackRegistry.process()` suppresses exceptions by default

Matplotlib uses instances of *`CallbackRegistry`* as a bridge between user input event from the GUI and user callbacks. Previously, any exceptions raised in a user call back would bubble out of the `process` method, which is typically in the GUI event loop. Most GUI frameworks simple print the traceback to the screen and continue as there is not always a clear method of getting the exception back to the user. However PyQt5 now exits the process when it receives an un-handled python exception in the event loop. Thus, *`process()`* now suppresses and prints tracebacks to stderr by default.

What *`process()`* does with exceptions is now user configurable via the `exception_handler` attribute and kwarg. To restore the previous behavior pass `None`

```
cb = CallbackRegistry(exception_handler=None)
```

A function which take and `Exception` as its only argument may also be passed

```
def maybe_reraise(exc):
    if isinstance(exc, RuntimeError):
        pass
    else:
        raise exc

cb = CallbackRegistry(exception_handler=maybe_reraise)
```

### Improved toggling of the axes grids

The `g` key binding now switches the states of the `x` and `y` grids independently (by cycling through all four on/off combinations).

The new `G` key binding switches the states of the minor grids.

Both bindings are disabled if only a subset of the grid lines (in either direction) is visible, to avoid making irreversible changes to the figure.

### Ticklabels are turned off instead of being invisible

Internally, the *`Tick`*'s `~matplotlib.axis.Tick.label1On` attribute is now used to hide tick labels instead of setting the visibility on the tick label objects. This improves overall performance and fixes some issues. As a consequence, in case those labels ought to be shown, *`tick_params()`* needs to be used, e.g.

```
ax.tick_params(labelbottom=True)
```

### Removal of warning on empty legends

*pyplot.legend* used to issue a warning when no labeled artist could be found. This warning has been removed.

### More accurate legend autopositioning

Automatic positioning of legends now prefers using the area surrounded by a *Line2D* rather than placing the legend over the line itself.

### Cleanup of stock sample data

The sample data of stocks has been cleaned up to remove redundancies and increase portability. The `AAPL.dat.gz`, `INTC.dat.gz` and `aapl.csv` files have been removed entirely and will also no longer be available from *matplotlib.cbook.get_sample_data*. If a CSV file is required, we suggest using the `msft.csv` that continues to be shipped in the sample data. If a NumPy binary file is acceptable, we suggest using one of the following two new files. The `aapl.npy.gz` and `goog.npy` files have been replaced by `aapl.npz` and `goog.npz`, wherein the first column's type has changed from `datetime.date` to `numpy.datetime64` for better portability across Python versions. Note that Matplotlib does not fully support `numpy.datetime64` as yet.

### Updated qhull to 2015.2

The version of qhull shipped with Matplotlib, which is used for Delaunay triangulation, has been updated from version 2012.1 to 2015.2.

### Improved Delaunay triangulations with large offsets

Delaunay triangulations now deal with large x,y offsets in a better way. This can cause minor changes to any triangulations calculated using Matplotlib, i.e. any use of *matplotlib.tri.Triangulation* that requests that a Delaunay triangulation is calculated, which includes *matplotlib.pyplot.tricontour*, *matplotlib.pyplot.tricontourf*, *matplotlib.pyplot.tripcolor*, *matplotlib.pyplot.triplot*, `matplotlib.mlab.griddata` and *mpl_toolkits.mplot3d.axes3d.Axes3D.plot_trisurf*.

### Use `backports.functools_lru_cache` instead of `functools32`

It's better maintained and more widely used (by pylint, jaraco, etc).

### `cbook.is_numlike` only performs an instance check

`matplotlib.cbook.is_numlike` now only checks that its argument is an instance of `(numbers.Number, np.Number)`. In particular, this means that arrays are now not num-like.

### Elliptical arcs now drawn between correct angles

The *matplotlib.patches.Arc* patch is now correctly drawn between the given angles.

Previously a circular arc was drawn and then stretched into an ellipse, so the resulting arc did not lie between *theta1* and *theta2*.

### `-d$backend` no longer sets the backend

It is no longer possible to set the backend by passing `-d$backend` at the command line. Use the `MPLBACKEND` environment variable instead.

### Path.intersects_bbox always treats the bounding box as filled

Previously, when `Path.intersects_bbox` was called with `filled` set to `False`, it would treat both the path and the bounding box as unfilled. This behavior was not well documented and it is usually not the desired behavior, since bounding boxes are used to represent more complex shapes located inside the bounding box. This behavior has now been changed: when `filled` is `False`, the path will be treated as unfilled, but the bounding box is still treated as filled. The old behavior was arguably an implementation bug.

When `Path.intersects_bbox` is called with `filled` set to `True` (the default value), there is no change in behavior. For those rare cases where `Path.intersects_bbox` was called with `filled` set to `False` and where the old behavior is actually desired, the suggested workaround is to call `Path.intersects_path` with a rectangle as the path:

```
from matplotlib.path import Path
from matplotlib.transforms import Bbox, BboxTransformTo
rect = Path.unit_rectangle().transformed(BboxTransformTo(bbox))
result = path.intersects_path(rect, filled=False)
```

### WX no longer calls generates `IdleEvent` events or calls `idle_event`

Removed unused private method `_onIdle` from `FigureCanvasWx`.

The `IdleEvent` class and `FigureCanvasBase.idle_event` method will be removed in 2.2

### Correct scaling of `magnitude_spectrum()`

The functions *matplotlib.mlab.magnitude_spectrum()* and *matplotlib.pyplot.magnitude_spectrum()* implicitly assumed the sum of windowing function values to be one. In Matplotlib and Numpy the standard windowing functions are scaled to have maximum value of one, which usually results in a sum of the order of n/2 for a n-point signal. Thus the amplitude scaling `magnitude_spectrum()` was off by that amount when using standard windowing functions (Bug 8417 ). Now the behavior is consistent with *matplotlib.pyplot.psd()* and `scipy.signal.welch()`. The following example demonstrates the new and old scaling:

```
import matplotlib.pyplot as plt
import numpy as np

tau, n = 10, 1024  # 10 second signal with 1024 points
T = tau/n  # sampling interval
t = np.arange(n)*T

a = 4  # amplitude
x = a*np.sin(40*np.pi*t)  # 20 Hz sine with amplitude a

# New correct behavior: Amplitude at 20 Hz is a/2
plt.magnitude_spectrum(x, Fs=1/T, sides='onesided', scale='linear')

# Original behavior: Amplitude at 20 Hz is (a/2)*(n/2) for a Hanning window
w = np.hanning(n)  # default window is a Hanning window
plt.magnitude_spectrum(x*np.sum(w), Fs=1/T, sides='onesided', scale='linear')
```

### Change to signatures of `bar()` & `barh()`

For 2.0 the *default value of *align** changed to `'center'`. However this caused the signature of *bar()* and *barh()* to be misleading as the first parameters were still *left* and *bottom* respectively:

```
bar(left, height, *, align='center', **kwargs)
barh(bottom, width, *, align='center', **kwargs)
```

despite behaving as the center in both cases. The methods now take `*args, **kwargs` as input and are documented to have the primary signatures of:

```
bar(x, height, *, align='center', **kwargs)
barh(y, width, *, align='center', **kwargs)
```

Passing *left* and *bottom* as keyword arguments to *bar()* and *barh()* respectively will warn. Support will be removed in Matplotlib 3.0.

### Font cache as json

The font cache is now saved as json, rather than a pickle.

### Invalid (Non-finite) Axis Limit Error

When using *set_xlim()* and *set_ylim()*, passing non-finite values now results in a `ValueError`. The previous behavior resulted in the limits being erroneously reset to `(-0.001, 0.001)`.

### `scatter` and `Collection` offsets are no longer implicitly flattened

*Collection* (and thus both 2D *scatter* and 3D *scatter*) no longer implicitly flattens its offsets. As a consequence, `scatter`'s `x` and `y` arguments can no longer be 2+-dimensional arrays.

## Deprecations

### `GraphicsContextBase`'s `linestyle` property.

The `GraphicsContextBase.get_linestyle` and `GraphicsContextBase.set_linestyle` methods, which had no effect, have been deprecated. All of the backends Matplotlib ships use `GraphicsContextBase.get_dashes` and `GraphicsContextBase.set_dashes` which are more general. Third-party backends should also migrate to the `*_dashes` methods.

### `NavigationToolbar2.dynamic_update`

Use *draw_idle* method on the `Canvas` instance instead.

## Testing

`matplotlib.testing.noseclasses` is deprecated and will be removed in 2.3

### `EngFormatter` *num* arg as string

Passing a string as *num* argument when calling an instance of *matplotlib.ticker.EngFormatter* is deprecated and will be removed in 2.3.

#### `mpl_toolkits.axes_grid` module

All functionally from `mpl_toolkits.axes_grid` can be found in either *`mpl_toolkits.axes_grid1`* or *`mpl_toolkits.axisartist`*. Axes classes from `mpl_toolkits.axes_grid` based on `Axis` from *`mpl_toolkits.axisartist`* can be found in *`mpl_toolkits.axisartist`*.

#### `Axes` collision in `Figure.add_axes`

Adding an axes instance to a figure by using the same arguments as for a previous axes instance currently reuses the earlier instance. This behavior has been deprecated in Matplotlib 2.1. In a future version, a *new* instance will always be created and returned. Meanwhile, in such a situation, a deprecation warning is raised by `matplotlib.figure.AxesStack`.

This warning can be suppressed, and the future behavior ensured, by passing a *unique* label to each axes instance. See the docstring of *`add_axes()`* for more information.

Additional details on the rationale behind this deprecation can be found in #7377 and #9024.

### Former validators for `contour.negative_linestyle`

The former public validation functions `validate_negative_linestyle` and `validate_negative_linestyle_legacy` will be deprecated in 2.1 and may be removed in 2.3. There are no public functions to replace them.

#### `cbook`

Many unused or near-unused *`matplotlib.cbook`* functions and classes have been deprecated: `converter`, `tostr`, `todatetime`, `todate`, `tofloat`, `toint`, `unique`, `is_string_like`, `is_sequence_of_strings`, `is_scalar`, `Sorter`, `Xlator`, `soundex`, `Null`, `dict_delall`, `RingBuffer`, `get_split_ind`, `wrap`, `get_recursive_filelist`, `pieces`, `exception_to_str`, `allequal`, `alltrue`, `onetrue`, `allpairs`, `finddir`, `reverse_dict`, `restrict_dict`, `issubclass_safe`, `recursive_remove`, `unmasked_index_ranges`.

### Code Removal

### qt4_compat.py

Moved to `qt_compat.py`. Renamed because it now handles Qt5 as well.

### Previously Deprecated methods

The `GraphicsContextBase.set_graylevel`, `FigureCanvasBase.onHilite` and `mpl_toolkits.axes_grid1.mpl_axes.Axes.toggle_axisline` methods have been removed.

The `ArtistInspector.findobj` method, which was never working due to the lack of a `get_children` method, has been removed.

The deprecated `point_in_path`, `get_path_extents`, `point_in_path_collection`, `path_intersects_path`, `convert_path_to_polygons`, `cleanup_path` and `clip_path_to_rect` functions in the `matplotlib.path` module have been removed. Their functionality remains exposed as methods on the `Path` class.

The deprecated `Artist.get_axes` and `Artist.set_axes` methods have been removed

The `matplotlib.backends.backend_ps.seq_allequal` function has been removed. Use `np.array_equal` instead.

The deprecated `matplotlib.rcsetup.validate_maskedarray`, `matplotlib.rcsetup.deprecate_savefig_extension` and `matplotlib.rcsetup.validate_tkpythoninspect` functions, and associated `savefig.extension` and `tk.pythoninspect` rcparams entries have been removed.

The keyword argument *resolution* of *`matplotlib.projections.polar.PolarAxes`* has been removed. It has deprecation with no effect from version *0.98.x*.

#### `Axes.set_aspect("normal")`

Support for setting an `Axes`'s aspect to `"normal"` has been removed, in favor of the synonym `"auto"`.

#### `shading` kwarg to `pcolor`

The `shading` kwarg to *`pcolor`* has been removed. Set `edgecolors` appropriately instead.

### Functions removed from the `lines` module

The *`matplotlib.lines`* module no longer imports the `pts_to_prestep`, `pts_to_midstep` and `pts_to_poststep` functions from *`matplotlib.cbook`*.

#### PDF backend functions

The methods `embedTeXFont` and `tex_font_mapping` of *matplotlib.backends.backend_pdf.PdfFile* have been removed. It is unlikely that external users would have called these methods, which are related to the font system internal to the PDF backend.

#### matplotlib.delaunay

Remove the delaunay triangulation code which is now handled by Qhull via *matplotlib.tri*.

## 9.13 Version 2.0

### 9.13.1 What's new in Matplotlib 2.0 (Jan 17, 2017)

**Note:** Matplotlib 2.0 supports Python 2.7, and 3.4+

#### Default style changes

The major changes in v2.0 are related to overhauling the default styles.

#### Changes to the default style

The most important changes in matplotlib 2.0 are the changes to the default style.

While it is impossible to select the best default for all cases, these are designed to work well in the most common cases.

A 'classic' style sheet is provided so reverting to the 1.x default values is a single line of python

```
import matplotlib.style
import matplotlib as mpl
mpl.style.use('classic')
```

See *The matplotlibrc file* for details about how to persistently and selectively revert many of these changes.

**Table of Contents**


- *Colors, color cycles, and colormaps*
  - *Colors in default property cycle*
  - *Colormap*

  - *Interactive figures*
  - *Grid lines*
- *Figure size, font size, and screen dpi*
- *Plotting functions*
  - `scatter`
  - `plot`
  - `errorbar`
  - `boxplot`
  - `fill_between` and `fill_betweenx`
  - *Patch edges and color*
  - `hexbin`
  - `bar` and `barh`
- *Hatching*
- *Fonts*
  - *Normal text*
  - *Math text*
- *Legends*
- *Image*
  - *Interpolation*
  - *Colormapping pipeline*
  - *Shading*
- *Plot layout*
  - *Auto limits*
  - *Z-order*
  - *Ticks*
  - *Tick label formatting*
- *mplot3d*

### Colors, color cycles, and colormaps

#### Colors in default property cycle

The colors in the default property cycle have been changed from `['b', 'g', 'r', 'c', 'm', 'y', 'k']` to the category10 color palette used by Vega and d3 originally developed at Tableau.

In addition to changing the colors, an additional method to specify colors was added. Previously, the default colors were the single character short-hand notations for red, green, blue, cyan, magenta, yellow, and black. This made them easy to type and usable in the abbreviated style string in `plot`, however the new default colors are only specified via hex values. To access these colors outside of the property cycling the notation for colors `'CN'`, where `N` takes values 0-9, was added to denote the first 10 colors in *`rcParams["axes.prop_cycle"]`* (default: `cycler('color', ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728', '#9467bd', '#8c564b', '#e377c2', '#7f7f7f', '#bcbd22', '#17becf'])`). See *Specifying colors* for more details.

To restore the old color cycle use

```
from cycler import cycler
mpl.rcParams['axes.prop_cycle'] = cycler(color='bgrcmyk')
```

or set

```
axes.prop_cycle    : cycler('color', 'bgrcmyk')
```

in your `matplotlibrc` file.

#### Colormap

The new default colormap used by *`matplotlib.cm.ScalarMappable`* instances is 'viridis' (aka option D).

For an introduction to color theory and how 'viridis' was generated watch Nathaniel Smith and Stéfan van der Walt's talk from SciPy2015. See here for many more details about the other alternatives and the tools used to create the color map. For details on all of the colormaps available in matplotlib see *Choosing Colormaps in Matplotlib*.

The previous default can be restored using

```
mpl.rcParams['image.cmap'] = 'jet'
```

or setting

```
image.cmap    : 'jet'
```

in your `matplotlibrc` file; however this is strongly discouraged.

### Interactive figures

The default interactive figure background color has changed from grey to white, which matches the default background color used when saving.

The previous defaults can be restored by

```
mpl.rcParams['figure.facecolor'] = '0.75'
```

or by setting

```
figure.facecolor : '0.75'
```

in your `matplotlibrc` file.

### Grid lines

The default style of grid lines was changed from black dashed lines to thicker solid light grey lines.

The previous default can be restored by using:

```
mpl.rcParams['grid.color'] = 'k'
mpl.rcParams['grid.linestyle'] = ':'
mpl.rcParams['grid.linewidth'] = 0.5
```

or by setting:

```
grid.color       :   k       # grid color
grid.linestyle   :   :       # dotted
grid.linewidth   :   0.5     # in points
```

in your `matplotlibrc` file.

### Figure size, font size, and screen dpi

The default dpi used for on-screen display was changed from 80 dpi to 100 dpi, the same as the default dpi for saving files. Due to this change, the on-screen display is now more what-you-see-is-what-you-get for saved files. To keep the figure the same size in terms of pixels, in order to maintain approximately the same size on the screen, the default figure size was reduced from 8x6 inches to 6.4x4.8 inches. As a consequence of this the default font sizes used for the title, tick labels, and axes labels were reduced to maintain their size relative to the overall size of the figure. By default the dpi of the saved image is now the dpi of the *Figure* instance being saved.

This will have consequences if you are trying to match text in a figure directly with external text.

The previous defaults can be restored by

```
mpl.rcParams['figure.figsize'] = [8.0, 6.0]
mpl.rcParams['figure.dpi'] = 80
mpl.rcParams['savefig.dpi'] = 100

mpl.rcParams['font.size'] = 12
mpl.rcParams['legend.fontsize'] = 'large'
mpl.rcParams['figure.titlesize'] = 'medium'
```

or by setting:

```
figure.figsize   : [8.0, 6.0]
figure.dpi       : 80
savefig.dpi      : 100

font.size        : 12.0
legend.fontsize  : 'large'
figure.titlesize : 'medium'
```

In your `matplotlibrc` file.

In addition, the `forward` kwarg to *`set_size_inches`* now defaults to `True` to improve the interactive experience. Backend canvases that adjust the size of their bound *`matplotlib.figure.Figure`* must pass `forward=False` to avoid circular behavior. This default is not configurable.

### Plotting functions

#### `scatter`

The following changes were made to the default behavior of *`scatter`*

- The default size of the elements in a scatter plot is now based on *`rcParams["lines.markersize"]`* (default: `6.0`) so it is consistent with `plot(X, Y, 'o')`. The old value was 20, and the new value is 36 (6^2).
- Scatter markers no longer have a black edge.
- If the color of the markers is not specified it will follow the property cycle, pulling from the 'patches' cycle on the `Axes`.

The classic default behavior of *`scatter`* can only be recovered through `mpl.style.use('classic')`. The marker size can be recovered via

```
mpl.rcParam['lines.markersize'] = np.sqrt(20)
```

however, this will also affect the default marker size of *`plot`*. To recover the classic behavior on a per-call basis pass the following kwargs:

```
classic_kwargs = {'s': 20, 'edgecolors': 'k', 'c': 'b'}
```

### plot

The following changes were made to the default behavior of *plot*

- the default linewidth increased from 1 to 1.5
- the dash patterns associated with `'--'`, `':'`, and `'-.'` have changed
- the dash patterns now scale with line width

The previous defaults can be restored by setting:

```
mpl.rcParams['lines.linewidth'] = 1.0
mpl.rcParams['lines.dashed_pattern'] = [6, 6]
mpl.rcParams['lines.dashdot_pattern'] = [3, 5, 1, 5]
mpl.rcParams['lines.dotted_pattern'] = [1, 3]
mpl.rcParams['lines.scale_dashes'] = False
```

or by setting:

```
lines.linewidth   : 1.0
lines.dashed_pattern : 6, 6
lines.dashdot_pattern : 3, 5, 1, 5
lines.dotted_pattern : 1, 3
lines.scale_dashes: False
```

in your `matplotlibrc` file.

### errorbar

By default, caps on the ends of errorbars are not present.

This also changes the return value of *errorbar()* as the list of 'caplines' will be empty by default.

The previous defaults can be restored by setting:

```
mpl.rcParams['errorbar.capsize'] = 3
```

or by setting

```
errorbar.capsize : 3
```

in your `matplotlibrc` file.

### boxplot

Previously, boxplots were composed of a mish-mash of styles that were, for better for worse, inherited from Matlab. Most of the elements were blue, but the medians were red. The fliers (outliers) were black plus-symbols ('+') and the whiskers were dashed lines, which created ambiguity if the (solid and black) caps were not drawn.

For the new defaults, everything is black except for the median and mean lines (if drawn), which are set to the first two elements of the current color cycle. Also, the default flier markers are now hollow circles, which maintain the ability of the plus-symbols to overlap without obscuring data too much.

The previous defaults can be restored by setting:

```
mpl.rcParams['boxplot.flierprops.color'] = 'k'
mpl.rcParams['boxplot.flierprops.marker'] = '+'
mpl.rcParams['boxplot.flierprops.markerfacecolor'] = 'none'
mpl.rcParams['boxplot.flierprops.markeredgecolor'] = 'k'
mpl.rcParams['boxplot.boxprops.color'] = 'b'
mpl.rcParams['boxplot.whiskerprops.color'] = 'b'
mpl.rcParams['boxplot.whiskerprops.linestyle'] = '--'
mpl.rcParams['boxplot.medianprops.color'] = 'r'
mpl.rcParams['boxplot.meanprops.color'] = 'r'
mpl.rcParams['boxplot.meanprops.marker'] = '^'
mpl.rcParams['boxplot.meanprops.markerfacecolor'] = 'r'
mpl.rcParams['boxplot.meanprops.markeredgecolor'] = 'k'
mpl.rcParams['boxplot.meanprops.markersize'] = 6
mpl.rcParams['boxplot.meanprops.linestyle'] = '--'
mpl.rcParams['boxplot.meanprops.linewidth'] = 1.0
```

or by setting:

```
boxplot.flierprops.color:           'k'
boxplot.flierprops.marker:          '+'
boxplot.flierprops.markerfacecolor: 'none'
boxplot.flierprops.markeredgecolor: 'k'
boxplot.boxprops.color:             'b'
boxplot.whiskerprops.color:         'b'
boxplot.whiskerprops.linestyle:     '--'
boxplot.medianprops.color:          'r'
boxplot.meanprops.color:            'r'
boxplot.meanprops.marker:           '^'
boxplot.meanprops.markerfacecolor:  'r'
boxplot.meanprops.markeredgecolor:  'k'
boxplot.meanprops.markersize:        6
boxplot.meanprops.linestyle:         '--'
boxplot.meanprops.linewidth:         1.0
```

in your `matplotlibrc` file.

#### `fill_between` and `fill_betweenx`

*`fill_between`* and *`fill_betweenx`* both follow the patch color cycle.

If the facecolor is set via the `facecolors` or `color` keyword argument, then the color is not cycled.

To restore the previous behavior, explicitly pass the keyword argument `facecolors='C0'` to the method call.

### Patch edges and color

Most artists drawn with a patch (`~matplotlib.axes.Axes.bar`, `~matplotlib.axes.Axes.pie`, etc) no longer have a black edge by default. The default face color is now `'C0'` instead of `'b'`.

The previous defaults can be restored by setting:

```
mpl.rcParams['patch.force_edgecolor'] = True
mpl.rcParams['patch.facecolor'] = 'b'
```

or by setting:

```
patch.facecolor        : b
patch.force_edgecolor  : True
```

in your `matplotlibrc` file.

#### `hexbin`

The default value of the *linecolor* keyword argument for *`hexbin`* has changed from `'none'` to `'face'`. If 'none' is now supplied, no line edges are drawn around the hexagons.

#### `bar` and `barh`

The default value of the `align` kwarg for both *`bar`* and *`barh`* is changed from `'edge'` to `'center'`.

To restore the previous behavior explicitly pass the keyword argument `align='edge'` to the method call.

### Hatching

The color of the lines in the hatch is now determined by

- If an edge color is explicitly set, use that for the hatch color
- If the edge color is not explicitly set, use *rcParams["hatch.color"]* (default: 'black') which is looked up at artist creation time.

The width of the lines in a hatch pattern is now configurable by the rcParams *rcParams["hatch.linewidth"]* (default: 1.0), which defaults to 1 point. The old behavior for the line width was different depending on backend:

- PDF: 0.1 pt
- SVG: 1.0 pt
- PS: 1 px
- Agg: 1 px

The old line width behavior cannot be restored across all backends simultaneously, but can be restored for a single backend by setting:

```
mpl.rcParams['hatch.linewidth'] = 0.1  # previous pdf hatch linewidth
mpl.rcParams['hatch.linewidth'] = 1.0  # previous svg hatch linewidth
```

The behavior of the PS and Agg backends was DPI dependent, thus:

```
mpl.rcParams['figure.dpi'] = dpi
mpl.rcParams['savefig.dpi'] = dpi  # or leave as default 'figure'
mpl.rcParams['hatch.linewidth'] = 1.0 / dpi  # previous ps and Agg hatch␣
 ↪linewidth
```

There is no direct API level control of the hatch color or linewidth.

Hatching patterns are now rendered at a consistent density, regardless of DPI. Formerly, high DPI figures would be more dense than the default, and low DPI figures would be less dense. This old behavior cannot be directly restored, but the density may be increased by repeating the hatch specifier.

### Fonts

#### Normal text

The default font has changed from "Bitstream Vera Sans" to "DejaVu Sans". DejaVu Sans has additional international and math characters, but otherwise has the same appearance as Bitstream Vera Sans. Latin, Greek, Cyrillic, Armenian, Georgian, Hebrew, and Arabic are all supported (but right-to-left rendering is still not handled by matplotlib). In addition, DejaVu contains a sub-set of emoji symbols.

See the DejaVu Sans PDF sample for full coverage.

### Math text

The default math font when using the built-in math rendering engine (mathtext) has changed from "Computer Modern" (i.e. LaTeX-like) to "DejaVu Sans". This change has no effect if the TeX backend is used (i.e. `text.usetex` is `True`).

To revert to the old behavior set the:

```
mpl.rcParams['mathtext.fontset'] = 'cm'
mpl.rcParams['mathtext.rm'] = 'serif'
```

or set:

```
mathtext.fontset: cm
mathtext.rm : serif
```

in your `matplotlibrc` file.

This `rcParam` is consulted when the text is drawn, not when the artist is created. Thus all mathtext on a given `canvas` will use the same fontset.

### Legends

- By default, the number of points displayed in a legend is now 1.
- The default legend location is `'best'`, so the legend will be automatically placed in a location to minimize overlap with data.
- The legend defaults now include rounded corners, a lighter boundary, and partially transparent boundary and background.

The previous defaults can be restored by setting:

```
mpl.rcParams['legend.fancybox'] = False
mpl.rcParams['legend.loc'] = 'upper right'
mpl.rcParams['legend.numpoints'] = 2
mpl.rcParams['legend.fontsize'] = 'large'
mpl.rcParams['legend.framealpha'] = None
mpl.rcParams['legend.scatterpoints'] = 3
mpl.rcParams['legend.edgecolor'] = 'inherit'
```

or by setting:

```
legend.fancybox      : False
legend.loc           : upper right
legend.numpoints     : 2      # the number of points in the legend line
legend.fontsize      : large
```

(continues on next page)

(continued from previous page)

```
legend.framealpha    : None    # opacity of legend frame
legend.scatterpoints : 3 # number of scatter points
legend.edgecolor     : inherit   # legend edge color ('inherit'
                                 # means it uses axes.edgecolor)
```

in your `matplotlibrc` file.

### Image

### Interpolation

The default interpolation method for *imshow* is now `'nearest'` and by default it resamples the data (both up and down sampling) before colormapping.

To restore the previous behavior set:

```
mpl.rcParams['image.interpolation'] = 'bilinear'
mpl.rcParams['image.resample'] = False
```

or set:

```
image.interpolation  : bilinear  # see help(imshow) for options
image.resample  : False
```

in your `matplotlibrc` file.

### Colormapping pipeline

Previously, the input data was normalized, then colormapped, and then resampled to the resolution required for the screen. This meant that the final resampling was being done in color space. Because the color maps are not generally linear in RGB space, colors not in the colormap may appear in the final image. This bug was addressed by an almost complete overhaul of the image handling code.

The input data is now normalized, then resampled to the correct resolution (in normalized dataspace), and then colormapped to RGB space. This ensures that only colors from the colormap appear in the final image. (If your viewer subsequently resamples the image, the artifact may reappear.)

The previous behavior cannot be restored.

### Shading

- The default shading mode for light source shading, in `matplotlib.colors.LightSource.shade`, is now `overlay`. Formerly, it was `hsv`.

### Plot layout

### Auto limits

The previous auto-scaling behavior was to find 'nice' round numbers as view limits that enclosed the data limits, but this could produce bad plots if the data happened to fall on a vertical or horizontal line near the chosen 'round number' limit. The new default sets the view limits to 5% wider than the data range.

The size of the padding in the x and y directions is controlled by the `'axes.xmargin'` and `'axes.ymargin'` rcParams respectively. Whether the view limits should be 'round numbers' is controlled by *rcParams["axes.autolimit_mode"]* (default: `'data'`). In the original `'round_number'` mode, the view limits coincide with ticks.

The previous default can be restored by using:

```
mpl.rcParams['axes.autolimit_mode'] = 'round_numbers'
mpl.rcParams['axes.xmargin'] = 0
mpl.rcParams['axes.ymargin'] = 0
```

or setting:

```
axes.autolimit_mode: round_numbers
axes.xmargin: 0
axes.ymargin: 0
```

in your `matplotlibrc` file.

### Z-order

- Ticks and grids are now plotted above solid elements such as filled contours, but below lines. To return to the previous behavior of plotting ticks and grids above lines, set `rcParams['axes.axisbelow'] = False`.

### Ticks

#### Direction

To reduce the collision of tick marks with data, the default ticks now point outward by default. In addition, ticks are now drawn only on the bottom and left spines to prevent a porcupine appearance, and for a cleaner separation between subplots.

To restore the previous behavior set:

```
mpl.rcParams['xtick.direction'] = 'in'
mpl.rcParams['ytick.direction'] = 'in'
mpl.rcParams['xtick.top'] = True
mpl.rcParams['ytick.right'] = True
```

or set:

```
xtick.top: True
xtick.direction: in

ytick.right: True
ytick.direction: in
```

in your `matplotlibrc` file.

#### Number of ticks

The default *Locator* used for the x and y axis is *AutoLocator* which tries to find, up to some maximum number, 'nicely' spaced ticks. The locator now includes an algorithm to estimate the maximum number of ticks that will leave room for the tick labels. By default it also ensures that there are at least two ticks visible.

There is no way, other than using `mpl.style.use('classic')`, to restore the previous behavior as the default. On an axis-by-axis basis you may either control the existing locator via:

```
ax.xaxis.get_major_locator().set_params(nbins=9, steps=[1, 2, 5, 10])
```

or create a new *MaxNLocator*:

```
import matplotlib.ticker as mticker
ax.set_major_locator(mticker.MaxNLocator(nbins=9, steps=[1, 2, 5, 10])
```

The algorithm used by *MaxNLocator* has been improved, and this may change the choice of tick locations in some cases. This also affects *AutoLocator*, which uses `MaxNLocator` internally.

For a log-scaled axis the default locator is the *LogLocator*. Previously the maximum number of ticks was set to 15, and could not be changed. Now there is a *numticks* kwarg for setting the maximum to any integer value, to the string 'auto', or to its default value of None which is equivalent to 'auto'. With the 'auto' setting

the maximum number will be no larger than 9, and will be reduced depending on the length of the axis in units of the tick font size. As in the case of the AutoLocator, the heuristic algorithm reduces the incidence of overlapping tick labels but does not prevent it.

### Tick label formatting

#### `LogFormatter` labeling of minor ticks

Minor ticks on a log axis are now labeled when the axis view limits span a range less than or equal to the interval between two major ticks. See *LogFormatter* for details. The minor tick labeling is turned off when using `mpl.style.use('classic')`, but cannot be controlled independently via *rcParams*.

#### `ScalarFormatter` tick label formatting with offsets

With the default *rcParams["axes.formatter.useoffset"]* (default: `True`), an offset will be used when it will save 4 or more digits. This can be controlled with the new *rcParams["axes.formatter.offset_threshold"]* (default: `4`). To restore the previous behavior of using an offset to save 2 or more digits, use `rcParams['axes.formatter.offset_threshold'] = 2`.

#### `AutoDateFormatter` format strings

The default date formats are now all based on ISO format, i.e., with the slowest-moving value first. The date formatters are configurable through the `date.autoformatter.*` rcParams.

| Threshold (tick interval >= than) | rcParam | classic | v2.0 |
|---|---|---|---|
| 365 days | `'date.autoformatter.year'` | `'%Y'` | `'%Y'` |
| 30 days | `'date.autoformatter.month'` | `'%b %Y'` | `'%Y-%m'` |
| 1 day | `'date.autoformatter.day'` | `'%b %d %Y'` | `'%Y-%m-%d'` |
| 1 hour | `'date.autoformatter.hour'` | `'%H:%M:%S'` | `'%H:%M'` |
| 1 minute | `'date.autoformatter.minute'` | `'%H:%M:%S.%f'` | `'%H:%M:%S'` |
| 1 second | `'date.autoformatter.second'` | `'%H:%M:%S.%f'` | `'%H:%M:%S'` |
| 1 microsecond | `'date.autoformatter.microsecond'` | `'%H:%M:%S.%f'` | `'%H:%M:%S.%f'` |

Python's `%x` and `%X` date formats may be of particular interest to format dates based on the current locale.

The previous default can be restored by:

```
mpl.rcParams['date.autoformatter.year'] = '%Y'
mpl.rcParams['date.autoformatter.month'] = '%b %Y'
mpl.rcParams['date.autoformatter.day'] = '%b %d %Y'
mpl.rcParams['date.autoformatter.hour'] = '%H:%M:%S'
mpl.rcParams['date.autoformatter.minute'] = '%H:%M:%S.%f'
mpl.rcParams['date.autoformatter.second'] = '%H:%M:%S.%f'
mpl.rcParams['date.autoformatter.microsecond'] = '%H:%M:%S.%f'
```

or setting

```
date.autoformatter.year   : %Y
date.autoformatter.month  : %b %Y
date.autoformatter.day    : %b %d %Y
date.autoformatter.hour   : %H:%M:%S
date.autoformatter.minute : %H:%M:%S.%f
date.autoformatter.second : %H:%M:%S.%f
date.autoformatter.microsecond : %H:%M:%S.%f
```

in your `matplotlibrc` file.

### mplot3d

- mplot3d now obeys some style-related rcParams, rather than using hard-coded defaults. These include:
  - xtick.major.width
  - ytick.major.width
  - xtick.color
  - ytick.color
  - axes.linewidth
  - axes.edgecolor
  - grid.color
  - grid.linewidth
  - grid.linestyle

### Improved color conversion API and RGBA support

The *colors* gained a new color conversion API with full support for the alpha channel. The main public functions are *is_color_like()*, *matplotlib.colors.to_rgba()*, *matplotlib.colors.to_rgba_array()* and *to_hex()*. RGBA quadruplets are encoded in hex format as "#rrggbbaa".

A side benefit is that the Qt options editor now allows setting the alpha channel of the artists as well.

### New Configuration (rcParams)

New rcparams added

| Parameter | Description |
|---|---|
| *rcParams["date.autoformatter.year"]* (default: '%Y') | format string for 'year' scale dates |
| *rcParams["date.autoformatter.month"]* (default: '%Y-%m') | format string for 'month' scale dates |
| *rcParams["date.autoformatter.day"]* (default: '%Y-%m-%d') | format string for 'day' scale dates |
| *rcParams["date.autoformatter.hour"]* (default: '%m-%d %H') | format string for 'hour' scale times |
| *rcParams["date.autoformatter.minute"]* (default: '%d %H:%M') | format string for 'minute' scale times |
| *rcParams["date.autoformatter.second"]* (default: '%H:%M:%S') | format string for 'second' scale times |
| *rcParams["date.autoformatter.microsecond"]* (default: '%M:%S.%f') | format string for 'microsecond' scale times |
| *rcParams["scatter.marker"]* (default: 'o') | default marker for scatter plot |
| *rcParams["svg.hashsalt"]* (default: None) | see note |
| *rcParams["xtick.top"]* (default: False), *rcParams["xtick.major.top"]* (default: True) *rcParams["xtick.minor.top"]* (default: True), *rcParams["xtick.bottom"]* (default: True), *rcParams["xtick.major.bottom"]* (default: True) *rcParams["xtick.minor.bottom"]* (default: True), *rcParams["ytick.left"]* (default: True), *rcParams["ytick.minor.left"]* (default: True), *rcParams["ytick.major.left"]* (default: True) *rcParams["ytick.right"]* (default: False), *rcParams["ytick.minor.right"]* (default: True), *rcParams["ytick.major.right"]* (default: True) | Control where major and minor ticks are drawn. The global values are anded with the corresponding major/minor values. corresponding major/minor values. |
| *rcParams["hist.bins"]* (default: 10) | The default number of bins to use in *hist*. This can be an int, a list of floats, or 'auto' if numpy >= 1.11 is installed. |
| *rcParams["lines.scale_dashes"]* (default: True) | Whether the line dash patterns should scale with linewidth. |
| *rcParams["axes.formatter.offset_threshold"]* (default: 4) | Minimum number of digits saved in tick labels that triggers using an offset. |

### Added `svg.hashsalt` key to rcParams

If `svg.hashsalt` is `None` (which it is by default), the svg backend uses `uuid4` to generate the hash salt. If it is not `None`, it must be a string that is used as the hash salt instead of `uuid4`. This allows for deterministic SVG output.

### Removed the `svg.image_noscale` rcParam

As a result of the extensive changes to image handling, the `svg.image_noscale` rcParam has been removed. The same functionality may be achieved by setting `interpolation='none'` on individual images or globally using the `image.interpolation` rcParam.

### Qualitative colormaps

ColorBrewer's "qualitative" colormaps ("Accent", "Dark2", "Paired", "Pastel1", "Pastel2", "Set1", "Set2", "Set3") were intended for discrete categorical data, with no implication of value, and therefore have been converted to `ListedColormap` instead of `LinearSegmentedColormap`, so the colors will no longer be interpolated and they can be used for choropleths, labeled image features, etc.

### Axis offset label now responds to *labelcolor*

Axis offset labels are now colored the same as axis tick markers when *labelcolor* is altered.

### Improved offset text choice

The default offset-text choice was changed to only use significant digits that are common to all ticks (e.g. 1231..1239 -> 1230, instead of 1231), except when they straddle a relatively large multiple of a power of ten, in which case that multiple is chosen (e.g. 1999..2001->2000).

### Style parameter blacklist

In order to prevent unexpected consequences from using a style, style files are no longer able to set parameters that affect things unrelated to style. These parameters include:

```
'interactive', 'backend', 'backend.qt4', 'webagg.port',
'webagg.port_retries', 'webagg.open_in_browser', 'backend_fallback',
'toolbar', 'timezone', 'datapath', 'figure.max_open_warning',
'savefig.directory', 'tk.window_focus', 'docstring.hardcopy'
```

### Change in default font

The default font used by matplotlib in text has been changed to DejaVu Sans and DejaVu Serif for the sans-serif and serif families, respectively. The DejaVu font family is based on the previous matplotlib default --Bitstream Vera-- but includes a much wider range of characters.

The default mathtext font has been changed from Computer Modern to the DejaVu family to maintain consistency with regular text. Two new options for the `mathtext.fontset` configuration parameter have been added: `dejavusans` (default) and `dejavuserif`. Both of these options use DejaVu glyphs whenever possible and fall back to STIX symbols when a glyph is not found in DejaVu. To return to the previous behavior, set the rcParam `mathtext.fontset` to `cm`.

### Faster text rendering

Rendering text in the Agg backend is now less fuzzy and about 20% faster to draw.

### Improvements for the Qt figure options editor

Various usability improvements were implemented for the Qt figure options editor, among which:

- Line style entries are now sorted without duplicates.
- The colormap and normalization limits can now be set for images.
- Line edits for floating values now display only as many digits as necessary to avoid precision loss. An important bug was also fixed regarding input validation using Qt5 and a locale where the decimal separator is ",".
- The axes selector now uses shorter, more user-friendly names for axes, and does not crash if there are no axes.
- Line and image entries using the default labels ("_lineX", "_imageX") are now sorted numerically even when there are more than 10 entries.

### Improved image support

Prior to version 2.0, matplotlib resampled images by first applying the colormap and then resizing the result. Since the resampling was performed on the colored image, this introduced colors in the output image that didn't actually exist in the colormap. Now, images are resampled first (and entirely in floating-point, if the input image is floating-point), and then the colormap is applied.

In order to make this important change, the image handling code was almost entirely rewritten. As a side effect, image resampling uses less memory and fewer datatype conversions than before.

The experimental private feature where one could "skew" an image by setting the private member `_image_skew_coordinate` has been removed. Instead, images will obey the transform of the axes on which they are drawn.

### Non-linear scales on image plots

*imshow* now draws data at the requested points in data space after the application of non-linear scales.

The image on the left demonstrates the new, correct behavior. The old behavior can be recreated using *pcolormesh* as demonstrated on the right.

This can be understood by analogy to plotting a histogram with linearly spaced bins with a logarithmic x-axis. Equal sized bins will be displayed as wider for small $x$ and narrower for large $x$.

### Support for HiDPI (Retina) displays in the NbAgg and WebAgg backends

The NbAgg and WebAgg backends will now use the full resolution of your high-pixel-density display.

### Change in the default animation codec

The default animation codec has been changed from `mpeg4` to `h264`, which is more efficient. It can be set via the `animation.codec` rcParam.

### Deprecated support for mencoder in animation

The use of mencoder for writing video files with mpl is problematic; switching to ffmpeg is strongly advised. All support for mencoder will be removed in version 2.2.

### Boxplot Zorder Keyword Argument

The *zorder* parameter now exists for *boxplot*. This allows the zorder of a boxplot to be set in the plotting function call.

```
boxplot(np.arange(10), zorder=10)
```

### Filled + and x markers

New fillable *plus* and *x* markers have been added. See the *markers* module and *marker reference* examples.

#### *rcount* and *ccount* for `plot_surface`

As of v2.0, mplot3d's *plot_surface* now accepts *rcount* and *ccount* arguments for controlling the sampling of the input data for plotting. These arguments specify the maximum number of evenly spaced samples to take from the input data. These arguments are also the new default sampling method for the function, and is considered a style change.

The old *rstride* and *cstride* arguments, which specified the size of the evenly spaced samples, become the default when 'classic' mode is invoked, and are still available for use. There are no plans for deprecating these arguments.

### Streamplot Zorder Keyword Argument Changes

The `zorder` parameter for *streamplot* now has default value of `None` instead of `2`. If `None` is given as `zorder`, *streamplot* has a default `zorder` of `matplotlib.lines.Line2D.zorder`.

### Extension to `matplotlib.backend_bases.GraphicsContextBase`

To support standardizing hatch behavior across the backends we ship the *matplotlib.backend_bases.GraphicsContextBase.get_hatch_color* method as added to *matplotlib.backend_bases.GraphicsContextBase*. This is only used during the render process in the backends we ship so will not break any third-party backends.

If you maintain a third-party backend which extends *GraphicsContextBase* this method is now available to you and should be used to color hatch patterns.

## 9.13.2 API Changes in 2.0.1

### Extensions to `matplotlib.backend_bases.GraphicsContextBase`

To better support controlling the color of hatches, the method *matplotlib.backend_bases.GraphicsContextBase.set_hatch_color* was added to the expected API of `GraphicsContext` classes. Calls to this method are currently wrapped with a `try:...except Attribute:` block to preserve back-compatibility with any third-party backends which do not extend *GraphicsContextBase*.

This value can be accessed in the backends via *matplotlib.backend_bases.GraphicsContextBase.get_hatch_color* (which was added in 2.0 see *Extension to matplotlib.backend_bases.GraphicsContextBase*) and should be used to color the hatches.

In the future there may also be `hatch_linewidth` and `hatch_density` related methods added. It is encouraged, but not required that third-party backends extend *GraphicsContextBase* to make adapting to these changes easier.

#### `afm.get_fontconfig_fonts` returns a list of paths and does not check for existence

`afm.get_fontconfig_fonts` used to return a set of paths encoded as a `{key: 1, ...}` dict, and checked for the existence of the paths. It now returns a list and dropped the existence check, as the same check is performed by the caller (`afm.findSystemFonts`) as well.

#### `bar` now returns rectangles of negative height or width if the corresponding input is negative

*`pyplot.bar`* used to normalize the coordinates of the rectangles that it created, to keep their height and width positives, even if the corresponding input was negative. This normalization has been removed to permit a simpler computation of the correct *`Artist.sticky_edges`* to use.

#### Do not clip line width when scaling dashes

The algorithm to scale dashes was changed to no longer clip the scaling factor: the dash patterns now continue to shrink at thin line widths. If the line width is smaller than the effective pixel size, this may result in dashed lines turning into solid gray-ish lines. This also required slightly tweaking the default patterns for '--', ':', and '.-' so that with the default line width the final patterns would not change.

There is no way to restore the old behavior.

#### Deprecate 'Vega' colormaps

The "Vega" colormaps are deprecated in Matplotlib 2.0.1 and will be removed in Matplotlib 2.2. Use the "tab" colormaps instead: "tab10", "tab20", "tab20b", "tab20c".

### 9.13.3 API Changes in 2.0.0

#### Deprecation and removal

#### Color of Axes

The `axisbg` and `axis_bgcolor` properties on *Axes* have been deprecated in favor of `facecolor`.

#### GTK and GDK backends deprecated

The GDK and GTK backends have been deprecated. These obsolete backends allow figures to be rendered via the GDK API to files and GTK2 figures. They are untested and known to be broken, and their use has been discouraged for some time. Instead, use the `GTKAgg` and `GTKCairo` backends for rendering to GTK2 windows.

### WX backend deprecated

The WX backend has been deprecated. It is untested, and its use has been discouraged for some time. Instead, use the `WXAgg` backend for rendering figures to WX windows.

### CocoaAgg backend removed

The deprecated and not fully functional CocoaAgg backend has been removed.

### `round` removed from TkAgg Backend

The TkAgg backend had its own implementation of the `round` function. This was unused internally and has been removed. Instead, use either the `round` builtin function or `numpy.around`.

### 'hold' functionality deprecated

The 'hold' keyword argument and all functions and methods related to it are deprecated, along with the `axes.hold` rcParams entry. The behavior will remain consistent with the default `hold=True` state that has long been in place. Instead of using a function or keyword argument (`hold=False`) to change that behavior, explicitly clear the axes or figure as needed prior to subsequent plotting commands.

### `Artist.update` has return value

The methods *matplotlib.artist.Artist.set*, *matplotlib.artist.Artist.update*, and the function *matplotlib.artist.setp* now use a common codepath to look up how to update the given artist properties (either using the setter methods or an attribute/property).

The behavior of *matplotlib.artist.Artist.update* is slightly changed to return a list of the values returned from the setter methods to avoid changing the API of *matplotlib.artist.Artist.set* and *matplotlib.artist.setp*.

The keys passed into *matplotlib.artist.Artist.update* are now converted to lower case before being processed, to match the behavior of *matplotlib.artist.Artist.set* and *matplotlib.artist.setp*. This should not break any user code because there are no set methods with capitals in their names, but this puts a constraint on naming properties in the future.

### `Legend` initializers gain *edgecolor* and *facecolor* keyword arguments

The *Legend* background patch (or 'frame') can have its `edgecolor` and `facecolor` determined by the corresponding keyword arguments to the *matplotlib.legend.Legend* initializer, or to any of the methods or functions that call that initializer. If left to their default values of `None`, their values will be taken from `matplotlib.rcParams`. The previously-existing `framealpha` kwarg still controls the alpha transparency of the patch.

### Qualitative colormaps

Colorbrewer's qualitative/discrete colormaps ("Accent", "Dark2", "Paired", "Pastel1", "Pastel2", "Set1", "Set2", "Set3") are now implemented as *ListedColormap* instead of *LinearSegmentedColormap*.

To use these for images where categories are specified as integers, for instance, use:

```
plt.imshow(x, cmap='Dark2', norm=colors.NoNorm())
```

### Change in the `draw_image` backend API

The `draw_image` method implemented by backends has changed its interface.

This change is only relevant if the backend declares that it is able to transform images by returning `True` from `option_scale_image`. See the `draw_image` docstring for more information.

### `matplotlib.ticker.LinearLocator` algorithm update

The *matplotlib.ticker.LinearLocator* is used to define the range and location of axis ticks when the user wants an exact number of ticks. `LinearLocator` thus differs from the default locator `MaxNLocator`, for which the user specifies a maximum number of intervals rather than a precise number of ticks.

The view range algorithm in `matplotlib.ticker.LinearLocator` has been changed so that more convenient tick locations are chosen. The new algorithm returns a plot view range that is a multiple of the user-requested number of ticks. This ensures tick marks will be located at whole integers more consistently. For example, when both y-axes of a``twinx`` plot use `matplotlib.ticker.LinearLocator` with the same number of ticks, their y-tick locations and grid lines will coincide.

### `matplotlib.ticker.LogLocator` gains numticks kwarg

The maximum number of ticks generated by the *LogLocator* can now be controlled explicitly via setting the new 'numticks' kwarg to an integer. By default the kwarg is None which internally sets it to the 'auto' string, triggering a new algorithm for adjusting the maximum according to the axis length relative to the ticklabel font size.

### `matplotlib.ticker.LogFormatter`: two new kwargs

Previously, minor ticks on log-scaled axes were not labeled by default. An algorithm has been added to the *`LogFormatter`* to control the labeling of ticks between integer powers of the base. The algorithm uses two parameters supplied in a kwarg tuple named 'minor_thresholds'. See the docstring for further explanation.

To improve support for axes using *`SymmetricalLogLocator`*, a *linthresh* keyword argument was added.

### New defaults for 3D quiver function in mpl_toolkits.mplot3d.axes3d.py

Matplotlib has both a 2D and a 3D `quiver` function. These changes affect only the 3D function and make the default behavior of the 3D function match the 2D version. There are two changes:

1) The 3D quiver function previously normalized the arrows to be the same length, which makes it unusable for situations where the arrows should be different lengths and does not match the behavior of the 2D function. This normalization behavior is now controlled with the `normalize` keyword, which defaults to False.
2) The `pivot` keyword now defaults to `tail` instead of `tip`. This was done in order to match the default behavior of the 2D quiver function.

To obtain the previous behavior with the 3D quiver function, one can call the function with

```
ax.quiver(x, y, z, u, v, w, normalize=True, pivot='tip')
```

where "ax" is an `Axes3d` object created with something like

```
import mpl_toolkits.mplot3d.axes3d
ax = plt.subplot(111, projection='3d')
```

### Stale figure behavior

Attempting to draw the figure will now mark it as not stale (independent if the draw succeeds). This change is to prevent repeatedly trying to re-draw a figure which is raising an error on draw. The previous behavior would only mark a figure as not stale after a full re-draw succeeded.

### The spectral colormap is now nipy_spectral

The colormaps formerly known as `spectral` and `spectral_r` have been replaced by `nipy_spectral` and `nipy_spectral_r` since Matplotlib 1.3.0. Even though the colormap was deprecated in Matplotlib 1.3.0, it never raised a warning. As of Matplotlib 2.0.0, using the old names raises a deprecation warning. In the future, using the old names will raise an error.

### Default install no longer includes test images

To reduce the size of wheels and source installs, the tests and baseline images are no longer included by default.

To restore installing the tests and images, use a `setup.cfg` with

```
[packages]
tests = True
toolkits_tests = True
```

in the source directory at build/install time.

## 9.14 Version 1.5

### 9.14.1 What's new in Matplotlib 1.5 (Oct 29, 2015)

**Table of Contents**


- *What's new in Matplotlib 1.5 (Oct 29, 2015)*
  - *Interactive OO usage*
  - *Working with labeled data like pandas DataFrames*
  - *Added `axes.prop_cycle` key to rcParams*
  - *New Colormaps*
  - *Styles*
  - *Backends*
  - *Configuration (rcParams)*
  - *Widgets*
  - *New plotting features*
  - *ToolManager*
  - *cbook.is_sequence_of_strings recognizes string objects*
  - *New `close-figs` argument for plot directive*
  - *Support for URL string arguments to `imread`*
  - *Display hook for animations in the IPython notebook*
  - *Prefixed pkg-config for building*

**Note:** matplotlib 1.5 supports Python 2.7, 3.4, and 3.5

### Interactive OO usage

All *Artist*s now keep track of if their internal state has been changed but not reflected in the display ('stale') by a call to `draw`. It is thus possible to pragmatically determine if a given *Figure* needs to be re-drawn in an interactive session.

To facilitate interactive usage a `draw_all` method has been added to `pyplot` which will redraw all of the figures which are 'stale'.

To make this convenient for interactive use matplotlib now registers a function either with IPython's 'post_execute' event or with the displayhook in the standard python REPL to automatically call `plt.draw_all` just before control is returned to the REPL. This ensures that the draw command is deferred and only called once.

The upshot of this is that for interactive backends (including `%matplotlib notebook`) in interactive mode (with `plt.ion()`)

```
import matplotlib.pyplot as plt
fig, ax = plt.subplots()
ln, = ax.plot([0, 1, 4, 9, 16])
plt.show()
ln.set_color('g')
```

will automatically update the plot to be green. Any subsequent modifications to the `Artist` objects will do likewise.

This is the first step of a larger consolidation and simplification of the pyplot internals.

### Working with labeled data like pandas DataFrames

Plot methods which take arrays as inputs can now also work with labeled data and unpack such data.

This means that the following two examples produce the same plot:

Example

```
df = pandas.DataFrame({"var1":[1,2,3,4,5,6], "var2":[1,2,3,4,5,6]})
plt.plot(df["var1"], df["var2"])
```

Example

```
plt.plot("var1", "var2", data=df)
```

This works for most plotting methods, which expect arrays/sequences as inputs. `data` can be anything which supports `__getitem__` (`dict`, `pandas.DataFrame`, `h5py`, ...) to access `array` like values with string keys.

In addition to this, some other changes were made, which makes working with labeled data (ex `pandas.Series`) easier:

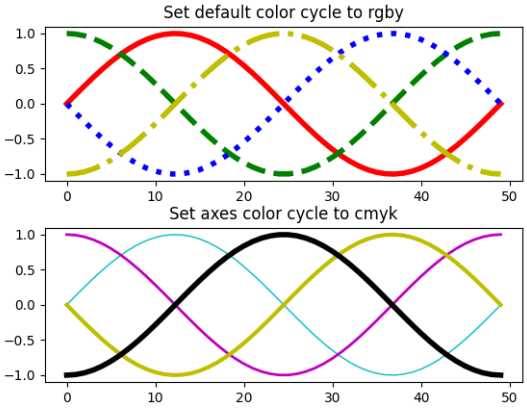
- For plotting methods with `label` keyword argument, one of the data inputs is designated as the label source. If the user does not supply a `label` that value object will be introspected for a label, currently by looking for a `name` attribute. If the value object does not have a `name` attribute but was specified by as a key into the `data` kwarg, then the key is used. In the above examples, this results in an implicit `label="var2"` for both cases.
- `plot()` now uses the index of a `Series` instead of `np.arange(len(y))`, if no `x` argument is supplied.

### Added `axes.prop_cycle` key to rcParams

This is a more generic form of the now-deprecated `axes.color_cycle` param. Now, we can cycle more than just colors, but also linestyles, hatches, and just about any other artist property. Cycler notation is used for defining property cycles. Adding cyclers together will be like you are `zip`-ing together two or more property cycles together:

```
axes.prop_cycle: cycler('color', 'rgb') + cycler('lw', [1, 2, 3])
```

You can even multiply cyclers, which is like using `itertools.product` on two or more property cycles.




### New Colormaps

All four of the colormaps proposed as the new default are available as `'viridis'` (the new default in 2.0), `'magma'`, `'plasma'`, and `'inferno'`

### Styles

Several new styles have been added, including many styles from the Seaborn project. Additionally, in order to prep for the upcoming 2.0 style-change release, a 'classic' and 'default' style has been added. For this release, the 'default' and 'classic' styles are identical. By using them now in your scripts, you can help ensure a smooth transition during future upgrades of matplotlib, so that you can upgrade to the snazzy new defaults when you are ready!

```
import matplotlib.style
matplotlib.style.use('classic')
```

The 'default' style will give you matplotlib's latest plotting styles:

```
matplotlib.style.use('default')
```

### Backends

#### New backend selection

The environment variable *MPLBACKEND* can now be used to set the matplotlib backend.

#### wx backend has been updated

The wx backend can now be used with both wxPython classic and Phoenix.

wxPython classic has to be at least version 2.8.12 and works on Python 2.x. As of May 2015 no official release of wxPython Phoenix is available but a current snapshot will work on Python 2.7+ and 3.4+.

If you have multiple versions of wxPython installed, then the user code is responsible setting the wxPython version. How to do this is explained in the comment at the beginning of the example *Embedding in wx #2*.

#### Configuration (rcParams)

Some parameters have been added, others have been improved.

| Parameter | Description |
|---|---|
| *rcParams["xaxis.labelpad"]*, *rcParams["yaxis.labelpad"]* | mplot3d now respects these parameters |
| *rcParams["axes.labelpad"]* (default: `4.0`) | Default space between the axis and the label |
| *rcParams["errorbar.capsize"]* (default: `0.0`) | Default length of end caps on error bars |
| *rcParams["xtick.minor.visible"]* (default: `False`), *rcParams["ytick.minor.visible"]* (default: `False`) | Default visibility of minor x/y ticks |
| *rcParams["legend.framealpha"]* (default: `0.8`) | Default transparency of the legend frame box |
| *rcParams["legend.facecolor"]* (default: `'inherit'`) | Default facecolor of legend frame box (or `'inherit'` from *rcParams["axes.facecolor"]* (default: `'white'`)) |
| *rcParams["legend.edgecolor"]* (default: `'0.8'`) | Default edgecolor of legend frame box (or `'inherit'` from *rcParams["axes.edgecolor"]* (default: `'black'`)) |
| *rcParams["figure.titlesize"]* (default: `'large'`) | Default font size for figure suptitles |
| *rcParams["figure.titleweight"]* (default: `'normal'`) | Default font weight for figure suptitles |
| *rcParams["image.composite_image"]* (default: `True`) | Whether a vector graphics backend should composite several images into a single image or not when saving. Useful when needing to edit the files further in Inkscape or other programs. |
| *rcParams["markers.fillstyle"]* (default: `'full'`) | Default fillstyle of markers. Possible values are `'full'` (the default), `'left'`, `'right'`, `'bottom'`, `'top'` and `'none'` |
| *rcParams["toolbar"]* (default: `'toolbar2'`) | Added `'toolmanager'` as a valid value, enabling the experimental `ToolManager` feature. |

### Widgets

#### Active state of Selectors

All selectors now implement `set_active` and `get_active` methods (also called when accessing the `active` property) to properly update and query whether they are active.

### Moved `ignore`, `set_active`, and `get_active` methods to base class `Widget`

Pushes up duplicate methods in child class to parent class to avoid duplication of code.

### Adds enable/disable feature to MultiCursor

A MultiCursor object can be disabled (and enabled) after it has been created without destroying the object. Example:

```
multi_cursor.active = False
```

### Improved RectangleSelector and new EllipseSelector Widget

Adds an *interactive* keyword which enables visible handles for manipulating the shape after it has been drawn.

Adds keyboard modifiers for:

- Moving the existing shape (default key = 'space')
- Making the shape square (default 'shift')
- Make the initial point the center of the shape (default 'control')
- Square and center can be combined

### Allow Artists to Display Pixel Data in Cursor

Adds *`get_cursor_data`* and *`format_cursor_data`* methods to artists which can be used to add zdata to the cursor display in the status bar. Also adds an implementation for Images.

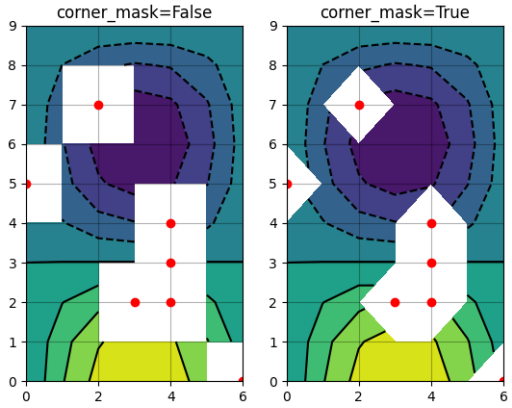
### New plotting features

### Auto-wrapping Text

Added the keyword argument "wrap" to Text, which automatically breaks long lines of text when being drawn. Works for any rotated text, different modes of alignment, and for text that are either labels or titles. This breaks at the `Figure`, not `Axes` edge.

### Contour plot corner masking

Ian Thomas rewrote the C++ code that calculates contours to add support for corner masking. This is controlled by a new keyword argument `corner_mask` in the functions *`contour()`* and *`contourf()`*. The previous behaviour, which is now obtained using `corner_mask=False`, was for a single masked point to completely mask out all four quads touching that point. The new behaviour, obtained using `corner_mask=True`, only masks the corners of those quads touching the point; any triangular corners comprising three unmasked points are contoured as usual. If the `corner_mask` keyword argument is not specified, the default value is taken from rcParams.




### Mostly unified linestyles for `Line2D`, `Patch` and `Collection`

The handling of linestyles for Lines, Patches and Collections has been unified. Now they all support defining linestyles with short symbols, like "--", as well as with full names, like "dashed". Also the definition using a dash pattern (`(0., [3., 3.])`) is supported for all methods using *`Line2D`*, *`Patch`* or *`Collection`*.

### Legend marker order

Added ability to place the label before the marker in a legend box with `markerfirst` keyword

### Support for legend for PolyCollection and stackplot

Added a *`legend_handler`* for *`PolyCollection`* as well as a *labels* argument to *`stackplot()`*.

### Support for alternate pivots in mplot3d quiver plot

Added a `pivot` kwarg to *`quiver`* that controls the pivot point around which the quiver line rotates. This also determines the placement of the arrow head along the quiver line.

### Logit Scale

Added support for the 'logit' axis scale, a nonlinear transformation

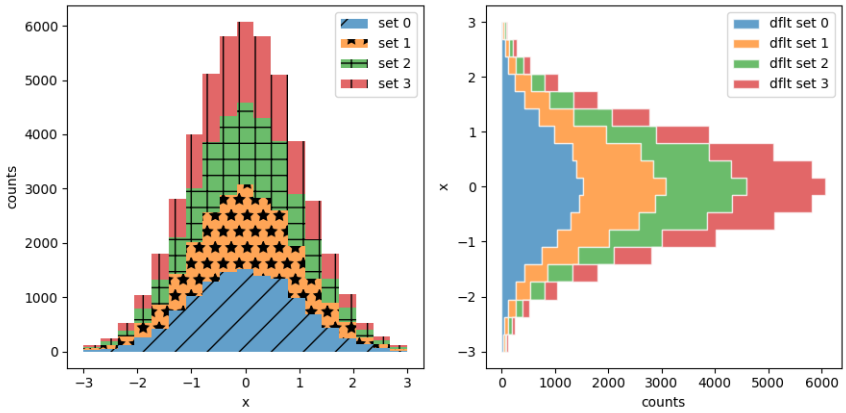
$$x- > \log 10(x/(1-x))$$

for data between 0 and 1 excluded.

### Add step kwargs to fill_between

Added `step` kwarg to *`Axes.fill_between`* to allow to fill between lines drawn using the 'step' draw style. The values of `step` match those of the `where` kwarg of *`Axes.step`*. The asymmetry of the kwargs names is not ideal, but *`Axes.fill_between`* already has a `where` kwarg.

This is particularly useful for plotting pre-binned histograms.




### Square Plot

Implemented square plots feature as a new parameter in the axis function. When argument 'square' is specified, equal scaling is set, and the limits are set such that `xmax-xmin == ymax-ymin`.

### Updated figimage to take optional resize parameter

Added the ability to plot simple 2D-Array using `plt.figimage(X, resize=True)`. This is useful for plotting simple 2D-Array without the Axes or whitespacing around the image.

### Updated Figure.savefig() can now use figure's dpi

Added support to save the figure with the same dpi as the figure on the screen using `dpi='figure'`:.

Example:

```
f = plt.figure(dpi=25)  # dpi set to 25
S = plt.scatter([1,2,3],[4,5,6])
f.savefig('output.png', dpi='figure')  # output savefig dpi set to 25 (same␣
 ↪as figure)
```

### Updated Table to control edge visibility

Added the ability to toggle the visibility of lines in Tables. Functionality added to the *pyplot.table* factory function under the keyword argument "edges". Values can be the strings "open", "closed", "horizontal", "vertical" or combinations of the letters "L", "R", "T", "B" which represent left, right, top, and bottom respectively.

Example:

```
table(..., edges="open")  # No line visible
table(..., edges="closed")  # All lines visible
table(..., edges="horizontal")  # Only top and bottom lines visible
table(..., edges="LT")  # Only left and top lines visible.
```

### Zero r/cstride support in plot_wireframe

Adam Hughes added support to mplot3d's plot_wireframe to draw only row or column line plots.

### Plot bar and barh with labels

Added kwarg *tick_label* to *bar* and *barh* to support plotting bar graphs with a text label for each bar.

### Added center and frame kwargs to pie

These control where the center of the pie graph are and if the Axes frame is shown.

### Fixed 3D filled contour plot polygon rendering

Certain cases of 3D filled contour plots that produce polygons with multiple holes produced improper rendering due to a loss of path information between *PolyCollection* and *Poly3DCollection*. A function *set_verts_and_codes()* was added to allow path information to be retained for proper rendering.

### Dense colorbars are rasterized

Vector file formats (pdf, ps, svg) are efficient for many types of plot element, but for some they can yield excessive file size and even rendering artifacts, depending on the renderer used for screen display. This is a problem for colorbars that show a large number of shades, as is most commonly the case. Now, if a colorbar is showing 50 or more colors, it will be rasterized in vector backends.

### DateFormatter strftime

*DateFormatter*'s `__call__` method will format a `datetime.datetime` object with the format string passed to the formatter's constructor. This method accepts datetimes with years before 1900, unlike `datetime.datetime.strftime()`.

### Artist-level {get,set}_usetex for text

Add `{get,set}_usetex` methods to *Text* objects which allow artist-level control of LaTeX rendering vs. the internal mathtex rendering.

### `Axes.remove()` works as expected

As with artists added to an *Axes*, *Axes* objects can be removed from their figure via *remove()*.

### API Consistency fix within Locators set_params() function

*set_params()* function, which sets parameters within a *Locator* type instance, is now available to all *Locator* types. The implementation also prevents unsafe usage by strictly defining the parameters that a user can set.

To use, call `set_params()` on a *Locator* instance with desired arguments:

```
loc = matplotlib.ticker.LogLocator()
# Set given attributes for loc.
loc.set_params(numticks=8, numdecs=8, subs=[2.0], base=8)
# The below will error, as there is no such parameter for LogLocator
# named foo
# loc.set_params(foo='bar')
```

### Date Locators

Date Locators (derived from *DateLocator*) now implement the *tick_values* method. This is expected of all Locators derived from *Locator*.

The Date Locators can now be used easily without creating axes

```
from datetime import datetime
from matplotlib.dates import YearLocator
t0 = datetime(2002, 10, 9, 12, 10)
tf = datetime(2005, 10, 9, 12, 15)
loc = YearLocator()
values = loc.tick_values(t0, tf)
```

### OffsetBoxes now support clipping

*Artist*s draw onto objects of type *OffsetBox* through *DrawingArea* and *TextArea*. The *TextArea* calculates the required space for the text and so the text is always within the bounds, for this nothing has changed.

However, *DrawingArea* acts as a parent for zero or more *Artist*s that draw on it and may do so beyond the bounds. Now child *Artist*s can be clipped to the bounds of the *DrawingArea*.

### OffsetBoxes now considered by tight_layout

When *tight_layout()* or *Figure.tight_layout* or *GridSpec.tight_layout()* is called, *OffsetBox*es that are anchored outside the axes will not get chopped out. The *OffsetBox*es will also not get overlapped by other axes in case of multiple subplots.

### Per-page pdf notes in multi-page pdfs (PdfPages)

Add a new method *attach_note()* to the PdfPages class, allowing the attachment of simple text notes to pages in a multi-page pdf of figures. The new note is visible in the list of pdf annotations in a viewer that has this facility (Adobe Reader, OSX Preview, Skim, etc.). Per default the note itself is kept off-page to prevent it to appear in print-outs.

*PdfPages.attach_note* needs to be called before *savefig* in order to be added to the correct figure.

### Updated fignum_exists to take figure name

Added the ability to check the existence of a figure using its name instead of just the figure number. Example:

```
figure('figure')
fignum_exists('figure') #true
```

### ToolManager

Federico Ariza wrote the new *ToolManager* that comes as replacement for *NavigationToolbar2*

*ToolManager* offers a new way of looking at the user interactions with the figures. Before we had the *NavigationToolbar2* with its own tools like `zoom/pan/home/save/...` and also we had the shortcuts like `yscale/grid/quit/.....` *ToolManager* relocate all those actions as Tools (located in *backend_tools*), and defines a way to access/trigger/reconfigure them.

The Toolbars are replaced by *ToolContainerBase*s that are just GUI interfaces to trigger the tools. But don't worry the default backends include a *ToolContainerBase* called `toolbar`

---

**Note:** At the moment, we release this primarily for feedback purposes and should be treated as experimental until further notice as API changes will occur. For the moment the *ToolManager* works only with the GTK3 and Tk backends. Make sure you use one of those. Port for the rest of the backends is coming soon.

To activate the *ToolManager* include the following at the top of your file

```
>>> matplotlib.rcParams['toolbar'] = 'toolmanager'
```

---

### Interact with the ToolContainer

The most important feature is the ability to easily reconfigure the ToolContainer (aka toolbar). For example, if we want to remove the "forward" button we would just do.

```
>>> fig.canvas.manager.toolmanager.remove_tool('forward')
```

Now if you want to programmatically trigger the "home" button

```
>>> fig.canvas.manager.toolmanager.trigger_tool('home')
```

### New Tools for ToolManager

It is possible to add new tools to the ToolManager

A very simple tool that prints "You're awesome" would be:

```
from matplotlib.backend_tools import ToolBase
class AwesomeTool(ToolBase):
    def trigger(self, *args, **kwargs):
        print("You're awesome")
```

To add this tool to *ToolManager*

```
>>> fig.canvas.manager.toolmanager.add_tool('Awesome', AwesomeTool)
```

If we want to add a shortcut ("d") for the tool

```
>>> fig.canvas.manager.toolmanager.update_keymap('Awesome', 'd')
```

To add it to the toolbar inside the group 'foo'

```
>>> fig.canvas.manager.toolbar.add_tool('Awesome', 'foo')
```

There is a second class of tools, "Toggleable Tools", this are almost the same as our basic tools, just that belong to a group, and are mutually exclusive inside that group. For tools derived from *ToolToggleBase* there are two basic methods *enable* and *disable* that are called automatically whenever it is toggled.

A full example is located in *Tool Manager*

### cbook.is_sequence_of_strings recognizes string objects

This is primarily how pandas stores a sequence of strings

```
import pandas as pd
import matplotlib.cbook as cbook

a = np.array(['a', 'b', 'c'])
print(cbook.is_sequence_of_strings(a))  # True

a = np.array(['a', 'b', 'c'], dtype=object)
print(cbook.is_sequence_of_strings(a))  # True

s = pd.Series(['a', 'b', 'c'])
print(cbook.is_sequence_of_strings(s))  # True
```

Previously, the last two prints returned false.

#### New `close-figs` argument for plot directive

Matplotlib has a sphinx extension `plot_directive` that creates plots for inclusion in sphinx documents. Matplotlib 1.5 adds a new option to the plot directive - `close-figs` - that closes any previous figure windows before creating the plots. This can help avoid some surprising duplicates of plots when using `plot_directive`.

#### Support for URL string arguments to `imread`

The *`imread()`* function now accepts URL strings that point to remote PNG files. This circumvents the generation of a HTTPResponse object directly.

#### Display hook for animations in the IPython notebook

*`Animation`* instances gained a `_repr_html_` method to support inline display of animations in the notebook. The method used to display is controlled by the `animation.html` rc parameter, which currently supports values of `none` and `html5`. `none` is the default, performing no display. `html5` converts the animation to an h264 encoded video, which is embedded directly in the notebook.

Users not wishing to use the `_repr_html_` display hook can also manually call the *`to_html5_video`* method to get the HTML and display using IPython's `HTML` display class:

```
from IPython.display import HTML
HTML(anim.to_html5_video())
```

#### Prefixed pkg-config for building

Handling of pkg-config has been fixed in so far as it is now possible to set it using the environment variable `PKG_CONFIG`. This is important if your toolchain is prefixed. This is done in a simpilar way as setting `CC` or `CXX` before building. An example follows:

```
export PKG_CONFIG=x86_64-pc-linux-gnu-pkg-config
```

### 9.14.2 API Changes in 1.5.3

#### `ax.plot(..., marker=None)` gives default marker

Prior to 1.5.3 keyword arguments passed to *`plot`* were handled in two parts -- default keyword arguments generated internal to *`plot`* (such as the cycled styles) and user supplied keyword arguments. The internally generated keyword arguments were passed to the *`matplotlib.lines.Line2D`* and the user keyword arguments were passed to `ln.set(**kwargs)` to update the artist after it was created. Now both sets of keyword arguments are merged and passed to *`Line2D`*. This change was made to allow *None* to be passed in via the user keyword arguments to mean 'do the default thing' as is the convention through out Matplotlib rather than raising an exception.

Unlike most *Line2D* setter methods *set_marker* did accept None as a valid input which was mapped to 'no marker'. Thus, by routing this marker=None through __init__ rather than set(...) the meaning of ax.plot(..., marker=None) changed from 'no markers' to 'default markers from rcparams'.

This is change is only evident if mpl.rcParams['lines.marker'] has a value other than 'None' (which is string 'None' which means 'no marker').

### 9.14.3 API Changes in 1.5.2

#### Default Behavior Changes

#### Changed default autorange behavior in boxplots

Prior to v1.5.2, the whiskers of boxplots would extend to the minimum and maximum values if the quartiles were all equal (i.e., Q1 = median = Q3). This behavior has been disabled by default to restore consistency with other plotting packages.

To restore the old behavior, simply set autorange=True when calling plt.boxplot.

### 9.14.4 API Changes in 1.5.0

#### Code Changes

#### Reversed matplotlib.cbook.ls_mapper, added ls_mapper_r

Formerly, *matplotlib.cbook.ls_mapper* was a dictionary with the long-form line-style names ("solid") as keys and the short forms ("-") as values. This long-to-short mapping is now done by *ls_mapper_r*, and the short-to-long mapping is done by the *ls_mapper*.

#### Prevent moving artists between Axes, Property-ify Artist.axes, deprecate Artist.{get,set}_axes

This was done to prevent an Artist that is already associated with an Axes from being moved/added to a different Axes. This was never supported as it causes havoc with the transform stack. The apparent support for this (as it did not raise an exception) was the source of multiple bug reports and questions on SO.

For almost all use-cases, the assignment of the axes to an artist should be taken care of by the axes as part of the Axes.add_* method, hence the deprecation of {get,set}_axes.

Removing the set_axes method will also remove the 'axes' line from the ACCEPTS kwarg tables (assuming that the removal date gets here before that gets overhauled).

### Tightened input validation on 'pivot' kwarg to quiver

Tightened validation so that only {'tip', 'tail', 'mid', and 'middle'} (but any capitalization) are valid values for the *pivot* keyword argument in the *Quiver* class (and hence *axes.Axes.quiver* and *pyplot.quiver* which both fully delegate to *Quiver*). Previously any input matching 'mid.*' would be interpreted as 'middle', 'tip.*' as 'tip' and any string not matching one of those patterns as 'tail'.

The value of `Quiver.pivot` is normalized to be in the set {'tip', 'tail', 'middle'} in *Quiver*.

### Reordered `Axes.get_children`

The artist order returned by *axes.Axes.get_children* did not match the one used by *axes.Axes.draw*. They now use the same order, as *axes.Axes.draw* now calls *axes.Axes.get_children*.

### Changed behaviour of contour plots

The default behaviour of *contour()* and *contourf()* when using a masked array is now determined by the new keyword argument *corner_mask*, or if this is not specified then the new *rcParams["contour.corner_mask"]* (default: `True`) instead. The new default behaviour is equivalent to using `corner_mask=True`; the previous behaviour can be obtained using `corner_mask=False` or by changing the rcParam. The example https://matplotlib.org/examples/pylab_examples/contour_corner_mask.html demonstrates the difference. Use of the old contouring algorithm, which is obtained with `corner_mask='legacy'`, is now deprecated.

Contour labels may now appear in different places than in earlier versions of Matplotlib.

In addition, the keyword argument *nchunk* now applies to *contour()* as well as *contourf()*, and it subdivides the domain into subdomains of exactly *nchunk* by *nchunk* quads, whereas previously it was only roughly *nchunk* by *nchunk* quads.

The C/C++ object that performs contour calculations used to be stored in the public attribute `QuadContourSet.Cntr`, but is now stored in a private attribute and should not be accessed by end users.

### Added set_params function to all Locator types

This was a bug fix targeted at making the api for Locators more consistent.

In the old behavior, only locators of type MaxNLocator have set_params() defined, causing its use on any other Locator to raise an AttributeError *( aside: set_params(args) is a function that sets the parameters of a Locator instance to be as specified within args)*. The fix involves moving set_params() to the Locator class such that all subtypes will have this function defined.

Since each of the Locator subtypes have their own modifiable parameters, a universal set_params() in Locator isn't ideal. Instead, a default no-operation function that raises a warning is implemented in Locator. Subtypes extending Locator will then override with their own implementations. Subtypes that do not have a need for set_params() will fall back onto their parent's implementation, which raises a warning as intended.

In the new behavior, Locator instances will not raise an AttributeError when set_params() is called. For Locators that do not implement set_params(), the default implementation in Locator is used.

### Disallow `None` as x or y value in ax.plot

Do not allow `None` as a valid input for the `x` or `y` args in *axes.Axes.plot*. This may break some user code, but this was never officially supported (ex documented) and allowing `None` objects through can lead to confusing exceptions downstream.

To create an empty line use

```
ln1, = ax.plot([], [], ...)
ln2, = ax.plot([], ...)
```

In either case to update the data in the *Line2D* object you must update both the `x` and `y` data.

### Removed *args* and *kwargs* from `MicrosecondLocator.__call__`

The call signature of `matplotlib.dates.MicrosecondLocator.__call__` has changed from `__call__(self, *args, **kwargs)` to `__call__(self)`. This is consistent with the superclass *Locator* and also all the other Locators derived from this superclass.

### No `ValueError` for the MicrosecondLocator and YearLocator

The *MicrosecondLocator* and *YearLocator* objects when called will return an empty list if the axes have no data or the view has no interval. Previously, they raised a `ValueError`. This is consistent with all the Date Locators.

### 'OffsetBox.DrawingArea' respects the 'clip' keyword argument

The call signature was `OffsetBox.DrawingArea(..., clip=True)` but nothing was done with the *clip* argument. The object did not do any clipping regardless of that parameter. Now the object can and does clip the child *Artist*s if they are set to be clipped.

You can turn off the clipping on a per-child basis using `child.set_clip_on(False)`.

### Add salt to clipPath id

Add salt to the hash used to determine the id of the `clipPath` nodes. This is to avoid conflicts when two svg documents with the same clip path are included in the same document (see https://github.com/ipython/ipython/issues/8133 and https://github.com/matplotlib/matplotlib/issues/4349 ), however this means that the svg output is no longer deterministic if the same figure is saved twice. It is not expected that this will affect any users as the current ids are generated from an md5 hash of properties of the clip path and any user would have a very difficult time anticipating the value of the id.

### Changed snap threshold for circle markers to inf

When drawing circle markers above some marker size (previously 6.0) the path used to generate the marker was snapped to pixel centers. However, this ends up distorting the marker away from a circle. By setting the snap threshold to inf snapping is never done on circles.

This change broke several tests, but is an improvement.

### Preserve units with Text position

Previously the 'get_position' method on Text would strip away unit information even though the units were still present. There was no inherent need to do this, so it has been changed so that unit data (if present) will be preserved. Essentially a call to 'get_position' will return the exact value from a call to 'set_position'.

If you wish to get the old behaviour, then you can use the new method called 'get_unitless_position'.

### New API for custom Axes view changes

Interactive pan and zoom were previously implemented using a Cartesian-specific algorithm that was not necessarily applicable to custom Axes. Three new private methods, `matplotlib.axes._base._AxesBase._get_view`, `matplotlib.axes._base._AxesBase._set_view`, and `matplotlib.axes._base._AxesBase._set_view_from_bbox`, allow for custom *Axes* classes to override the pan and zoom algorithms. Implementors of custom *Axes* who override these methods may provide suitable behaviour for both pan and zoom as well as the view navigation buttons on the interactive toolbars.

### MathTex visual changes

The spacing commands in mathtext have been changed to more closely match vanilla TeX.

### Improved spacing in mathtext

The extra space that appeared after subscripts and superscripts has been removed.

### No annotation coordinates wrap

In #2351 for 1.4.0 the behavior of ['axes points', 'axes pixel', 'figure points', 'figure pixel'] as coordinates was change to no longer wrap for negative values. In 1.4.3 this change was reverted for 'axes points' and 'axes pixel' and in addition caused 'axes fraction' to wrap. For 1.5 the behavior has been reverted to as it was in 1.4.0-1.4.2, no wrapping for any type of coordinate.

### Deprecation

#### Deprecated `GraphicsContextBase.set_graylevel`

The `GraphicsContextBase.set_graylevel` function has been deprecated in 1.5 and will be removed in 1.6. It has been unused. The *`GraphicsContextBase.set_foreground`* could be used instead.

#### deprecated idle_event

The `idle_event` was broken or missing in most backends and causes spurious warnings in some cases, and its use in creating animations is now obsolete due to the animations module. Therefore code involving it has been removed from all but the wx backend (where it partially works), and its use is deprecated. The *`animation`* module may be used instead to create animations.

#### `color_cycle` deprecated

In light of the new property cycling feature, the Axes method `set_color_cycle` is now deprecated. Calling this method will replace the current property cycle with one that cycles just the given colors.

Similarly, the rc parameter *axes.color_cycle* is also deprecated in lieu of the new *`rcParams["axes.prop_cycle"]`* (default: `cycler('color', ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728', '#9467bd', '#8c564b', '#e377c2', '#7f7f7f', '#bcbd22', '#17becf'])`) parameter. Having both parameters in the same rc file is not recommended as the result cannot be predicted. For compatibility, setting *axes.color_cycle* will replace the cycler in *`rcParams["axes.prop_cycle"]`* (default: `cycler('color', ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728', '#9467bd', '#8c564b', '#e377c2', '#7f7f7f', '#bcbd22', '#17becf'])`) with a color cycle. Accessing *axes.color_cycle* will return just the color portion of the property cycle, if it exists.

Timeline for removal has not been set.

### Bundled jquery

The version of jquery bundled with the webagg backend has been upgraded from 1.7.1 to 1.11.3. If you are using the version of jquery bundled with webagg you will need to update your html files as such

```
-    <script src="_static/jquery/js/jquery-1.7.1.min.js"></script>
+    <script src="_static/jquery/js/jquery-1.11.3.min.js"></script>
```

### Code Removed

#### Removed `Image` from main namespace

`Image` was imported from PIL/pillow to test if PIL is available, but there is no reason to keep `Image` in the namespace once the availability has been determined.

#### Removed `lod` from Artist

Removed the method `set_lod` and all references to the attribute `_lod` as they are not used anywhere else in the code base. It appears to be a feature stub that was never built out.

#### Removed threading related classes from cbook

The classes `Scheduler`, `Timeout`, and `Idle` were in cbook, but are not used internally. They appear to be a prototype for the idle event system which was not working and has recently been pulled out.

#### Removed *Lena* images from sample_data

The `lena.png` and `lena.jpg` images have been removed from Matplotlib's sample_data directory. The images are also no longer available from *matplotlib.cbook.get_sample_data*. We suggest using `matplotlib.cbook.get_sample_data('grace_hopper.png')` or `matplotlib.cbook.get_sample_data('grace_hopper.jpg')` instead.

#### Legend

Removed handling of *loc* as a positional argument to *Legend*

#### Legend handlers

Remove code to allow legend handlers to be callable. They must now implement a method `legend_artist`.

#### Axis

Removed method `set_scale`. This is now handled via a private method which should not be used directly by users. It is called via `Axes.set_{x,y}scale` which takes care of ensuring the related changes are also made to the Axes object.

### finance.py

Removed functions with ambiguous argument order from finance.py

### Annotation

Removed `textcoords` and `xytext` proprieties from Annotation objects.

### sphinxext.ipython_*.py

Both `ipython_console_highlighting` and `ipython_directive` have been moved to IPython.

Change your import from `matplotlib.sphinxext.ipython_directive` to `IPython.sphinxext.ipython_directive` and from `matplotlib.sphinxext.ipython_directive` to `IPython.sphinxext.ipython_directive`

### LineCollection.color

Deprecated in 2005, use `set_color`

### remove `'faceted'` as a valid value for *shading* in `tri.tripcolor`

Use *edgecolor* instead. Added validation on *shading* to only be valid values.

### Remove `faceted` kwarg from scatter

Remove support for the `faceted` kwarg. This was deprecated in d48b34288e9651ff95c3b8a071ef5ac5cf50bae7 (2008-04-18!) and replaced by `edgecolor`.

### Remove `set_colorbar` method from `ScalarMappable`

Remove `set_colorbar` method, use *`colorbar`* attribute directly.

#### patheffects.svg

- remove `get_proxy_renderer` method from `AbstractPathEffect` class
- remove `patch_alpha` and `offset_xy` from `SimplePatchShadow`

#### Remove `testing.image_util.py`

Contained only a no-longer used port of functionality from PIL

#### Remove `mlab.FIFOBuffer`

Not used internally and not part of core mission of mpl.

#### Remove `mlab.prepca`

Deprecated in 2009.

#### Remove `NavigationToolbar2QTAgg`

Added no functionality over the base `NavigationToolbar2Qt`

#### mpl.py

Remove the module `matplotlib.mpl`. Deprecated in 1.3 by PR #1670 and commit 78ce67d161625833cacff23cfe5d74920248c5b2

## 9.15 Version 1.4

### 9.15.1 What's new in Matplotlib 1.4 (Aug 25, 2014)

Thomas A. Caswell served as the release manager for the 1.4 release.

**Table of Contents**


- *What's new in Matplotlib 1.4 (Aug 25, 2014)*
  - *New colormap*
  - *The nbagg backend*
  - *New plotting features*

- *Date handling*
- *Configuration (rcParams)*
- `style` *package added*
- *Backends*
- *Text*
- *Sphinx extensions*
- *Legend and PathEffects documentation*
- *Widgets*
- *GAE integration*


**Note:** matplotlib 1.4 supports Python 2.6, 2.7, 3.3, and 3.4

### New colormap

In heatmaps, a green-to-red spectrum is often used to indicate intensity of activity, but this can be problematic for the red/green colorblind. A new, colorblind-friendly colormap is now available at `matplotlib.cm.Wistia`. This colormap maintains the red/green symbolism while achieving deuteranopic legibility through brightness variations. See here for more information.

### The nbagg backend

Phil Elson added a new backend, named "nbagg", which enables interactive figures in a live IPython notebook session. The backend makes use of the infrastructure developed for the webagg backend, which itself gives standalone server backed interactive figures in the browser, however nbagg does not require a dedicated matplotlib server as all communications are handled through the IPython Comm machinery.

As with other backends nbagg can be enabled inside the IPython notebook with:

```
import matplotlib
matplotlib.use('nbagg')
```

Once figures are created and then subsequently shown, they will placed in an interactive widget inside the notebook allowing panning and zooming in the same way as any other matplotlib backend. Because figures require a connection to the IPython notebook server for their interactivity, once the notebook is saved, each figure will be rendered as a static image - thus allowing non-interactive viewing of figures on services such as nbviewer.

### New plotting features

#### Power-law normalization

Ben Gamari added a power-law normalization method, *PowerNorm*. This class maps a range of values to the interval [0,1] with power-law scaling with the exponent provided by the constructor's *gamma* argument. Power law normalization can be useful for, e.g., emphasizing small populations in a histogram.

#### Fully customizable boxplots

Paul Hobson overhauled the *boxplot()* method such that it is now completely customizable in terms of the styles and positions of the individual artists. Under the hood, *boxplot()* relies on a new function (*boxplot_stats()*), which accepts any data structure currently compatible with *boxplot()*, and returns a list of dictionaries containing the positions for each element of the boxplots. Then a second method, *bxp* is called to draw the boxplots based on the stats.

The *boxplot()* function can be used as before to generate boxplots from data in one step. But now the user has the flexibility to generate the statistics independently, or to modify the output of *boxplot_stats()* prior to plotting with *bxp*.

Lastly, each artist (e.g., the box, outliers, cap, notches) can now be toggled on or off and their styles can be passed in through individual kwargs. See the examples: *Artist customization in box plots* and *Separate calculation and plotting of boxplots*

Added a bool kwarg, `manage_xticks`, which if False disables the management of the ticks and limits on the x-axis by *bxp()*.

#### Support for datetime axes in 2d plots

Andrew Dawson added support for datetime axes to *contour()*, *contourf()*, *pcolormesh()* and *pcolor()*.

#### Support for additional spectrum types

Todd Jennings added support for new types of frequency spectrum plots: *magnitude_spectrum()*, *phase_spectrum()*, and *angle_spectrum()*, as well as corresponding functions in mlab.

He also added these spectrum types to *specgram()*, as well as adding support for linear scaling there (in addition to the existing dB scaling). Support for additional spectrum types was also added to *specgram()*.

He also increased the performance for all of these functions and plot types.

### Support for detrending and windowing 2D arrays in mlab

Todd Jennings added support for 2D arrays in the *detrend_mean()*, *detrend_none()*, and *detrend()*, as well as adding `matplotlib.mlab.apply_window` which support windowing 2D arrays.

### Support for strides in mlab

Todd Jennings added some functions to mlab to make it easier to use NumPy strides to create memory-efficient 2D arrays. This includes `matplotlib.mlab.stride_repeat`, which repeats an array to create a 2D array, and `matplotlib.mlab.stride_windows`, which uses a moving window to create a 2D array from a 1D array.

### Formatter for new-style formatting strings

Added *StrMethodFormatter* which does the same job as *FormatStrFormatter*, but accepts new-style formatting strings instead of printf-style formatting strings

### Consistent grid sizes in streamplots

*streamplot()* uses a base grid size of 30x30 for both `density=1` and `density=(1, 1)`. Previously a grid size of 30x30 was used for `density=1`, but a grid size of 25x25 was used for `density=(1, 1)`.

### Get a list of all tick labels (major and minor)

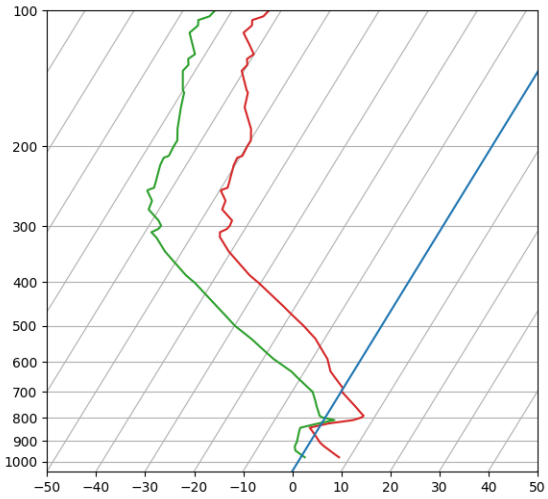
Added the kwarg 'which' to *Axes.get_xticklabels*, *Axes.get_yticklabels* and *Axis.get_ticklabels*. 'which' can be 'major', 'minor', or 'both' select which ticks to return, like *set_ticks_position*. If 'which' is `None` then the old behaviour (controlled by the bool *minor*).

### Separate horizontal/vertical axes padding support in ImageGrid

The kwarg 'axes_pad' to *mpl_toolkits.axes_grid1.axes_grid.ImageGrid* can now be a tuple if separate horizontal/vertical padding is needed. This is supposed to be very helpful when you have a labelled legend next to every subplot and you need to make some space for legend's labels.

### Support for skewed transformations

The *Affine2D* gained additional methods *skew* and *skew_deg* to create skewed transformations. Additionally, matplotlib internals were cleaned up to support using such transforms in *Axes*. This transform is important for some plot types, specifically the Skew-T used in meteorology.




### Support for specifying properties of wedge and text in pie charts.

Added the kwargs 'wedgeprops' and 'textprops' to *pie* to accept properties for wedge and text objects in a pie. For example, one can specify wedgeprops = {'linewidth':3} to specify the width of the borders of the wedges in the pie. For more properties that the user can specify, look at the docs for the wedge and text objects.

### Fixed the direction of errorbar upper/lower limits

Larry Bradley fixed the *errorbar()* method such that the upper and lower limits (*lolims*, *uplims*, *xlolims*, *xuplims*) now point in the correct direction.

### More consistent add-object API for Axes

Added the Axes method *add_image* to put image handling on a par with artists, collections, containers, lines, patches, and tables.

### Violin Plots

Per Parker, Gregory Kelsie, Adam Ortiz, Kevin Chan, Geoffrey Lee, Deokjae Donald Seo, and Taesu Terry Lim added a basic implementation for violin plots. Violin plots can be used to represent the distribution of sample data. They are similar to box plots, but use a kernel density estimation function to present a smooth approximation of the data sample used. The added features are:

*violin* - Renders a violin plot from a collection of statistics. *violin_stats()* - Produces a collection of statistics suitable for rendering a violin plot. *violinplot()* - Creates a violin plot from a set of sample data. This method makes use of *violin_stats()* to process the input data, and *violin_stats()* to do the actual rendering. Users are also free to modify or replace the output of *violin_stats()* in order to customize the violin plots to their liking.

This feature was implemented for a software engineering course at the University of Toronto, Scarborough, run in Winter 2014 by Anya Tafliovich.

### More *markevery* options to show only a subset of markers

Rohan Walker extended the *markevery* property in *Line2D*. You can now specify a subset of markers to show with an int, slice object, numpy fancy indexing, or float. Using a float shows markers at approximately equal display-coordinate-distances along the line.

### Added size related functions to specialized `Collection`s

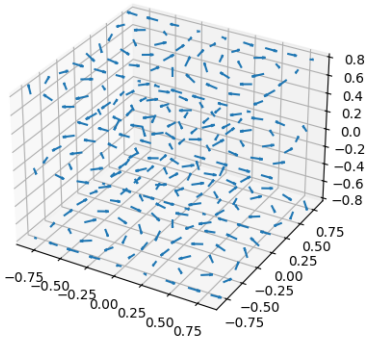
Added the `get_size` and `set_size` functions to control the size of elements of specialized collections ( *AsteriskPolygonCollection* matplotlib.collections.BrokenBarHCollection *CircleCollection* *PathCollection* *PolyCollection* *RegularPolyCollection* *StarPolygonCollection*).

### Fixed the mouse coordinates giving the wrong theta value in Polar graph

Added code to *transform_non_affine* to ensure that the calculated theta value was between the range of 0 and 2 * pi since the problem was that the value can become negative after applying the direction and rotation to the theta calculation.

### Simple quiver plot for mplot3d toolkit

A team of students in an *Engineering Large Software Systems* course, taught by Prof. Anya Tafliovich at the University of Toronto, implemented a simple version of a quiver plot in 3D space for the mplot3d toolkit as one of their term project. This feature is documented in *quiver*. The team members are: Ryan Steve D'Souza, Victor B, xbtsw, Yang Wang, David, Caradec Bisesar and Vlad Vassilovski.




### polar-plot r-tick locations

Added the ability to control the angular position of the r-tick labels on a polar plot via *set_rlabel_position*.

### Date handling

### n-d array support for date conversion

Andrew Dawson added support for n-d array handling to *matplotlib.dates.num2date()*, *matplotlib.dates.date2num()* and *matplotlib.dates.datestr2num()*. Support is also added to the unit conversion interfaces *matplotlib.dates.DateConverter* and *matplotlib.units.Registry*.

### Configuration (rcParams)

#### `savefig.transparent` added

Controls whether figures are saved with a transparent background by default. Previously *savefig* always defaulted to a non-transparent background.

#### `axes.titleweight`

Added rcParam to control the weight of the title

#### `axes.formatter.useoffset` added

Controls the default value of *useOffset* in *`ScalarFormatter`*. If `True` and the data range is much smaller than the data average, then an offset will be determined such that the tick labels are meaningful. If `False` then the full number will be formatted in all conditions.

#### `nbagg.transparent` added

Controls whether nbagg figures have a transparent background. `nbagg.transparent` is `True` by default.

### XDG compliance

Matplotlib now looks for configuration files (both rcparams and style) in XDG compliant locations.

#### `style` package added

You can now easily switch between different styles using the new `style` package:

```
>>> from matplotlib import style
>>> style.use('dark_background')
```

Subsequent plots will use updated colors, sizes, etc. To list all available styles, use:

```
>>> print style.available
```

You can add your own custom `<style name>.mplstyle` files to `~/.matplotlib/stylelib` or call `use` with a URL pointing to a file with `matplotlibrc` settings.

*Note that this is an experimental feature*, and the interface may change as users test out this new feature.

### Backends

### Qt5 backend

Martin Fitzpatrick and Tom Badran implemented a Qt5 backend. The differences in namespace locations between Qt4 and Qt5 was dealt with by shimming Qt4 to look like Qt5, thus the Qt5 implementation is the primary implementation. Backwards compatibility for Qt4 is maintained by wrapping the Qt5 implementation.

The Qt5Agg backend currently does not work with IPython's %matplotlib magic.

The 1.4.0 release has a known bug where the toolbar is broken. This can be fixed by:

```
cd path/to/installed/matplotlib
wget https://github.com/matplotlib/matplotlib/pull/3322.diff
# unix2dos 3322.diff (if on windows to fix line endings)
patch -p2 < 3322.diff
```

### Qt4 backend

Rudolf Höfler changed the appearance of the subplottool. All sliders are vertically arranged now, buttons for tight layout and reset were added. Furthermore, the subplottool is now implemented as a modal dialog. It was previously a QMainWindow, leaving the SPT open if one closed the plot window.

In the figure options dialog one can now choose to (re-)generate a simple automatic legend. Any explicitly set legend entries will be lost, but changes to the curves' label, linestyle, et cetera will now be updated in the legend.

Interactive performance of the Qt4 backend has been dramatically improved under windows.

The mapping of key-signals from Qt to values matplotlib understands was greatly improved (For both Qt4 and Qt5).

### Cairo backends

The Cairo backends are now able to use the cairocffi bindings which are more actively maintained than the pycairo bindings.

### Gtk3Agg backend

The Gtk3Agg backend now works on Python 3.x, if the cairocffi bindings are installed.

### PDF backend

Added context manager for saving to multi-page PDFs.

### Text

### Text URLs supported by SVG backend

The SVG backend will now render *Text* objects' url as a link in output SVGs. This allows one to make clickable text in saved figures using the url kwarg of the *Text* class.

#### Anchored sizebar font

Added the `fontproperties` kwarg to *`AnchoredSizeBar`* to control the font properties.

#### Sphinx extensions

The `:context:` directive in the *`plot_directive`* Sphinx extension can now accept an optional `reset` setting, which will cause the context to be reset. This allows more than one distinct context to be present in documentation. To enable this option, use `:context: reset` instead of `:context:` any time you want to reset the context.

#### Legend and PathEffects documentation

The *Legend guide* and *Path effects guide* have both been updated to better reflect the full potential of each of these powerful features.

#### Widgets

#### Span Selector

Added an option `span_stays` to the *`SpanSelector`* which makes the selector rectangle stay on the axes after you release the mouse.

#### GAE integration

Matplotlib will now run on google app engine.

### 9.15.2 API Changes in 1.4.x

#### Code changes

- A major refactoring of the axes module was made. The axes module has been split into smaller modules:
  - the `_base` module, which contains a new private `_AxesBase` class. This class contains all methods except plotting and labelling methods.
  - the *`axes`* module, which contains the *`axes.Axes`* class. This class inherits from `_AxesBase`, and contains all plotting and labelling methods.
  - the `_subplot` module, with all the classes concerning subplotting.

  There are a couple of things that do not exists in the *`axes`* module's namespace anymore. If you use them, you need to import them from their original location:
  - `math` -> `import math`

  - ma -> from numpy import ma
  - cbook -> from matplotlib import cbook
  - docstring -> from matplotlib import docstring
  - is_sequence_of_strings -> from matplotlib.cbook import is_sequence_of_strings
  - is_string_like -> from matplotlib.cbook import is_string_like
  - iterable -> from matplotlib.cbook import iterable
  - itertools -> import itertools
  - martist -> from matplotlib import artist as martist
  - matplotlib -> import matplotlib
  - mcoll -> from matplotlib import collections as mcoll
  - mcolors -> from matplotlib import colors as mcolors
  - mcontour -> from matplotlib import contour as mcontour
  - mpatches -> from matplotlib import patches as mpatches
  - mpath -> from matplotlib import path as mpath
  - mquiver -> from matplotlib import quiver as mquiver
  - mstack -> from matplotlib import stack as mstack
  - mstream -> from matplotlib import stream as mstream
  - mtable -> from matplotlib import table as mtable

- As part of the refactoring to enable Qt5 support, the module matplotlib.backends.qt4_compat was renamed to matplotlib.backends.qt_compat. qt4_compat is deprecated in 1.4 and will be removed in 1.5.
- The *errorbar()* method has been changed such that the upper and lower limits (*lolims*, *uplims*, *xlolims*, *xuplims*) now point in the correct direction.
- The *fmt* kwarg for *errorbar()* now supports the string 'none' to suppress drawing of a line and markers; use of the *None* object for this is deprecated. The default *fmt* value is changed to the empty string (''), so the line and markers are governed by the *plot()* defaults.
- A bug has been fixed in the path effects rendering of fonts, which now means that the font size is consistent with non-path effect fonts. See https://github.com/matplotlib/matplotlib/issues/2889 for more detail.
- The Sphinx extensions ipython_directive and ipython_console_highlighting have been moved to the IPython project itself. While they remain in Matplotlib for this release, they have been deprecated. Update your extensions in conf.py to point to IPython.sphinxext.ipython_directive instead of matplotlib.sphinxext.ipython_directive.

- In `matplotlib.finance`, almost all functions have been deprecated and replaced with a pair of functions name `*_ochl` and `*_ohlc`. The former is the 'open-close-high-low' order of quotes used previously in this module, and the latter is the 'open-high-low-close' order that is standard in finance.
- For consistency the `face_alpha` keyword to *matplotlib.patheffects.SimplePatchShadow* has been deprecated in favour of the `alpha` keyword. Similarly, the keyword `offset_xy` is now named `offset` across all *AbstractPathEffect*s. `matplotlib.patheffects._Base` has been renamed to *matplotlib.patheffects.AbstractPathEffect*. `matplotlib.patheffect.ProxyRenderer` has been renamed to *matplotlib.patheffects.PathEffectRenderer* and is now a full RendererBase subclass.
- The artist used to draw the outline of a *Figure.colorbar* has been changed from a *matplotlib.lines.Line2D* to *matplotlib.patches.Polygon*, thus `colorbar.ColorbarBase.outline` is now a *matplotlib.patches.Polygon* object.
- The legend handler interface has changed from a callable, to any object which implements the `legend_artists` method (a deprecation phase will see this interface be maintained for v1.4). See *Legend guide* for further details. Further legend changes include:
  - `matplotlib.axes.Axes._get_legend_handles` now returns a generator of handles, rather than a list.
  - The *legend()* function's *loc* positional argument has been deprecated. Use the *loc* keyword argument instead.
- The *rcParams["savefig.transparent"]* (default: `False`) has been added to control default transparency when saving figures.
- Slightly refactored the *Annotation* family. The text location in *Annotation* is now entirely handled by the underlying *Text* object so `.set_position` works as expected. The attributes *xytext* and *textcoords* have been deprecated in favor of *xyann* and *anncoords* so that *Annotation* and *AnnotationBbox* can share a common sensibly named api for getting/setting the location of the text or box.
  - *xyann* -> set the location of the annotation
  - *xy* -> set where the arrow points to
  - *anncoords* -> set the units of the annotation location
  - *xycoords* -> set the units of the point location
  - `set_position()` -> *Annotation* only set location of annotation
- *matplotlib.mlab.specgram*, *matplotlib.mlab.psd*, *matplotlib.mlab.csd*, *matplotlib.mlab.cohere*, `matplotlib.mlab.cohere_pairs`, *matplotlib.pyplot.specgram*, *matplotlib.pyplot.psd*, *matplotlib.pyplot.csd*, and *matplotlib.pyplot.cohere* now raise ValueError where they previously raised AssertionError.
- For *matplotlib.mlab.psd*, *matplotlib.mlab.csd*, *matplotlib.mlab.cohere*, `matplotlib.mlab.cohere_pairs`, *matplotlib.pyplot.specgram*, *matplotlib.pyplot.psd*, *matplotlib.pyplot.csd*, and *matplotlib.pyplot.cohere*, in cases where a shape (n, 1) array is returned, this is now converted to a (n, ) array. Previously, (n, m) arrays

were averaged to an (n, ) array, but (n, 1) arrays were returned unchanged. This change makes the dimensions consistent in both cases.

- Added the *rcParams["axes.formatter.useoffset"]* (default: `True`) to control the default value of *useOffset* in *ticker.ScalarFormatter*
- Added *Formatter* sub-class *StrMethodFormatter* which does the exact same thing as *FormatStrFormatter*, but for new-style formatting strings.
- Deprecated `matplotlib.testing.image_util` and the only function within, `matplotlib.testing.image_util.autocontrast`. These will be removed completely in v1.5.0.
- The `fmt` argument of *plot_date()* has been changed from `bo` to just `o`, so color cycling can happen by default.
- Removed the class `FigureManagerQTAgg` and deprecated `NavigationToolbar2QTAgg` which will be removed in 1.5.
- Removed formerly public (non-prefixed) attributes `rect` and `drawRect` from `FigureCanvasQTAgg`; they were always an implementation detail of the (preserved) `drawRectangle()` function.
- The function signatures of `matplotlib.tight_bbox.adjust_bbox` and `matplotlib.tight_bbox.process_figure_for_rasterizing` have been changed. A new *fixed_dpi* parameter allows for overriding the `figure.dpi` setting instead of trying to deduce the intended behaviour from the file format.
- Added support for horizontal/vertical axes padding to *mpl_toolkits.axes_grid1.axes_grid.ImageGrid* --- argument *axes_pad* can now be tuple-like if separate axis padding is required. The original behavior is preserved.
- Added support for skewed transforms to *matplotlib.transforms.Affine2D*, which can be created using the *skew* and *skew_deg* methods.
- Added clockwise parameter to control sectors direction in *axes.Axes.pie*
- In *matplotlib.lines.Line2D* the *markevery* functionality has been extended. Previously an integer start-index and stride-length could be specified using either a two-element-list or a two-element-tuple. Now this can only be done using a two-element-tuple. If a two-element-list is used then it will be treated as NumPy fancy indexing and only the two markers corresponding to the given indexes will be shown.
- Removed *prop* keyword argument from *mpl_toolkits.axes_grid1.anchored_artists.AnchoredSizeBar* call. It was passed through to the base-class `__init__` and is only used for setting padding. Now *fontproperties* (which is what is really used to set the font properties of *AnchoredSizeBar*) is passed through in place of *prop*. If *fontproperties* is not passed in, but *prop* is, then *prop* is used in place of *fontproperties*. If both are passed in, *prop* is silently ignored.
- The use of the index 0 in *pyplot.subplot* and related commands is deprecated. Due to a lack of validation, calling `plt.subplots(2, 2, 0)` does not raise an exception, but puts an axes in the _last_ position. This is due to the indexing in subplot being 1-based (to mirror MATLAB) so before indexing into the *GridSpec* object used to determine where the axes should go, 1 is subtracted off. Passing in 0 results in passing -1 to *GridSpec* which results in getting the last position back. Even though this behavior is clearly wrong and not intended, we are going through a deprecation cycle in

an abundance of caution that any users are exploiting this 'feature'. The use of 0 as an index will raise a warning in 1.4 and an exception in 1.5.

- Clipping is now off by default on offset boxes.
- Matplotlib now uses a less-aggressive call to `gc.collect(1)` when closing figures to avoid major delays with large numbers of user objects in memory.
- The default clip value of *all* pie artists now defaults to `False`.

### Code removal

- Removed `mlab.levypdf`. The code raised a NumPy error (and has for a long time) and was not the standard form of the Levy distribution. `scipy.stats.levy` should be used instead

## 9.16 Version 1.3

### 9.16.1 What's new in Matplotlib 1.3 (Aug 01, 2013)

**Table of Contents**


- *What's new in Matplotlib 1.3 (Aug 01, 2013)*
  - *New in 1.3.1*
  - *New plotting features*
  - *Updated Axes3D.contour methods*
  - *Drawing*
  - *Text*
  - *Configuration (rcParams)*
  - *Backends*
  - *Documentation and examples*
  - *Infrastructure*


**Note:** matplotlib 1.3 supports Python 2.6, 2.7, 3.2, and 3.3

### New in 1.3.1

1.3.1 is a micro release, primarily dealing with improved setup and handling of dependencies, and correcting and enhancing the documentation.

The following changes were made in 1.3.1 since 1.3.0.

### Enhancements

- Added a context manager for creating multi-page pdfs (see *matplotlib.backends.backend_pdf.PdfPages*).
- The WebAgg backend should now have lower latency over heterogeneous Internet connections.

### Bug fixes

- Histogram plots now contain the endline.
- Fixes to the Molleweide projection.
- Handling recent fonts from Microsoft and Macintosh-style fonts with non-ascii metadata is improved.
- Hatching of fill between plots now works correctly in the PDF backend.
- Tight bounding box support now works in the PGF backend.
- Transparent figures now display correctly in the Qt4Agg backend.
- Drawing lines from one subplot to another now works.
- Unit handling on masked arrays has been improved.

### Setup and dependencies

- Now works with any version of pyparsing 1.5.6 or later, without displaying hundreds of warnings.
- Now works with 64-bit versions of Ghostscript on MS-Windows.
- When installing from source into an environment without Numpy, Numpy will first be downloaded and built and then used to build matplotlib.
- Externally installed backends are now always imported using a fully-qualified path to the module.
- Works with newer version of wxPython.
- Can now build with a PyCXX installed globally on the system from source.
- Better detection of Gtk3 dependencies.

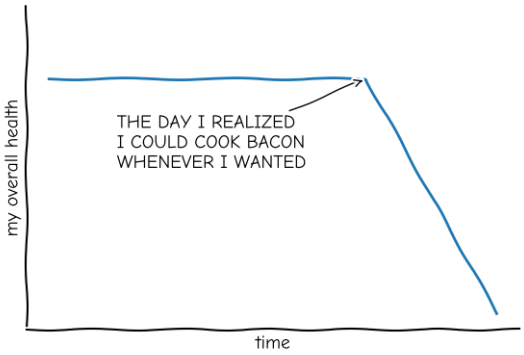
### Testing

- Tests should now work in non-English locales.
- PEP8 conformance tests now report on locations of issues.

### New plotting features

#### xkcd-style sketch plotting

To give your plots a sense of authority that they may be missing, Michael Droettboom (inspired by the work of many others in PR #1329) has added an xkcd-style sketch plotting mode. To use it, simply call *matplotlib.pyplot.xkcd* before creating your plot. For really fine control, it is also possible to modify each artist's sketch parameters individually with *matplotlib.artist.Artist.set_sketch_params()*.




"Stove Ownership" from xkcd by Randall Munroe

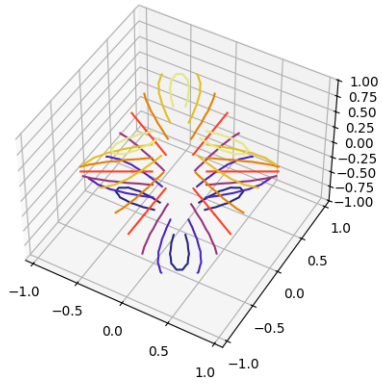
### Updated Axes3D.contour methods

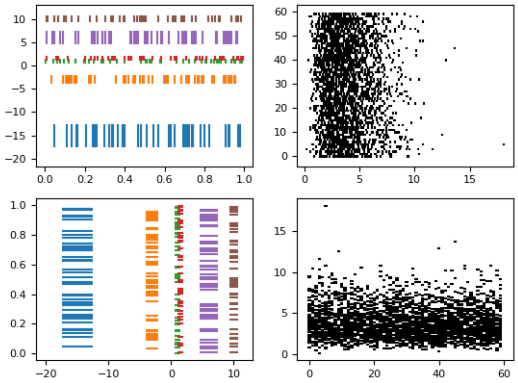
Damon McDougall updated the *tricontour()* and *tricontourf()* methods to allow 3D contour plots on arbitrary unstructured user-specified triangulations.

### New eventplot plot type

Todd Jennings added a *eventplot()* function to create multiple rows or columns of identical line segments

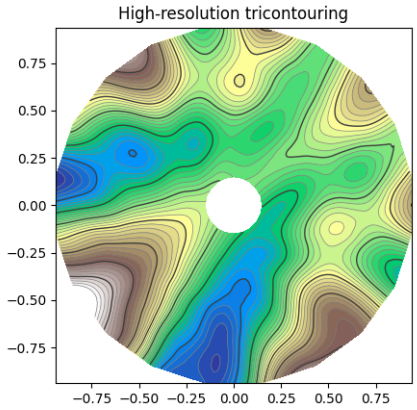
As part of this feature, there is a new *EventCollection* class that allows for plotting and manipulating rows or columns of identical line segments.


1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0


10
5
0
−5
−10
−15
−20
0.0
0.2
0.4
0.6
0.8
1.0
60
50
40
30
20
10
0
0
5
10
15
1.0
0.8
0.6
0.4
0.2
0.0
−20
−10
0
10
15
10
5
0
0
20
40
60

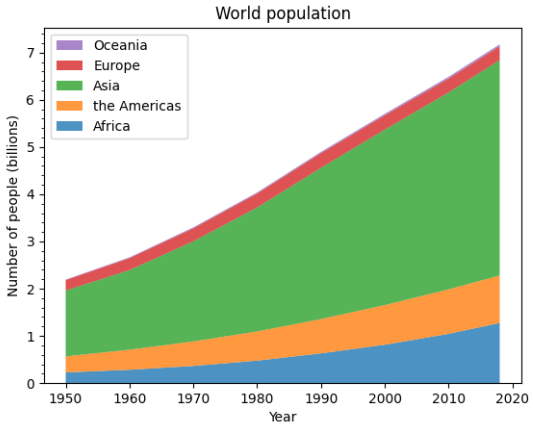
### Triangular grid interpolation

Geoffroy Billotey and Ian Thomas added classes to perform interpolation within triangular grids: (`LinearTriInterpolator` and `CubicTriInterpolator`) and a utility class to find the triangles in which points lie (`TrapezoidMapTriFinder`). A helper class to perform mesh refinement and smooth contouring was also added (`UniformTriRefiner`). Finally, a class implementing some basic tools for triangular mesh improvement was added (`TriAnalyzer`).




### Baselines for stackplot

Till Stensitzki added non-zero baselines to `stackplot()`. They may be symmetric or weighted.



### Rectangular colorbar extensions

Andrew Dawson added a new keyword argument *extendrect* to `colorbar()` to optionally make colorbar extensions rectangular instead of triangular.

### More robust boxplots

Paul Hobson provided a fix to the `boxplot()` method that prevent whiskers from being drawn inside the box for oddly distributed data sets.

### Calling subplot() without arguments

A call to `subplot()` without any arguments now acts the same as `subplot(111)` or `subplot(1, 1, 1)` -- it creates one axes for the whole figure. This was already the behavior for both `axes()` and `subplots()`, and now this consistency is shared with `subplot()`.

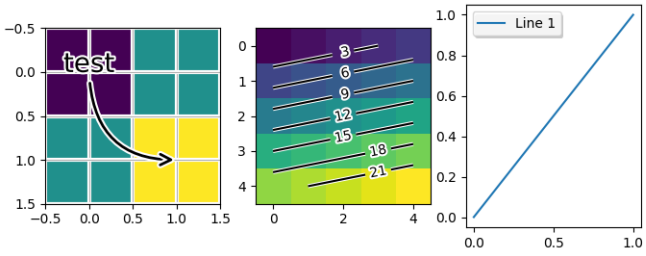
### Drawing

### Independent alpha values for face and edge colors

Wes Campaigne modified how `Patch` objects are drawn such that (for backends supporting transparency) you can set different alpha values for faces and edges, by specifying their colors in RGBA format. Note that if you set the alpha attribute for the patch object (e.g. using `set_alpha()` or the `alpha` keyword argument), that value will override the alpha components set in both the face and edge colors.

### Path effects on lines

Thanks to Jae-Joon Lee, path effects now also work on plot lines.



### Easier creation of colormap and normalizer for levels with colors

Phil Elson added the *matplotlib.colors.from_levels_and_colors()* function to easily create a colormap and normalizer for representation of discrete colors for plot types such as *matplotlib.pyplot.pcolormesh()*, with a similar interface to that of *matplotlib.pyplot.contourf*.

### Full control of the background color

Wes Campaigne and Phil Elson fixed the Agg backend such that PNGs are now saved with the correct background color when `fig.patch.get_alpha()` is not 1.

### Improved `bbox_inches="tight"` functionality

Passing `bbox_inches="tight"` through to *pyplot.savefig* now takes into account *all* artists on a figure - this was previously not the case and led to several corner cases which did not function as expected.

### Initialize a rotated rectangle

Damon McDougall extended the *Rectangle* constructor to accept an *angle* kwarg, specifying the rotation of a rectangle in degrees.

## Text

### Anchored text support

The SVG and pgf backends are now able to save text alignment information to their output formats. This allows to edit text elements in saved figures, using Inkscape for example, while preserving their intended position. For SVG please note that you'll have to disable the default text-to-path conversion (`mpl.rc('svg', fonttype='none')`).

### Better vertical text alignment and multi-line text

The vertical alignment of text is now consistent across backends. You may see small differences in text placement, particularly with rotated text.

If you are using a custom backend, note that the *draw_text* renderer method is now passed the location of the baseline, not the location of the bottom of the text bounding box.

Multi-line text will now leave enough room for the height of very tall or very low text, such as superscripts and subscripts.

### Left and right side axes titles

Andrew Dawson added the ability to add axes titles flush with the left and right sides of the top of the axes using a new keyword argument *loc* to *title()*.

### Improved manual contour plot label positioning

Brian Mattern modified the manual contour plot label positioning code to interpolate along line segments and find the actual closest point on a contour to the requested position. Previously, the closest path vertex was used, which, in the case of straight contours was sometimes quite distant from the requested location. Much more precise label positioning is now possible.

### Configuration (rcParams)

### Quickly find rcParams

Phil Elson made it easier to search for rcParameters by passing a valid regular expression to *matplotlib.RcParams.find_all()*. *matplotlib.RcParams* now also has a pretty repr and str representation so that search results are printed prettily:

```
>>> import matplotlib
>>> print(matplotlib.rcParams.find_all('\.size'))
RcParams({'font.size': 12,
          'xtick.major.size': 4,
          'xtick.minor.size': 2,
          'ytick.major.size': 4,
          'ytick.minor.size': 2})
```

### `axes.xmargin` and `axes.ymargin` added to rcParams

*rcParams["axes.xmargin"]* (default: `0.05`) and *rcParams["axes.ymargin"]* (default: `0.05`) were added to configure the default margins used. Previously they were hard-coded to default to 0, default value of both rcParam values is 0.

### Changes to font rcParams

The `font.*` rcParams now affect only text objects created after the rcParam has been set, and will not retroactively affect already existing text objects. This brings their behavior in line with most other rcParams.

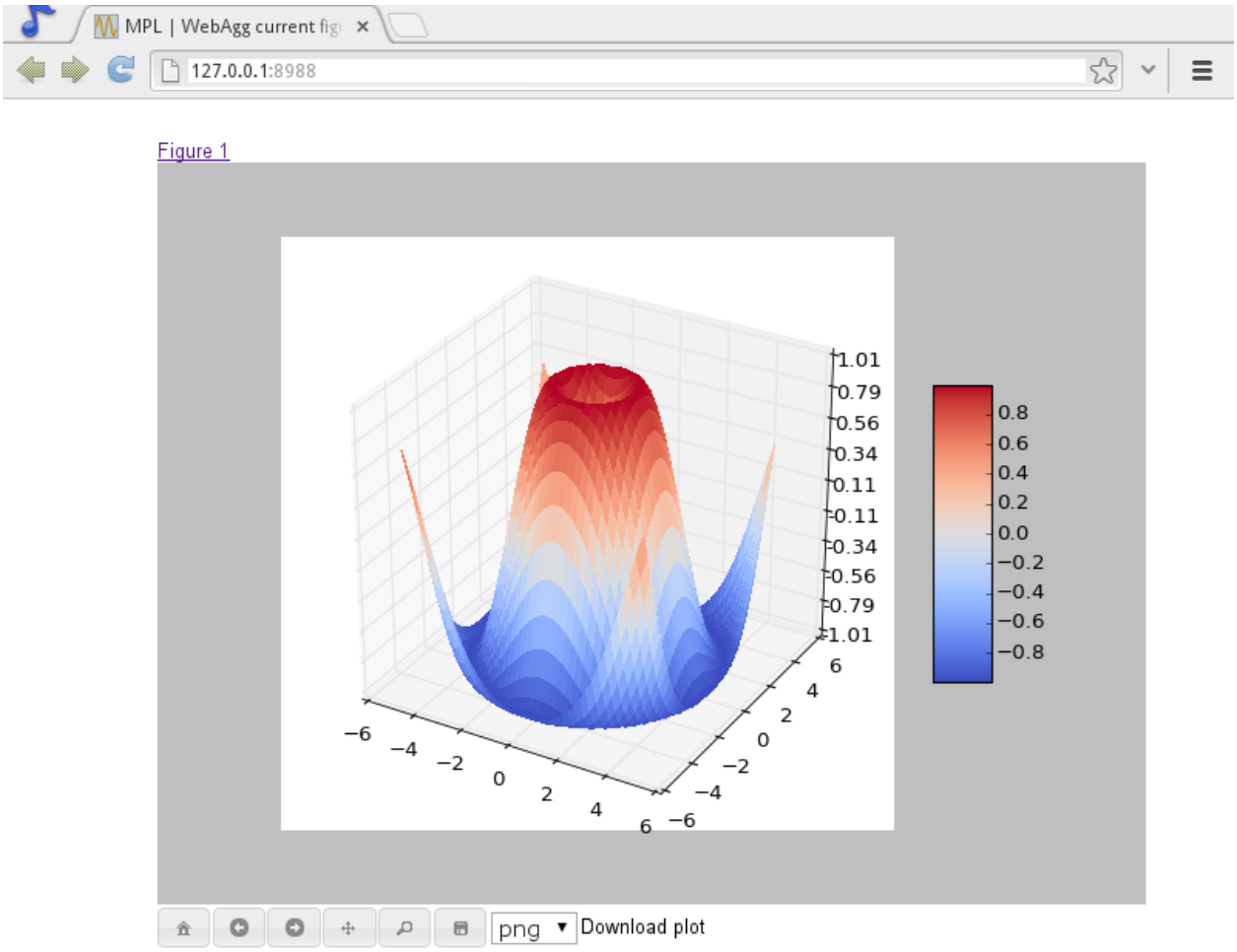
### Added `rcParams["savefig.jpeg_quality"]`

rcParam value `rcParams["savefig.jpeg_quality"]` was added so that the user can configure the default quality used when a figure is written as a JPEG. The default quality is 95; previously, the default quality was 75. This change minimizes the artifacting inherent in JPEG images, particularly with images that have sharp changes in color as plots often do.

## Backends

### WebAgg backend

Michael Droettboom, Phil Elson and others have developed a new backend, WebAgg, to display figures in a web browser. It works with animations as well as being fully interactive.




Future versions of matplotlib will integrate this backend with the IPython notebook for a fully web browser based plotting frontend.

### Remember save directory

Martin Spacek made the save figure dialog remember the last directory saved to. The default is configurable with the new *rcParams["savefig.directory"]* (default: '~') rcParam in matplotlibrc.

### Documentation and examples

### Numpydoc docstrings

Nelle Varoquaux has started an ongoing project to convert matplotlib's docstrings to numpydoc format. See MEP10 for more information.

### Example reorganization

Tony Yu has begun work reorganizing the examples into more meaningful categories. The new gallery page is the fruit of this ongoing work. See MEP12 for more information.

### Examples now use subplots()

For the sake of brevity and clarity, most of the *examples* now use the newer *subplots()*, which creates a figure and one (or multiple) axes object(s) in one call. The old way involved a call to *figure()*, followed by one (or multiple) *subplot()* calls.

### Infrastructure

### Housecleaning

A number of features that were deprecated in 1.2 or earlier, or have not been in a working state for a long time have been removed. Highlights include removing the Qt version 3 backends, and the FltkAgg and Emf backends. See *API Changes in 1.3.x* for a complete list.

### New setup script

matplotlib 1.3 includes an entirely rewritten setup script. We now ship fewer dependencies with the tarballs and installers themselves. Notably, pytz, dateutil, pyparsing and six are no longer included with matplotlib. You can either install them manually first, or let pip install them as dependencies along with matplotlib. It is now possible to not include certain subcomponents, such as the unit test data, in the install. See `setup.cfg.template` for more information.

#### XDG base directory support

On Linux, matplotlib now uses the XDG base directory specification to find the `matplotlibrc` configuration file. `matplotlibrc` should now be kept in `~/.config/matplotlib`, rather than `~/.matplotlib`. If your configuration is found in the old location, it will still be used, but a warning will be displayed.

#### Catch opening too many figures using pyplot

Figures created through *`pyplot.figure`* are retained until they are explicitly closed. It is therefore common for new users of matplotlib to run out of memory when creating a large series of figures in a loop without closing them.

matplotlib will now display a `RuntimeWarning` when too many figures have been opened at once. By default, this is displayed for 20 or more figures, but the exact number may be controlled using the `figure.max_open_warning` rcParam.

### 9.16.2 API Changes in 1.3.x

#### Changes in 1.3.1

It is rare that we make an API change in a micro release, however, for 1.3.1 since 1.3.0 the following change was made:

- `text.Text.cached` (used to cache font objects) has been made into a private variable. Among the obvious encapsulation benefit, this removes this confusing-looking member from the documentation.
- The method *`hist()`* now always returns bin occupancies as an array of type `float`. Previously, it was sometimes an array of type `int`, depending on the call.

#### Code removal

- The following items that were deprecated in version 1.2 or earlier have now been removed completely.
  - The Qt 3.x backends (`qt` and `qtagg`) have been removed in favor of the Qt 4.x backends (`qt4` and `qt4agg`).
  - The FltkAgg and Emf backends have been removed.
  - The `matplotlib.nxutils` module has been removed. Use the functionality on *`matplotlib.path.Path.contains_point`* and friends instead.
  - Instead of `axes.Axes.get_frame`, use `axes.Axes.patch`.
  - The following keyword arguments to the *`legend`* function have been renamed:
    - *pad* -> *borderpad*
    - *labelsep* -> *labelspacing*

    * *handlelen -> handlelength*
    * *handletextsep -> handletextpad*
    * *axespad -> borderaxespad*

    Related to this, the following rcParams have been removed:

    * `legend.pad`,
    * `legend.labelsep`,
    * `legend.handlelen`,
    * `legend.handletextsep` and
    * `legend.axespad`

  - For the *hist* function, instead of *width*, use *rwidth* (relative width).
  - On *patches.Circle*, the *resolution* keyword argument has been removed. For a circle made up of line segments, use *patches.CirclePolygon*.
  - The printing functions in the Wx backend have been removed due to the burden of keeping them up-to-date.
  - `mlab.liaupunov` has been removed.
  - `mlab.save`, `mlab.load`, `pylab.save` and `pylab.load` have been removed. We recommend using `numpy.savetxt` and `numpy.loadtxt` instead.
  - `widgets.HorizontalSpanSelector` has been removed. Use *widgets.SpanSelector* instead.

### Code deprecation

- The CocoaAgg backend has been deprecated, with the possibility for deletion or resurrection in a future release.
- The top-level functions in *matplotlib.path* that are implemented in C++ were never meant to be public. Instead, users should use the Pythonic wrappers for them in the *path.Path* and *collections.Collection* classes. Use the following mapping to update your code:
  - `point_in_path` -> *path.Path.contains_point*
  - `get_path_extents` -> *path.Path.get_extents*
  - `point_in_path_collection` -> *collections.Collection.contains*
  - `path_in_path` -> *path.Path.contains_path*
  - `path_intersects_path` -> *path.Path.intersects_path*
  - `convert_path_to_polygons` -> *path.Path.to_polygons*
  - `cleanup_path` -> *path.Path.cleaned*
  - `points_in_path` -> *path.Path.contains_points*

  - clip_path_to_rect -> *path.Path.clip_to_bbox*

- matplotlib.colors.normalize and matplotlib.colors.no_norm have been deprecated in favour of *matplotlib.colors.Normalize* and *matplotlib.colors.NoNorm* respectively.
- The *ScalarMappable* class' set_colorbar method is now deprecated. Instead, the *matplotlib.cm.ScalarMappable.colorbar* attribute should be used. In previous Matplotlib versions this attribute was an undocumented tuple of (colorbar_instance, colorbar_axes) but is now just colorbar_instance. To get the colorbar axes it is possible to just use the matplotlib.colorbar.ColorbarBase.ax attribute on a colorbar instance.
- The matplotlib.mpl module is now deprecated. Those who relied on this module should transition to simply using import matplotlib as mpl.

### Code changes

- *Patch* now fully supports using RGBA values for its facecolor and edgecolor attributes, which enables faces and edges to have different alpha values. If the *Patch* object's alpha attribute is set to anything other than None, that value will override any alpha-channel value in both the face and edge colors. Previously, if *Patch* had alpha=None, the alpha component of edgecolor would be applied to both the edge and face.
- The optional isRGB argument to *set_foreground()* (and the other GraphicsContext classes that descend from it) has been renamed to isRGBA, and should now only be set to True if the fg color argument is known to be an RGBA tuple.
- For *Patch*, the capstyle used is now butt, to be consistent with the default for most other objects, and to avoid problems with non-solid linestyle appearing solid when using a large linewidth. Previously, *Patch* used capstyle='projecting'.
- *Path* objects can now be marked as *readonly* by passing readonly=True to its constructor. The built-in path singletons, obtained through Path.unit* class methods return readonly paths. If you have code that modified these, you will need to make a deepcopy first, using either:

```
import copy
path = copy.deepcopy(Path.unit_circle())

# or

path = Path.unit_circle().deepcopy()
```

  Deep copying a *Path* always creates an editable (i.e. non-readonly) *Path*.

- The list at Path.NUM_VERTICES was replaced by a dictionary mapping Path codes to the number of expected vertices at *NUM_VERTICES_FOR_CODE*.
- To support XKCD style plots, the matplotlib.path.cleanup_path method's signature was updated to require a sketch argument. Users of matplotlib.path.cleanup_path are encouraged to use the new *cleaned()* Path method.

- Data limits on a plot now start from a state of having "null" limits, rather than limits in the range (0, 1). This has an effect on artists that only control limits in one direction, such as *axes.Axes.axvline* and *axes.Axes.axhline*, since their limits will no longer also include the range (0, 1). This fixes some problems where the computed limits would be dependent on the order in which artists were added to the axes.
- Fixed a bug in setting the position for the right/top spine with data position type. Previously, it would draw the right or top spine at +1 data offset.
- In *FancyArrow*, the default arrow head width, `head_width`, has been made larger to produce a visible arrow head. The new value of this kwarg is `head_width = 20 * width`.
- It is now possible to provide `number of levels + 1` colors in the case of `extend='both'` for contourf (or just `number of levels` colors for an extend value `min` or `max`) such that the resulting colormap's `set_under` and `set_over` are defined appropriately. Any other number of colors will continue to behave as before (if more colors are provided than levels, the colors will be unused). A similar change has been applied to contour, where `extend='both'` would expect `number of levels + 2` colors.
- A new keyword *extendrect* in *colorbar()* and *ColorbarBase* allows one to control the shape of colorbar extensions.
- The extension of *MultiCursor* to both vertical (default) and/or horizontal cursor implied that `self.line` is replaced by `self.vline` for vertical cursors lines and `self.hline` is added for the horizontal cursors lines.
- On POSIX platforms, the `matplotlib.cbook.report_memory` function raises `NotImplementedError` instead of `OSError` if the **`ps`** command cannot be run.
- The `matplotlib.cbook.check_output` function has been moved to `matplotlib.compat.subprocess`.

### Configuration and rcParams

- On Linux, the user-specific `matplotlibrc` configuration file is now located in `~/.config/matplotlib/matplotlibrc` to conform to the XDG Base Directory Specification.
- The `font.*` rcParams now affect only text objects created after the rcParam has been set, and will not retroactively affect already existing text objects. This brings their behavior in line with most other rcParams.
- Removed call of *grid()* in `matplotlib.pyplot.plotfile`. To draw the axes grid, set the `axes.grid` rcParam to *True*, or explicitly call *grid()*.

## 9.17 Version 1.2

### 9.17.1 What's new in Matplotlib 1.2.2

**Table of Contents**


- *What's new in Matplotlib 1.2.2*
  - *Improved collections*
  - *Multiple images on same axes are correctly transparent*


#### Improved collections

The individual items of a collection may now have different alpha values and be rendered correctly. This also fixes a bug where collections were always filled in the PDF backend.

#### Multiple images on same axes are correctly transparent

When putting multiple images onto the same axes, the background color of the axes will now show through correctly.

### 9.17.2 What's new in Matplotlib 1.2 (Nov 9, 2012)

**Table of Contents**


- *What's new in Matplotlib 1.2 (Nov 9, 2012)*
  - *Python 3.x support*
  - *PGF/TikZ backend*
  - *Locator interface*
  - *Tri-Surface Plots*
  - *Control the lengths of colorbar extensions*
  - *Figures are picklable*
  - *Set default bounding box in matplotlibrc*
  - *New Boxplot Functionality*
  - *New RC parameter functionality*
  - *Streamplot*

- *New hist functionality*
- *Updated shipped dependencies*
- *Face-centred colors in tripcolor plots*
- *Hatching patterns in filled contour plots, with legends*
- *Known issues in the matplotlib 1.2 release*


**Note:** matplotlib 1.2 supports Python 2.6, 2.7, and 3.1

### Python 3.x support

Matplotlib 1.2 is the first version to support Python 3.x, specifically Python 3.1 and 3.2. To make this happen in a reasonable way, we also had to drop support for Python versions earlier than 2.6.

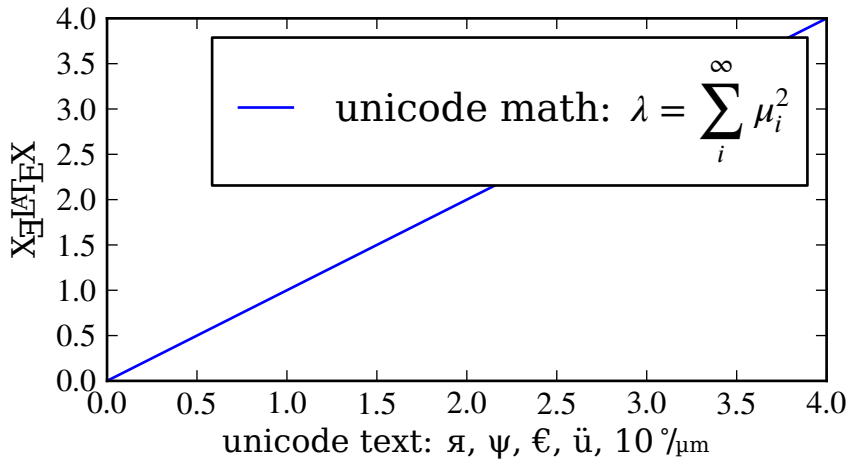
This work was done by Michael Droettboom, the Cape Town Python Users' Group, many others and supported financially in part by the SAGE project.

The following GUI backends work under Python 3.x: Gtk3Cairo, Qt4Agg, TkAgg and MacOSX. The other GUI backends do not yet have adequate bindings for Python 3.x, but continue to work on Python 2.6 and 2.7, particularly the Qt and QtAgg backends (which have been deprecated). The non-GUI backends, such as PDF, PS and SVG, work on both Python 2.x and 3.x.

Features that depend on the Python Imaging Library, such as JPEG handling, do not work, since the version of PIL for Python 3.x is not sufficiently mature.

### PGF/TikZ backend

Peter Würtz wrote a backend that allows matplotlib to export figures as drawing commands for LaTeX. These can be processed by PdfLaTeX, XeLaTeX or LuaLaTeX using the PGF/TikZ package. Usage examples and documentation are found in *Text rendering with XeLaTeX/LuaLaTeX via the pgf backend*.



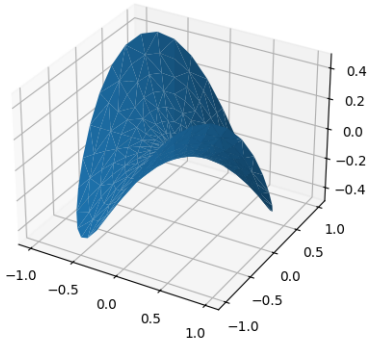
### Locator interface

Philip Elson exposed the intelligence behind the tick Locator classes with a simple interface. For instance, to get no more than 5 sensible steps which span the values 10 and 19.5:

```
>>> import matplotlib.ticker as mticker
>>> locator = mticker.MaxNLocator(nbins=5)
>>> print(locator.tick_values(10, 19.5))
[ 10.  12.  14.  16.  18.  20.]
```

### Tri-Surface Plots

Damon McDougall added a new plotting method for the *mplot3d* toolkit called *plot_trisurf()*.




### Control the lengths of colorbar extensions

Andrew Dawson added a new keyword argument *extendfrac* to *colorbar()* to control the length of minimum and maximum colorbar extensions.

### Figures are picklable

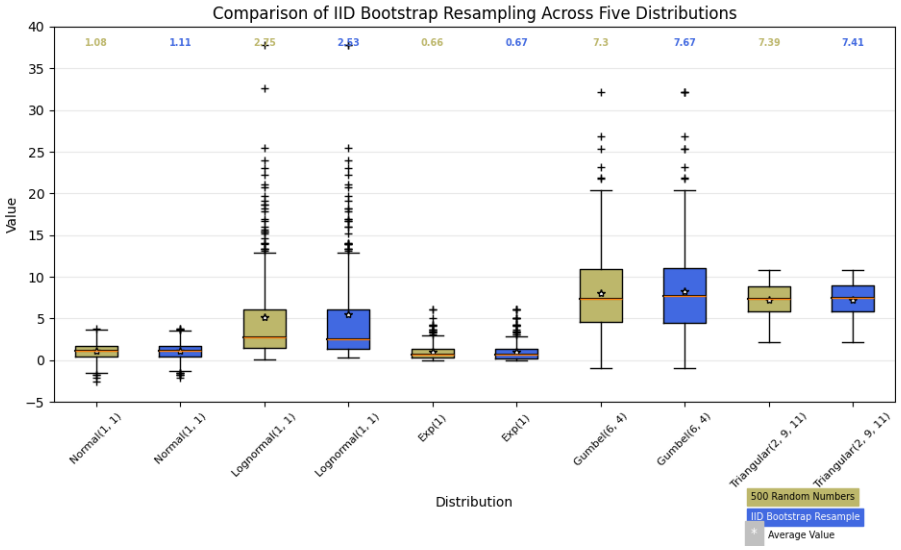
Philip Elson added an experimental feature to make figures picklable for quick and easy short-term storage of plots. Pickle files are not designed for long term storage, are unsupported when restoring a pickle saved in another matplotlib version and are insecure when restoring a pickle from an untrusted source. Having said this, they are useful for short term storage for later modification inside matplotlib.

### Set default bounding box in matplotlibrc

Two new defaults are available in the matplotlibrc configuration file: `savefig.bbox`, which can be set to 'standard' or 'tight', and `savefig.pad_inches`, which controls the bounding box padding.

### New Boxplot Functionality

Users can now incorporate their own methods for computing the median and its confidence intervals into the *`boxplot`* method. For every column of data passed to boxplot, the user can specify an accompanying median and confidence interval.




### New RC parameter functionality

Matthew Emmett added a function and a context manager to help manage RC parameters: *`rc_file()`* and *`rc_context`*. To load RC parameters from a file:

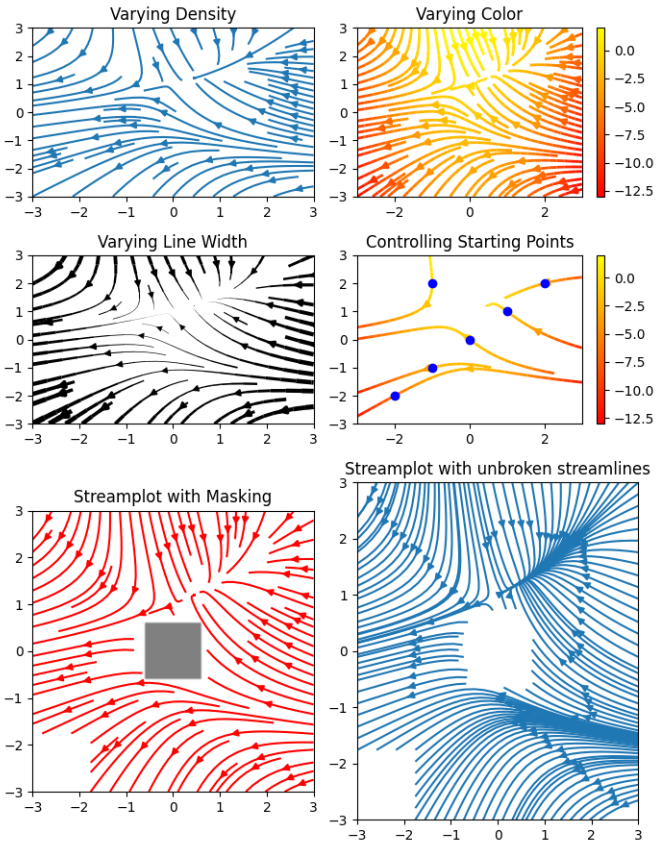
```
>>> mpl.rc_file('mpl.rc')
```

To temporarily use RC parameters:

```
>>> with mpl.rc_context(fname='mpl.rc', rc={'text.usetex': True}):
>>>     ...
```

### Streamplot

Tom Flannaghan and Tony Yu have added a new *streamplot()* function to plot the streamlines of a vector field. This has been a long-requested feature and complements the existing *quiver()* function for plotting vector fields. In addition to simply plotting the streamlines of the vector field, *streamplot()* allows users to map the colors and/or line widths of the streamlines to a separate parameter, such as the speed or local intensity of the vector field.




### New hist functionality

Nic Eggert added a new *stacked* kwarg to *hist()* that allows creation of stacked histograms using any of the histogram types. Previously, this functionality was only available by using the "barstacked" histogram type. Now, when `stacked=True` is passed to the function, any of the histogram types can be stacked. The "barstacked" histogram type retains its previous functionality for backwards compatibility.

### Updated shipped dependencies

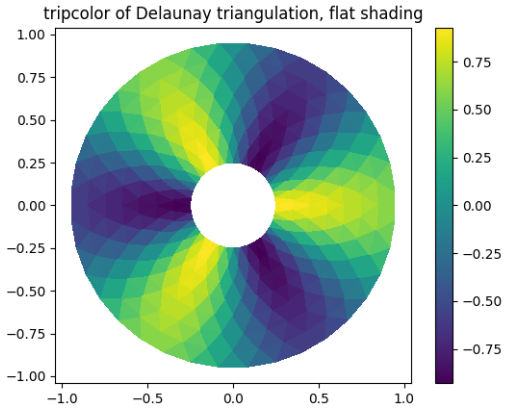
The following dependencies that ship with matplotlib and are optionally installed alongside it have been updated:

- pytz 2012d
- **dateutil 1.5 on Python 2.x,**

  and 2.1 on Python 3.x

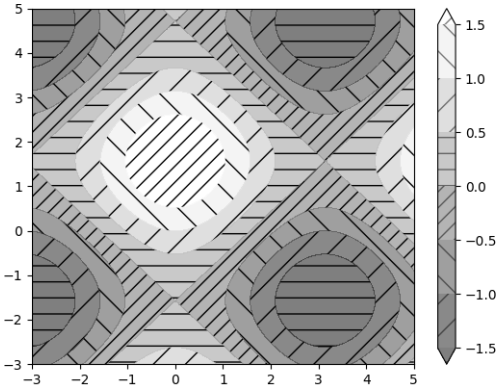
### Face-centred colors in tripcolor plots

Ian Thomas extended `tripcolor()` to allow one color value to be specified for each triangular face rather than for each point in a triangulation.




### Hatching patterns in filled contour plots, with legends

Phil Elson added support for hatching to `contourf()`, together with the ability to use a legend to identify contoured ranges.



#### Known issues in the matplotlib 1.2 release

- When using the Qt4Agg backend with IPython 0.11 or later, the save dialog will not display. This should be fixed in a future version of IPython.

### 9.17.3 API Changes in 1.2.x

- The `classic` option of the rc parameter `toolbar` is deprecated and will be removed in the next release.
- The `matplotlib.cbook.isvector` method has been removed since it is no longer functional.
- The `rasterization_zorder` property on *Axes* sets a zorder below which artists are rasterized. This has defaulted to -30000.0, but it now defaults to *None*, meaning no artists will be rasterized. In order to rasterize artists below a given zorder value, *set_rasterization_zorder* must be explicitly called.
- In *scatter()*, and *scatter*, when specifying a marker using a tuple, the angle is now specified in degrees, not radians.
- Using *twinx()* or *twiny()* no longer overrides the current locaters and formatters on the axes.
- In *contourf()*, the handling of the *extend* kwarg has changed. Formerly, the extended ranges were mapped after to 0, 1 after being normed, so that they always corresponded to the extreme values of the colormap. Now they are mapped outside this range so that they correspond to the special colormap values determined by the *set_under()* and *set_over()* methods, which default to the colormap end points.
- The new rc parameter `savefig.format` replaces `cairo.format` and `savefig.extension`, and sets the default file format used by *matplotlib.figure.Figure.savefig()*.
- In *pyplot.pie()* and *axes.Axes.pie()*, one can now set the radius of the pie; setting the *radius* to 'None' (the default value), will result in a pie with a radius of 1 as before.
- Use of `matplotlib.projections.projection_factory` is now deprecated in favour of axes class identification using `matplotlib.projections.process_projection_requirements` followed by direct axes class invocation (at the time of writing, functions which do this are: *add_axes()*, *add_subplot()* and *gca()*). Therefore:

```
key = figure._make_key(*args, **kwargs)
ispolar = kwargs.pop('polar', False)
projection = kwargs.pop('projection', None)
if ispolar:
    if projection is not None and projection != 'polar':
        raise ValueError('polar and projection args are inconsistent')
    projection = 'polar'
ax = projection_factory(projection, self, rect, **kwargs)
key = self._make_key(*args, **kwargs)

# is now
```

(continues on next page)

(continued from previous page)

```
projection_class, kwargs, key = \
                     process_projection_requirements(self, *args, **kwargs)
ax = projection_class(self, rect, **kwargs)
```

This change means that third party objects can expose themselves as Matplotlib axes by providing a `_as_mpl_axes` method. See *matplotlib.projections* for more detail.

- A new keyword *extendfrac* in *colorbar()* and *ColorbarBase* allows one to control the size of the triangular minimum and maximum extensions on colorbars.
- A new keyword *capthick* in *errorbar()* has been added as an intuitive alias to the *markeredgewidth* and *mew* keyword arguments, which indirectly controlled the thickness of the caps on the errorbars. For backwards compatibility, specifying either of the original keyword arguments will override any value provided by *capthick*.
- Transform subclassing behaviour is now subtly changed. If your transform implements a non-affine transformation, then it should override the `transform_non_affine` method, rather than the generic `transform` method. Previously transforms would define `transform` and then copy the method into `transform_non_affine`:

```
class MyTransform(mtrans.Transform):
    def transform(self, xy):
        ...
    transform_non_affine = transform
```

This approach will no longer function correctly and should be changed to:

```
class MyTransform(mtrans.Transform):
    def transform_non_affine(self, xy):
        ...
```

- Artists no longer have `x_isdata` or `y_isdata` attributes; instead any artist's transform can be interrogated with `artist_instance.get_transform().contains_branch(ax.transData)`
- Lines added to an axes now take into account their transform when updating the data and view limits. This means transforms can now be used as a pre-transform. For instance:

```
>>> import matplotlib.pyplot as plt
>>> import matplotlib.transforms as mtrans
>>> ax = plt.axes()
>>> ax.plot(range(10), transform=mtrans.Affine2D().scale(10) + ax.
 ↪transData)
>>> print(ax.viewLim)
Bbox('array([[  0.,   0.],\n       [ 90.,  90.]])')
```

- One can now easily get a transform which goes from one transform's coordinate system to another, in an optimized way, using the new subtract method on a transform. For instance, to go from data coordinates to axes coordinates:

```
>>> import matplotlib.pyplot as plt
>>> ax = plt.axes()
>>> data2ax = ax.transData - ax.transAxes
>>> print(ax.transData.depth, ax.transAxes.depth)
3, 1
>>> print(data2ax.depth)
2
```

for versions before 1.2 this could only be achieved in a sub-optimal way, using `ax.transData + ax.transAxes.inverted()` (depth is a new concept, but had it existed it would return 4 for this example).

- `twinx` and `twiny` now returns an instance of SubplotBase if parent axes is an instance of SubplotBase.
- All Qt3-based backends are now deprecated due to the lack of py3k bindings. Qt and QtAgg backends will continue to work in v1.2.x for py2.6 and py2.7. It is anticipated that the Qt3 support will be completely removed for the next release.
- `matplotlib.colors.ColorConverter`, *`Colormap`* and *`Normalize`* now subclasses `object`
- ContourSet instances no longer have a `transform` attribute. Instead, access the transform with the `get_transform` method.

## 9.18 Version 1.1

### 9.18.1 What's new in Matplotlib 1.1 (Nov 02, 2011)

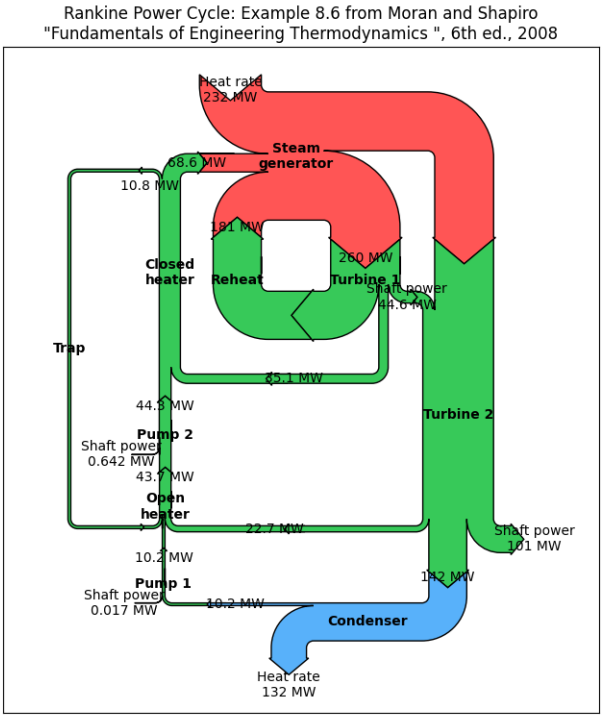
**Table of Contents**


- *What's new in Matplotlib 1.1 (Nov 02, 2011)*
  - *Sankey Diagrams*
  - *Animation*
  - *Tight Layout*
  - *PyQT4, PySide, and IPython*
  - *Legend*
  - *mplot3d*
  - *Numerix support removed*
  - *Markers*
  - *Other improvements*

**Note:** matplotlib 1.1 supports Python 2.4 to 2.7

## Sankey Diagrams

Kevin Davies has extended Yannick Copin's original Sankey example into a module (*sankey*) and provided new examples (*The Sankey class*, *Long chain of connections using Sankey*, *Rankine power cycle*).




## Animation

Ryan May has written a backend-independent framework for creating animated figures. The *animation* module is intended to replace the backend-specific examples formerly in the *Examples* listings. Examples using the new framework are in *Animation*; see the entrancing `double pendulum <gallery/animation/double_pendulum_sgskip.py>` which uses *matplotlib.animation.Animation.save()* to create the movie below.

This should be considered as a beta release of the framework; please try it and provide feedback.

### Tight Layout

A frequent issue raised by users of matplotlib is the lack of a layout engine to nicely space out elements of the plots. While matplotlib still adheres to the philosophy of giving users complete control over the placement of plot elements, Jae-Joon Lee created the `matplotlib.tight_layout` module and introduced a new command *`tight_layout()`* to address the most common layout issues.

The usage of this functionality can be as simple as

```
plt.tight_layout()
```

and it will adjust the spacing between subplots so that the axis labels do not overlap with neighboring subplots. A *Tight layout guide* has been created to show how to use this new tool.

### PyQT4, PySide, and IPython

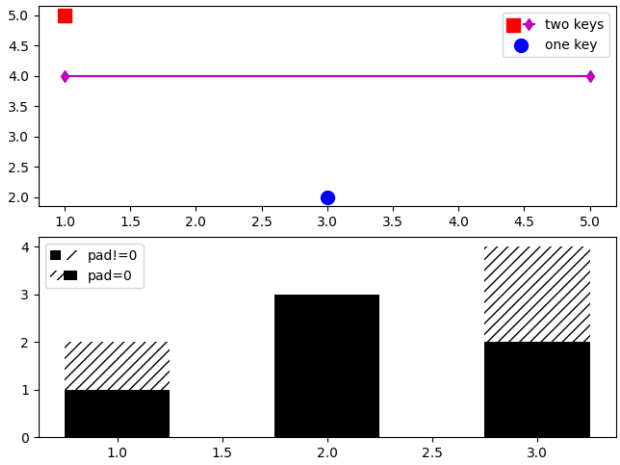
Gerald Storer made the Qt4 backend compatible with PySide as well as PyQT4. At present, however, PySide does not support the PyOS_InputHook mechanism for handling gui events while waiting for text input, so it cannot be used with the new version 0.11 of IPython. Until this feature appears in PySide, IPython users should use the PyQT4 wrapper for QT4, which remains the matplotlib default.

An rcParam entry, "backend.qt4", has been added to allow users to select PyQt4, PyQt4v2, or PySide. The latter two use the Version 2 Qt API. In most cases, users can ignore this rcParam variable; it is available to aid in testing, and to provide control for users who are embedding matplotlib in a PyQt4 or PySide app.

### Legend

Jae-Joon Lee has improved plot legends. First, legends for complex plots such as *`stem()`* plots will now display correctly. Second, the 'best' placement of a legend has been improved in the presence of NANs.

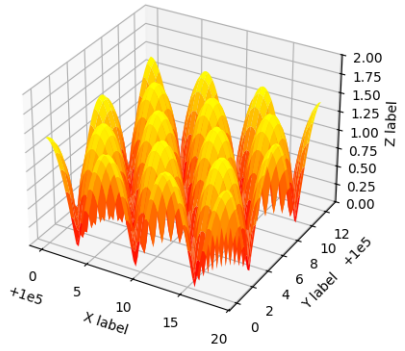
See the *Legend guide* for more detailed explanation and examples.



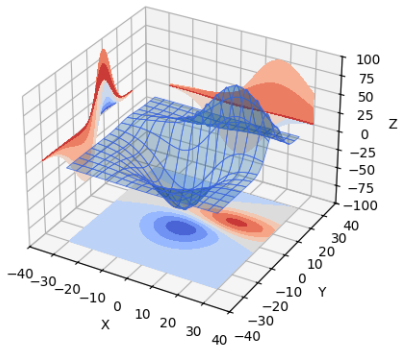
### mplot3d

In continuing the efforts to make 3D plotting in matplotlib just as easy as 2D plotting, Ben Root has made several improvements to the `mplot3d` module.

- `Axes3D` has been improved to bring the class towards feature-parity with regular Axes objects
- Documentation for *The mplot3d toolkit* was significantly expanded
- Axis labels and orientation improved
- Most 3D plotting functions now support empty inputs
- Ticker offset display added:




- `contourf()` gains *zdir* and *offset* kwargs. You can now do this:



### Numerix support removed

After more than two years of deprecation warnings, Numerix support has now been completely removed from matplotlib.

### Markers

The list of available markers for *plot()* and *scatter()* has now been merged. While they were mostly similar, some markers existed for one function, but not the other. This merge did result in a conflict for the 'd' diamond marker. Now, 'd' will be interpreted to always mean "thin" diamond while 'D' will mean "regular" diamond.

Thanks to Michael Droettboom for this effort.

### Other improvements

- Unit support for polar axes and *arrow()*
- *PolarAxes* gains getters and setters for "theta_direction", and "theta_offset" to allow for theta to go in either the clock-wise or counter-clockwise direction and to specify where zero degrees should be placed. *set_theta_zero_location()* is an added convenience function.
- Fixed error in argument handling for tri-functions such as *tripcolor()*
- `axes.labelweight` parameter added to rcParams.
- For *imshow()*, *interpolation='nearest'* will now always perform an interpolation. A "none" option has been added to indicate no interpolation at all.
- An error in the Hammer projection has been fixed.
- *clabel* for *contour()* now accepts a callable. Thanks to Daniel Hyams for the original patch.
- Jae-Joon Lee added the *HBoxDivider* and *VBoxDivider* classes.
- Christoph Gohlke reduced memory usage in *imshow()*.
- *scatter()* now accepts empty inputs.
- The behavior for 'symlog' scale has been fixed, but this may result in some minor changes to existing plots. This work was refined by ssyr.
- Peter Butterworth added named figure support to *figure()*.
- Michiel de Hoon has modified the MacOSX backend to make its interactive behavior consistent with the other backends.
- Pim Schellart added a new colormap called "cubehelix". Sameer Grover also added a colormap called "coolwarm". See it and all other colormaps *here*.
- Many bug fixes and documentation improvements.

### 9.18.2 API Changes in 1.1.x

- Added new `matplotlib.sankey.Sankey` for generating Sankey diagrams.
- In `imshow()`, setting *interpolation* to 'nearest' will now always mean that the nearest-neighbor interpolation is performed. If you want the no-op interpolation to be performed, choose 'none'.
- There were errors in how the tri-functions were handling input parameters that had to be fixed. If your tri-plots are not working correctly anymore, or you were working around apparent mistakes, please see issue #203 in the github tracker. When in doubt, use kwargs.
- The 'symlog' scale had some bad behavior in previous versions. This has now been fixed and users should now be able to use it without frustrations. The fixes did result in some minor changes in appearance for some users who may have been depending on the bad behavior.
- There is now a common set of markers for all plotting functions. Previously, some markers existed only for `scatter()` or just for `plot()`. This is now no longer the case. This merge did result in a conflict. The string 'd' now means "thin diamond" while 'D' will mean "regular diamond".

## 9.19 Version 1.0

### 9.19.1 What's new in Matplotlib 1.0 (Jul 06, 2010)

**Table of Contents**

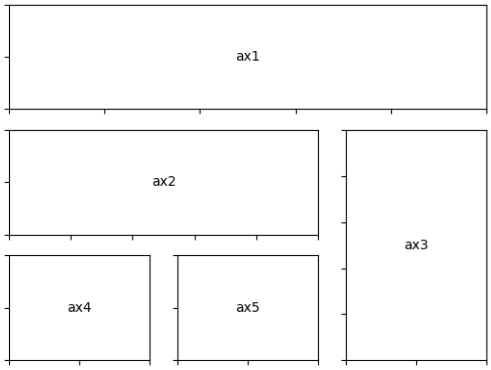

- *What's new in Matplotlib 1.0 (Jul 06, 2010)*
  - *HTML5/Canvas backend*
  - *Sophisticated subplot grid layout*
  - *Easy pythonic subplots*
  - *Contour fixes and triplot*
  - *multiple calls to show supported*
  - *mplot3d graphs can be embedded in arbitrary axes*
  - *tick_params*
  - *Lots of performance and feature enhancements*
  - *Much improved software carpentry*
  - *Bugfix marathon*

### HTML5/Canvas backend

Simon Ratcliffe and Ludwig Schwardt have released an HTML5/Canvas backend for matplotlib. The backend is almost feature complete, and they have done a lot of work comparing their html5 rendered images with our core renderer Agg. The backend features client/server interactive navigation of matplotlib figures in an html5 compliant browser.

### Sophisticated subplot grid layout

Jae-Joon Lee has written *gridspec*, a new module for doing complex subplot layouts, featuring row and column spans and more. See *Arranging multiple Axes in a Figure* for a tutorial overview.




### Easy pythonic subplots

Fernando Perez got tired of all the boilerplate code needed to create a figure and multiple subplots when using the matplotlib API, and wrote a *subplots()* helper function. Basic usage allows you to create the figure and an array of subplots with numpy indexing (starts with 0). e.g.:

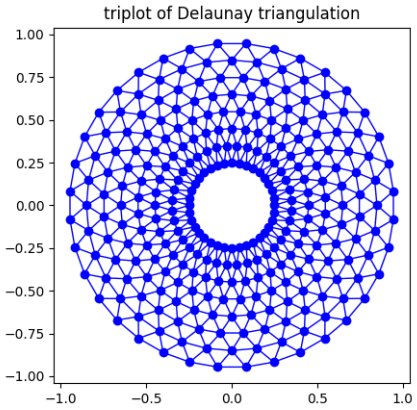
```
fig, axarr = plt.subplots(2, 2)
axarr[0,0].plot([1,2,3])   # upper, left
```

See *Multiple subplots* for several code examples.

### Contour fixes and triplot

Ian Thomas has fixed a long-standing bug that has vexed our most talented developers for years. *contourf()* now handles interior masked regions, and the boundaries of line and filled contours coincide.

Additionally, he has contributed a new module *tri* and helper function *triplot()* for creating and plotting unstructured triangular grids.




### multiple calls to show supported

A long standing request is to support multiple calls to `show()`. This has been difficult because it is hard to get consistent behavior across operating systems, user interface toolkits and versions. Eric Firing has done a lot of work on rationalizing show across backends, with the desired behavior to make show raise all newly created figures and block execution until they are closed. Repeated calls to show should raise newly created figures since the last call. Eric has done a lot of testing on the user interface toolkits and versions and platforms he has access to, but it is not possible to test them all, so please report problems to the mailing list and bug tracker.

### mplot3d graphs can be embedded in arbitrary axes

You can now place an mplot3d graph into an arbitrary axes location, supporting mixing of 2D and 3D graphs in the same figure, and/or multiple 3D graphs in a single figure, using the "projection" keyword argument to add_axes or add_subplot. Thanks Ben Root.

### tick_params

Eric Firing wrote tick_params, a convenience method for changing the appearance of ticks and tick labels. See pyplot function `tick_params()` and associated Axes method `tick_params()`.

### Lots of performance and feature enhancements

- Faster magnification of large images, and the ability to zoom in to a single pixel
- Local installs of documentation work better
- Improved "widgets" -- mouse grabbing is supported
- More accurate snapping of lines to pixel boundaries
- More consistent handling of color, particularly the alpha channel, throughout the API

### Much improved software carpentry

The matplotlib trunk is probably in as good a shape as it has ever been, thanks to improved software carpentry. We now have a buildbot which runs a suite of nose regression tests on every svn commit, auto-generating a set of images and comparing them against a set of known-goods, sending emails to developers on failures with a pixel-by-pixel image comparison. Releases and release bugfixes happen in branches, allowing active new feature development to happen in the trunk while keeping the release branches stable. Thanks to Andrew Straw, Michael Droettboom and other matplotlib developers for the heavy lifting.

### Bugfix marathon

Eric Firing went on a bug fixing and closing marathon, closing over 100 bugs on the (now-closed) SourceForge bug tracker with help from Jae-Joon Lee, Michael Droettboom, Christoph Gohlke and Michiel de Hoon.

## 9.20 Version 0.x

### 9.20.1 List of changes to Matplotlib prior to 2015

This is a list of the changes made to Matplotlib from 2003 to 2015. For more recent changes, please refer to the *Release notes*.

**2015-11-16**

Levels passed to contour(f) and tricontour(f) must be in increasing order.

**2015-10-21**

Added TextBox widget

**2015-10-21**

Added get_ticks_direction()

**2015-02-27**

Added the rcParam 'image.composite_image' to permit users to decide whether they want the vector graphics backends to combine all images within a set of axes into a single composite image. (If images do not get combined, users can open vector graphics files in Adobe Illustrator or Inkscape and edit each image individually.)

**2015-02-19**

Rewrite of C++ code that calculates contours to add support for corner masking. This is controlled by the 'corner_mask' keyword in plotting commands 'contour' and 'contourf'. - IMT

**2015-01-23**

Text bounding boxes are now computed with advance width rather than ink area. This may result in slightly different placement of text.

**2014-10-27**

Allowed selection of the backend using the *MPLBACKEND* environment variable. Added documentation on backend selection methods.

**2014-09-27**

Overhauled *colors.LightSource*. Added *LightSource.hillshade* to allow the independent generation of illumination maps. Added new types of blending for creating more visually appealing shaded relief plots (e.g. `blend_mode="overlay"`, etc, in addition to the legacy "hsv" mode).

**2014-06-10**

Added Colorbar.remove()

**2014-06-07**

Fixed bug so radial plots can be saved as ps in py3k.

**2014-06-01**

Changed the fmt kwarg of errorbar to support the mpl convention that "none" means "don't draw it", and to default to the empty string, so that plotting of data points is done with the plot() function defaults. Deprecated use of the None object in place "none".

**2014-05-22**

Allow the linscale keyword parameter of symlog scale to be smaller than one.

**2014-05-20**

Added logic in FontManager to invalidate font-cache if font-family rcparams have changed.

**2014-05-16**

Fixed the positioning of multi-line text in the PGF backend.

**2014-05-14**

Added Axes.add_image() as the standard way to add AxesImage instances to Axes. This improves the consistency with add_artist(), add_collection(), add_container(), add_line(), add_patch(), and add_table().

**2014-05-02**

Added colorblind-friendly colormap, named 'Wistia'.

**2014-04-27**

Improved input clean up in Axes.{h|v}lines Coerce input into a 1D ndarrays (after dealing with units).

**2014-04-27**

removed un-needed cast to float in stem

**2014-04-23**

Updated references to "ipython -pylab" The preferred method for invoking pylab is now using the "%pylab" magic. -Chris G.

**2014-04-22**

Added (re-)generate a simple automatic legend to "Figure Options" dialog of the Qt4Agg backend.

**2014-04-22**

Added an example showing the difference between interpolation = 'none' and interpolation = 'nearest' in *imshow* when saving vector graphics files.

**2014-04-22**

Added violin plotting functions. See *Axes.violinplot*, *Axes.violin*, *cbook.violin_stats* and *mlab.GaussianKDE* for details.

**2014-04-10**

Fixed the triangular marker rendering error. The "Up" triangle was rendered instead of "Right" triangle and vice-versa.

**2014-04-08**

Fixed a bug in parasite_axes.py by making a list out of a generator at line 263.

**2014-04-02**

Added `clipon=False` to patch creation of wedges and shadows in *pie*.

**2014-02-25**

In backend_qt4agg changed from using update -> repaint under windows. See comment in source near `self._priv_update` for longer explanation.

**2014-03-27**

Added tests for pie ccw parameter. Removed pdf and svg images from tests for pie linewidth parameter.

**2014-03-24**

Changed the behaviour of axes to not ignore leading or trailing patches of height 0 (or width 0) while calculating the x and y axis limits. Patches having both height == 0 and width == 0 are ignored.

**2014-03-24**

Added bool kwarg (manage_xticks) to boxplot to enable/disable the management of the xlimits and ticks when making a boxplot. Default in True which maintains current behavior by default.

**2014-03-23**

Fixed a bug in projections/polar.py by making sure that the theta value being calculated when given the mouse coordinates stays within the range of 0 and 2 * pi.

**2014-03-22**

Added the keyword arguments wedgeprops and textprops to pie. Users can control the wedge and text properties of the pie in more detail, if they choose.

**2014-03-17**

Bug was fixed in append_axes from the AxesDivider class would not append axes in the right location with respect to the reference locator axes

**2014-03-13**

Add parameter 'clockwise' to function pie, True by default.

**2014-02-28**

Added 'origin' kwarg to *spy*

**2014-02-27**

Implemented separate horizontal/vertical axes padding to the ImageGrid in the AxesGrid toolkit

**2014-02-27**

Allowed markevery property of matplotlib.lines.Line2D to be, an int numpy fancy index, slice object, or float. The float behaviour turns on markers at approximately equal display-coordinate-distances along the line.

**2014-02-25**

In backend_qt4agg changed from using update -> repaint under windows. See comment in source near `self._priv_update` for longer explanation.

**2014-01-02**

*triplot* now returns the artist it adds and support of line and marker kwargs has been improved. GBY

**2013-12-30**

Made streamplot grid size consistent for different types of density argument. A 30x30 grid is now used for both density=1 and density=(1, 1).

**2013-12-03**

Added a pure boxplot-drawing method that allow a more complete customization of boxplots. It takes a list of dicts contains stats. Also created a function (*cbook.boxplot_stats*) that generates the stats needed.

**2013-11-28**

Added qhull extension module to perform Delaunay triangulation more robustly than before. It is used by tri.Triangulation (and hence all pyplot.tri* methods) and mlab.griddata. Deprecated matplotlib.delaunay module. - IMT

**2013-11-05**

Add power-law normalization method. This is useful for, e.g., showing small populations in a "hist2d" histogram.

**2013-10-27**

Added get_rlabel_position and set_rlabel_position methods to PolarAxes to control angular position of radial tick labels.

**2013-10-06**

Add stride-based functions to mlab for easy creation of 2D arrays with less memory.

**2013-10-06**

Improve window and detrend functions in mlab, particular support for 2D arrays.

**2013-10-06**

Improve performance of all spectrum-related mlab functions and plots.

**2013-10-06**

Added support for magnitude, phase, and angle spectrums to axes.specgram, and support for magnitude, phase, angle, and complex spectrums to mlab-specgram.

**2013-10-06**

Added magnitude_spectrum, angle_spectrum, and phase_spectrum plots, as well as magnitude_spectrum, angle_spectrum, phase_spectrum, and complex_spectrum functions to mlab

**2013-07-12**

Added support for datetime axes to 2d plots. Axis values are passed through Axes.convert_xunits/Axes.convert_yunits before being used by contour/contourf, pcolormesh and pcolor.

**2013-07-12**

Allowed matplotlib.dates.date2num, matplotlib.dates.num2date, and matplotlib.dates.datestr2num to accept n-d inputs. Also factored in support for n-d arrays to matplotlib.dates.DateConverter and matplotlib.units.Registry.

**2013-06-26**

Refactored the axes module: the axes module is now a folder, containing the following submodule:

- _subplots.py, containing all the subplots helper methods
- _base.py, containing several private methods and a new _AxesBase class. This _AxesBase class contains all the methods that are not directly linked to plots of the "old" Axes
- _axes.py contains the Axes class. This class now inherits from _AxesBase: it contains all "plotting" methods and labelling methods.

This refactoring should not affect the API. Only private methods are not importable from the axes module anymore.

**2013-05-18**

Added support for arbitrary rasterization resolutions to the SVG backend. Previously the resolution was hard coded to 72 dpi. Now the backend class takes a image_dpi argument for its constructor, adjusts the image bounding box accordingly and forwards a magnification factor to the image renderer. The code and results now resemble those of the PDF backend. - MW

**2013-05-08**

Changed behavior of hist when given stacked=True and normed=True. Histograms are now stacked first, then the sum is normalized. Previously, each histogram was normalized, then they were stacked.

**2013-04-25**

Changed all instances of:

```
from matplotlib import MatplotlibDeprecationWarning as mplDeprecation
```

to:

```
from cbook import mplDeprecation
```

and removed the import into the matplotlib namespace in __init__.py - Thomas Caswell

**2013-04-15**

Added 'axes.xmargin' and 'axes.ymargin' to rpParams to set default margins on auto-scaling. - TAC

**2013-04-16**

Added patheffect support for Line2D objects. -JJL

**2013-03-31**

Added support for arbitrary unstructured user-specified triangulations to Axes3D.tricontour[f] - Damon McDougall

**2013-03-19**

Added support for passing *linestyle* kwarg to `step` so all `plot` kwargs are passed to the underlying `plot` call. -TAC

**2013-02-25**

Added classes CubicTriInterpolator, UniformTriRefiner, TriAnalyzer to matplotlib.tri module. - GBy

**2013-01-23**

Add 'savefig.directory' to rcParams to remember and fill in the last directory saved to for figure save dialogs - Martin Spacek

**2013-01-13**

Add eventplot method to axes and pyplot and EventCollection class to collections.

**2013-01-08**

Added two extra titles to axes which are flush with the left and right edges of the plot respectively. Andrew Dawson

**2013-01-07**

Add framealpha keyword argument to legend - PO

**2013-01-16**

Till Stensitzki added a baseline feature to stackplot

**2012-12-22**

Added classes for interpolation within triangular grids (LinearTriInterpolator) and to find the triangles in which points lie (TrapezoidMapTriFinder) to matplotlib.tri module. - IMT

**2012-12-05**

Added MatplotlibDeprecationWarning class for signaling deprecation. Matplotlib developers can use this class as follows:

```
from matplotlib import MatplotlibDeprecationWarning as mplDeprecation
```

In light of the fact that Python builtin DeprecationWarnings are ignored by default as of Python 2.7, this class was put in to allow for the signaling of deprecation, but via UserWarnings which are not ignored by default. - PI

**2012-11-27**

Added the *mtext* parameter for supplying matplotlib.text.Text instances to RendererBase.draw_tex and RendererBase.draw_text. This allows backends to utilize additional text attributes, like the alignment of text elements. - pwuertz

**2012-11-26**

deprecate matplotlib/mpl.py, which was used only in pylab.py and is now replaced by the more suitable `import matplotlib as mpl`. - PI

**2012-11-25**

Make rc_context available via pyplot interface - PI

**2012-11-16**

plt.set_cmap no longer throws errors if there is not already an active colorable artist, such as an image, and just sets up the colormap to use from that point forward. - PI

**2012-11-16**

Added the function _get_rbga_face, which is identical to _get_rbg_face except it return a (r,g,b,a) tuble, to line2D. Modified Line2D.draw to use _get_rbga_face to get the markerface color so that any alpha set by markerfacecolor will respected. - Thomas Caswell

**2012-11-13**

Add a symmetric log normalization class to colors.py. Also added some tests for the normalization class. Till Stensitzki

**2012-11-12**

Make axes.stem take at least one argument. Uses a default range(n) when the first arg not provided. Damon McDougall

**2012-11-09**

Make plt.subplot() without arguments act as subplot(111) - PI

**2012-11-08**

Replaced plt.figure and plt.subplot calls by the newer, more convenient single call to plt.subplots() in the documentation examples - PI

**2012-10-05**

Add support for saving animations as animated GIFs. - JVDP

**2012-08-11**

Fix path-closing bug in patches.Polygon, so that regardless of whether the path is the initial one or was subsequently set by set_xy(), get_xy() will return a closed path if and only if get_closed() is True. Thanks to Jacob Vanderplas. - EF

**2012-08-05**

When a norm is passed to contourf, either or both of the vmin, vmax attributes of that norm are now respected. Formerly they were respected only if both were specified. In addition, vmin and/or vmax can now be passed to contourf directly as kwargs. - EF

**2012-07-24**

Contourf handles the extend kwarg by mapping the extended ranges outside the normed 0-1 range so that they are handled by colormap colors determined by the set_under and set_over methods. Previously the extended ranges were mapped to 0 or 1 so that the "under" and "over" colormap colors were ignored. This change also increases slightly the color contrast for a given set of contour levels. - EF

**2012-06-24**

Make use of mathtext in tick labels configurable - DSD

**2012-06-05**

Images loaded through PIL are now ordered correctly - CG

**2012-06-02**

Add new Axes method and pyplot function, hist2d. - PO

**2012-05-31**

Remove support for 'cairo.<format>' style of backend specification. Deprecate 'cairo.format' and 'savefig.extension' rcParams and replace with 'savefig.format'. - Martin Spacek

**2012-05-29**

pcolormesh now obeys the passed in "edgecolor" kwarg. To support this, the "shading" argument to pcolormesh now only takes "flat" or "gouraud". To achieve the old "faceted" behavior, pass "edgecolors='k'". - MGD

**2012-05-22**

Added radius kwarg to pie charts. - HH

**2012-05-22**

Collections now have a setting "offset_position" to select whether the offsets are given in "screen" coordinates (default, following the old behavior) or "data" coordinates. This is currently used internally to improve the performance of hexbin.

As a result, the "draw_path_collection" backend methods have grown a new argument "offset_position". - MGD

**2012-05-04**

Add a new argument to pie charts - startingangle - that allows one to specify the angle offset for the first wedge of the chart. - EP

**2012-05-03**

symlog scale now obeys the logarithmic base. Previously, it was completely ignored and always treated as base e. - MGD

**2012-05-03**

Allow linscalex/y keyword to symlog scale that allows the size of the linear portion relative to the logarithmic portion to be adjusted. - MGD

**2012-04-14**

Added new plot style: stackplot. This new feature supports stacked area plots. - Damon McDougall

**2012-04-06**

When path clipping changes a LINETO to a MOVETO, it also changes any CLOSEPOLY command to a LINETO to the initial point. This fixes a problem with pdf and svg where the CLOSEPOLY would then draw a line to the latest MOVETO position instead of the intended initial position. - JKS

**2012-03-27**

Add support to ImageGrid for placing colorbars only at one edge of each column/row. - RMM

**2012-03-07**

Refactor movie writing into useful classes that make use of pipes to write image data to ffmpeg or mencoder. Also improve settings for these and the ability to pass custom options. - RMM

**2012-02-29**

errorevery keyword added to errorbar to enable errorbar subsampling. fixes issue #600.

**2012-02-28**

Added plot_trisurf to the mplot3d toolkit. This supports plotting three dimensional surfaces on an irregular grid. - Damon McDougall

**2012-01-23**

The radius labels in polar plots no longer use a fixed padding, but use a different alignment depending on the quadrant they are in. This fixes numerical problems when (rmax - rmin) gets too small. - MGD

**2012-01-08**

Add axes.streamplot to plot streamlines of a velocity field. Adapted from Tom Flannaghan streamplot implementation. -TSY

**2011-12-29**

ps and pdf markers are now stroked only if the line width is nonzero for consistency with agg, fixes issue #621. - JKS

**2011-12-27**

Work around an EINTR bug in some versions of subprocess. - JKS

**2011-10-25**

added support for operatorname to mathtext, including the ability to insert spaces, such as $operatorname{arg,max}$ - PI

**2011-08-18**

Change api of Axes.get_tightbbox and add an optional keyword parameter *call_axes_locator*. - JJL

**2011-07-29**

A new rcParam "axes.formatter.use_locale" was added, that, when True, will use the current locale to format tick labels. This means that, for example, in the fr_FR locale, ',' will be used as a decimal separator. - MGD

**2011-07-15**

The set of markers available in the plot() and scatter() commands has been unified. In general, this gives more options to both than were previously available, however, there is one backward-incompatible change to the markers in scatter:

> "d" used to mean "diamond", it now means "narrow diamond". "D" can be used for a "diamond".

-MGD

**2011-07-13**

Fix numerical problems in symlog scale, particularly when linthresh <= 1.0. Symlog plots may look different if one was depending on the old broken behavior - MGD

**2011-07-10**

Fixed argument handling error in tripcolor/triplot/tricontour, issue #203. - IMT

**2011-07-08**

Many functions added to mplot3d.axes3d to bring Axes3D objects more feature-parity with regular Axes objects. Significant revisions to the documentation as well. - BVR

**2011-07-07**

Added compatibility with IPython strategy for picking a version of Qt4 support, and an rcParam for making the choice explicitly: backend.qt4. - EF

**2011-07-07**

Modified AutoMinorLocator to improve automatic choice of the number of minor intervals per major interval, and to allow one to specify this number via a kwarg. - EF

**2011-06-28**

3D versions of scatter, plot, plot_wireframe, plot_surface, bar3d, and some other functions now support empty inputs. - BVR

**2011-06-22**

Add set_theta_offset, set_theta_direction and set_theta_zero_location to polar axes to control the location of 0 and directionality of theta. - MGD

**2011-06-22**

Add axes.labelweight parameter to set font weight to axis labels - MGD.

**2011-06-20**

Add pause function to pyplot. - EF

**2011-06-16**

Added *bottom* keyword parameter for the stem command. Also, implemented a legend handler for the stem plot. - JJL

**2011-06-16**

Added legend.frameon rcParams. - Mike Kaufman

**2011-05-31**

Made backend_qt4 compatible with PySide . - Gerald Storer

**2011-04-17**

Disable keyboard auto-repeat in qt4 backend by ignoring key events resulting from auto-repeat. This makes constrained zoom/pan work. - EF

**2011-04-14**

interpolation="nearest" always interpolate images. A new mode "none" is introduced for no interpolation - JJL

**2011-04-03**

Fixed broken pick interface to AsteriskCollection objects used by scatter. - EF

**2011-04-01**

The plot directive Sphinx extension now supports all of the features in the Numpy fork of that extension. These include doctest formatting, an 'include-source' option, and a number of new configuration options. - MGD

**2011-03-29**

Wrapped ViewVCCachedServer definition in a factory function. This class now inherits from urllib2.HTTPSHandler in order to fetch data from github, but HTTPSHandler is not defined if python was built without SSL support. - DSD

**2011-03-10**

Update pytz version to 2011c, thanks to Simon Cross. - JKS

**2011-03-06**

Add standalone tests.py test runner script. - JKS

**2011-03-06**

Set edgecolor to 'face' for scatter asterisk-type symbols; this fixes a bug in which these symbols were not responding to the c kwarg. The symbols have no face area, so only the edgecolor is visible. - EF

**2011-02-27**

Support libpng version 1.5.x; suggestion by Michael Albert. Changed installation specification to a minimum of libpng version 1.2. - EF

**2011-02-20**

clabel accepts a callable as an fmt kwarg; modified patch by Daniel Hyams. - EF

**2011-02-18**

scatter([], []) is now valid. Also fixed issues with empty collections - BVR

**2011-02-07**

Quick workaround for dviread bug #3175113 - JKS

**2011-02-05**

Add cbook memory monitoring for Windows, using tasklist. - EF

**2011-02-05**

Speed up Normalize and LogNorm by using in-place operations and by using float32 for float32 inputs and for ints of 2 bytes or shorter; based on patch by Christoph Gohlke. - EF

**2011-02-04**

Changed imshow to use rgba as uint8 from start to finish, instead of going through an intermediate step as double precision; thanks to Christoph Gohlke. - EF

**2011-01-13**

Added zdir and offset arguments to contourf3d to bring contourf3d in feature parity with contour3d. - BVR

**2011-01-04**

Tag 1.0.1 for release at r8896

**2011-01-03**

Added display of ticker offset to 3d plots. - BVR

**2011-01-03**

Turn off tick labeling on interior subplots for pyplots.subplots when sharex/sharey is True. - JDH

**2010-12-29**

Implement axes_divider.HBox and VBox. -JJL

**2010-11-22**

Fixed error with Hammer projection. - BVR

**2010-11-12**

Fixed the placement and angle of axis labels in 3D plots. - BVR

**2010-11-07**

New rc parameters examples.download and examples.directory allow bypassing the download mechanism in get_sample_data. - JKS

**2010-10-04**

Fix JPEG saving bug: only accept the kwargs documented by PIL for JPEG files. - JKS

**2010-09-15**

Remove unused _wxagg extension and numerix.h. - EF

**2010-08-25**

Add new framework for doing animations with examples.- RM

**2010-08-21**

Remove unused and inappropriate methods from Tick classes: set_view_interval, get_minpos, and get_data_interval are properly found in the Axis class and don't need to be duplicated in XTick and YTick. - EF

**2010-08-21**

Change Axis.set_view_interval() so that when updating an existing interval, it respects the orientation of that interval, and can enlarge but not reduce the interval. This fixes a bug in which Axis.set_ticks would change the view limits of an inverted axis. Whether set_ticks should be affecting the viewLim at all remains an open question. - EF

**2010-08-16**

Handle NaN's correctly in path analysis routines. Fixes a bug where the best location for a legend was not calculated correctly when the line contains NaNs. - MGD

**2010-08-14**

Fix bug in patch alpha handling, and in bar color kwarg - EF

**2010-08-12**

Removed all traces of numerix module after 17 months of deprecation warnings. - EF

**2010-08-05**

Added keyword arguments 'thetaunits' and 'runits' for polar plots. Fixed PolarAxes so that when it set default Formatters, it marked them as such. Fixed semilogx and semilogy to no longer blindly reset the ticker information on the non-log axis. Axes.arrow can now accept unitized data. - JRE

**2010-08-03**

Add support for MPLSETUPCFG variable for custom setup.cfg filename. Used by sage buildbot to build an mpl w/ no gui support - JDH

**2010-08-01**

Create directory specified by MPLCONFIGDIR if it does not exist. - ADS

**2010-07-20**

Return Qt4's default cursor when leaving the canvas - DSD

**2010-07-06**

Tagging for mpl 1.0 at r8502

**2010-07-05**

Added Ben Root's patch to put 3D plots in arbitrary axes, allowing you to mix 3d and 2d in different axes/subplots or to have multiple 3D plots in one figure. See examples/mplot3d/subplot3d_demo.py - JDH

**2010-07-05**

Preferred kwarg names in set_xlim are now 'left' and 'right'; in set_ylim, 'bottom' and 'top'; original kwargs are still accepted without complaint. - EF

**2010-07-05**

TkAgg and FltkAgg backends are now consistent with other interactive backends: when used in scripts from the command line (not from ipython -pylab), show blocks, and can be called more than once. - EF

**2010-07-02**

Modified CXX/WrapPython.h to fix "swab bug" on solaris so mpl can compile on Solaris with CXX6 in the trunk. Closes tracker bug 3022815 - JDH

**2010-06-30**

Added autoscale convenience method and corresponding pyplot function for simplified control of autoscaling; and changed axis, set_xlim, and set_ylim so that by default, they turn off the autoscaling on the relevant axis or axes. Therefore one can call set_xlim before plotting a line, for example, and the limits will be retained. - EF

**2010-06-20**

Added Axes.tick_params and corresponding pyplot function to control tick and tick label appearance after an Axes has been created. - EF

**2010-06-09**

Allow Axes.grid to control minor gridlines; allow Axes.grid and Axis.grid to control major and minor gridlines in the same method call. - EF

**2010-06-06**

Change the way we do split/dividend adjustments in finance.py to handle dividends and fix the zero division bug reported in sf bug 2949906 and 2123566. Note that volume is not adjusted because the Yahoo CSV does not distinguish between share split and dividend adjustments making it near impossible to get volume adjustment right (unless we want to guess based on the size of the adjustment or scrape the html tables, which we don't) - JDH

**2010-06-06**

Updated dateutil to 1.5 and pytz to 2010h.

**2010-06-02**

Add error_kw kwarg to Axes.bar(). - EF

**2010-06-01**

Fix pcolormesh() and QuadMesh to pass on kwargs as appropriate. - RM

**2010-05-18**

Merge mpl_toolkits.gridspec into the main tree. - JJL

**2010-05-04**

Improve backend_qt4 so it displays figures with the correct size - DSD

**2010-04-20**

Added generic support for connecting to a timer for events. This adds TimerBase, TimerGTK, TimerQT, TimerWx, and TimerTk to the backends and a new_timer() method to each backend's canvas to allow ease of creating a new timer. - RM

**2010-04-20**

Added margins() Axes method and pyplot function. - EF

**2010-04-18**

update the axes_grid documentation. -JJL

**2010-04-18**

Control MaxNLocator parameters after instantiation, and via Axes.locator_params method, with corresponding pyplot function. -EF

**2010-04-18**

Control ScalarFormatter offsets directly and via the Axes.ticklabel_format() method, and add that to pyplot. -EF

**2010-04-16**

Add a close_event to the backends. -RM

**2010-04-06**

modify axes_grid examples to use axes_grid1 and axisartist. -JJL

**2010-04-06**

rebase axes_grid using axes_grid1 and axisartist modules. -JJL

**2010-04-06**

axes_grid toolkit is split into two separate modules, axes_grid1 and axisartist. -JJL

**2010-04-05**

Speed up import: import pytz only if and when it is needed. It is not needed if the rc timezone is UTC. - EF

**2010-04-03**

Added color kwarg to Axes.hist(), based on work by Jeff Klukas. - EF

**2010-03-24**

refactor colorbar code so that no cla() is necessary when mappable is changed. -JJL

**2010-03-22**

fix incorrect rubber band during the zoom mode when mouse leaves the axes. -JJL

**2010-03-21**

x/y key during the zoom mode only changes the x/y limits. -JJL

**2010-03-20**

Added pyplot.sca() function suggested by JJL. - EF

**2010-03-20**

Added conditional support for new Tooltip API in gtk backend. - EF

**2010-03-20**

Changed plt.fig_subplot() to plt.subplots() after discussion on list, and changed its API to return axes as a numpy object array (with control of dimensions via squeeze keyword). FP.

**2010-03-13**

Manually brought in commits from branch:

```
------------------------------------------------------------------------
r8191 | leejjoon | 2010-03-13
17:27:57 -0500 (Sat, 13 Mar 2010) | 1 line
```

fix the bug that handles for scatter are incorrectly set when dpi!=72. Thanks to Ray Speth for the bug report.

**2010-03-03**

Manually brought in commits from branch via diff/patch (svnmerge is broken):

```
------------------------------------------------------------------------
r8175 | leejjoon | 2010-03-03
10:03:30 -0800 (Wed, 03 Mar 2010) | 1 line

fix arguments of allow_rasterization.draw_wrapper
------------------------------------------------------------------------
r8174 | jdh2358 | 2010-03-03
09:15:58 -0800 (Wed, 03 Mar 2010) | 1 line

added support for favicon in docs build
------------------------------------------------------------------------
r8173 | jdh2358 | 2010-03-03
08:56:16 -0800 (Wed, 03 Mar 2010) | 1 line
```

(continues on next page)

(continued from previous page)

```
applied Mattias get_bounds patch
------------------------------------------------------------------------
r8172 | jdh2358 | 2010-03-03
08:31:42 -0800 (Wed, 03 Mar 2010) | 1 line

fix svnmerge download instructions
------------------------------------------------------------------------
r8171 | jdh2358 | 2010-03-03
07:47:48 -0800 (Wed, 03 Mar 2010) | 1 line
```

**2010-02-25**

add annotation_demo3.py that demonstrates new functionality. -JJL

**2010-02-25**

refactor Annotation to support arbitrary Transform as xycoords or textcoords. Also, if a tuple of two coordinates is provided, they are interpreted as coordinates for each x and y position. -JJL

**2010-02-24**

Added pyplot.fig_subplot(), to create a figure and a group of subplots in a single call. This offers an easier pattern than manually making figures and calling add_subplot() multiple times. FP

**2010-02-17**

Added Gokhan's and Mattias' customizable keybindings patch for the toolbar. You can now set the keymap.* properties in the matplotlibrc file. Newbindings were added for toggling log scaling on the x-axis. JDH

**2010-02-16**

Committed TJ's filled marker patch for left|right|bottom|top|full filled markers. See examples/pylab_examples/filledmarker_demo.py. JDH

**2010-02-11**

Added 'bootstrap' option to boxplot. This allows bootstrap estimates of median confidence intervals. Based on an initial patch by Paul Hobson. - ADS

**2010-02-06**

Added setup.cfg "basedirlist" option to override setting in setupext.py "basedir" dictionary; added "gnu0" platform requested by Benjamin Drung. - EF

**2010-02-06**

Added 'xy' scaling option to EllipseCollection. - EF

**2010-02-03**

Made plot_directive use a custom PlotWarning category, so that warnings can be turned into fatal errors easily if desired. - FP

**2010-01-29**

Added draggable method to Legend to allow mouse drag placement. Thanks Adam Fraser. JDH

**2010-01-25**

Fixed a bug reported by Olle Engdegard, when using histograms with stepfilled and log=True - MM

**2010-01-16**

Upgraded CXX to 6.1.1 - JDH

**2009-01-16**

Don't create minor ticks on top of existing major ticks. Patch by Neil Crighton. -ADS

**2009-01-16**

Ensure three minor ticks always drawn (SF# 2924245). Patch by Neil Crighton. -ADS

**2010-01-16**

Applied patch by Ian Thomas to fix two contouring problems: now contourf handles interior masked regions, and the boundaries of line and filled contours coincide. - EF

**2009-01-11**

The color of legend patch follows the rc parameters axes.facecolor and axes.edgecolor. -JJL

**2009-01-11**

adjustable of Axes can be "box-forced" which allow sharing axes. -JJL

**2009-01-11**

Add add_click and pop_click methods in BlockingContourLabeler. -JJL

**2010-01-03**

Added rcParams['axes.color_cycle'] - EF

**2010-01-03**

Added Pierre's qt4 formlayout editor and toolbar button - JDH

**2009-12-31**

Add support for using math text as marker symbols (Thanks to tcb) - MGD

**2009-12-31**

Commit a workaround for a regression in PyQt4-4.6.{0,1} - DSD

**2009-12-22**

Fix cmap data for gist_earth_r, etc. -JJL

**2009-12-20**

spines: put spines in data coordinates, add set_bounds() call. -ADS

**2009-12-18**

Don't limit notch size in boxplot to q1-q3 range, as this is effectively making the data look better than it is. - ADS

**2009-12-18**

mlab.prctile handles even-length data, such that the median is the mean of the two middle values. - ADS

**2009-12-15**

Add raw-image (unsampled) support for the ps backend. - JJL

**2009-12-14**

Add patch_artist kwarg to boxplot, but keep old default. Convert boxplot_demo2.py to use the new patch_artist. - ADS

**2009-12-06**

axes_grid: reimplemented AxisArtist with FloatingAxes support. Added new examples. - JJL

**2009-12-01**

Applied Laurent Dufrechou's patch to improve blitting with the qt4 backend - DSD

**2009-11-13**

The pdf backend now allows changing the contents of a pdf file's information dictionary via PdfPages.infodict. - JKS

**2009-11-12**

font_manager.py should no longer cause EINTR on Python 2.6 (but will on the 2.5 version of subprocess). Also the fc-list command in that file was fixed so now it should actually find the list of fontconfig fonts. - JKS

**2009-11-10**

Single images, and all images in renderers with option_image_nocomposite (i.e. agg, macosx and the svg backend when rcParams['svg.image_noscale'] is True), are now drawn respecting the zorder relative to other artists. (Note that there may now be inconsistencies across backends when more than one image is drawn at varying zorders, but this change introduces correct behavior for the backends in which it's easy to do so.)

**2009-10-21**

Make AutoDateLocator more configurable by adding options to control the maximum and minimum number of ticks. Also add control of the intervals to be used for ticking. This does not change behavior but opens previously hard-coded behavior to runtime modification`. - RMM

**2009-10-19**

Add "path_effects" support for Text and Patch. See examples/pylab_examples/patheffect_demo.py -JJL

**2009-10-19**

Add "use_clabeltext" option to clabel. If True, clabels will be created with ClabelText class, which recalculates rotation angle of the label during the drawing time. -JJL

**2009-10-16**

Make AutoDateFormatter actually use any specified timezone setting.This was only working correctly when no timezone was specified. - RMM

**2009-09-27**

Beginnings of a capability to test the pdf backend. - JKS

**2009-09-27**

Add a savefig.extension rcparam to control the default filename extension used by savefig. - JKS

---

**2009-09-21**

Tagged for release 0.99.1

**2009-09-20**

Fix usetex spacing errors in pdf backend. - JKS

**2009-09-20**

Add Sphinx extension to highlight IPython console sessions, originally authored (I think) by Michael Droetboom. - FP

**2009-09-20**

Fix off-by-one error in dviread.Tfm, and additionally protect against exceptions in case a dvi font is missing some metrics. - JKS

**2009-09-15**

Implement draw_text and draw_tex method of backend_base using the textpath module. Implement draw_tex method of the svg backend. - JJL

**2009-09-15**

Don't fail on AFM files containing floating-point bounding boxes - JKS

**2009-09-13**

AxesGrid : add modified version of colorbar. Add colorbar location howto. - JJL

**2009-09-07**

AxesGrid : implemented axisline style. Added a demo examples/axes_grid/demo_axisline_style.py- JJL

**2009-09-04**

Make the textpath class as a separate module (textpath.py). Add support for mathtext and tex.- JJL

**2009-09-01**

Added support for Gouraud interpolated triangles. pcolormesh now accepts shading='gouraud' as an option. - MGD

**2009-08-29**

Added matplotlib.testing package, which contains a Nose plugin and a decorator that lets tests be marked as KnownFailures - ADS

**2009-08-20**

Added scaled dict to AutoDateFormatter for customized scales - JDH

**2009-08-15**

Pyplot interface: the current image is now tracked at the figure and axes level, addressing tracker item 1656374. - EF

**2009-08-15**

Docstrings are now manipulated with decorators defined in a new module, docstring.py, thanks to Jason Coombs. - EF

**2009-08-14**

Add support for image filtering for agg back end. See the example demo_agg_filter.py. -JJL

**2009-08-09**

AnnotationBbox added. Similar to Annotation, but works with OffsetBox instead of Text. See the example demo_annotation_box.py. -JJL

**2009-08-07**

BboxImage implemented. Two examples, demo_bboximage.py and demo_ribbon_box.py added. -JJL

**2009-08-07**

In an effort to simplify the backend API, all clipping rectangles and paths are now passed in using GraphicsContext objects, even on collections and images. Therefore:

```
draw_path_collection(self, master_transform, cliprect, clippath,
                     clippath_trans, paths, all_transforms, offsets,
                     offsetTrans, facecolors, edgecolors, linewidths,
                     linestyles, antialiaseds, urls)
```

becomes:

```
draw_path_collection(self, gc, master_transform, paths, all_transforms,
                     offsets, offsetTrans, facecolors, edgecolors,
                     linewidths, linestyles, antialiaseds, urls)
```

```
draw_quad_mesh(self, master_transform, cliprect, clippath,
               clippath_trans, meshWidth, meshHeight, coordinates,
               offsets, offsetTrans, facecolors, antialiased,
               showedges)
```

becomes:

```
draw_quad_mesh(self, gc, master_transform, meshWidth, meshHeight,
               coordinates, offsets, offsetTrans, facecolors,
               antialiased, showedges)
```

```
draw_image(self, x, y, im, bbox, clippath=None, clippath_trans=None)
```

becomes:

```
draw_image(self, gc, x, y, im)
```

- MGD

**2009-08-06**

Tagging the 0.99.0 release at svn r7397 - JDH

- fixed an alpha colormapping bug posted on sf 2832575
- fix typo in axes_divider.py. use nanmin, nanmax in angle_helper.py (patch by Christoph Gohlke)
- remove dup gui event in enter/leave events in gtk
- lots of fixes for os x binaries (Thanks Russell Owen)
- attach gtk events to mpl events -- fixes sf bug 2816580
- applied sf patch 2815064 (middle button events for wx) and patch 2818092 (resize events for wx)
- fixed boilerplate.py so it doesn't break the ReST docs.
- removed a couple of cases of mlab.load
- fixed rec2csv win32 file handle bug from sf patch 2831018
- added two examples from Josh Hemann: examples/pylab_examples/barchart_demo2.py and examples/pylab_examples/boxplot_demo2.py
- handled sf bugs 2831556 and 2830525; better bar error messages and backend driver configs
- added miktex win32 patch from sf patch 2820194
- apply sf patches 2830233 and 2823885 for osx setup and 64 bit; thanks Michiel

**2009-08-04**

Made cbook.get_sample_data make use of the ETag and Last-Modified headers of mod_dav_svn. - JKS

**2009-08-03**

Add PathCollection; modify contourf to use complex paths instead of simple paths with cuts. - EF

**2009-08-03**

Fixed boilerplate.py so it doesn't break the ReST docs. - JKS

**2009-08-03**

pylab no longer provides a load and save function. These are available in matplotlib.mlab, or you can use numpy.loadtxt and numpy.savetxt for text files, or np.save and np.load for binary numpy arrays. - JDH

**2009-07-31**

Added cbook.get_sample_data for urllib enabled fetching and caching of data needed for examples. See examples/misc/sample_data_demo.py - JDH

**2009-07-31**

Tagging 0.99.0.rc1 at 7314 - MGD

**2009-07-30**

Add set_cmap and register_cmap, and improve get_cmap, to provide convenient handling of user-generated colormaps. Reorganized _cm and cm modules. - EF

**2009-07-28**

Quiver speed improved, thanks to tip by Ray Speth. -EF

**2009-07-27**

Simplify argument handling code for plot method. -EF

**2009-07-25**

Allow "plot(1, 2, 'r*')" to work. - EF

**2009-07-22**

Added an 'interp' keyword to griddata so the faster linear interpolation method can be chosen. Default is 'nn', so default behavior (using natural neighbor method) is unchanged (JSW)

**2009-07-22**

Improved boilerplate.py so that it generates the correct signatures for pyplot functions. - JKS

**2009-07-19**

Fixed the docstring of Axes.step to reflect the correct meaning of the kwargs "pre" and "post" - See SF bug https://sourceforge.net/tracker/index.php?func=detail&aid=2823304&group_id=80706&atid=560720 - JDH

**2009-07-18**

Fix support for hatches without color fills to pdf and svg backends. Add an example of that to hatch_demo.py. - JKS

**2009-07-17**

Removed fossils from swig version of agg backend. - EF

**2009-07-14**

initial submission of the annotation guide. -JJL

**2009-07-14**

axes_grid : minor improvements in anchored_artists and inset_locator. -JJL

**2009-07-14**

Fix a few bugs in ConnectionStyle algorithms. Add ConnectionPatch class. -JJL

**2009-07-11**

Added a fillstyle Line2D property for half filled markers -- see examples/pylab_examples/fillstyle_demo.py JDH

**2009-07-08**

Attempt to improve performance of qt4 backend, do not call qApp.processEvents while processing an event. Thanks Ole Streicher for tracking this down - DSD

**2009-06-24**

Add withheader option to mlab.rec2csv and changed use_mrecords default to False in mlab.csv2rec since this is partially broken - JDH

**2009-06-24**

backend_agg.draw_marker quantizes the main path (as in the draw_path). - JJL

**2009-06-24**

axes_grid: floating axis support added. - JJL

**2009-06-14**

Add new command line options to backend_driver.py to support running only some directories of tests - JKS

**2009-06-13**

partial cleanup of mlab and its importation in pylab - EF

**2009-06-13**

Introduce a rotation_mode property for the Text artist. See examples/pylab_examples/demo_text_rotation_mode.py -JJL

**2009-06-07**

add support for bz2 files per sf support request 2794556 - JDH

**2009-06-06**

added a properties method to the artist and inspector to return a dict mapping property name -> value; see sf feature request 2792183 - JDH

**2009-06-06**

added Neil's auto minor tick patch; sf patch #2789713 - JDH

**2009-06-06**

do not apply alpha to rgba color conversion if input is already rgba - JDH

**2009-06-03**

axes_grid : Initial check-in of curvelinear grid support. See examples/axes_grid/demo_curvelinear_grid.py - JJL

**2009-06-01**

Add set_color method to Patch - EF

**2009-06-01**

Spine is now derived from Patch - ADS

**2009-06-01**

use cbook.is_string_like() instead of isinstance() for spines - ADS

**2009-06-01**

cla() support for spines - ADS

**2009-06-01**

Removed support for gtk < 2.4. - EF

**2009-05-29**

Improved the animation_blit_qt4 example, which was a mix of the object-oriented and pylab interfaces. It is now strictly object-oriented - DSD

**2009-05-28**

Fix axes_grid toolkit to work with spine patch by ADS. - JJL

**2009-05-28**

Applied fbianco's patch to handle scroll wheel events in the qt4 backend - DSD

**2009-05-26**

Add support for "axis spines" to have arbitrary location. -ADS

**2009-05-20**

Add an empty matplotlibrc to the tests/ directory so that running tests will use the default set of rcparams rather than the user's config. - RMM

**2009-05-19**

Axis.grid(): allow use of which='major,minor' to have grid on major and minor ticks. -ADS

**2009-05-18**

Make psd(), csd(), and cohere() wrap properly for complex/two-sided versions, like specgram() (SF #2791686) - RMM

**2009-05-18**

Fix the linespacing bug of multiline text (#1239682). See examples/pylab_examples/multiline.py -JJL

**2009-05-18**

Add *annotation_clip* attr. for text.Annotation class. If True, annotation is only drawn when the annotated point is inside the axes area. -JJL

**2009-05-17**

Fix bug(#2749174) that some properties of minor ticks are not conserved -JJL

**2009-05-17**

applied Michiel's sf patch 2790638 to turn off gtk event loop in setupext for pygtk>=2.15.10 - JDH

**2009-05-17**

applied Michiel's sf patch 2792742 to speed up Cairo and macosx collections; speedups can be 20x. Also fixes some bugs in which gc got into inconsistent state

---

**2008-05-17**

Release 0.98.5.3 at r7107 from the branch - JDH

**2009-05-13**

An optional offset and bbox support in restore_bbox. Add animation_blit_gtk2.py. -JJL

**2009-05-13**

psfrag in backend_ps now uses baseline-alignment when preview.sty is used ((default is bottom-alignment). Also, a small API improvement in OffsetBox-JJL

**2009-05-13**

When the x-coordinate of a line is monotonically increasing, it is now automatically clipped at the stage of generating the transformed path in the draw method; this greatly speeds up zooming and panning when one is looking at a short segment of a long time series, for example. - EF

**2009-05-11**

aspect=1 in log-log plot gives square decades. -JJL

**2009-05-08**

clabel takes new kwarg, rightside_up; if False, labels will not be flipped to keep them rightside-up. This allows the use of clabel to make streamfunction arrows, as requested by Evan Mason. - EF

**2009-05-07**

'labelpad' can now be passed when setting x/y labels. This allows controlling the spacing between the label and its axis. - RMM

**2009-05-06**

print_ps now uses mixed-mode renderer. Axes.draw rasterize artists whose zorder smaller than rasterization_zorder. -JJL

**2009-05-06**

Per-artist Rasterization, originally by Eric Bruning. -JJ

**2009-05-05**

Add an example that shows how to make a plot that updates using data from another process. Thanks to Robert Cimrman - RMM

**2009-05-05**

Add Axes.get_legend_handles_labels method. - JJL

**2009-05-04**

Fix bug that Text.Annotation is still drawn while set to not visible. - JJL

**2009-05-04**

Added TJ's fill_betweenx patch - JDH

**2009-05-02**

Added options to plotfile based on question from Joseph Smidt and patch by Matthias Michler. - EF

**2009-05-01**

Changed add_artist and similar Axes methods to return their argument. - EF

**2009-04-30**

Incorrect eps bbox for landscape mode fixed - JJL

**2009-04-28**

Fixed incorrect bbox of eps output when usetex=True. - JJL

**2009-04-24**

Changed use of os.open* to instead use subprocess.Popen. os.popen* are deprecated in 2.6 and are removed in 3.0. - RMM

**2009-04-20**

Worked on axes_grid documentation. Added axes_grid.inset_locator. - JJL

**2009-04-17**

Initial check-in of the axes_grid toolkit. - JJL

**2009-04-17**

Added a support for bbox_to_anchor in offsetbox.AnchoredOffsetbox. Improved a documentation. - JJL

**2009-04-16**

Fixed a offsetbox bug that multiline texts are not correctly aligned. - JJL

**2009-04-16**

Fixed a bug in mixed mode renderer that images produced by an rasterizing backend are placed with incorrect size. - JJL

**2009-04-14**

Added Jonathan Taylor's Reinier Heeres' port of John Porters' mplot3d to svn trunk. Package in mpl_toolkits.mplot3d and demo is examples/mplot3d/demo.py. Thanks Reiner

**2009-04-06**

The pdf backend now escapes newlines and linefeeds in strings. Fixes sf bug #2708559; thanks to Tiago Pereira for the report.

**2009-04-06**

texmanager.make_dvi now raises an error if LaTeX failed to create an output file. Thanks to Joao Luis Silva for reporting this. - JKS

**2009-04-05**

_png.read_png() reads 12 bit PNGs (patch from Tobias Wood) - ADS

**2009-04-04**

Allow log axis scale to clip non-positive values to small positive value; this is useful for errorbars. - EF

**2009-03-28**

Make images handle nan in their array argument. A helper, cbook.safe_masked_invalid() was added. - EF

**2009-03-25**

Make contour and contourf handle nan in their Z argument. - EF

**2009-03-20**

Add AuxTransformBox in offsetbox.py to support some transformation. anchored_text.py example is enhanced and renamed (anchored_artists.py). - JJL

**2009-03-20**

Add "bar" connection style for annotation - JJL

**2009-03-17**

Fix bugs in edge color handling by contourf, found by Jae-Joon Lee. - EF

**2009-03-14**

Added 'LightSource' class to colors module for creating shaded relief maps. shading_example.py added to illustrate usage. - JSW

**2009-03-11**

Ensure wx version >= 2.8; thanks to Sandro Tosi and Chris Barker. - EF

**2009-03-10**

Fix join style bug in pdf. - JKS

**2009-03-07**

Add pyplot access to figure number list - EF

**2009-02-28**

hashing of FontProperties accounts current rcParams - JJL

**2009-02-28**

Prevent double-rendering of shared axis in twinx, twiny - EF

**2009-02-26**

Add optional bbox_to_anchor argument for legend class - JJL

**2009-02-26**

Support image clipping in pdf backend. - JKS

**2009-02-25**

Improve tick location subset choice in FixedLocator. - EF

**2009-02-24**

Deprecate numerix, and strip out all but the numpy part of the code. - EF

**2009-02-21**

Improve scatter argument handling; add an early error message, allow inputs to have more than one dimension. - EF

**2009-02-16**

Move plot_directive.py to the installed source tree. Add support for inline code content - MGD

**2009-02-16**

Move mathmpl.py to the installed source tree so it is available to other projects. - MGD

**2009-02-14**

Added the legend title support - JJL

**2009-02-10**

Fixed a bug in backend_pdf so it doesn't break when the setting pdf.use14corefonts=True is used. Added test case in unit/test_pdf_use14corefonts.py. - NGR

**2009-02-08**

Added a new imsave function to image.py and exposed it in the pyplot interface - GR

**2009-02-04**

Some reorganization of the legend code. anchored_text.py added as an example. - JJL

**2009-02-04**

Add extent keyword arg to hexbin - ADS

**2009-02-04**

Fix bug in mathtext related to dots and ldots - MGD

**2009-02-03**

Change default joinstyle to round - MGD

**2009-02-02**

Reduce number of marker XObjects in pdf output - JKS

**2009-02-02**

Change default resolution on polar plot to 1 - MGD

**2009-02-02**

Avoid malloc errors in ttconv for fonts that don't have e.g., PostName (a version of Tahoma triggered this) - JKS

**2009-01-30**

Remove support for pyExcelerator in exceltools -- use xlwt instead - JDH

**2009-01-29**

Document 'resolution' kwarg for polar plots. Support it when using pyplot.polar, not just Figure.add_axes. - MGD

**2009-01-29**

Rework the nan-handling/clipping/quantizing/simplification framework so each is an independent part of a pipeline. Expose the C++-implementation of all of this so it can be used from all Python backends. Add rcParam "path.simplify_threshold" to control the threshold of similarity below which vertices will be removed.

**2009-01-26**

Improved tight bbox option of the savefig. - JJL

**2009-01-26**

Make curves and NaNs play nice together - MGD

**2009-01-21**

Changed the defaults of acorr and xcorr to use usevlines=True, maxlags=10 and normed=True since these are the best defaults

**2009-01-19**

Fix bug in quiver argument handling. - EF

**2009-01-19**

Fix bug in backend_gtk: don't delete nonexistent toolbar. - EF

**2009-01-16**

Implement bbox_inches option for savefig. If bbox_inches is "tight", try to determine the tight bounding box. - JJL

**2009-01-16**

Fix bug in is_string_like so it doesn't raise an unnecessary exception. - EF

**2009-01-16**

Fix an infinite recursion in the unit registry when searching for a converter for a sequence of strings. Add a corresponding test. - RM

**2009-01-16**

Bugfix of C typedef of MPL_Int64 that was failing on Windows XP 64 bit, as reported by George Goussard on numpy mailing list. - ADS

**2009-01-16**

Added helper function LinearSegmentedColormap.from_list to facilitate building simple custom colomaps. See examples/pylab_examples/custom_cmap_fromlist.py - JDH

**2009-01-16**

Applied Michiel's patch for macosx backend to fix rounding bug. Closed sf bug 2508440 - JSW

**2009-01-10**

Applied Michiel's hatch patch for macosx backend and draw_idle patch for qt. Closes sf patched 2497785 and 2468809 - JDH

**2009-01-10**

Fix bug in pan/zoom with log coordinates. - EF

**2009-01-06**

Fix bug in setting of dashed negative contours. - EF

**2009-01-06**

Be fault tolerant when len(linestyles)>NLev in contour. - MM

**2009-01-06**

Added marginals kwarg to hexbin to plot marginal densities JDH

**2009-01-06**

Change user-visible multipage pdf object to PdfPages to avoid accidents with the file-like PdfFile. - JKS

**2009-01-05**

Fix a bug in pdf usetex: allow using non-embedded fonts. - JKS

**2009-01-05**

optional use of preview.sty in usetex mode. - JJL

**2009-01-02**

Allow multipage pdf files. - JKS

**2008-12-31**

Improve pdf usetex by adding support for font effects (slanting and extending). - JKS

**2008-12-29**

Fix a bug in pdf usetex support, which occurred if the same Type-1 font was used with different encodings, e.g., with Minion Pro and MnSymbol. - JKS

**2008-12-20**

fix the dpi-dependent offset of Shadow. - JJL

**2008-12-20**

fix the hatch bug in the pdf backend. minor update in docs and example - JJL

**2008-12-19**

Add axes_locator attribute in Axes. Two examples are added. - JJL

**2008-12-19**

Update Axes.legend documentation. /api/api_changes.rst is also updated to describe changes in keyword parameters. Issue a warning if old keyword parameters are used. - JJL

**2008-12-18**

add new arrow style, a line + filled triangles. -JJL

---

**2008-12-18**

Re-Released 0.98.5.2 from v0_98_5_maint at r6679 Released 0.98.5.2 from v0_98_5_maint at r6667

**2008-12-18**

Removed configobj, experimental traits and doc/mpl_data link - JDH

**2008-12-18**

Fix bug where a line with NULL data limits prevents subsequent data limits from calculating correctly - MGD

**2008-12-17**

Major documentation generator changes - MGD

**2008-12-17**

Applied macosx backend patch with support for path collections, quadmesh, etc... - JDH

**2008-12-17**

fix dpi-dependent behavior of text bbox and arrow in annotate -JJL

**2008-12-17**

Add group id support in artist. Two examples which demonstrate svg filter are added. -JJL

**2008-12-16**

Another attempt to fix dpi-dependent behavior of Legend. -JJL

**2008-12-16**

Fixed dpi-dependent behavior of Legend and fancybox in Text.

**2008-12-16**

Added markevery property to Line2D to support subsampling of markers - JDH

**2008-12-15**

Removed mpl_data symlink in docs. On platforms that do not support symlinks, these become copies, and the font files are large, so the distro becomes unnecessarily bloated. Keeping the mpl_examples dir because relative links are harder for the plot directive and the *.py files are not so large. - JDH

**2008-12-15**

Fix $ in non-math text with usetex off. Document differences between usetex on/off - MGD

**2008-12-15**

Fix anti-aliasing when auto-snapping - MGD

**2008-12-15**

Fix grid lines not moving correctly during pan and zoom - MGD

**2008-12-12**

Preparations to eliminate maskedarray rcParams key: its use will now generate a warning. Similarly, importing the obsolete numerix.npyma will generate a warning. - EF

**2008-12-12**

Added support for the numpy.histogram() weights parameter to the axes hist() method. Docs taken from numpy - MM

**2008-12-12**

Fixed warning in hist() with numpy 1.2 - MM

**2008-12-12**

Removed external packages: configobj and enthought.traits which are only required by the experimental traited config and are somewhat out of date. If needed, install them independently, see http://code.enthought.com/pages/traits.html and http://www.voidspace.org.uk/python/configobj.html

**2008-12-12**

Added support to assign labels to histograms of multiple data. - MM

---

**2008-12-11**

Released 0.98.5 at svn r6573

**2008-12-11**

Use subprocess.Popen instead of os.popen in dviread (Windows problem reported by Jorgen Stenarson) - JKS

**2008-12-10**

Added Michael's font_manager fix and Jae-Joon's figure/subplot fix. Bumped version number to 0.98.5 - JDH

---

**2008-12-09**

Released 0.98.4 at svn r6536

**2008-12-08**

Added mdehoon's native macosx backend from sf patch 2179017 - JDH

**2008-12-08**

Removed the prints in the set_*style commands. Return the list of pprinted strings instead - JDH

**2008-12-08**

Some of the changes Michael made to improve the output of the property tables in the rest docs broke of made difficult to use some of the interactive doc helpers, e.g., setp and getp. Having all the rest markup in the ipython shell also confused the docstrings. I added a new rc param docstring.hardcopy, to format the docstrings differently for hard copy and other use. The ArtistInspector could use a little refactoring now since there is duplication of effort between the rest out put and the non-rest output - JDH

**2008-12-08**

Updated spectral methods (psd, csd, etc.) to scale one-sided densities by a factor of 2 and, optionally, scale all densities by the sampling frequency. This gives better MatLab compatibility. -RM

**2008-12-08**

Fixed alignment of ticks in colorbars. -MGD

**2008-12-07**

drop the deprecated "new" keyword of np.histogram() for numpy 1.2 or later. -JJL

**2008-12-06**

Fixed a bug in svg backend that new_figure_manager() ignores keywords arguments such as figsize, etc. -JJL

**2008-12-05**

Fixed a bug that the handlelength of the new legend class set too short when numpoints=1 -JJL

**2008-12-04**

Added support for data with units (e.g., dates) to Axes.fill_between. -RM

**2008-12-04**

Added fancybox keyword to legend. Also applied some changes for better look, including baseline adjustment of the multiline texts so that it is center aligned. -JJL

**2008-12-02**

The transmuter classes in the patches.py are reorganized as subclasses of the Style classes. A few more box and arrow styles are added. -JJL

**2008-12-02**

Fixed a bug in the new legend class that didn't allowed a tuple of coordinate values as loc. -JJL

**2008-12-02**

Improve checks for external dependencies, using subprocess (instead of deprecated popen*) and distutils (for version checking) - DSD

**2008-11-30**

Reimplementation of the legend which supports baseline alignment, multi-column, and expand mode. - JJL

**2008-12-01**

Fixed histogram autoscaling bug when bins or range are given explicitly (fixes Debian bug 503148) - MM

**2008-11-25**

Added rcParam axes.unicode_minus which allows plain hyphen for minus when False - JDH

**2008-11-25**

Added scatterpoints support in Legend. patch by Erik Tollerud - JJL

**2008-11-24**

Fix crash in log ticking. - MGD

**2008-11-20**

Added static helper method BrokenHBarCollection.span_where and Axes/pyplot method fill_between. See examples/pylab/fill_between.py - JDH

**2008-11-12**

Add x_isdata and y_isdata attributes to Artist instances, and use them to determine whether either or both coordinates are used when updating dataLim. This is used to fix autoscaling problems that had been triggered by axhline, axhspan, axvline, axvspan. - EF

**2008-11-11**

Update the psd(), csd(), cohere(), and specgram() methods of Axes and the csd() cohere(), and specgram() functions in mlab to be in sync with the changes to psd(). In fact, under the hood, these all call the same core to do computations. - RM

**2008-11-11**

Add 'pad_to' and 'sides' parameters to mlab.psd() to allow controlling of zero padding and returning of negative frequency components, respectively. These are added in a way that does not change the API. - RM

**2008-11-10**

Fix handling of c kwarg by scatter; generalize is_string_like to accept numpy and numpy.ma string array scalars. - RM and EF

**2008-11-09**

Fix a possible EINTR problem in dviread, which might help when saving pdf files from the qt backend. - JKS

**2008-11-05**

Fix bug with zoom to rectangle and twin axes - MGD

**2008-10-24**

Added Jae Joon's fancy arrow, box and annotation enhancements -- see examples/pylab_examples/annotation_demo2.py

**2008-10-23**

Autoscaling is now supported with shared axes - EF

**2008-10-23**

Fixed exception in dviread that happened with Minion - JKS

**2008-10-21**

set_xlim, ylim now return a copy of the viewlim array to avoid modify inplace surprises

**2008-10-20**

Added image thumbnail generating function matplotlib.image.thumbnail. See examples/misc/image_thumbnail.py - JDH

**2008-10-20**

Applied scatleg patch based on ideas and work by Erik Tollerud and Jae-Joon Lee. - MM

**2008-10-11**

Fixed bug in pdf backend: if you pass a file object for output instead of a filename, e.g., in a wep app, we now flush the object at the end. - JKS

**2008-10-08**

Add path simplification support to paths with gaps. - EF

**2008-10-05**

Fix problem with AFM files that don't specify the font's full name or family name. - JKS

**2008-10-04**

Added 'scilimits' kwarg to Axes.ticklabel_format() method, for easy access to the set_powerlimits method of the major ScalarFormatter. - EF

**2008-10-04**

Experimental new kwarg borderpad to replace pad in legend, based on suggestion by Jae-Joon Lee. - EF

**2008-09-27**

Allow spy to ignore zero values in sparse arrays, based on patch by Tony Yu. Also fixed plot to handle empty data arrays, and fixed handling of markers in figlegend. - EF

**2008-09-24**

Introduce drawstyles for lines. Transparently split linestyles like 'steps--' into drawstyle 'steps' and linestyle '--'. Legends always use drawstyle 'default'. - MM

**2008-09-18**

Fixed quiver and quiverkey bugs (failure to scale properly when resizing) and added additional methods for determining the arrow angles - EF

**2008-09-18**

Fix polar interpolation to handle negative values of theta - MGD

**2008-09-14**

Reorganized cbook and mlab methods related to numerical calculations that have little to do with the goals of those two modules into a separate module numerical_methods.py Also, added ability to select points and stop point selection with keyboard in ginput and manual contour labeling code. Finally, fixed contour labeling bug. - DMK

**2008-09-11**

Fix backtick in Postscript output. - MGD

**2008-09-10**

[ 2089958 ] Path simplification for vector output backends Leverage the simplification code exposed through path_to_polygons to simplify certain well-behaved paths in the vector backends (PDF, PS and SVG). "path.simplify" must be set to True in matplotlibrc for this to work. - MGD

**2008-09-10**

Add "filled" kwarg to Path.intersects_path and Path.intersects_bbox. - MGD

**2008-09-07**

Changed full arrows slightly to avoid an xpdf rendering problem reported by Friedrich Hagedorn. - JKS

**2008-09-07**

Fix conversion of quadratic to cubic Bezier curves in PDF and PS backends. Patch by Jae-Joon Lee. - JKS

**2008-09-06**

Added 5-point star marker to plot command - EF

**2008-09-05**

Fix hatching in PS backend - MGD

**2008-09-03**

Fix log with base 2 - MGD

**2008-09-01**

Added support for bilinear interpolation in NonUniformImage; patch by Gregory Lielens. - EF

**2008-08-28**

Added support for multiple histograms with data of different length - MM

**2008-08-28**

Fix step plots with log scale - MGD

**2008-08-28**

Fix masked arrays with markers in non-Agg backends - MGD

**2008-08-28**

Fix clip_on kwarg so it actually works correctly - MGD

**2008-08-25**

Fix locale problems in SVG backend - MGD

**2008-08-22**

fix quiver so masked values are not plotted - JSW

**2008-08-18**

improve interactive pan/zoom in qt4 backend on windows - DSD

**2008-08-11**

Fix more bugs in NaN/inf handling. In particular, path simplification (which does not handle NaNs or infs) will be turned off automatically when infs or NaNs are present. Also masked arrays are now converted to arrays with NaNs for consistent handling of masks and NaNs - MGD and EF

---

**2008-08-03**

Released 0.98.3 at svn r5947

**2008-08-01**

Backported memory leak fixes in _ttconv.cpp - MGD

**2008-07-31**

Added masked array support to griddata. - JSW

**2008-07-26**

Added optional C and reduce_C_function arguments to axes.hexbin(). This allows hexbin to accumulate the values of C based on the x,y coordinates and display in hexagonal bins. - ADS

**2008-07-24**

Deprecated (raise NotImplementedError) all the mlab2 functions from matplotlib.mlab out of concern that some of them were not clean room implementations. JDH

**2008-07-24**

Rewrite of a significant portion of the clabel code (class ContourLabeler) to improve inlining. - DMK

**2008-07-22**

Added Barbs polygon collection (similar to Quiver) for plotting wind barbs. Added corresponding helpers to Axes and pyplot as well. (examples/pylab_examples/barb_demo.py shows it off.) - RMM

**2008-07-21**

Added scikits.delaunay as matplotlib.delaunay. Added griddata function in matplotlib.mlab, with example (griddata_demo.py) in pylab_examples. griddata function will use mpl_toolkits._natgrid if installed. - JSW

**2008-07-21**

Re-introduced offset_copy that works in the context of the new transforms. - MGD

**2008-07-21**

Committed patch by Ryan May to add get_offsets and set_offsets to Collections base class - EF

**2008-07-21**

Changed the "asarray" strategy in image.py so that colormapping of masked input should work for all image types (thanks Klaus Zimmerman) - EF

**2008-07-20**

Rewrote cbook.delete_masked_points and corresponding unit test to support rgb color array inputs, datetime inputs, etc. - EF

**2008-07-20**

Renamed unit/axes_unit.py to cbook_unit.py and modified in accord with Ryan's move of delete_masked_points from axes to cbook. - EF

**2008-07-18**

Check for nan and inf in axes.delete_masked_points(). This should help hexbin and scatter deal with nans. - ADS

**2008-07-17**

Added ability to manually select contour label locations. Also added a waitforbuttonpress function. - DMK

**2008-07-17**

Fix bug with NaNs at end of path (thanks, Andrew Straw for the report) - MGD

**2008-07-16**

Improve error handling in texmanager, thanks to Ian Henry for reporting - DSD

**2008-07-12**

Added support for external backends with the "module://my_backend" syntax - JDH

**2008-07-11**

Fix memory leak related to shared axes. Grouper should store weak references. - MGD

**2008-07-10**

Bugfix: crash displaying fontconfig pattern - MGD

**2008-07-10**

Bugfix: [ 2013963 ] update_datalim_bounds in Axes not works - MGD

**2008-07-10**

Bugfix: [ 2014183 ] multiple imshow() causes gray edges - MGD

**2008-07-09**

Fix rectangular axes patch on polar plots bug - MGD

**2008-07-09**

Improve mathtext radical rendering - MGD

**2008-07-08**

Improve mathtext superscript placement - MGD

**2008-07-07**

Fix custom scales in pcolormesh (thanks Matthew Turk) - MGD

**2008-07-03**

Implemented findobj method for artist and pyplot - see examples/pylab_examples/findobj_demo.py - JDH

**2008-06-30**

Another attempt to fix TextWithDash - DSD

**2008-06-30**

Removed Qt4 NavigationToolbar2.destroy -- it appears to have been unnecessary and caused a bug reported by P. Raybaut - DSD

**2008-06-27**

Fixed tick positioning bug - MM

**2008-06-27**

Fix dashed text bug where text was at the wrong end of the dash - MGD

**2008-06-26**

Fix mathtext bug for expressions like $x_{leftarrow}$ - MGD

**2008-06-26**

Fix direction of horizontal/vertical hatches - MGD

**2008-06-25**

Figure.figurePatch renamed Figure.patch, Axes.axesPatch renamed Axes.patch, Axes.axesFrame renamed Axes.frame, Axes.get_frame, which returns Axes.patch, is deprecated. Examples and users guide updated - JDH

**2008-06-25**

Fix rendering quality of pcolor - MGD

---

**2008-06-24**

Released 0.98.2 at svn r5667 - (source only for debian) JDH

**2008-06-24**

Added "transparent" kwarg to savefig. - MGD

**2008-06-24**

Applied Stefan's patch to draw a single centered marker over a line with numpoints==1 - JDH

**2008-06-23**

Use splines to render circles in scatter plots - MGD

---

**2008-06-22**

Released 0.98.1 at revision 5637

**2008-06-22**

Removed axes3d support and replaced it with a NotImplementedError for one release cycle

**2008-06-21**

fix marker placement bug in backend_ps - DSD

**2008-06-20**

[ 1978629 ] scale documentation missing/incorrect for log - MGD

**2008-06-20**

Added closed kwarg to PolyCollection. Fixes bug [ 1994535 ] still missing lines on graph with svn (r 5548). - MGD

**2008-06-20**

Added set/get_closed method to Polygon; fixes error in hist - MM

**2008-06-19**

Use relative font sizes (e.g., 'medium' and 'large') in rcsetup.py and matplotlibrc.template so that text will be scaled by default when changing rcParams['font.size'] - EF

**2008-06-17**

Add a generic PatchCollection class that can contain any kind of patch. - MGD

**2008-06-13**

Change pie chart label alignment to avoid having labels overwrite the pie - MGD

**2008-06-12**

Added some helper functions to the mathtext parser to return bitmap arrays or write pngs to make it easier to use mathtext outside the context of an mpl figure. modified the mathpng sphinxext to use the mathtext png save functionality - see examples/api/mathtext_asarray.py - JDH

**2008-06-11**

Use matplotlib.mathtext to render math expressions in online docs - MGD

**2008-06-11**

Move PNG loading/saving to its own extension module, and remove duplicate code in _backend_agg.cpp and _image.cpp that does the same thing - MGD

**2008-06-11**

Numerous mathtext bugfixes, primarily related to dpi-independence - MGD

**2008-06-10**

Bar now applies the label only to the first patch only, and sets '_nolegend_' for the other patch labels. This lets autolegend work as expected for hist and bar - see https://sourceforge.net/tracker/index.php?func=detail&aid=1986597&group_id=80706&atid=560720 JDH

**2008-06-10**

Fix text baseline alignment bug. [ 1985420 ] Repair of baseline alignment in Text._get_layout. Thanks Stan West - MGD

**2008-06-09**

Committed Gregor's image resample patch to downsampling images with new rcparam image.resample - JDH

**2008-06-09**

Don't install Enthought.Traits along with matplotlib. For matplotlib developers convenience, it can still be installed by setting an option in setup.cfg while we figure decide if there is a future for the traited config - DSD

**2008-06-09**

Added range keyword arg to hist() - MM

**2008-06-07**

Moved list of backends to rcsetup.py; made use of lower case for backend names consistent; use validate_backend when importing backends subpackage - EF

**2008-06-06**

hist() revision, applied ideas proposed by Erik Tollerud and Olle Engdegard: make histtype='step' unfilled by default and introduce histtype='stepfilled'; use default color cycle; introduce reverse cumulative histogram; new align keyword - MM

**2008-06-06**

Fix closed polygon patch and also provide the option to not close the polygon - MGD

**2008-06-05**

Fix some dpi-changing-related problems with PolyCollection, as called by Axes.scatter() - MGD

**2008-06-05**

Fix image drawing so there is no extra space to the right or bottom - MGD

**2006-06-04**

Added a figure title command suptitle as a Figure method and pyplot command -- see examples/figure_title.py - JDH

**2008-06-02**

Added support for log to hist with histtype='step' and fixed a bug for log-scale stacked histograms - MM

---

**2008-05-29**

Released 0.98.0 at revision 5314

**2008-05-29**

matplotlib.image.imread now no longer always returns RGBA -- if the image is luminance or RGB, it will return a MxN or MxNx3 array if possible. Also uint8 is no longer always forced to float.

**2008-05-29**

Implement path clipping in PS backend - JDH

**2008-05-29**

Fixed two bugs in texmanager.py: improved comparison of dvipng versions fixed a bug introduced when get_grey method was added - DSD

**2008-05-28**

Fix crashing of PDFs in xpdf and ghostscript when two-byte characters are used with Type 3 fonts - MGD

**2008-05-28**

Allow keyword args to configure widget properties as requested in http://sourceforge.net/tracker/index.php?func=detail&aid=1866207&group_id=80706&atid=560722 - JDH

**2008-05-28**

Replaced '-' with u'\u2212' for minus sign as requested in http://sourceforge.net/tracker/index.php?func=detail&aid=1962574&group_id=80706&atid=560720

**2008-05-28**

zero width/height Rectangles no longer influence the autoscaler. Useful for log histograms with empty bins - JDH

**2008-05-28**

Fix rendering of composite glyphs in Type 3 conversion (particularly as evidenced in the Eunjin.ttf Korean font) Thanks Jae-Joon Lee for finding this!

**2008-05-27**

Rewrote the cm.ScalarMappable callback infrastructure to use cbook.CallbackRegistry rather than custom callback handling. Any users of add_observer/notify of the cm.ScalarMappable should use the cm.ScalarMappable.callbacksSM CallbackRegistry instead. JDH

**2008-05-27**

Fix TkAgg build on Ubuntu 8.04 (and hopefully a more general solution for other platforms, too.)

**2008-05-24**

Added PIL support for loading images to imread (if PIL is available) - JDH

**2008-05-23**

Provided a function and a method for controlling the plot color cycle. - EF

**2008-05-23**

Major revision of hist(). Can handle 2D arrays and create stacked histogram plots; keyword 'width' deprecated and rwidth (relative width) introduced; align='edge' changed to center of bin - MM

**2008-05-22**

Added support for ReST-based documentation using Sphinx. Documents are located in doc/, and are broken up into a users guide and an API reference. To build, run the make.py files. Sphinx-0.4 is needed to build generate xml, which will be useful for rendering equations with mathml, use sphinx from svn until 0.4 is released - DSD

**2008-05-21**

Fix segfault in TkAgg backend - MGD

**2008-05-21**

Fix a "local variable unreferenced" bug in plotfile - MM

**2008-05-19**

Fix crash when Windows cannot access the registry to determine font path [Bug 1966974, thanks Patrik Simons] - MGD

**2008-05-16**

removed some unneeded code w/ the python 2.4 requirement. cbook no longer provides compatibility for reversed, enumerate, set or izip. removed lib/subprocess, mpl1, sandbox/units, and the swig code. This stuff should remain on the maintenance branch for archival purposes. JDH

**2008-05-16**

Reorganized examples dir - JDH

**2008-05-16**

Added 'elinewidth' keyword arg to errorbar, based on patch by Christopher Brown - MM

**2008-05-16**

Added 'cumulative' keyword arg to hist to plot cumulative histograms. For normed hists, this is normalized to one - MM

**2008-05-15**

Fix Tk backend segfault on some machines - MGD

**2008-05-14**

Don't use stat on Windows (fixes font embedding problem) - MGD

**2008-05-09**

Fix /singlequote (') in Postscript backend - MGD

**2008-05-08**

Fix kerning in SVG when embedding character outlines - MGD

**2008-05-07**

Switched to future numpy histogram semantic in hist - MM

**2008-05-06**

Fix strange colors when blitting in QtAgg and Qt4Agg - MGD

**2008-05-05**

pass notify_axes_change to the figure's add_axobserver in the qt backends, like we do for the other backends. Thanks Glenn Jones for the report - DSD

**2008-05-02**

Added step histograms, based on patch by Erik Tollerud. - MM

**2008-05-02**

On PyQt <= 3.14 there is no way to determine the underlying Qt version. [1851364] - MGD

**2008-05-02**

Don't call sys.exit() when pyemf is not found [1924199] - MGD

**2008-05-02**

Update _subprocess.c from upstream Python 2.5.2 to get a few memory and reference-counting-related bugfixes. See bug 1949978. - MGD

**2008-04-30**

Added some record array editing widgets for gtk -- see examples/rec_edit*.py - JDH

**2008-04-29**

Fix bug in mlab.sqrtm - MM

**2008-04-28**

Fix bug in SVG text with Mozilla-based viewers (the symbol tag is not supported) - MGD

**2008-04-27**

Applied patch by Michiel de Hoon to add hexbin axes method and pyplot function - EF

**2008-04-25**

Enforce python >= 2.4; remove subprocess build - EF

**2008-04-25**

Enforce the numpy requirement at build time - JDH

**2008-04-24**

Make numpy 1.1 and python 2.3 required when importing matplotlib - EF

**2008-04-24**

Fix compilation issues on VS2003 (Thanks Martin Spacek for all the help) - MGD

**2008-04-24**

Fix sub/superscripts when the size of the font has been changed - MGD

**2008-04-22**

Use "svg.embed_char_paths" consistently everywhere - MGD

**2008-04-20**

Add support to MaxNLocator for symmetric axis autoscaling. - EF

**2008-04-20**

Fix double-zoom bug. - MM

**2008-04-15**

Speed up colormapping. - EF

**2008-04-12**

Speed up zooming and panning of dense images. - EF

**2008-04-11**

Fix global font rcParam setting after initialization time. - MGD

**2008-04-11**

Revert commits 5002 and 5031, which were intended to avoid an unnecessary call to draw(). 5002 broke saving figures before show(). 5031 fixed the problem created in 5002, but broke interactive plotting. Unnecessary call to draw still needs resolution - DSD

**2008-04-07**

Improve color validation in rc handling, suggested by Lev Givon - EF

**2008-04-02**

Allow to use both linestyle definition arguments, '-' and 'solid' etc. in plots/collections - MM

**2008-03-27**

Fix saving to Unicode filenames with Agg backend (other backends appear to already work...) (Thanks, Christopher Barker) - MGD

**2008-03-26**

Fix SVG backend bug that prevents copying and pasting in Inkscape (thanks Kaushik Ghose) - MGD

**2008-03-24**

Removed an unnecessary call to draw() in the backend_qt* mouseReleaseEvent. Thanks to Ted Drain - DSD

**2008-03-23**

Fix a pdf backend bug which sometimes caused the outermost gsave to not be balanced with a grestore. - JKS

**2008-03-20**

Fixed a minor bug in ContourSet._process_linestyles when len(linestyles)==Nlev - MM

**2008-03-19**

Changed ma import statements to "from numpy import ma"; this should work with past and future versions of numpy, whereas "import numpy.ma as ma" will work only with numpy >= 1.05, and "import numerix.npyma as ma" is obsolete now that maskedarray is replacing the earlier implementation, as of numpy 1.05.

**2008-03-14**

Removed an apparently unnecessary call to FigureCanvasAgg.draw in backend_qt*agg. Thanks to Ted Drain - DSD

**2008-03-10**

Workaround a bug in backend_qt4agg's blitting due to a buffer width/bbox width mismatch in _backend_agg's copy_from_bbox - DSD

**2008-02-29**

Fix class Wx toolbar pan and zoom functions (Thanks Jeff Peery) - MGD

**2008-02-16**

Added some new rec array functionality to mlab (rec_summarize, rec2txt and rec_groupby). See examples/rec_groupby_demo.py. Thanks to Tim M for rec2txt.

**2008-02-12**

Applied Erik Tollerud's span selector patch - JDH

**2008-02-11**

Update plotting() doc string to refer to getp/setp. - JKS

**2008-02-10**

Fixed a problem with square roots in the pdf backend with usetex. - JKS

**2008-02-08**

Fixed minor __str__ bugs so getp(gca()) works. - JKS

**2008-02-05**

Added getters for title, xlabel, ylabel, as requested by Brandon Kieth - EF

**2008-02-05**

Applied Gael's ginput patch and created examples/ginput_demo.py - JDH

**2008-02-03**

Expose interpnames, a list of valid interpolation methods, as an AxesImage class attribute. - EF

**2008-02-03**

Added BoundaryNorm, with examples in colorbar_only.py and image_masked.py. - EF

**2008-02-03**

Force dpi=72 in pdf backend to fix picture size bug. - JKS

**2008-02-01**

Fix doubly-included font problem in Postscript backend - MGD

**2008-02-01**

Fix reference leak in ft2font Glyph objects. - MGD

**2008-01-31**

Don't use unicode strings with usetex by default - DSD

**2008-01-31**

Fix text spacing problems in PDF backend with *some* fonts, such as STIXGeneral.

**2008-01-31**

Fix sqrt with radical number (broken by making [ and ] work below) - MGD

**2008-01-27**

Applied Martin Teichmann's patch to improve the Qt4 backend. Uses Qt's builtin toolbars and statusbars. See bug 1828848 - DSD

**2008-01-10**

Moved toolkits to mpl_toolkits, made mpl_toolkits a namespace package - JSWHIT

**2008-01-10**

Use setup.cfg to set the default parameters (tkagg, numpy) when building windows installers - DSD

**2008-01-10**

Fix bug displaying [ and ] in mathtext - MGD

**2008-01-10**

Fix bug when displaying a tick value offset with scientific notation. (Manifests itself as a warning that the times symbol cannot be found). - MGD

**2008-01-10**

Use setup.cfg to set the default parameters (tkagg, numpy) when building windows installers - DSD

---

**2008-01-06**

Released 0.91.2 at revision 4802

**2007-12-26**

Reduce too-late use of matplotlib.use() to a warning instead of an exception, for backwards compatibility - EF

**2007-12-25**

Fix bug in errorbar, identified by Noriko Minakawa - EF

**2007-12-25**

Changed masked array importing to work with the upcoming numpy 1.05 (now the maskedarray branch) as well as with earlier versions. - EF

**2007-12-16**

rec2csv saves doubles without losing precision. Also, it does not close filehandles passed in open. - JDH,ADS

**2007-12-13**

Moved rec2gtk to matplotlib.toolkits.gtktools and rec2excel to matplotlib.toolkits.exceltools - JDH

**2007-12-12**

Support alpha-blended text in the Agg and Svg backends - MGD

**2007-12-10**

Fix SVG text rendering bug. - MGD

**2007-12-10**

Increase accuracy of circle and ellipse drawing by using an 8-piece bezier approximation, rather than a 4-piece one. Fix PDF, SVG and Cairo backends so they can draw paths (meaning ellipses as well). - MGD

**2007-12-07**

Issue a warning when drawing an image on a non-linear axis. - MGD

**2007-12-06**

let widgets.Cursor initialize to the lower x and y bounds rather than 0,0, which can cause havoc for dates and other transforms - DSD

**2007-12-06**

updated references to mpl data directories for py2exe - DSD

**2007-12-06**

fixed a bug in rcsetup, see bug 1845057 - DSD

**2007-12-05**

Fix how fonts are cached to avoid loading the same one multiple times. (This was a regression since 0.90 caused by the refactoring of font_manager.py) - MGD

**2007-12-05**

Support arbitrary rotation of usetex text in Agg backend. - MGD

**2007-12-04**

Support '|' as a character in mathtext - MGD

---

**2007-11-27**

Released 0.91.1 at revision 4517

---

**2007-11-27**

Released 0.91.0 at revision 4478

**2007-11-13**

All backends now support writing to a file-like object, not just a regular file. savefig() can be passed a file-like object in place of a file path. - MGD

**2007-11-13**

Improved the default backend selection at build time: SVG -> Agg -> TkAgg -> WXAgg -> GTK -> GTKAgg. The last usable backend in this progression will be chosen in the default config file. If a backend is defined in setup.cfg, that will be the default backend - DSD

**2007-11-13**

Improved creation of default config files at build time for traited config package - DSD

**2007-11-12**

Exposed all the build options in setup.cfg. These options are read into a dict called "options" by setupext.py. Also, added "-mpl" tags to the version strings for packages provided by matplotlib. Versions provided by mpl will be identified and updated on subsequent installs - DSD

**2007-11-12**

Added support for STIX fonts. A new rcParam, mathtext.fontset, can be used to choose between:

**'cm'**

The TeX/LaTeX Computer Modern fonts

**'stix'**

The STIX fonts (see stixfonts.org)

**'stixsans'**

The STIX fonts, using sans-serif glyphs by default

**'custom'**

A generic Unicode font, in which case the mathtext font must be specified using mathtext.bf, mathtext.it, mathtext.sf etc.

Added a new example, stix_fonts_demo.py to show how to access different fonts and unusual symbols. - MGD

**2007-11-12**

Options to disable building backend extension modules moved from setup.py to setup.cfg - DSD

**2007-11-09**

Applied Martin Teichmann's patch 1828813: a QPainter is used in paintEvent, which has to be destroyed using the method end(). If matplotlib raises an exception before the call to end - and it does if you feed it with bad data - this method end() is never called and Qt4 will start spitting error messages

**2007-11-09**

Moved pyparsing back into matplotlib namespace. Don't use system pyparsing, API is too variable from one release to the next - DSD

**2007-11-08**

Made pylab use straight numpy instead of oldnumeric by default - EF

**2007-11-08**

Added additional record array utilities to mlab (rec2excel, rec2gtk, rec_join, rec_append_field, rec_drop_field) - JDH

**2007-11-08**

Updated pytz to version 2007g - DSD

**2007-11-08**

Updated pyparsing to version 1.4.8 - DSD

**2007-11-08**

Moved csv2rec to recutils and added other record array utilities - JDH

**2007-11-08**

If available, use existing pyparsing installation - DSD

**2007-11-07**

Removed old enthought.traits from lib/matplotlib, added Gael Varoquaux's enthought.traits-2.6b1, which is stripped of setuptools. The package is installed to site-packages if not already available - DSD

**2007-11-05**

Added easy access to minor tick properties; slight mod of patch by Pierre G-M - EF

**2007-11-02**

Committed Phil Thompson's patch 1599876, fixes to Qt4Agg backend and qt4 blitting demo - DSD

**2007-11-02**

Committed Phil Thompson's patch 1599876, fixes to Qt4Agg backend and qt4 blitting demo - DSD

**2007-10-31**

Made log color scale easier to use with contourf; automatic level generation now works. - EF

**2007-10-29**

TRANSFORMS REFACTORING

The primary goal of this refactoring was to make it easier to extend matplotlib to support new kinds of projections. This is primarily an internal improvement, and the possible user-visible changes it allows are yet to come.

The transformation framework was completely rewritten in Python (with Numpy). This will make it easier to add news kinds of transformations without writing C/C++ code.

Transforms are composed into a 'transform tree', made of transforms whose value depends on other transforms (their children). When the contents of children change, their parents are automatically updated to reflect those changes. To do this an "invalidation" method is used: when children change, all of their ancestors are marked as "invalid". When the value of a transform is accessed at a later time, its value is recomputed only if it is invalid, otherwise a cached value may be used. This prevents unnecessary recomputations of transforms, and contributes to better interactive performance.

The framework can be used for both affine and non-affine transformations. However, for speed, we want use the backend renderers to perform affine transformations whenever possible. Therefore, it is possible to perform just the affine or non-affine part of a transformation on a set of data. The affine is always assumed to occur after the non-affine. For any transform:

```
full transform == non-affine + affine
```

Much of the drawing has been refactored in terms of compound paths. Therefore, many methods have been removed from the backend interface and replaced with a handful to draw compound paths. This will make updating the backends easier, since there is less to update. It also should make the backends more consistent in terms of functionality.

User visible changes:

- POLAR PLOTS: Polar plots are now interactively zoomable, and the r-axis labels can be interactively rotated. Straight line segments are now interpolated to follow the curve of the r-axis.
- Non-rectangular clipping works in more backends and with more types of objects.
- Sharing an axis across figures is now done in exactly the same way as sharing an axis between two axes in the same figure:

```
fig1 = figure()
fig2 = figure()

ax1 = fig1.add_subplot(111)
ax2 = fig2.add_subplot(111, sharex=ax1, sharey=ax1)
```

- linestyles now include steps-pre, steps-post and steps-mid. The old step still works and is equivalent to step-pre.
- Multiple line styles may be provided to a collection.

See API_CHANGES for more low-level information about this refactoring.

**2007-10-24**

Added ax kwarg to Figure.colorbar and pyplot.colorbar - EF

**2007-10-19**

Removed a gsave/grestore pair surrounding _draw_ps, which was causing a loss graphics state info (see "EPS output problem - scatter & edgecolors" on mpl-dev, 2007-10-29) - DSD

**2007-10-15**

Fixed a bug in patches.Ellipse that was broken for aspect='auto'. Scale free ellipses now work properly for equal and auto on Agg and PS, and they fall back on a polygonal approximation for nonlinear transformations until we convince ourselves that the spline approximation holds for nonlinear transformations. Added unit/ellipse_compare.py to compare spline with vertex approx for both aspects. JDH

**2007-10-05**

remove generator expressions from texmanager and mpltraits. generator expressions are not supported by python-2.3 - DSD

**2007-10-01**

Made matplotlib.use() raise an exception if called after backends has been imported. - EF

**2007-09-30**

Modified update* methods of Bbox and Interval so they work with reversed axes. Prior to this, trying to set the ticks on a reversed axis failed with an uninformative error message. - EF

**2007-09-30**

Applied patches to axes3d to fix index error problem - EF

**2007-09-24**

Applied Eike Welk's patch reported on mpl-dev on 2007-09-22 Fixes a bug with multiple plot windows in the qt backend, ported the changes to backend_qt4 as well - DSD

**2007-09-21**

Changed cbook.reversed to yield the same result as the python reversed builtin - DSD

**2007-09-13**

The usetex support in the pdf backend is more usable now, so I am enabling it. - JKS

**2007-09-12**

Fixed a Axes.bar unit bug - JDH

**2007-09-10**

Made skiprows=1 the default on csv2rec - JDH

**2007-09-09**

Split out the plotting part of pylab and put it in pyplot.py; removed numerix from the remaining pylab.py, which imports everything from pyplot.py. The intention is that apart from cleanups, the result of importing from pylab is nearly unchanged, but there is the new alternative of importing from pyplot to get the state-engine graphics without all the numeric functions. Numpified examples; deleted two that were obsolete; modified some to use pyplot. - EF

**2007-09-08**

Eliminated gd and paint backends - EF

**2007-09-06**

.bmp file format is now longer an alias for .raw

**2007-09-07**

Added clip path support to pdf backend. - JKS

**2007-09-06**

Fixed a bug in the embedding of Type 1 fonts in PDF. Now it doesn't crash Preview.app. - JKS

**2007-09-06**

Refactored image saving code so that all GUI backends can save most image types. See FILETYPES for a matrix of backends and their supported file types. Backend canvases should no longer write their own print_figure() method -- instead they should write a print_xxx method for each filetype they can output and add an entry to their class-scoped filetypes dictionary. - MGD

**2007-09-05**

Fixed Qt version reporting in setupext.py - DSD

**2007-09-04**

Embedding Type 1 fonts in PDF, and thus usetex support via dviread, sort of works. To test, enable it by renaming _draw_tex to draw_tex. - JKS

**2007-09-03**

Added ability of errorbar show limits via caret or arrowhead ends on the bars; patch by Manual Metz. - EF

**2007-09-03**

Created type1font.py, added features to AFM and FT2Font (see API_CHANGES), started work on embedding Type 1 fonts in pdf files. - JKS

**2007-09-02**

Continued work on dviread.py. - JKS

**2007-08-16**

Added a set_extent method to AxesImage, allow data extent to be modified after initial call to imshow - DSD

**2007-08-14**

Fixed a bug in pyqt4 subplots-adjust. Thanks to Xavier Gnata for the report and suggested fix - DSD

**2007-08-13**

Use pickle to cache entire fontManager; change to using font_manager module-level function findfont wrapper for the fontManager.findfont method - EF

**2007-08-11**

Numpification and cleanup of mlab.py and some examples - EF

**2007-08-06**

Removed mathtext2

**2007-07-31**

Refactoring of distutils scripts.

- Will not fail on the entire build if an optional Python package (e.g., Tkinter) is installed but its development headers are not (e.g., tk-devel). Instead, it will continue to build all other extensions.
- Provide an overview at the top of the output to display what dependencies and their versions were found, and (by extension) what will be built.
- Use pkg-config, when available, to find freetype2, since this was broken on Mac OS-X when using MacPorts in a non- standard location.

**2007-07-30**

Reorganized configuration code to work with traited config objects. The new config system is located in the matplotlib.config package, but it is disabled by default. To enable it, set NEWCONFIG=True in matplotlib.__init__.py. The new configuration system will still use the old matplotlibrc files by default. To switch to the experimental, traited configuration, set USE_TRAITED_CONFIG=True in config.__init__.py.

**2007-07-29**

Changed default pcolor shading to flat; added aliases to make collection kwargs agree with setter names, so updating works; related minor cleanups. Removed quiver_classic, scatter_classic, pcolor_classic. - EF

**2007-07-26**

Major rewrite of mathtext.py, using the TeX box layout model.

There is one (known) backward incompatible change. The font commands (cal, rm, it, tt) now behave as TeX does: they are in effect until the next font change command or the end of the grouping. Therefore uses of $cal{R}$ should be changed to ${cal R}$. Alternatively, you may use the new LaTeX-style font commands (mathcal, mathrm, mathit, mathtt) which do affect the following group, e.g., $mathcal{R}$.

Other new features include:

- Math may be interspersed with non-math text. Any text with an even number of $'s (non-escaped) will be sent to the mathtext parser for layout.
- Sub/superscripts are less likely to accidentally overlap.
- Support for sub/superscripts in either order, e.g., $x^i_j$ and $x_j^i$ are equivalent.
- Double sub/superscripts (e.g., $x_i_j$) are considered ambiguous and raise an exception. Use braces to disambiguate.
- $frac{x}{y}$ can be used for displaying fractions.
- $sqrt[3]{x}$ can be used to display the radical symbol with a root number and body.
- $left(frac{x}{y}right)$ may be used to create parentheses and other delimiters that automatically resize to the height of their contents.
- Spacing around operators etc. is now generally more like TeX.
- Added support (and fonts) for boldface (bf) and sans-serif (sf) symbols.
- Log-like function name shortcuts are supported. For example, $sin(x)$ may be used instead of ${rm sin}(x)$
- Limited use of kerning for the easy case (same font)

Behind the scenes, the pyparsing.py module used for doing the math parsing was updated to the latest stable version (1.4.6). A lot of duplicate code was refactored out of the Font classes.

- MGD

**2007-07-19**

completed numpification of most trivial cases - NN

**2007-07-19**

converted non-numpy relicts throughout the code - NN

**2007-07-19**

replaced the Python code in numerix/ by a minimal wrapper around numpy that explicitly mentions all symbols that need to be addressed for further numpification - NN

**2007-07-18**

make usetex respect changes to rcParams. texmanager used to only configure itself when it was created, now it reconfigures when rcParams are changed. Thank you Alexander Schmolck for contributing a patch - DSD

**2007-07-17**

added validation to setting and changing rcParams - DSD

**2007-07-17**

bugfix segfault in transforms module. Thanks Ben North for the patch. - ADS

**2007-07-16**

clean up some code in ticker.ScalarFormatter, use unicode to render multiplication sign in offset ticklabel - DSD

**2007-07-16**

fixed a formatting bug in ticker.ScalarFormatter's scientific notation (10^0 was being rendered as 10 in some cases) - DSD

**2007-07-13**

Add MPL_isfinite64() and MPL_isinf64() for testing doubles in (the now misnamed) MPL_isnan.h. - ADS

**2007-07-13**

The matplotlib._isnan module removed (use numpy.isnan) - ADS

**2007-07-13**

Some minor cleanups in _transforms.cpp - ADS

**2007-07-13**

Removed the rest of the numerix extension code detritus, numpified axes.py, and cleaned up the imports in axes.py - JDH

**2007-07-13**

Added legend.loc as configurable option that could in future default to 'best'. - NN

**2007-07-12**

Bugfixes in mlab.py to coerce inputs into numpy arrays. -ADS

**2007-07-11**

Added linespacing kwarg to text.Text - EF

**2007-07-11**

Added code to store font paths in SVG files. - MGD

**2007-07-10**

Store subset of TTF font as a Type 3 font in PDF files. - MGD

**2007-07-09**

Store subset of TTF font as a Type 3 font in PS files. - MGD

**2007-07-09**

Applied Paul's pick restructure pick and add pickers, sourceforge patch 1749829 - JDH

**2007-07-09**

Applied Allan's draw_lines agg optimization. JDH

**2007-07-08**

Applied Carl Worth's patch to fix cairo draw_arc - SC

**2007-07-07**

fixed bug 1712099: xpdf distiller on windows - DSD

**2007-06-30**

Applied patches to tkagg, gtk, and wx backends to reduce memory leakage. Patches supplied by Mike Droettboom; see tracker numbers 1745400, 1745406, 1745408. Also made unit/memleak_gui.py more flexible with command-line options. - EF

**2007-06-30**

Split defaultParams into separate file rcdefaults (together with validation code). Some heavy refactoring was necessary to do so, but the overall behavior should be the same as before. - NN

**2007-06-27**

Added MPLCONFIGDIR for the default location for mpl data and configuration. useful for some apache installs where HOME is not writable. Tried to clean up the logic in _get_config_dir to support non-writable HOME where are writable HOME/.matplotlib already exists - JDH

**2007-06-27**

Fixed locale bug reported at http://sourceforge.net/tracker/index.php?func=detail&aid=1744154&group_id=80706&a
by adding a cbook.unicode_safe function - JDH

**2007-06-27**

Applied Micheal's tk savefig bugfix described at http://sourceforge.net/tracker/index.php?func=detail&aid=1716732&g
Thanks Michael!

**2007-06-27**

Patch for get_py2exe_datafiles() to work with new directory layout. (Thanks Tocer and also Werner Bruhin.) -ADS

**2007-06-27**

Added a scroll event to the mpl event handling system and implemented it for backends GTK* -- other backend users/developers/maintainers, please add support for your backend. - JDH

**2007-06-25**

Changed default to clip=False in colors.Normalize; modified ColorbarBase for easier colormap display - EF

**2007-06-13**

Added maskedarray option to rc, numerix - EF

**2007-06-11**

Python 2.5 compatibility fix for mlab.py - EF

**2007-06-10**

In matplotlibrc file, use 'dashed' | 'solid' instead of a pair of floats for contour.negative_linestyle - EF

**2007-06-08**

Allow plot and fill fmt string to be any mpl string colorspec - EF

**2007-06-08**

Added gnuplot file plotfile function to pylab -- see examples/plotfile_demo.py - JDH

**2007-06-07**

Disable build of numarray and Numeric extensions for internal MPL use and the numerix layer. - ADS

**2007-06-07**

Added csv2rec to matplotlib.mlab to support automatically converting csv files to record arrays using type introspection, and turned on native datetime support using the new units support in matplotlib.dates. See examples/loadrec.py ! JDH

**2007-06-07**

Simplified internal code of _auto_legend_data - NN

**2007-06-04**

Added labeldistance arg to Axes.pie to control the raidal distance of the wedge labels - JDH

**2007-06-03**

Turned mathtext in SVG into single <text> with multiple <tspan> objects (easier to edit in inkscape). - NN

**2007-06-02**

Released 0.90.1 at revision 3352

**2007-06-02**

Display only meaningful labels when calling legend() without args. - NN

**2007-06-02**

Have errorbar follow the color cycle even if line is not plotted. Suppress plotting of errorbar caps for capsize=0. - NN

**2007-06-02**

Set markers to same alpha value as line. - NN

**2007-06-02**

Fix mathtext position in svg backend. - NN

**2007-06-01**

Deprecate Numeric and numarray for use as numerix. Props to Travis -- job well done. - ADS

**2007-05-18**

Added LaTeX unicode support. Enable with the 'text.latex.unicode' rcParam. This requires the ucs and inputenc LaTeX packages. - ADS

**2007-04-23**

Fixed some problems with polar -- added general polygon clipping to clip the lines and grids to the polar axes. Added support for set_rmax to easily change the maximum radial grid. Added support for polar legend - JDH

**2007-04-16**

Added Figure.autofmt_xdate to handle adjusting the bottom and rotating the tick labels for date plots when the ticks often overlap - JDH

**2007-04-09**

Beginnings of usetex support for pdf backend. -JKS

**2007-04-07**

Fixed legend/LineCollection bug. Added label support to collections. - EF

**2007-04-06**

Removed deprecated support for a float value as a gray-scale; now it must be a string, like '0.5'. Added alpha kwarg to ColorConverter.to_rgba_list. - EF

**2007-04-06**

Fixed rotation of ellipses in pdf backend (sf bug #1690559) -JKS

**2007-04-04**

More matshow tweaks; documentation updates; new method set_bounds() for formatters and locators. - EF

**2007-04-02**

Fixed problem with imshow and matshow of integer arrays; fixed problems with changes to color autoscaling. - EF

**2007-04-01**

Made image color autoscaling work correctly with a tracking colorbar; norm.autoscale now scales unconditionally, while norm.autoscale_None changes only None-valued vmin, vmax. - EF

**2007-03-31**

Added a qt-based subplot-adjustment dialog - DSD

**2007-03-30**

Fixed a bug in backend_qt4, reported on mpl-dev - DSD

**2007-03-26**

Removed colorbar_classic from figure.py; fixed bug in Figure.clear() in which _axobservers was not getting cleared. Modernization and cleanups. - EF

**2007-03-26**

Refactored some of the units support -- units now live in the respective x and y Axis instances. See also API_CHANGES for some alterations to the conversion interface. JDH

**2007-03-25**

Fix masked array handling in quiver.py for numpy. (Numeric and numarray support for masked arrays is broken in other ways when using quiver. I didn't pursue that.) - ADS

**2007-03-23**

Made font_manager.py close opened files. - JKS

**2007-03-22**

Made imshow default extent match matshow - EF

**2007-03-22**

Some more niceties for xcorr -- a maxlags option, normed now works for xcorr as well as axorr, usevlines is supported, and a zero correlation hline is added. See examples/xcorr_demo.py. Thanks Sameer for the patch. - JDH

**2007-03-21**

Axes.vlines and Axes.hlines now create and returns a LineCollection, not a list of lines. This is much faster. The kwarg signature has changed, so consult the docs. Modified Axes.errorbar which uses vlines and hlines. See API_CHANGES; the return signature for these three functions is now different

**2007-03-20**

Refactored units support and added new examples - JDH

**2007-03-19**

Added Mike's units patch - JDH

**2007-03-18**

Matshow as an Axes method; test version matshow1() in pylab; added 'integer' Boolean kwarg to MaxNLocator initializer to force ticks at integer locations. - EF

**2007-03-17**

Preliminary support for clipping to paths agg - JDH

**2007-03-17**

Text.set_text() accepts anything convertible with '%s' - EF

**2007-03-14**

Add masked-array support to hist. - EF

**2007-03-03**

Change barh to take a kwargs dict and pass it to bar. Fixes sf bug #1669506.

**2007-03-02**

Add rc parameter pdf.inheritcolor, which disables all color-setting operations in the pdf backend. The idea is that you include the resulting file in another program and set the colors (both stroke and fill color) there, so you can use the same pdf file for e.g., a paper and a presentation and have them in the surrounding color. You will probably not want to draw figure and axis frames in that case, since they would be filled in the same color. - JKS

**2007-02-26**

Prevent building _wxagg.so with broken Mac OS X wxPython. - ADS

**2007-02-23**

Require setuptools for Python 2.3 - ADS

**2007-02-22**

WXAgg accelerator updates - KM

WXAgg's C++ accelerator has been fixed to use the correct wxBitmap constructor.

The backend has been updated to use new wxPython functionality to provide fast blit() animation without the C++ accelerator. This requires wxPython 2.8 or later. Previous versions of wxPython can use the C++ accelerator or the old pure Python routines.

setup.py no longer builds the C++ accelerator when wxPython >= 2.8 is present.

The blit() method is now faster regardless of which agg/wxPython conversion routines are used.

**2007-02-21**

Applied the PDF backend patch by Nicolas Grilly. This impacts several files and directories in matplotlib:

- Created the directory lib/matplotlib/mpl-data/fonts/pdfcorefonts, holding AFM files for the 14 PDF core fonts. These fonts are embedded in every PDF viewing application.
- setup.py: Added the directory pdfcorefonts to package_data.
- lib/matplotlib/__init__.py: Added the default parameter 'pdf.use14corefonts'. When True, the PDF backend uses only the 14 PDF core fonts.
- lib/matplotlib/afm.py: Added some keywords found in recent AFM files. Added a little workaround to handle Euro symbol.
- lib/matplotlib/fontmanager.py: Added support for the 14 PDF core fonts. These fonts have a dedicated cache (file pdfcorefont.cache), not the same as for other AFM files (file .afmfont.cache). Also cleaned comments to conform to CODING_GUIDE.
- lib/matplotlib/backends/backend_pdf.py: Added support for 14 PDF core fonts. Fixed some issues with incorrect character widths and encodings (works only for the most common encoding, WinAnsiEncoding, defined by the official PDF Reference). Removed parameter 'dpi' because it causes alignment issues.

-JKS (patch by Nicolas Grilly)

**2007-02-17**

Changed ft2font.get_charmap, and updated all the files where get_charmap is mentioned - ES

**2007-02-13**

Added barcode demo- JDH

**2007-02-13**

Added binary colormap to cm - JDH

**2007-02-13**

Added twiny to pylab - JDH

**2007-02-12**

Moved data files into lib/matplotlib so that setuptools' develop mode works. Re-organized the mpl-data layout so that this source structure is maintained in the installation. (i.e., the 'fonts' and 'images' sub-directories are maintained in site-packages.) Suggest removing site-packages/matplotlib/mpl-data and ~/.matplotlib/ttffont.cache before installing - ADS

**2007-02-07**

Committed Rob Hetland's patch for qt4: remove references to text()/latin1(), plus some improvements to the toolbar layout - DSD

---

**2007-02-06**

Released 0.90.0 at revision 3003

**2007-01-22**

Extended the new picker API to text, patches and patch collections. Added support for user customizable pick hit testing and attribute tagging of the PickEvent - Details and examples in examples/pick_event_demo.py - JDH

**2007-01-16**

Begun work on a new pick API using the mpl event handling framework. Artists will define their own pick method with a configurable epsilon tolerance and return pick attrs. All artists that meet the tolerance threshold will fire a PickEvent with artist dependent attrs; e.g., a Line2D can set the indices attribute that shows the indices into the line that are within epsilon of the pick point. See examples/pick_event_demo.py. The implementation of pick for the remaining Artists remains to be done, but the core infrastructure at the level of event handling is in place with a proof-of-concept implementation for Line2D - JDH

**2007-01-16**

src/_image.cpp: update to use Py_ssize_t (for 64-bit systems). Use return value of fread() to prevent warning messages - SC.

**2007-01-15**

src/_image.cpp: combine buffer_argb32() and buffer_bgra32() into a new method color_conv(format) - SC

**2007-01-14**

backend_cairo.py: update draw_arc() so that examples/arctest.py looks correct - SC

**2007-01-12**

backend_cairo.py: enable clipping. Update draw_image() so that examples/contour_demo.py looks correct - SC

**2007-01-12**

backend_cairo.py: fix draw_image() so that examples/image_demo.py now looks correct - SC

**2007-01-11**

Added Axes.xcorr and Axes.acorr to plot the cross correlation of x vs. y or the autocorrelation of x. pylab wrappers also provided. See examples/xcorr_demo.py - JDH

**2007-01-10**

Added "Subplot.label_outer" method. It will set the visibility of the ticklabels so that yticklabels are only visible in the first column and xticklabels are only visible in the last row - JDH

**2007-01-02**

Added additional kwarg documentation - JDH

**2006-12-28**

Improved error message for nonpositive input to log transform; added log kwarg to bar, barh, and hist, and modified bar method to behave sensibly by default when the ordinate has a log scale. (This only works if the log scale is set before or by the call to bar, hence the utility of the log kwarg.) - EF

**2006-12-27**

backend_cairo.py: update draw_image() and _draw_mathtext() to work with numpy - SC

**2006-12-20**

Fixed xpdf dependency check, which was failing on windows. Removed ps2eps dependency check. - DSD

**2006-12-19**

Added Tim Leslie's spectral patch - JDH

**2006-12-17**

Added rc param 'axes.formatter.limits' to control the default threshold for switching to scientific notation. Added convenience method Axes.ticklabel_format() for turning scientific notation on or off on either or both axes. - EF

**2006-12-16**

Added ability to turn control scientific notation in ScalarFormatter - EF

**2006-12-16**

Enhanced boxplot to handle more flexible inputs - EF

**2006-12-13**

Replaced calls to where() in colors.py with much faster clip() and putmask() calls; removed inappropriate uses of getmaskorNone (which should be needed only very rarely); all in response to profiling by David Cournapeau. Also fixed bugs in my 2-D array support from 12-09. - EF

**2006-12-09**

Replaced spy and spy2 with the new spy that combines marker and image capabilities - EF

**2006-12-09**

Added support for plotting 2-D arrays with plot: columns are plotted as in Matlab - EF

**2006-12-09**

Added linewidth kwarg to bar and barh; fixed arg checking bugs - EF

**2006-12-07**

Made pcolormesh argument handling match pcolor; fixed kwarg handling problem noted by Pierre GM - EF

**2006-12-06**

Made pcolor support vector X and/or Y instead of requiring 2-D arrays - EF

**2006-12-05**

Made the default Artist._transform None (rather than invoking identity_transform for each artist only to have it overridden later). Use artist.get_transform() rather than artist._transform, even in derived classes, so that the default transform will be created lazily as needed - JDH

**2006-12-03**

Added LogNorm to colors.py as illustrated by examples/pcolor_log.py, based on suggestion by Jim McDonald. Colorbar modified to handle LogNorm. Norms have additional "inverse" method. - EF

**2006-12-02**

Changed class names in colors.py to match convention: normalize -> Normalize, no_norm -> NoNorm. Old names are still available. Changed __init__.py rc defaults to match those in matplotlibrc - EF

**2006-11-22**

Fixed bug in set_*lim that I had introduced on 11-15 - EF

**2006-11-22**

Added examples/clippedline.py, which shows how to clip line data based on view limits -- it also changes the marker style when zoomed in - JDH

**2006-11-21**

Some spy bug-fixes and added precision arg per Robert C's suggestion - JDH

**2006-11-19**

Added semi-automatic docstring generation detailing all the kwargs that functions take using the artist introspection tools; e.g., 'help text now details the scatter kwargs that control the Text properties - JDH

**2006-11-17**

Removed obsolete scatter_classic, leaving a stub to raise NotImplementedError; same for pcolor_classic - EF

**2006-11-15**

Removed obsolete pcolor_classic - EF

**2006-11-15**

Fixed 1588908 reported by Russel Owen; factored nonsingular method out of ticker.py, put it into transforms.py as a function, and used it in set_xlim and set_ylim. - EF

**2006-11-14**

Applied patch 1591716 by Ulf Larssen to fix a bug in apply_aspect. Modified and applied patch 1594894 by mdehoon to fix bugs and improve formatting in lines.py. Applied patch 1573008 by Greg Willden to make psd etc. plot full frequency range for complex inputs. - EF

**2006-11-14**

Improved the ability of the colorbar to track changes in corresponding image, pcolor, or contourf. - EF

**2006-11-11**

Fixed bug that broke Numeric compatibility; added support for alpha to colorbar. The alpha information is taken from the mappable object, not specified as a kwarg. - EF

**2006-11-05**

Added broken_barh function for making a sequence of horizontal bars broken by gaps -- see examples/broken_barh.py

**2006-11-05**

Removed lineprops and markerprops from the Annotation code and replaced them with an arrow configurable with kwarg arrowprops. See examples/annotation_demo.py - JDH

**2006-11-02**

Fixed a pylab subplot bug that was causing axes to be deleted with hspace or wspace equals zero in subplots_adjust - JDH

**2006-10-31**

Applied axes3d patch 1587359 http://sourceforge.net/tracker/index.php?func=detail&aid=1587359&group_id=80706
JDH

---

**2006-10-26**

Released 0.87.7 at revision 2835

**2006-10-25**

Made "tiny" kwarg in Locator.nonsingular much smaller - EF

**2006-10-17**

Closed sf bug 1562496 update line props dash/solid/cap/join styles - JDH

**2006-10-17**

Complete overhaul of the annotations API and example code - See matplotlib.text.Annotation and examples/annotation_demo.py JDH

**2006-10-12**

Committed Manuel Metz's StarPolygon code and examples/scatter_star_poly.py - JDH

**2006-10-11**

commented out all default values in matplotlibrc.template Default values should generally be taken from defaultParam in __init__.py - the file matplotlib should only contain those values that the user wants to explicitly change from the default. (see thread "marker color handling" on matplotlib-devel)

**2006-10-10**

Changed default comment character for load to '#' - JDH

**2006-10-10**

deactivated rcfile-configurability of markerfacecolor and markeredgecolor. Both are now hardcoded to the special value 'auto' to follow the line color. Configurability at run-time (using function arguments) remains functional. - NN

**2006-10-07**

introduced dummy argument magnification=1.0 to FigImage.make_image to satisfy unit test figimage_demo.py The argument is not yet handled correctly, which should only show up when using non-standard DPI settings in PS backend, introduced by patch #1562394. - NN

**2006-10-06**

add backend-agnostic example: simple3d.py - NN

**2006-09-29**

fix line-breaking for SVG-inline images (purely cosmetic) - NN

**2006-09-29**

reworked set_linestyle and set_marker markeredgecolor and markerfacecolor now default to a special value "auto" that keeps the color in sync with the line color further, the intelligence of axes.plot is cleaned up, improved and simplified. Complete compatibility cannot be guaranteed, but the new behavior should be much more predictable (see patch #1104615 for details) - NN

**2006-09-29**

changed implementation of clip-path in SVG to work around a limitation in inkscape - NN

**2006-09-29**

added two options to matplotlibrc:

- svg.image_inline
- svg.image_noscale

see patch #1533010 for details - NN

**2006-09-29**

axes.py: cleaned up kwargs checking - NN

**2006-09-29**

setup.py: cleaned up setup logic - NN

**2006-09-29**

setup.py: check for required pygtk versions, fixes bug #1460783 - SC

---

**2006-09-27**

Released 0.87.6 at revision 2783

**2006-09-24**

Added line pointers to the Annotation code, and a pylab interface. See matplotlib.text.Annotation, examples/annotation_demo.py and examples/annotation_demo_pylab.py - JDH

**2006-09-18**

mathtext2.py: The SVG backend now supports the same things that the AGG backend does. Fixed some bugs with rendering, and out of bounds errors in the AGG backend - ES. Changed the return values of math_parse_s_ft2font_svg to support lines (fractions etc.)

**2006-09-17**

Added an Annotation class to facilitate annotating objects and an examples file examples/annotation_demo.py. I want to add dash support as in TextWithDash, but haven't decided yet whether inheriting from TextWithDash is the right base class or if another approach is needed - JDH

---

**2006-09-05**

Released 0.87.5 at revision 2761

**2006-09-04**

Added nxutils for some numeric add-on extension code -- specifically a better/more efficient inside polygon tester (see unit/inside_poly_*.py) - JDH

**2006-09-04**

Made bitstream fonts the rc default - JDH

**2006-08-31**

Fixed alpha-handling bug in ColorConverter, affecting collections in general and contour/contourf in particular. - EF

**2006-08-30**

ft2font.cpp: Added draw_rect_filled method (now used by mathtext2 to draw the fraction bar) to FT2Font - ES

**2006-08-29**

setupext.py: wrap calls to tk.getvar() with str(). On some systems, getvar returns a Tcl_Obj instead of a string - DSD

**2006-08-28**

mathtext2.py: Sub/superscripts can now be complex (i.e. fractions etc.). The demo is also updated - ES

**2006-08-28**

font_manager.py: Added /usr/local/share/fonts to list of X11 font directories - DSD

**2006-08-28**

mathtext2.py: Initial support for complex fractions. Also, rendering is now completely separated from parsing. The sub/superscripts now work better. Updated the mathtext2_demo.py - ES

**2006-08-27**

qt backends: don't create a QApplication when backend is imported, do it when the FigureCanvasQt is created. Simplifies applications where mpl is embedded in qt. Updated embedding_in_qt* examples - DSD

**2006-08-27**

mathtext2.py: Now the fonts are searched in the OS font dir and in the mpl-data dir. Also env is not a dict anymore. - ES

**2006-08-26**

minor changes to __init__.py, mathtex2_demo.py. Added matplotlibrc key "mathtext.mathtext2" (removed the key "mathtext2") - ES

**2006-08-21**

mathtext2.py: Initial support for fractions Updated the mathtext2_demo.py _mathtext_data.py: removed "" from the unicode dicts mathtext.py: Minor modification (because of _mathtext_data.py)- ES

**2006-08-20**

Added mathtext2.py: Replacement for mathtext.py. Supports _ ^, rm, cal etc., sin, cos etc., unicode, recursive nestings, inline math mode. The only backend currently supported is Agg __init__.py: added new rc params for mathtext2 added mathtext2_demo.py example - ES

**2006-08-19**

Added embedding_in_qt4.py example - DSD

**2006-08-11**

Added scale free Ellipse patch for Agg - CM

**2006-08-10**

Added converters to and from julian dates to matplotlib.dates (num2julian and julian2num) - JDH

**2006-08-08**

Fixed widget locking so multiple widgets could share the event handling - JDH

**2006-08-07**

Added scale free Ellipse patch to SVG and PS - CM

**2006-08-05**

Re-organized imports in numerix for numpy 1.0b2 -- TEO

**2006-08-04**

Added draw_markers to PDF backend. - JKS

**2006-08-01**

Fixed a bug in postscript's rendering of dashed lines - DSD

**2006-08-01**

figure.py: savefig() update docstring to add support for 'format' argument. backend_cairo.py: print_figure() add support 'format' argument. - SC

**2006-07-31**

Don't let postscript's xpdf distiller compress images - DSD

**2006-07-31**

Added shallowcopy() methods to all Transformations; removed copy_bbox_transform and copy_bbox_transform_shallow from transforms.py; added offset_copy() function to transforms.py to facilitate positioning artists with offsets. See examples/transoffset.py. - EF

**2006-07-31**

Don't let postscript's xpdf distiller compress images - DSD

**2006-07-29**

Fixed numerix polygon bug reported by Nick Fotopoulos. Added inverse_numerix_xy() transform method. Made autoscale_view() preserve axis direction (e.g., increasing down).- EF

**2006-07-28**

Added shallow bbox copy routine for transforms -- mainly useful for copying transforms to apply offset to. - JDH

**2006-07-28**

Added resize method to FigureManager class for Qt and Gtk backend - CM

**2006-07-28**

Added subplots_adjust button to Qt backend - CM

**2006-07-26**

Use numerix more in collections. Quiver now handles masked arrays. - EF

**2006-07-22**

Fixed bug #1209354 - DSD

**2006-07-22**

make scatter() work with the kwarg "color". Closes bug 1285750 - DSD

**2006-07-20**

backend_cairo.py: require pycairo 1.2.0. print_figure() update to output SVG using cairo.

**2006-07-19**

Added blitting for Qt4Agg - CM

**2006-07-19**

Added lasso widget and example examples/lasso_demo.py - JDH

**2006-07-18**

Added blitting for QtAgg backend - CM

**2006-07-17**

Fixed bug #1523585: skip nans in semilog plots - DSD

**2006-07-12**

Add support to render the scientific notation label over the right-side y-axis - DSD

---

**2006-07-11**

Released 0.87.4 at revision 2558

**2006-07-07**

Fixed a usetex bug with older versions of latex - DSD

**2006-07-07**

Add compatibility for NumPy 1.0 - TEO

**2006-06-29**

Added a Qt4Agg backend. Thank you James Amundson - DSD

**2006-06-26**

Fixed a usetex bug. On Windows, usetex will process postscript output in the current directory rather than in a temp directory. This is due to the use of spaces and tildes in windows paths, which cause problems with latex. The subprocess module is no longer used. - DSD

**2006-06-22**

Various changes to bar(), barh(), and hist(). Added 'edgecolor' keyword arg to bar() and barh(). The x and y args in barh() have been renamed to width and bottom respectively, and their order has been swapped to maintain a (position, value) order ala matlab. left, height, width and bottom args can now all be scalars or sequences. barh() now defaults to edge alignment instead of center alignment. Added a keyword arg 'align' to bar(), barh() and hist() that controls between edge or center bar alignment. Fixed ignoring the rcParams['patch.facecolor'] for bar color in bar() and barh(). Fixed ignoring the rcParams['lines.color'] for error bar color in bar() and barh(). Fixed a bug where patches would be cleared when error bars were plotted if rcParams['axes.hold'] was False. - MAS

**2006-06-22**

Added support for numerix 2-D arrays as alternatives to a sequence of (x,y) tuples for specifying paths in collections, quiver, contour, pcolor, transforms. Fixed contour bug involving setting limits for colormapping. Added numpy-style all() to numerix. - EF

**2006-06-20**

Added custom FigureClass hook to pylab interface - see examples/custom_figure_class.py

**2006-06-16**

Added colormaps from gist (gist_earth, gist_stern, gist_rainbow, gist_gray, gist_yarg, gist_heat, gist_ncar) - JW

**2006-06-16**

Added a pointer to parent in figure canvas so you can access the container with fig.canvas.manager. Useful if you want to set the window title, e.g., in gtk fig.canvas.manager.window.set_title, though a GUI neutral method would be preferable JDH

**2006-06-16**

Fixed colorbar.py to handle indexed colors (i.e., norm = no_norm()) by centering each colored region on its index. - EF

**2006-06-15**

Added scalex and scaley to Axes.autoscale_view to support selective autoscaling just the x or y axis, and supported these command in plot so you can say plot(something, scaley=False) and just the x axis will be autoscaled. Modified axvline and axhline to support this, so for example axvline will no longer autoscale the y axis. JDH

**2006-06-13**

Fix so numpy updates are backward compatible - TEO

**2006-06-12**

Updated numerix to handle numpy restructuring of oldnumeric - TEO

**2006-06-12**

Updated numerix.fft to handle numpy restructuring Added ImportError to numerix.linear_algebra for numpy -TEO

**2006-06-11**

Added quiverkey command to pylab and Axes, using QuiverKey class in quiver.py. Changed pylab and Axes to use quiver2 if possible, but drop back to the newly-renamed quiver_classic if necessary. Modified examples/quiver_demo.py to illustrate the new quiver and quiverkey. Changed LineCollection implementation slightly to improve compatibility with PolyCollection. - EF

**2006-06-11**

Fixed a usetex bug for windows, running latex on files with spaces in their names or paths was failing - DSD

**2006-06-09**

Made additions to numerix, changes to quiver to make it work with all numeric flavors. - EF

**2006-06-09**

Added quiver2 function to pylab and method to axes, with implementation via a Quiver class in quiver.py. quiver2 will replace quiver before the next release; it is placed alongside it initially to facilitate testing and transition. See also examples/quiver2_demo.py. - EF

**2006-06-08**

Minor bug fix to make ticker.py draw proper minus signs with usetex - DSD

**2006-06-06**

Released 0.87.3 at revision 2432

**2006-05-30**

More partial support for polygons with outline or fill, but not both. Made LineCollection inherit from ScalarMappable. - EF

**2006-05-29**

Yet another revision of aspect-ratio handling. - EF

**2006-05-27**

Committed a patch to prevent stroking zero-width lines in the svg backend - DSD

**2006-05-24**

Fixed colorbar positioning bug identified by Helge Avlesen, and improved the algorithm; added a 'pad' kwarg to control the spacing between colorbar and parent axes. - EF

**2006-05-23**

Changed color handling so that collection initializers can take any mpl color arg or sequence of args; deprecated float as grayscale, replaced by string representation of float. - EF

**2006-05-19**

Fixed bug: plot failed if all points were masked - EF

**2006-05-19**

Added custom symbol option to scatter - JDH

**2006-05-18**

New example, multi_image.py; colorbar fixed to show offset text when the ScalarFormatter is used; FixedFormatter augmented to accept and display offset text. - EF

**2006-05-14**

New colorbar; old one is renamed to colorbar_classic. New colorbar code is in colorbar.py, with wrappers in figure.py and pylab.py. Fixed aspect-handling bug reported by Michael Mossey. Made backend_bases.draw_quad_mesh() run.- EF

**2006-05-08**

Changed handling of end ranges in contourf: replaced "clip-ends" kwarg with "extend". See docstring for details. -EF

**2006-05-08**

Added axisbelow to rc - JDH

**2006-05-08**

If using PyGTK require version 2.2+ - SC

**2006-04-19**

Added compression support to PDF backend, controlled by new pdf.compression rc setting. - JKS

**2006-04-19**

Added Jouni's PDF backend

**2006-04-18**

Fixed a bug that caused agg to not render long lines

**2006-04-16**

Masked array support for pcolormesh; made pcolormesh support the same combinations of X,Y,C dimensions as pcolor does; improved (I hope) description of grid used in pcolor, pcolormesh. - EF

**2006-04-14**

Reorganized axes.py - EF

**2006-04-13**

Fixed a bug Ryan found using usetex with sans-serif fonts and exponential tick labels - DSD

**2006-04-11**

Refactored backend_ps and backend_agg to prevent module-level texmanager imports. Now these imports only occur if text.usetex rc setting is true - DSD

**2006-04-10**

Committed changes required for building mpl on win32 platforms with visual studio. This allows wxpython blitting for fast animations. - CM

**2006-04-10**

Fixed an off-by-one bug in Axes.change_geometry.

**2006-04-10**

Fixed bug in pie charts where wedge wouldn't have label in legend. Submitted by Simon Hildebrandt. - ADS

**2006-05-06**

Usetex makes temporary latex and dvi files in a temporary directory, rather than in the user's current working directory - DSD

**2006-04-05**

Applied Ken's wx deprecation warning patch closing sf patch #1465371 - JDH

**2006-04-05**

Added support for the new API in the postscript backend. Allows values to be masked using nan's, and faster file creation - DSD

**2006-04-05**

Use python's subprocess module for usetex calls to external programs. subprocess catches when they exit abnormally so an error can be raised. - DSD

**2006-04-03**

Fixed the bug in which widgets would not respond to events. This regressed the twinx functionality, so I also updated subplots_adjust to update axes that share an x or y with a subplot instance. - CM

**2006-04-02**

Moved PBox class to transforms and deleted pbox.py; made pylab axis command a thin wrapper for Axes.axis; more tweaks to aspect-ratio handling; fixed Axes.specgram to account for the new imshow default of unit aspect ratio; made contour set the Axes.dataLim. - EF

**2006-03-31**

Fixed the Qt "Underlying C/C++ object deleted" bug. - JRE

**2006-03-31**

Applied Vasily Sulatskov's Qt Navigation Toolbar enhancement. - JRE

**2006-03-31**

Ported Norbert's rewriting of Halldor's stineman_interp algorithm to make it numerix compatible and added code to matplotlib.mlab. See examples/interp_demo.py - JDH

**2006-03-30**

Fixed a bug in aspect ratio handling; blocked potential crashes when panning with button 3; added axis('image') support. - EF

**2006-03-28**

More changes to aspect ratio handling; new PBox class in new file pbox.py to facilitate resizing and repositioning axes; made PolarAxes maintain unit aspect ratio. - EF

**2006-03-23**

Refactored TextWithDash class to inherit from, rather than delegate to, the Text class. Improves object inspection and closes bug # 1357969 - DSD

**2006-03-22**

Improved aspect ratio handling, including pylab interface. Interactive resizing, pan, zoom of images and plots (including panels with a shared axis) should work. Additions and possible refactoring are still likely. - EF

**2006-03-21**

Added another colorbrewer colormap (RdYlBu) - JSWHIT

**2006-03-21**

Fixed tickmarks for logscale plots over very large ranges. Closes bug # 1232920 - DSD

**2006-03-21**

Added Rob Knight's arrow code; see examples/arrow_demo.py - JDH

**2006-03-20**

Added support for masking values with nan's, using ADS's isnan module and the new API. Works for *Agg backends - DSD

**2006-03-20**

Added contour.negative_linestyle rcParam - ADS

**2006-03-20**

Added _isnan extension module to test for nan with Numeric - ADS

**2006-03-17**

Added Paul and Alex's support for faceting with quadmesh in sf patch 1411223 - JDH

**2006-03-17**

Added Charle Twardy's pie patch to support colors=None. Closes sf patch 1387861 - JDH

**2006-03-17**

Applied sophana's patch to support overlapping axes with toolbar navigation by toggling activation with the 'a' key. Closes sf patch 1432252 - JDH

**2006-03-17**

Applied Aarre's linestyle patch for backend EMF; closes sf patch 1449279 - JDH

**2006-03-17**

Applied Jordan Dawe's patch to support kwarg properties for grid lines in the grid command. Closes sf patch 1451661 - JDH

**2006-03-17**

Center postscript output on page when using usetex - DSD

**2006-03-17**

subprocess module built if Python <2.4 even if subprocess can be imported from an egg - ADS

**2006-03-17**

Added _subprocess.c from Python upstream and hopefully enabled building (without breaking) on Windows, although not tested. - ADS

**2006-03-17**

Updated subprocess.py to latest Python upstream and reverted name back to subprocess.py - ADS

**2006-03-16**

Added John Porter's 3D handling code

---

**2006-03-16**

Released 0.87.2 at revision 2150

**2006-03-15**

Fixed bug in MaxNLocator revealed by daigos@infinito.it. The main change is that Locator.nonsingular now adjusts vmin and vmax if they are nearly the same, not just if they are equal. A new kwarg, "tiny", sets the threshold. - EF

**2006-03-14**

Added import of compatibility library for newer numpy linear_algebra - TEO

**2006-03-12**

Extended "load" function to support individual columns and moved "load" and "save" into matplotlib.mlab so they can be used outside of pylab -- see examples/load_converter.py - JDH

**2006-03-12**

Added AutoDateFormatter and AutoDateLocator submitted by James Evans. Try the load_converter.py example for a demo. - ADS

**2006-03-11**

Added subprocess module from python-2.4 - DSD

**2006-03-11**

Fixed landscape orientation support with the usetex option. The backend_ps print_figure method was getting complicated, I added a _print_figure_tex method to maintain some degree of sanity - DSD

**2006-03-11**

Added "papertype" savefig kwarg for setting postscript papersizes. papertype and ps.papersize rc setting can also be set to "auto" to autoscale pagesizes - DSD

**2006-03-09**

Apply P-J's patch to make pstoeps work on windows patch report # 1445612 - DSD

**2006-03-09**

Make backend rc parameter case-insensitive - DSD

**2006-03-07**

Fixed bug in backend_ps related to C0-C6 papersizes, which were causing problems with postscript viewers. Supported page sizes include letter, legal, ledger, A0-A10, and B0-B10 - DSD

---

**2006-03-07**

Released 0.87.1

**2006-03-04**

backend_cairo.py: fix get_rgb() bug reported by Keith Briggs. Require pycairo 1.0.2. Support saving png to file-like objects. - SC

**2006-03-03**

Fixed pcolor handling of vmin, vmax - EF

**2006-03-02**

improve page sizing with usetex with the latex geometry package. Closes bug # 1441629 - DSD

**2006-03-02**

Fixed dpi problem with usetex png output. Accepted a modified version of patch # 1441809 - DSD

**2006-03-01**

Fixed axis('scaled') to deal with case xmax < xmin - JSWHIT

**2006-03-01**

Added reversed colormaps (with '_r' appended to name) - JSWHIT

**2006-02-27**

Improved eps bounding boxes with usetex - DSD

**2006-02-27**

Test svn commit, again!

**2006-02-27**

Fixed two dependency checking bugs related to usetex on Windows - DSD

**2006-02-27**

Made the rc deprecation warnings a little more human readable.

**2006-02-26**

Update the previous gtk.main_quit() bug fix to use gtk.main_level() - SC

**2006-02-24**

Implemented alpha support in contour and contourf - EF

**2006-02-22**

Fixed gtk main quit bug when quit was called before mainloop. - JDH

**2006-02-22**

Small change to colors.py to workaround apparent bug in numpy masked array module - JSWHIT

**2006-02-22**

Fixed bug in ScalarMappable.to_rgba() reported by Ray Jones, and fixed incorrect fix found by Jeff Whitaker - EF

---

**2006-02-22**

Released 0.87

**2006-02-21**

Fixed portrait/landscape orientation in postscript backend - DSD

**2006-02-21**

Fix bug introduced in yesterday's bug fix - SC

**2006-02-20**

backend_gtk.py FigureCanvasGTK.draw(): fix bug reported by David Tremouilles - SC

**2006-02-20**

Remove the "pygtk.require('2.4')" error from examples/embedding_in_gtk2.py - SC

**2006-02-18**

backend_gtk.py FigureCanvasGTK.draw(): simplify to use (rather than duplicate) the expose_event() drawing code - SC

**2006-02-12**

Added stagger or waterfall plot capability to LineCollection; illustrated in examples/collections.py. - EF

**2006-02-11**

Massive cleanup of the usetex code in the postscript backend. Possibly fixed the clipping issue users were reporting with older versions of ghostscript - DSD

**2006-02-11**

Added autolim kwarg to axes.add_collection. Changed collection get_verts() methods accordingly. - EF

**2006-02-09**

added a temporary rc parameter text.dvipnghack, to allow Mac users to get nice results with the usetex option. - DSD

**2006-02-09**

Fixed a bug related to setting font sizes with the usetex option. - DSD

**2006-02-09**

Fixed a bug related to usetex's latex code. - DSD

**2006-02-09**

Modified behavior of font.size rc setting. You should define font.size in pts, which will set the "medium" or default fontsize. Special text sizes like axis labels or tick labels can be given relative font sizes like small, large, x-large, etc. and will scale accordingly. - DSD

**2006-02-08**

Added py2exe specific datapath check again. Also added new py2exe helper function get_py2exe_datafiles for use in py2exe setup.py scripts. - CM

**2006-02-02**

Added box function to pylab

**2006-02-02**

Fixed a problem in setupext.py, tk library formatted in unicode caused build problems - DSD

**2006-02-01**

Dropped TeX engine support in usetex to focus on LaTeX. - DSD

**2006-01-29**

Improved usetex option to respect the serif, sans-serif, monospace, and cursive rc settings. Removed the font.latex.package rc setting, it is no longer required - DSD

**2006-01-29**

Fixed tex's caching to include font.family rc information - DSD

**2006-01-29**

Fixed subpixel rendering bug in *Agg that was causing uneven gridlines - JDH

**2006-01-28**

Added fontcmd to backend_ps's RendererPS.draw_tex, to support other font families in eps output - DSD

**2006-01-28**

Added MaxNLocator to ticker.py, and changed contour.py to use it by default. - EF

**2006-01-28**

Added fontcmd to backend_ps's RendererPS.draw_tex, to support other font families in eps output - DSD

**2006-01-27**

Buffered reading of matplotlibrc parameters in order to allow 'verbose' settings to be processed first (allows verbose.report during rc validation process) - DSD

**2006-01-27**

Removed setuptools support from setup.py and created a separate setupegg.py file to replace it. - CM

**2006-01-26**

Replaced the ugly datapath logic with a cleaner approach from http://wiki.python.org/moin/DistutilsInstallDataScattered. Overrides the install_data command. - CM

**2006-01-24**

Don't use character typecodes in cntr.c --- changed to use defined typenumbers instead. - TEO

**2006-01-24**

Fixed some bugs in usetex's and ps.usedistiller's dependency

**2006-01-24**

Added masked array support to scatter - EF

**2006-01-24**

Fixed some bugs in usetex's and ps.usedistiller's dependency checking - DSD

---

**2006-01-24**

Released 0.86.2

**2006-01-20**

Added a converters dict to pylab load to convert selected columns to float -- especially useful for files with date strings, uses a datestr2num converter - JDH

**2006-01-20**

Added datestr2num to matplotlib dates to convert a string or sequence of strings to a matplotlib datenum

**2006-01-18**

Added quadrilateral pcolormesh patch 1409190 by Alex Mont and Paul Kienzle -- this is *Agg only for now. See examples/quadmesh_demo.py - JDH

**2006-01-18**

Added Jouni's boxplot patch - JDH

**2006-01-18**

Added comma delimiter for pylab save - JDH

**2006-01-12**

Added Ryan's legend patch - JDH

**2006-01-12**

Fixed numpy / numeric to use .dtype.char to keep in SYNC with numpy SVN

---

**2006-01-11**

Released 0.86.1

**2006-01-11**

Fixed setup.py for win32 build and added rc template to the MANIFEST.in

**2006-01-10**

Added xpdf distiller option. matplotlibrc ps.usedistiller can now be none, false, ghostscript, or xpdf. Validation checks for dependencies. This needs testing, but the xpdf option should produce the highest-quality output and small file sizes - DSD

**2006-01-10**

For the usetex option, backend_ps now does all the LaTeX work in the os's temp directory - DSD

**2006-01-10**

Added checks for usetex dependencies. - DSD

---

**2006-01-09**

Released 0.86

**2006-01-04**

Changed to support numpy (new name for scipy_core) - TEO

**2006-01-04**

Added Mark's scaled axes patch for shared axis

**2005-12-28**

Added Chris Barker's build_wxagg patch - JDH

**2005-12-27**

Altered numerix/scipy to support new scipy package structure - TEO

**2005-12-20**

Fixed Jame's Boyles date tick reversal problem - JDH

**2005-12-20**

Added Jouni's rc patch to support lists of keys to set on - JDH

**2005-12-12**

Updated pyparsing and mathtext for some speed enhancements (Thanks Paul McGuire) and minor fixes to scipy numerix and setuptools

**2005-12-12**

Matplotlib data is now installed as package_data in the matplotlib module. This gets rid of checking the many possibilities in matplotlib._get_data_path() - CM

**2005-12-11**

Support for setuptools/pkg_resources to build and use matplotlib as an egg. Still allows matplotlib to exist using a traditional distutils install. - ADS

**2005-12-03**

Modified setup to build matplotlibrc based on compile time findings. It will set numerix in the order of scipy, numarray, Numeric depending on which are founds, and backend as in preference order GTKAgg, WXAgg, TkAgg, GTK, Agg, PS

**2005-12-03**

Modified scipy patch to support Numeric, scipy and numarray Some work remains to be done because some of the scipy imports are broken if only the core is installed. e.g., apparently we need from scipy.basic.fftpack import * rather than from scipy.fftpack import *

**2005-12-03**

Applied some fixes to Nicholas Young's nonuniform image patch

**2005-12-01**

Applied Alex Gontmakher hatch patch - PS only for now

**2005-11-30**

Added Rob McMullen's EMF patch

**2005-11-30**

Added Daishi's patch for scipy

**2005-11-30**

Fixed out of bounds draw markers segfault in agg

**2005-11-28**

Got TkAgg blitting working 100% (cross fingers) correctly. - CM

**2005-11-27**

Multiple changes in cm.py, colors.py, figure.py, image.py, contour.py, contour_demo.py; new _cm.py, examples/image_masked.py.

1. Separated the color table data from cm.py out into a new file, _cm.py, to make it easier to find the actual code in cm.py and to add new colormaps. Also added some line breaks to the color data dictionaries. Everything from _cm.py is imported by cm.py, so the split should be transparent.
2. Enabled automatic generation of a colormap from a list of colors in contour; see modified examples/contour_demo.py.
3. Support for imshow of a masked array, with the ability to specify colors (or no color at all) for masked regions, and for regions that are above or below the normally mapped region. See examples/image_masked.py.
4. In support of the above, added two new classes, ListedColormap, and no_norm, to colors.py, and modified the Colormap class to include common functionality. Added a clip kwarg to the normalize class. Reworked color handling in contour.py, especially in the ContourLabeller mixin.

- EF

**2005-11-25**

Changed text.py to ensure color is hashable. EF

---

**2005-11-16**

Released 0.85

**2005-11-16**

Changed the default linewidth in rc to 1.0

**2005-11-16**

Replaced agg_to_gtk_drawable with pure pygtk pixbuf code in backend_gtkagg. When the equivalent is doe for blit, the agg extension code will no longer be needed

**2005-11-16**

Added a maxdict item to cbook to prevent caches from growing w/o bounds

**2005-11-15**

Fixed a colorup/colordown reversal bug in finance.py -- Thanks Gilles

**2005-11-15**

Applied Jouni K Steppanen's boxplot patch SF patch#1349997 - JDH

**2005-11-09**

added axisbelow attr for Axes to determine whether ticks and such are above or below the actors

**2005-11-08**

Added Nicolas' irregularly spaced image patch

**2005-11-08**

Deprecated HorizontalSpanSelector and replaced with SpanSelection that takes a third arg, direction. The new SpanSelector supports horizontal and vertical span selection, and the appropriate min/max is returned. - CM

**2005-11-08**

Added lineprops dialog for gtk

**2005-11-03**

Added FIFOBuffer class to mlab to support real time feeds and examples/fifo_buffer.py

**2005-11-01**

Contributed Nickolas Young's patch for afm mathtext to support mathtext based upon the standard postscript Symbol font when ps.usetex = True.

**2005-10-26**

Added support for scatter legends - thanks John Gill

**2005-10-20**

Fixed image clipping bug that made some tex labels disappear. JDH

**2005-10-14**

Removed sqrt from dvipng 1.6 alpha channel mask.

**2005-10-14**

Added width kwarg to hist function

**2005-10-10**

Replaced all instances of os.rename with shutil.move

**2005-10-05**

Added Michael Brady's ydate patch

**2005-10-04**

Added rkern's texmanager patch

**2005-09-25**

contour.py modified to use a single ContourSet class that handles filled contours, line contours, and labels; added keyword arg (clip_ends) to contourf. Colorbar modified to work with new ContourSet object; if the ContourSet has lines rather than polygons, the colorbar will follow suit. Fixed a bug introduced in 0.84, in which contourf(...,colors=...) was broken - EF

---

**2005-09-19**

Released 0.84

**2005-09-14**

Added a new 'resize_event' which triggers a callback with a backend_bases.ResizeEvent object - JDH

**2005-09-14**

font_manager.py: removed chkfontpath from x11FontDirectory() - SC

**2005-09-14**

Factored out auto date locator/formatter factory code into matplotlib.date.date_ticker_factory; applies John Bryne's quiver patch.

**2005-09-13**

Added Mark's axes positions history patch #1286915

**2005-09-09**

Added support for auto canvas resizing with:

```
fig.set_figsize_inches(9,5,forward=True) # inches
```

OR:

```
fig.resize(400,300)   # pixels
```

**2005-09-07**

figure.py: update Figure.draw() to use the updated renderer.draw_image() so that examples/figimage_demo.py works again. examples/stock_demo.py: remove data_clipping (which no longer exists) - SC

**2005-09-06**

Added Eric's tick.direction patch: in or out in rc

**2005-09-06**

Added Martin's rectangle selector widget

**2005-09-04**

Fixed a logic err in text.py that was preventing rgxsuper from matching - JDH

**2005-08-29**

Committed Ken's wx blit patch #1275002

**2005-08-26**

colorbar modifications - now uses contourf instead of imshow so that colors used by contourf are displayed correctly. Added two new keyword args (cspacing and clabels) that are only relevant for ContourMappable images - JSWHIT

**2005-08-24**

Fixed a PS image bug reported by Darren - JDH

**2005-08-23**

colors.py: change hex2color() to accept unicode strings as well as normal strings. Use isinstance() instead of types.IntType etc - SC

**2005-08-16**

removed data_clipping line and rc property - JDH

**2005-08-22**

backend_svg.py: Remove redundant "x=0.0 y=0.0" from svg element. Increase svg version from 1.0 to 1.1. Add viewBox attribute to svg element to allow SVG documents to scale-to-fit into an arbitrary viewport - SC

**2005-08-16**

Added Eric's dot marker patch - JDH

**2005-08-08**

Added blitting/animation for TkAgg - CM

**2005-08-05**

Fixed duplicate tickline bug - JDH

**2005-08-05**

Fixed a GTK animation bug that cropped up when doing animations in gtk//gtkagg canvases that had widgets packed above them

**2005-08-05**

Added Clovis Goldemberg patch to the tk save dialog

**2005-08-04**

Removed origin kwarg from backend.draw_image. origin is handled entirely by the frontend now.

**2005-07-03**

Fixed a bug related to TeX commands in backend_ps

**2005-08-03**

Fixed SVG images to respect upper and lower origins.

**2005-08-03**

Added flipud method to image and removed it from to_str.

**2005-07-29**

Modified figure.figaspect to take an array or number; modified backend_svg to write utf-8 - JDH

**2005-07-30**

backend_svg.py: embed png image files in svg rather than linking to a separate png file, fixes bug #1245306 (thanks to Norbert Nemec for the patch) - SC

---

**2005-07-29**

Released 0.83.2

**2005-07-27**

Applied SF patch 1242648: minor rounding error in IndexDateFormatter in dates.py

**2005-07-27**

Applied sf patch 1244732: Scale axis such that circle looks like circle - JDH

**2005-07-29**

Improved message reporting in texmanager and backend_ps - DSD

**2005-07-28**

backend_gtk.py: update FigureCanvasGTK.draw() (needed due to the recent expose_event() change) so that examples/anim.py works in the usual way - SC

**2005-07-26**

Added new widgets Cursor and HorizontalSpanSelector to matplotlib.widgets. See examples/widgets/cursor.py and examples/widgets/span_selector.py - JDH

**2005-07-26**

added draw event to mpl event hierarchy -- triggered on figure.draw

**2005-07-26**

backend_gtk.py: allow 'f' key to toggle window fullscreen mode

**2005-07-26**

backend_svg.py: write "<.../>" elements all on one line and remove surplus spaces - SC

**2005-07-25**

backend_svg.py: simplify code by deleting GraphicsContextSVG and RendererSVG.new_gc(), and moving the gc.get_capstyle() code into RendererSVG._get_gc_props_svg() - SC

**2005-07-24**

backend_gtk.py: call FigureCanvasBase.motion_notify_event() on all motion-notify-events, not just ones where a modifier key or button has been pressed (fixes bug report from Niklas Volbers) - SC

**2005-07-24**

backend_gtk.py: modify print_figure() use own pixmap, fixing problems where print_figure() overwrites the display pixmap. return False from all button/key etc events - to allow the event to propagate further - SC

**2005-07-23**

backend_gtk.py: change expose_event from using set_back_pixmap(); clear() to draw_drawable() - SC

**2005-07-23**

backend_gtk.py: removed pygtk.require() matplotlib/__init__.py: delete 'FROZEN' and 'McPLError' which are no longer used - SC

**2005-07-22**

backend_gdk.py: removed pygtk.require() - SC

**2005-07-21**

backend_svg.py: Remove unused imports. Remove methods doc strings which just duplicate the docs from backend_bases.py. Rename draw_mathtext to _draw_mathtext. - SC

**2005-07-17**

examples/embedding_in_gtk3.py: new example demonstrating placing a FigureCanvas in a gtk.ScrolledWindow - SC

**2005-07-14**

Fixed a Windows related bug (#1238412) in texmanager - DSD

**2005-07-11**

Fixed color kwarg bug, setting color=1 or 0 caused an exception - DSD

**2005-07-07**

Added Eric's MA set_xdata Line2D fix - JDH

**2005-07-06**

Made HOME/.matplotlib the new config dir where the matplotlibrc file, the ttf.cache, and the tex.cache live. The new default filenames in .matplotlib have no leading dot and are not hidden. e.g., the new

names are matplotlibrc tex.cache ttffont.cache. This is how ipython does it so it must be right. If old files are found, a warning is issued and they are moved to the new location. Also fixed texmanager to put all files, including temp files in ~/.matplotlib/tex.cache, which allows you to usetex in non-writable dirs.

**2005-07-05**

Fixed bug #1231611 in subplots adjust layout. The problem was that the text caching mechanism was not using the transformation affine in the key. - JDH

**2005-07-05**

Fixed default backend import problem when using API (SF bug # 1209354 - see API_CHANGES for more info - JDH

**2005-07-04**

backend_gtk.py: require PyGTK version 2.0.0 or higher - SC

**2005-06-30**

setupext.py: added numarray_inc_dirs for building against numarray when not installed in standard location - ADS

**2005-06-27**

backend_svg.py: write figure width, height as int, not float. Update to fix some of the pychecker warnings - SC

**2005-06-23**

Updated examples/agg_test.py to demonstrate curved paths and fills - JDH

**2005-06-21**

Moved some texmanager and backend_agg tex caching to class level rather than instance level - JDH

**2005-06-20**

setupext.py: fix problem where _nc_backend_gdk is installed to the wrong directory - SC

**2005-06-19**

Added 10.4 support for CocoaAgg. - CM

**2005-06-18**

Move Figure.get_width_height() to FigureCanvasBase and return int instead of float. - SC

**2005-06-18**

Applied Ted Drain's QtAgg patch: 1) Changed the toolbar to be a horizontal bar of push buttons instead of a QToolbar and updated the layout algorithms in the main window accordingly. This eliminates the ability to drag and drop the toolbar and detach it from the window. 2) Updated the resize algorithm in the main window to show the correct size for the plot widget as requested. This works almost correctly right now. It looks to me like the final size of the widget is off by the border of the main window but I haven't figured out a way to get that information yet. We could just add a small margin to the new size but that seems a little hacky. 3) Changed the x/y location label to be in the toolbar like the Tk backend instead of as a status line at the bottom of the widget. 4) Changed the toolbar pixmaps to use the ppm

files instead of the png files. I noticed that the Tk backend buttons looked much nicer and it uses the ppm files so I switched them.

**2005-06-17**

Modified the gtk backend to not queue mouse motion events. This allows for live updates when dragging a slider. - CM

**2005-06-17**

Added starter CocoaAgg backend. Only works on OS 10.3 for now and requires PyObjC. (10.4 is high priority) - CM

**2005-06-17**

Upgraded pyparsing and applied Paul McGuire's suggestions for speeding things up. This more than doubles the speed of mathtext in my simple tests. JDH

**2005-06-16**

Applied David Cooke's subplot make_key patch

---

### 0.82 (2005-06-15)

**2005-06-15**

Added subplot config tool to GTK* backends -- note you must now import the NavigationToolbar2 from your backend of choice rather than from backend_gtk because it needs to know about the backend specific canvas -- see examples/embedding_in_gtk2.py. Ditto for wx backend -- see examples/embedding_in_wxagg.py

**2005-06-15**

backend_cairo.py: updated to use pycairo 0.5.0 - SC

**2005-06-14**

Wrote some GUI neutral widgets (Button, Slider, RadioButtons, CheckButtons) in matplotlib.widgets. See examples/widgets/*.py - JDH

**2005-06-14**

Exposed subplot parameters as rc vars and as the fig SubplotParams instance subplotpars. See figure.SubplotParams, figure.Figure.subplots_adjust and the pylab method subplots_adjust and examples/subplots_adjust.py . Also added a GUI neutral widget for adjusting subplots, see examples/subplot_toolbar.py - JDH

**2005-06-13**

Exposed cap and join style for lines with new rc params and line properties:

```
lines.dash_joinstyle : miter        # miter|round|bevel
lines.dash_capstyle : butt          # butt|round|projecting
lines.solid_joinstyle : miter       # miter|round|bevel
lines.solid_capstyle : projecting   # butt|round|projecting
```

**2005-06-13**

Added kwargs to Axes init

**2005-06-13**

Applied Baptiste's tick patch - JDH

**2005-06-13**

Fixed rc alias 'l' bug reported by Fernando by removing aliases for mainlevel rc options. - JDH

**2005-06-10**

Fixed bug #1217637 in ticker.py - DSD

**2005-06-07**

Fixed a bug in texmanager.py: .aux files not being removed - DSD

**2005-06-08**

Added Sean Richard's hist binning fix -- see API_CHANGES - JDH

**2005-06-07**

Fixed a bug in texmanager.py: .aux files not being removed - DSD

---

### 0.81 (2005-06-07)

**2005-06-06**

Added autoscale_on prop to axes

**2005-06-06**

Added Nick's picker "among" patch - JDH

**2005-06-05**

Fixed a TeX/LaTeX font discrepancy in backend_ps. - DSD

**2005-06-05**

Added a ps.distill option in rc settings. If True, postscript output will be distilled using ghostscript, which should trim the file size and allow it to load more quickly. Hopefully this will address the issue of large ps files due to font definitions. Tested with gnu-ghostscript-8.16. - DSD

**2005-06-03**

Improved support for tex handling of text in backend_ps. - DSD

**2005-06-03**

Added rc options to render text with tex or latex, and to select the latex font package. - DSD

**2005-06-03**

Fixed a bug in ticker.py causing a ZeroDivisionError

**2005-06-02**

backend_gtk.py remove DBL_BUFFER, add line to expose_event to try to fix pygtk 2.6 redraw problem - SC

**2005-06-01**

The default behavior of ScalarFormatter now renders scientific notation and large numerical offsets in a label at the end of the axis. - DSD

**2005-06-01**

Added Nicholas' frombyte image patch - JDH

**2005-05-31**

Added vertical TeX support for agg - JDH

**2005-05-31**

Applied Eric's cntr patch - JDH

**2005-05-27**

Finally found the pesky agg bug (which Maxim was kind enough to fix within hours) that was causing a segfault in the win32 cached marker drawing. Now windows users can get the enormous performance benefits of cached markers w/o those occasional pesy screenshots. - JDH

**2005-05-27**

Got win32 build system working again, using a more recent version of gtk and pygtk in the win32 build, gtk 2.6 from https://web.archive.org/web/20050527002647/https://www.gimp.org/~tml/gimp/win32/downloads.html (you will also need libpng12.dll to use these). I haven't tested whether this binary build of mpl for win32 will work with older gtk runtimes, so you may need to upgrade.

**2005-05-27**

Fixed bug where 2nd wxapp could be started if using wxagg backend. - ADS

**2005-05-26**

Added Daishi text with dash patch -- see examples/dashtick.py

**2005-05-26**

Moved backend_latex functionality into backend_ps. If text.usetex=True, the PostScript backend will use LaTeX to generate the .ps or .eps file. Ghostscript is required for eps output. - DSD

**2005-05-24**

Fixed alignment and color issues in latex backend. - DSD

**2005-05-21**

Fixed raster problem for small rasters with dvipng -- looks like it was a premultiplied alpha problem - JDH

**2005-05-20**

Added linewidth and faceted kwarg to scatter to control edgewidth and color. Also added autolegend patch to inspect line segments.

**2005-05-18**

Added Orsay and JPL qt fixes - JDH

**2005-05-17**

Added a psfrag latex backend -- some alignment issues need to be worked out. Run with -dLaTeX and a *.tex file and *.eps file are generated. latex and dvips the generated latex file to get ps output. Note xdvi *does* not work, you must generate ps.- JDH

**2005-05-13**

Added Florent Rougon's Axis set_label1 patch

**2005-05-17**

pcolor optimization, fixed bug in previous pcolor patch - JSWHIT

**2005-05-16**

Added support for masked arrays in pcolor - JSWHIT

**2005-05-12**

Started work on TeX text for antigrain using pngdvi -- see examples/tex_demo.py and the new module matplotlib.texmanager. Rotated text not supported and rendering small glyphs is not working right yet. But large fontsizes and/or high dpi saved figs work great.

**2005-05-10**

New image resize options interpolation options. New values for the interp kwarg are

> 'nearest', 'bilinear', 'bicubic', 'spline16', 'spline36', 'hanning', 'hamming', 'hermite', 'kaiser', 'quadric', 'catrom', 'gaussian', 'bessel', 'mitchell', 'sinc', 'lanczos', 'blackman'

See help(imshow) for details, particularly the interpolation, filternorm and filterrad kwargs

**2005-05-10**

Applied Eric's contour mem leak fixes - JDH

**2005-05-10**

Extended python agg wrapper and started implementing backend_agg2, an agg renderer based on the python wrapper. This will be more flexible and easier to extend than the current backend_agg. See also examples/agg_test.py - JDH

**2005-05-09**

Added Marcin's no legend patch to exclude lines from the autolegend builder:

```
plot(x, y, label='nolegend')
```

**2005-05-05**

Upgraded to agg23

**2005-05-05**

Added newscalarformatter_demo.py to examples. -DSD

**2005-05-04**

Added NewScalarFormatter. Improved formatting of ticklabels, scientific notation, and the ability to plot large numbers with small ranges, by determining a numerical offset. See ticker.NewScalarFormatter for more details. -DSD

**2005-05-03**

Added the option to specify a delimiter in pylab.load -DSD

**2005-04-28**

Added Darren's line collection example

**2005-04-28**

Fixed aa property in agg - JDH

**2005-04-27**

Set postscript page size in .matplotlibrc - DSD

**2005-04-26**

Added embedding in qt example. - JDH

**2005-04-14**

Applied Michael Brady's qt backend patch: 1) fix a bug where keyboard input was grabbed by the figure and not released 2) turn on cursor changes 3) clean up a typo and commented-out print statement. - JDH

**2005-04-14**

Applied Eric Firing's masked data lines patch and contour patch. Support for masked arrays has been added to the plot command and to the Line2D object. Only the valid points are plotted. A "valid_only" kwarg was added to the get_xdata() and get_ydata() methods of Line2D; by default it is False, so that the original data arrays are returned. Setting it to True returns the plottable points. - see examples/masked_demo.py - JDH

**2005-04-13**

Applied Tim Leslie's arrow key event handling patch - JDH

### 0.80

**2005-04-11**

Applied a variant of rick's xlim/ylim/axis patch. These functions now take kwargs to let you selectively alter only the min or max if desired. e.g., xlim(xmin=2) or axis(ymax=3). They always return the new lim. - JDH

**2005-04-11**

Incorporated Werner's wx patch -- wx backend should be compatible with wxpython2.4 and recent versions of 2.5. Some early versions of wxpython 2.5 will not work because there was a temporary change in the dc API that was rolled back to make it 2.4 compliant

**2005-04-11**

modified tkagg show so that new figure window pops up on call to figure

**2005-04-11**

fixed wxapp init bug

**2005-04-02**

updated backend_ps.draw_lines, draw_markers for use with the new API - DSD

**2005-04-01**

Added editable polygon example

---

### 0.74 (2005-03-31)

**2005-03-30**

Fixed and added checks for floating point inaccuracy in ticker.Base - DSD

**2005-03-30**

updated /ellipse definition in backend_ps.py to address bug #1122041 - DSD

**2005-03-29**

Added unicode support for Agg and PS - JDH

**2005-03-28**

Added Jarrod's svg patch for text - JDH

**2005-03-28**

Added Ludal's arrow and quiver patch - JDH

**2005-03-28**

Added label kwarg to Axes to facilitate forcing the creation of new Axes with otherwise identical attributes

**2005-03-28**

Applied boxplot and OSX font search patches

**2005-03-27**

Added ft2font NULL check to fix Japanese font bug - JDH

**2005-03-27**

Added sprint legend patch plus John Gill's tests and fix -- see examples/legend_auto.py - JDH

---

### 0.73.1 (2005-03-19)

**2005-03-19**

Reverted wxapp handling because it crashed win32 - JDH

**2005-03-18**

Add .number attribute to figure objects returned by figure() - FP

---

### 0.73 (2005-03-18)

**2005-03-16**

Fixed labelsep bug

**2005-03-16**

Applied Darren's ticker fix for small ranges - JDH

**2005-03-16**

Fixed tick on horiz colorbar - JDH

**2005-03-16**

Added Japanese winreg patch - JDH

**2005-03-15**

backend_gtkagg.py: changed to use double buffering, this fixes the problem reported Joachim Berdal Haga - "Parts of plot lagging from previous frame in animation". Tested with anim.py and it makes no noticeable difference to performance (23.7 before, 23.6 after) - SC

**2005-03-14**

add src/_backend_gdk.c extension to provide a substitute function for pixbuf.get_pixels_array(). Currently pixbuf.get_pixels_array() only works with Numeric, and then only works if pygtk has been compiled with Numeric support. The change provides a function pixbuf_get_pixels_array() which works with Numeric and numarray and is always available. It means that backend_gtk should be able to display images and mathtext in all circumstances. - SC

**2005-03-11**

Upgraded CXX to 5.3.1

**2005-03-10**

remove GraphicsContextPS.set_linestyle() and GraphicsContextSVG.set_linestyle() since they do no more than the base class GraphicsContext.set_linestyle() - SC

**2005-03-09**

Refactored contour functionality into dedicated module

**2005-03-09**

Added Eric's contourf updates and Nadia's clabel functionality

**2005-03-09**

Moved colorbar to figure.Figure to expose it for API developers - JDH

**2005-03-09**

backend_cairo.py: implemented draw_markers() - SC

**2005-03-09**

cbook.py: only use enumerate() (the python version) if the builtin version is not available. Add new function 'izip' which is set to itertools.izip if available and the python equivalent if not available. - SC

**2005-03-07**

backend_gdk.py: remove PIXELS_PER_INCH from points_to_pixels(), but still use it to adjust font sizes. This allows the GTK version of line_styles.py to more closely match GTKAgg, previously the markers were being drawn too large. - SC

**2005-03-01**

Added Eric's contourf routines

**2005-03-01**

Added start of proper agg SWIG wrapper. I would like to expose agg functionality directly a the user level and this module will serve that purpose eventually, and will hopefully take over most of the functionality of the current _image and _backend_agg modules. - JDH

**2005-02-28**

Fixed polyfit / polyval to convert input args to float arrays - JDH

**2005-02-25**

Add experimental feature to backend_gtk.py to enable/disable double buffering (DBL_BUFFER=True/False) - SC

**2005-02-24**

colors.py change ColorConverter.to_rgb() so it always returns rgb (and not rgba), allow cnames keys to be cached, change the exception raised from RuntimeError to ValueError (like hex2color()) hex2color() use a regular expression to check the color string is valid - SC

**2005-02-23**

Added rc param ps.useafm so backend ps can use native afm fonts or truetype. afme breaks mathtext but causes much smaller font sizes and may result in images that display better in some contexts (e.g., pdfs incorporated into latex docs viewed in acrobat reader). I would like to extend this approach to allow the user to use truetype only for mathtext, which should be easy.

**2005-02-23**

Used sequence protocol rather than tuple in agg collection drawing routines for greater flexibility - JDH

---

### 0.72.1 (2005-02-22)

**2005-02-21**

fixed linestyles for collections -- contour now dashes for levels <0

**2005-02-21**

fixed ps color bug - JDH

**2005-02-15**

fixed missing qt file

**2005-02-15**

banished error_msg and report_error. Internal backend methods like error_msg_gtk are preserved. backend writers, check your backends, and diff against 0.72 to make sure I did the right thing! - JDH

**2005-02-14**

Added enthought traits to matplotlib tree - JDH

---

### 0.72 (2005-02-14)

**2005-02-14**

fix bug in cbook alltrue() and onetrue() - SC

**2005-02-11**

updated qtagg backend from Ted - JDH

**2005-02-11**

matshow fixes for figure numbering, return value and docs - FP

**2005-02-09**

new zorder example for fine control in zorder_demo.py - FP

**2005-02-09**

backend renderer draw_lines now has transform in backend, as in draw_markers; use numerix in _backend_agg, added small line optimization to agg

**2005-02-09**

subplot now deletes axes that it overlaps

**2005-02-08**

Added transparent support for gzipped files in load/save - Fernando Perez (FP from now on).

**2005-02-08**

Small optimizations in PS backend. They may have a big impact for large plots, otherwise they don't hurt - FP

**2005-02-08**

Added transparent support for gzipped files in load/save - Fernando Perez (FP from now on).

**2005-02-07**

Added newstyle path drawing for markers - only implemented in agg currently - JDH

**2005-02-05**

Some superscript text optimizations for ticking log plots

**2005-02-05**

Added some default key press events to pylab figures: 'g' toggles grid - JDH

**2005-02-05**

Added some support for handling log switching for lines that have nonpos data - JDH

**2005-02-04**

Added Nadia's contour patch - contour now has matlab compatible syntax; this also fixed an unequal sized contour array bug- JDH

**2005-02-04**

Modified GTK backends to allow the FigureCanvas to be resized smaller than its original size - SC

**2005-02-02**

Fixed a bug in dates mx2num - JDH

**2005-02-02**

Incorporated Fernando's matshow - JDH

**2005-02-01**

Added Fernando's figure num patch, including experimental support for pylab backend switching, LineCOllection.color warns, savefig now a figure method, fixed a close(fig) bug - JDH

**2005-01-31**

updated datalim in contour - JDH

**2005-01-30**

Added backend_qtagg.py provided by Sigve Tjora - SC

**2005-01-28**

Added tk.inspect rc param to .matplotlibrc. IDLE users should set tk.pythoninspect:True and interactive:True and backend:TkAgg

**2005-01-28**

Replaced examples/interactive.py with an updated script from Fernando Perez - SC

**2005-01-27**

Added support for shared x or y axes. See examples/shared_axis_demo.py and examples/ganged_plots.py

**2005-01-27**

Added Lee's patch for missing symbols leq and LEFTbracket to _mathtext_data - JDH

**2005-01-26**

Added Baptiste's two scales patch -- see help(twinx) in the pylab interface for more info. See also examples/two_scales.py

**2005-01-24**

Fixed a mathtext parser bug that prevented font changes in sub/superscripts - JDH

**2005-01-24**

Fixed contour to work w/ interactive changes in colormaps, clim, etc - JDH

---

### 0.71 (2005-01-21)

**2005-01-21**

Refactored numerix to solve vexing namespace issues - JDH

**2005-01-21**

Applied Nadia's contour bug fix - JDH

**2005-01-20**

Made some changes to the contour routine - particularly region=1 seems t fix a lot of the zigzag strangeness. Added colormaps as default for contour - JDH

**2005-01-19**

Restored builtin names which were overridden (min, max, abs, round, and sum) in pylab. This is a potentially significant change for those who were relying on an array version of those functions that previously overrode builtin function names. - ADS

**2005-01-18**

Added accents to mathtext: hat, breve, grave, bar, acute, tilde, vec, dot, ddot. All of them have the same syntax, e.g., to make an overbar you do bar{o} or to make an o umlaut you do ddot{o}. The shortcuts are also provided, e.g., "o 'e `e ~n .x ^y - JDH

**2005-01-18**

Plugged image resize memory leaks - JDH

**2005-01-18**

Fixed some mathtext parser problems relating to superscripts

**2005-01-17**

Fixed a yticklabel problem for colorbars under change of clim - JDH

**2005-01-17**

Cleaned up Destroy handling in wx reducing memleak/fig from approx 800k to approx 6k- JDH

**2005-01-17**

Added kappa to latex_to_bakoma - JDH

**2005-01-15**

Support arbitrary colorbar axes and horizontal colorbars - JDH

**2005-01-15**

Fixed colormap number of colors bug so that the colorbar has the same discretization as the image - JDH

**2005-01-15**

Added Nadia's x,y contour fix - JDH

**2005-01-15**

backend_cairo: added PDF support which requires pycairo 0.1.4. Its not usable yet, but is ready for when the Cairo PDF backend matures - SC

**2005-01-15**

Added Nadia's x,y contour fix

**2005-01-12**

Fixed set clip_on bug in artist - JDH

**2005-01-11**

Reverted pythoninspect in tkagg - JDH

**2005-01-09**

Fixed a backend_bases event bug caused when an event is triggered when location is None - JDH

**2005-01-07**

Add patch from Stephen Walton to fix bug in pylab.load() when the % character is included in a comment. - ADS

**2005-01-07**

Added markerscale attribute to Legend class. This allows the marker size in the legend to be adjusted relative to that in the plot. - ADS

**2005-01-06**

Add patch from Ben Vanhaeren to make the FigureManagerGTK vbox a public attribute - SC

---

**2004-12-30**

Release 0.70

**2004-12-28**

Added coord location to key press and added a examples/picker_demo.py

**2004-12-28**

Fixed coords notification in wx toolbar - JDH

**2004-12-28**

Moved connection and disconnection event handling to the FigureCanvasBase. Backends now only need to connect one time for each of the button press, button release and key press/release functions. The base class deals with callbacks and multiple connections. This fixes flakiness on some backends (tk, wx) in the presence of multiple connections and/or disconnect - JDH

**2004-12-27**

Fixed PS mathtext bug where color was not set - Jochen please verify correct - JDH

**2004-12-27**

Added Shadow class and added shadow kwarg to legend and pie for shadow effect - JDH

**2004-12-27**

Added pie charts and new example/pie_demo.py

**2004-12-23**

Fixed an agg text rotation alignment bug, fixed some text kwarg processing bugs, and added examples/text_rotation.py to explain and demonstrate how text rotations and alignment work in matplotlib. - JDH

---

### 0.65.1 (2004-12-22)

**2004-12-22**

Fixed colorbar bug which caused colorbar not to respond to changes in colormap in some instances - JDH

**2004-12-22**

Refactored NavigationToolbar in tkagg to support app embedding , init now takes (canvas, window) rather than (canvas, figman) - JDH

**2004-12-21**

Refactored axes and subplot management - removed add_subplot and add_axes from the FigureManager. classic toolbar updates are done via an observer pattern on the figure using add_axobserver. Figure now maintains the axes stack (for gca) and supports axes deletion. Ported changes to GTK, Tk, Wx, and FLTK. Please test! Added delaxes - JDH

**2004-12-21**

Lots of image optimizations - 4x performance boost over 0.65 JDH

**2004-12-20**

Fixed a figimage bug where the axes is shown and modified tkagg to move the destroy binding into the show method.

**2004-12-18**

Minor refactoring of NavigationToolbar2 to support embedding in an application - JDH

**2004-12-14**

Added linestyle to collections (currently broken) - JDH

**2004-12-14**

Applied Nadia's setupext patch to fix libstdc++ link problem with contour and solaris -JDH

**2004-12-14**

A number of pychecker inspired fixes, including removal of True and False from cbook which I erroneously thought was needed for python2.2 - JDH

**2004-12-14**

Finished porting doc strings for set introspection. Used silent_list for many get funcs that return lists. JDH

**2004-12-13**

dates.py: removed all timezone() calls, except for UTC - SC

### 0.65 (2004-12-13)

**2004-12-13**

colors.py: rgb2hex(), hex2color() made simpler (and faster), also rgb2hex() - added round() instead of integer truncation hex2color() - changed 256.0 divisor to 255.0, so now '#ffffff' becomes (1.0,1.0,1.0) not (0.996,0.996,0.996) - SC

**2004-12-11**

Added ion and ioff to pylab interface - JDH

**2004-12-11**

backend_template.py: delete FigureCanvasTemplate.realize() - most backends don't use it and its no longer needed

backend_ps.py, backend_svg.py: delete show() and draw_if_interactive() - they are not needed for image backends

backend_svg.py: write direct to file instead of StringIO

- SC

**2004-12-10**

Added zorder to artists to control drawing order of lines, patches and text in axes. See examples/zoder_demo.py - JDH

**2004-12-10**

Fixed colorbar bug with scatter - JDH

**2004-12-10**

Added Nadia Dencheva <dencheva@stsci.edu> contour code - JDH

**2004-12-10**

backend_cairo.py: got mathtext working - SC

**2004-12-09**

Added Norm Peterson's svg clipping patch

**2004-12-09**

Added Matthew Newville's wx printing patch

**2004-12-09**

Migrated matlab to pylab - JDH

**2004-12-09**

backend_gtk.py: split into two parts

- backend_gdk.py - an image backend
- backend_gtk.py - A GUI backend that uses GDK - SC

**2004-12-08**

backend_gtk.py: remove quit_after_print_xvfb(*args), show_xvfb(), Dialog_MeasureTool(gtk.Dialog) one month after sending mail to matplotlib-users asking if anyone still uses these functions - SC

**2004-12-02**

backend_bases.py, backend_template.py: updated some of the method documentation to make them consistent with each other - SC

**2004-12-04**

Fixed multiple bindings per event for TkAgg mpl_connect and mpl_disconnect. Added a "test_disconnect" command line parameter to coords_demo.py JTM

**2004-12-04**

Fixed some legend bugs JDH

**2004-11-30**

Added over command for oneoff over plots. e.g., over(plot, x, y, lw=2). Works with any plot function.

**2004-11-30**

Added bbox property to text - JDH

**2004-11-29**

Zoom to rect now respect reversed axes limits (for both linear and log axes). - GL

**2004-11-29**

Added the over command to the matlab interface. over allows you to add an overlay plot regardless of hold state. - JDH

**2004-11-25**

Added Printf to mplutils for printf style format string formatting in C++ (should help write better exceptions)

**2004-11-24**

IMAGE_FORMAT: remove from agg and gtkagg backends as its no longer used - SC

**2004-11-23**

Added matplotlib compatible set and get introspection. See set_and_get.py

**2004-11-23**

applied Norbert's patched and exposed legend configuration to kwargs - JDH

**2004-11-23**

backend_gtk.py: added a default exception handler - SC

**2004-11-18**

backend_gtk.py: change so that the backend knows about all image formats and does not need to use IMAGE_FORMAT in other backends - SC

**2004-11-18**

Fixed some report_error bugs in string interpolation as reported on SF bug tracker- JDH

**2004-11-17**

backend_gtkcairo.py: change so all print_figure() calls render using Cairo and get saved using backend_gtk.print_figure() - SC

**2004-11-13**

backend_cairo.py: Discovered the magic number (96) required for Cairo PS plots to come out the right size. Restored Cairo PS output and added support for landscape mode - SC

**2004-11-13**

Added ishold - JDH

**2004-11-12**

Added many new matlab colormaps - autumn bone cool copper flag gray hot hsv jet pink prism spring summer winter - PG

**2004-11-11**

greatly simplify the emitted postscript code - JV

**2004-11-12**

Added new plotting functions spy, spy2 for sparse matrix visualization - JDH

**2004-11-11**

Added rgrids, thetragrids for customizing the grid locations and labels for polar plots - JDH

**2004-11-11**

make the Gtk backends build without an X-server connection - JV

**2004-11-10**

matplotlib/__init__.py: Added FROZEN to signal we are running under py2exe (or similar) - is used by backend_gtk.py - SC

**2004-11-09**

backend_gtk.py: Made fix suggested by maffew@cat.org.au to prevent problems when py2exe calls pygtk.require(). - SC

**2004-11-09**

backend_cairo.py: Added support for printing to a fileobject. Disabled cairo PS output which is not working correctly. - SC

---

### 0.64 (2004-11-08)

**2004-11-04**

Changed -dbackend processing to only use known backends, so we don't clobber other non-matplotlib uses of -d, like -debug.

**2004-11-04**

backend_agg.py: added IMAGE_FORMAT to list the formats that the backend can save to. backend_gtkagg.py: added support for saving JPG files by using the GTK backend - SC

**2004-10-31**

backend_cairo.py: now produces png and ps files (although the figure sizing needs some work). pycairo did not wrap all the necessary functions, so I wrapped them myself, they are included in the backend_cairo.py doc string. - SC

**2004-10-31**

backend_ps.py: clean up the generated PostScript code, use the PostScript stack to hold intermediate values instead of storing them in the dictionary. - JV

**2004-10-30**

backend_ps.py, ft2font.cpp, ft2font.h: fix the position of text in the PostScript output. The new FT2Font method get_descent gives the distance between the lower edge of the bounding box and the baseline of a string. In backend_ps the text is shifted upwards by this amount. - JV

**2004-10-30**

backend_ps.py: clean up the code a lot. Change the PostScript output to be more DSC compliant. All definitions for the generated PostScript are now in a PostScript dictionary 'mpldict'. Moved the long comment about drawing ellipses from the PostScript output into a Python comment. - JV

**2004-10-30**

backend_gtk.py: removed FigureCanvasGTK.realize() as its no longer needed. Merged ColorManager into GraphicsContext backend_bases.py: For set_capstyle/joinstyle() only set cap or joinstyle if there is no error. - SC

**2004-10-30**

backend_gtk.py: tidied up print_figure() and removed some of the dependency on widget events - SC

**2004-10-28**

backend_cairo.py: The renderer is complete except for mathtext, draw_image() and clipping. gtkcairo works reasonably well. cairo does not yet create any files since I can't figure how to set the 'target surface', I don't think pycairo wraps the required functions - SC

**2004-10-28**

backend_gtk.py: Improved the save dialog (GTK 2.4 only) so it presents the user with a menu of supported image formats - SC

**2004-10-28**

backend_svg.py: change print_figure() to restore original face/edge color backend_ps.py : change print_figure() to ensure original face/edge colors are restored even if there's an IOError - SC

**2004-10-27**

Applied Norbert's errorbar patch to support barsabove kwarg

**2004-10-27**

Applied Norbert's legend patch to support None handles

**2004-10-27**

Added two more backends: backend_cairo.py, backend_gtkcairo.py They are not complete yet, currently backend_gtkcairo just renders polygons, rectangles and lines - SC

**2004-10-21**

Added polar axes and plots - JDH

**2004-10-20**

Fixed corrcoef bug exposed by corrcoef(X) where X is matrix - JDH

**2004-10-19**

Added kwarg support to xticks and yticks to set ticklabel text properties -- thanks to T. Edward Whalen for the suggestion

**2004-10-19**

Added support for PIL images in imshow(), image.py - ADS

**2004-10-19**

Re-worked exception handling in _image.py and _transforms.py to avoid masking problems with shared libraries. - JTM

**2004-10-16**

Streamlined the matlab interface wrapper, removed the noplot option to hist - just use mlab.hist instead.

**2004-09-30**

Added Andrew Dalke's strftime code to extend the range of dates supported by the DateFormatter - JDH

**2004-09-30**

Added barh - JDH

**2004-09-30**

Removed fallback to alternate array package from numerix so that ImportErrors are easier to debug. - JTM

**2004-09-30**

Add GTK+ 2.4 support for the message in the toolbar. SC

**2004-09-30**

Made some changes to support python22 - lots of doc fixes. - JDH

**2004-09-29**

Added a Verbose class for reporting - JDH

---

**2004-09-28**

Released 0.63.0

**2004-09-28**

Added save to file object for agg - see examples/print_stdout.py

**2004-09-24**

Reorganized all py code to lib subdir

**2004-09-24**

Fixed axes resize image edge effects on interpolation - required upgrade to agg22 which fixed an agg bug related to this problem

**2004-09-20**

Added toolbar2 message display for backend_tkagg. JTM

**2004-09-17**

Added coords formatter attributes. These must be callable, and return a string for the x or y data. These will be used to format the x and y data for the coords box. Default is the axis major formatter. e.g.:

```
# format the coords message box
def price(x): return '$%1.2f'%x
ax.format_xdata = DateFormatter('%Y-%m-%d')
ax.format_ydata = price
```

**2004-09-17**

Total rewrite of dates handling to use python datetime with num2date, date2num and drange. pytz for timezone handling, dateutils for spohisticated ticking. date ranges from 0001-9999 are supported. rrules allow arbitrary date ticking. examples/date_demo*.py converted to show new usage. new example examples/date_demo_rrule.py shows how to use rrules in date plots. The date locators are much more general and almost all of them have different constructors. See matplotlib.dates for more info.

**2004-09-15**

Applied Fernando's backend __init__ patch to support easier backend maintenance. Added his numutils to mlab. JDH

**2004-09-16**

Re-designated all files in matplotlib/images as binary and w/o keyword substitution using "cvs admin -kb *.svg ...". See binary files in "info cvs" under Linux. This was messing up builds from CVS on windows since CVS was doing lf -> cr/lf and keyword substitution on the bitmaps. - JTM

**2004-09-15**

Modified setup to build array-package-specific extensions for those extensions which are array-aware. Setup builds extensions automatically for either Numeric, numarray, or both, depending on what you have installed. Python proxy modules for the array-aware extensions import the version optimized for numarray or Numeric determined by numerix. - JTM

**2004-09-15**

Moved definitions of infinity from mlab to numerix to avoid divide by zero warnings for numarray - JTM

**2004-09-09**

Added axhline, axvline, axhspan and axvspan

---

### 0.62.4 (2004-08-30)

**2004-08-30**

Fixed a multiple images with different extent bug, Fixed markerfacecolor as RGB tuple

**2004-08-27**

Mathtext now more than 5x faster. Thanks to Paul Mcguire for fixes both to pyparsing and to the matplotlib grammar! mathtext broken on python2.2

**2004-08-25**

Exposed Darren's and Greg's log ticking and formatting options to semilogx and friends

**2004-08-23**

Fixed grid w/o args to toggle grid state - JDH

**2004-08-11**

Added Gregory's log patches for major and minor ticking

**2004-08-18**

Some pixel edge effects fixes for images

**2004-08-18**

Fixed TTF files reads in backend_ps on win32.

**2004-08-18**

Added base and subs properties for logscale plots, user modifiable using set_[x,y]scale('log',base=b,subs=[mt1,mt2,...]) - GL

**2004-08-18**

fixed a bug exposed by trying to find the HOME dir on win32 thanks to Alan Issac for pointing to the light - JDH

**2004-08-18**

fixed errorbar bug in setting ecolor - JDH

**2004-08-12**

Added Darren Dale's exponential ticking patch

**2004-08-11**

Added Gregory's fltkagg backend

---

### 0.61.0 (2004-08-09)

**2004-08-08**

backend_gtk.py: get rid of the final PyGTK deprecation warning by replacing gtkOptionMenu with gtkMenu in the 2.4 version of the classic toolbar.

**2004-08-06**

Added Tk zoom to rect rectangle, proper idle drawing, and keybinding - JDH

**2004-08-05**

Updated installing.html and INSTALL - JDH

**2004-08-01**

backend_gtk.py: move all drawing code into the expose_event()

**2004-07-28**

Added Greg's toolbar2 and backend_*agg patches - JDH

**2004-07-28**

Added image.imread with support for loading png into numerix arrays

**2004-07-28**

Added key modifiers to events - implemented dynamic updates and rubber banding for interactive pan/zoom - JDH

**2004-07-27**

did a readthrough of SVG, replacing all the string additions with string interps for efficiency, fixed some layout problems, added font and image support (through external pngs) - JDH

**2004-07-25**

backend_gtk.py: modify toolbar2 to make it easier to support GTK+ 2.4. Add GTK+ 2.4 toolbar support. - SC

**2004-07-24**

backend_gtk.py: Simplified classic toolbar creation - SC

**2004-07-24**

Added images/matplotlib.svg to be used when GTK+ windows are minimised - SC

**2004-07-22**

Added right mouse click zoom for NavigationToolbar2 panning mode. - JTM

**2004-07-22**

Added NavigationToolbar2 support to backend_tkagg. Minor tweak to backend_bases. - JTM

**2004-07-22**

Incorporated Gergory's renderer cache and buffer object cache - JDH

**2004-07-22**

Backend_gtk.py: Added support for GtkFileChooser, changed FileSelection/FileChooser so that only one instance pops up, and made them both modal. - SC

**2004-07-21**

Applied backend_agg memory leak patch from hayden - jocallo@online.no. Found and fixed a leak in binary operations on transforms. Moral of the story: never incref where you meant to decref! Fixed several leaks in ft2font: moral of story: almost always return Py::asObject over Py::Object - JDH

**2004-07-21**

Fixed a to string memory allocation bug in agg and image modules - JDH

**2004-07-21**

Added mpl_connect and mpl_disconnect to matlab interface - JDH

**2004-07-21**

Added beginnings of users_guide to CVS - JDH

**2004-07-20**

ported toolbar2 to wx

**2004-07-20**

upgraded to agg21 - JDH

**2004-07-20**

Added new icons for toolbar2 - JDH

**2004-07-19**

Added vertical mathtext for *Agg and GTK - thanks Jim Benson! - JDH

**2004-07-16**

Added ps/eps/svg savefig options to wx and gtk JDH

**2004-07-15**

Fixed python framework tk finder in setupext.py - JDH

**2004-07-14**

Fixed layer images demo which was broken by the 07/12 image extent fixes - JDH

**2004-07-13**

Modified line collections to handle arbitrary length segments for each line segment. - JDH

**2004-07-13**

Fixed problems with image extent and origin - set_image_extent deprecated. Use imshow(blah, blah, extent=(xmin, xmax, ymin, ymax) instead - JDH

**2004-07-12**

Added prototype for new nav bar with codified event handling. Use mpl_connect rather than connect for matplotlib event handling. toolbar style determined by rc toolbar param. backend status: gtk: prototype, wx: in progress, tk: not started - JDH

**2004-07-11**

backend_gtk.py: use builtin round() instead of redefining it. - SC

**2004-07-10**

Added embedding_in_wx3 example - ADS

**2004-07-09**

Added dynamic_image_wxagg to examples - ADS

**2004-07-09**

added support for embedding TrueType fonts in PS files - PEB

**2004-07-09**

fixed a sfnt bug exposed if font cache is not built

**2004-07-09**

added default arg None to matplotlib.matlab grid command to toggle current grid state

---

### 0.60.2 (2004-07-08)

**2004-07-08**

fixed a mathtext bug for '6'

**2004-07-08**

added some numarray bug workarounds

---

### 0.60 (2004-07-07)

**2004-07-07**

Fixed a bug in dynamic_demo_wx

**2004-07-07**

backend_gtk.py: raise SystemExit immediately if 'import pygtk' fails - SC

**2004-07-05**

Added new mathtext commands over{sym1}{sym2} and under{sym1}{sym2}

**2004-07-05**

Unified image and patch collections colormapping and scaling args. Updated docstrings for all - JDH

**2004-07-05**

Fixed a figure legend bug and added examples/figlegend_demo.py - JDH

**2004-07-01**

Fixed a memory leak in image and agg to string methods

**2004-06-25**

Fixed fonts_demo spacing problems and added a kwargs version of the fonts_demo fonts_demo_kw.py - JDH

**2004-06-25**

finance.py: handle case when urlopen() fails - SC

**2004-06-24**

Support for multiple images on axes and figure, with blending. Support for upper and lower image origins. clim, jet and gray functions in matlab interface operate on current image - JDH

**2004-06-23**

ported code to Perry's new colormap and norm scheme. Added new rc attributes image.aspect, image.interpolation, image.cmap, image.lut, image.origin

**2004-06-20**

backend_gtk.py: replace gtk.TRUE/FALSE with True/False. simplified _make_axis_menu(). - SC

**2004-06-19**

anim_tk.py: Updated to use TkAgg by default (not GTK) backend_gtk_py: Added '_' in front of private widget creation functions - SC

**2004-06-17**

backend_gtk.py: Create a GC once in realise(), not every time draw() is called. - SC

**2004-06-16**

Added new py2exe FAQ entry and added frozen support in get_data_path for py2exe - JDH

**2004-06-16**

Removed GTKGD, which was always just a proof-of-concept backend - JDH

**2004-06-16**

backend_gtk.py updates to replace deprecated functions gtk.mainquit(), gtk.mainloop(). Update NavigationToolbar to use the new GtkToolbar API - SC

**2004-06-15**

removed set_default_font from font_manager to unify font customization using the new function rc. See API_CHANGES for more info. The examples fonts_demo.py and fonts_demo_kw.py are ported to the new API - JDH

**2004-06-15**

Improved (yet again!) axis scaling to properly handle singleton plots - JDH

**2004-06-15**

Restored the old FigureCanvasGTK.draw() - SC

**2004-06-11**

More memory leak fixes in transforms and ft2font - JDH

**2004-06-11**

Eliminated numerix .numerix file and environment variable NUMERIX. Fixed bug which prevented command line overrides: --numarray or --numeric. - JTM

**2004-06-10**

Added rc configuration function rc; deferred all rc param setting until object creation time; added new rc attrs: lines.markerfacecolor, lines.markeredgecolor, lines.markeredgewidth, patch.linewidth, patch.facecolor, patch.edgecolor, patch.antialiased; see examples/customize_rc.py for usage - JDH

---

### 0.54.2 (2004-06-09)

**2004-06-08**

Rewrote ft2font using CXX as part of general memory leak fixes; also fixed transform memory leaks - JDH

**2004-06-07**

Fixed several problems with log ticks and scaling - JDH

**2004-06-07**

Fixed width/height issues for images - JDH

**2004-06-03**

Fixed draw_if_interactive bug for semilogx;

**2004-06-02**

Fixed text clipping to clip to axes - JDH

**2004-06-02**

Fixed leading newline text and multiple newline text - JDH

**2004-06-02**

Fixed plot_date to return lines - JDH

**2004-06-01**

Fixed plot to work with x or y having shape N,1 or 1,N - JDH

**2004-05-31**

Added renderer markeredgewidth attribute of Line2D. - ADS

**2004-05-29**

Fixed tick label clipping to work with navigation.

**2004-05-28**

**Added renderer grouping commands to support groups in**

SVG/PS. - JDH

**2004-05-28**

Fixed, this time I really mean it, the singleton plot plot([0]) scaling bug; Fixed Flavio's shape = N,1 bug - JDH

**2004-05-28**

added colorbar - JDH

**2004-05-28**

Made some changes to the matplotlib.colors.Colormap to properly support clim - JDH

---

### 0.54.1 (2004-05-27)

**2004-05-27**

Lots of small bug fixes: rotated text at negative angles, errorbar capsize and autoscaling, right tick label position, gtkagg on win98, alpha of figure background, singleton plots - JDH

**2004-05-26**

Added Gary's errorbar stuff and made some fixes for length one plots and constant data plots - JDH

**2004-05-25**

Tweaked TkAgg backend so that canvas.draw() works more like the other backends. Fixed a bug resulting in 2 draws per figure manager show(). - JTM

### 0.54 (2004-05-19)

**2004-05-18**

Added newline separated text with rotations to text.Text layout - JDH

**2004-05-16**

Added fast pcolor using PolyCollections. - JDH

**2004-05-14**

Added fast polygon collections - changed scatter to use them. Added multiple symbols to scatter. 10x speedup on large scatters using *Agg and 5X speedup for ps. - JDH

**2004-05-14**

On second thought... created an "nx" namespace in numerix which maps type names onto typecodes the same way for both numarray and Numeric. This undoes my previous change immediately below. To get a typename for Int16 usable in a Numeric extension: say nx.Int16. - JTM

**2004-05-15**

Rewrote transformation class in extension code, simplified all the artist constructors - JDH

**2004-05-14**

Modified the type definitions in the numarray side of numerix so that they are Numeric typecodes and can be used with Numeric compilex extensions. The original numarray types were renamed to type<old_name>. - JTM

**2004-05-06**

Gary Ruben sent me a bevy of new plot symbols and markers. See matplotlib.matlab.plot - JDH

**2004-05-06**

Total rewrite of mathtext - factored ft2font stuff out of layout engine and defined abstract class for font handling to lay groundwork for ps mathtext. Rewrote parser and made layout engine much more precise. Fixed all the layout hacks. Added spacing commands / and hspace. Added composite chars and defined angstrom. - JDH

**2004-05-05**

Refactored text instances out of backend; aligned text with arbitrary rotations is now supported - JDH

**2004-05-05**

Added a Matrix capability for numarray to numerix. JTM

**2004-05-04**

Updated whats_new.html.template to use dictionary and template loop, added anchors for all versions and items; updated goals.txt to use those for links. PG

**2004-05-04**

Added fonts_demo.py to backend_driver, and AFM and TTF font caches to font_manager.py - PEB

**2004-05-03**

Redid goals.html.template to use a goals.txt file that has a pseudo restructured text organization. PG

**2004-05-03**

Removed the close buttons on all GUIs and added the python #! bang line to the examples following Steve Chaplin's advice on matplotlib dev

**2004-04-29**

Added CXX and rewrote backend_agg using it; tracked down and fixed agg memory leak - JDH

**2004-04-29**

Added stem plot command - JDH

**2004-04-28**

Fixed PS scaling and centering bug - JDH

**2004-04-26**

Fixed errorbar autoscale problem - JDH

**2004-04-22**

Fixed copy tick attribute bug, fixed singular datalim ticker bug; fixed mathtext fontsize interactive bug. - JDH

**2004-04-21**

Added calls to draw_if_interactive to axes(), legend(), and pcolor(). Deleted duplicate pcolor(). - JTM

---

**2004-04-21**

matplotlib 0.53 release

**2004-04-19**

Fixed vertical alignment bug in PS backend - JDH

**2004-04-17**

Added support for two scales on the "same axes" with tick different ticking and labeling left right or top bottom. See examples/two_scales.py - JDH

**2004-04-17**

Added default dirs as list rather than single dir in setupext.py - JDH

**2004-04-16**

Fixed wx exception swallowing bug (and there was much rejoicing!) - JDH

**2004-04-16**

Added new ticker locator a formatter, fixed default font return - JDH

**2004-04-16**

Added get_name method to FontProperties class. Fixed font lookup in GTK and WX backends. - PEB

**2004-04-16**

Added get- and set_fontstyle methods. - PEB

**2004-04-10**

Mathtext fixes: scaling with dpi, - JDH

**2004-04-09**

Improved font detection algorithm. - PEB

**2004-04-09**

Move deprecation warnings from text.py to __init__.py - PEB

**2004-04-09**

Added default font customization - JDH

**2004-04-08**

Fixed viewlim set problem on axes and axis. - JDH

**2004-04-07**

Added validate_comma_sep_str and font properties parameters to __init__. Removed font families and added rcParams to FontProperties __init__ arguments in font_manager. Added default font property parameters to .matplotlibrc file with descriptions. Added deprecation warnings to the get_ - and set_fontXXX methods of the Text object. - PEB

**2004-04-06**

Added load and save commands for ASCII data - JDH

**2004-04-05**

Improved font caching by not reading AFM fonts until needed. Added better documentation. Changed the behaviour of the get_family, set_family, and set_name methods of FontProperties. - PEB

**2004-04-05**

Added WXAgg backend - JDH

**2004-04-04**

Improved font caching in backend_agg with changes to font_manager - JDH

**2004-03-29**

Fixed fontdicts and kwargs to work with new font manager - JDH

This is the Old, stale, never used changelog

**2002-12-10**

- Added a TODO file and CHANGELOG. Lots to do -- get crackin'!
- Fixed y zoom tool bug
- Adopted a compromise fix for the y data clipping problem. The problem was that for solid lines, the y data clipping (as opposed to the gc clipping) caused artifactual horizontal solid lines near the ylim boundaries. I did a 5% offset hack in Axes set_ylim functions which helped, but didn't cure the problem for very high gain y zooms. So I disabled y data clipping for connected lines . If you need extensive y clipping, either plot(y,x) because x data clipping is always enabled, or change the _set_clip code to 'if 1' as indicated in the lines.py src. See _set_clip in lines.py and set_ylim in figure.py for more information.

**2002-12-11**

- Added a measurement dialog to the figure window to measure axes position and the delta x delta y with a left mouse drag. These defaults can be overridden by deriving from Figure and overriding button_press_event, button_release_event, and motion_notify_event, and _dialog_measure_tool.
- fixed the navigation dialog so you can check the axes the navigation buttons apply to.

**2003-04-23**

Released matplotlib v0.1

**2003-04-24**

Added a new line style PixelLine2D which is the plots the markers as pixels (as small as possible) with format symbol ','

Added a new class Patch with derived classes Rectangle, RegularPolygon and Circle

**2003-04-25**

Implemented new functions errorbar, scatter and hist

Added a new line type '|' which is a vline. syntax is plot(x, Y, '|') where y.shape = len(x),2 and each row gives the ymin,ymax for the respective values of x. Previously I had implemented vlines as a list of lines, but I needed the efficiency of the numeric clipping for large numbers of vlines outside the viewport, so I wrote a dedicated class Vline2D which derives from Line2D

**2003-05-01**

Fixed ytick bug where grid and tick show outside axis viewport with gc clip

**2003-05-14**

Added new ways to specify colors 1) matlab format string 2) html-style hex string, 3) rgb tuple. See examples/color_demo.py

**2003-05-28**

Changed figure rendering to draw form a pixmap to reduce flicker. See examples/system_monitor.py for an example where the plot is continuously updated w/o flicker. This example is meant to simulate a system monitor that shows free CPU, RAM, etc...

**2003-08-04**

Added Jon Anderson's GTK shell, which doesn't require pygtk to have threading built-in and looks nice!

**2003-08-25**

Fixed deprecation warnings for python2.3 and pygtk-1.99.18

**2003-08-26**

Added figure text with new example examples/figtext.py

**2003-08-27**

Fixed bugs in figure text with font override dictionaries and fig text that was placed outside the window bounding box

**2003-09-01 through 2003-09-15**

Added a postscript and a GD module backend

**2003-09-16**

Fixed font scaling and point scaling so circles, squares, etc on lines will scale with DPI as will fonts. Font scaling is not fully implemented on the gtk backend because I have not figured out how to scale fonts to arbitrary sizes with GTK

**2003-09-17**

Fixed figure text bug which crashed X windows on long figure text extending beyond display area. This was, I believe, due to the vestigial erase functionality that was no longer needed since I began rendering to a pixmap

**2003-09-30**

Added legend

**2003-10-01**

Fixed bug when colors are specified with rgb tuple or hex string.

**2003-10-21**

Andrew Straw provided some legend code which I modified and incorporated. Thanks Andrew!

**2003-10-27**

Fixed a bug in axis.get_view_distance that affected zoom in versus out with interactive scrolling, and a bug in the axis text reset system that prevented the text from being redrawn on a interactive gtk view lim set with the widget

Fixed a bug in that prevented the manual setting of ticklabel strings from working properly

**2003-11-02**

- Do a nearest neighbor color pick on GD when allocate fails

**2003-11-02**

- Added pcolor plot
- Added MRI example
- Fixed bug that screwed up label position if xticks or yticks were empty
- added nearest neighbor color picker when GD max colors exceeded
- fixed figure background color bug in GD backend

**2003-11-10 - 2003-11-11**

major refactoring.

- Ticks (with labels, lines and grid) handled by dedicated class
- Artist now know bounding box and dpi
- Bounding boxes and transforms handled by dedicated classes
- legend in dedicated class. Does a better job of alignment and bordering. Can be initialized with specific line instances. See examples/legend_demo2.py

**2003-11-14**

Fixed legend positioning bug and added new position args

**2003-11-16**

Finished porting GD to new axes API

**2003-11-20**

- add TM for matlab on website and in docs

**2003-11-20**

- make a nice errorbar and scatter screenshot

**2003-11-20**

- auto line style cycling for multiple line types broken

**2003-11-18**

(using inkrect) :logical rect too big on gtk backend

**2003-11-18**

ticks don't reach edge of axes in gtk mode -- rounding error?

**2003-11-20**

- port Gary's errorbar code to new API before 0.40

**2003-11-20**

- problem with stale _set_font. legend axes box doesn't resize on save in GTK backend -- see htdocs legend_demo.py

**2003-11-21**

- make a dash-dot dict for the GC

**2003-12-15**

- fix install path bug

### 9.20.2 What's new in Matplotlib 0.99 (Aug 29, 2009)

**Table of Contents**


- *What's new in Matplotlib 0.99 (Aug 29, 2009)*
  - *New documentation*
  - *mplot3d*
  - *axes grid toolkit*
  - *Axis spine placement*


#### New documentation

Jae-Joon Lee has written two new guides *Legend guide* and *Advanced annotation*. Michael Sarahan has written *Image tutorial*. John Hunter has written two new tutorials on working with paths and transformations: *Path Tutorial* and *Transformations Tutorial*.

#### mplot3d

Reinier Heeres has ported John Porter's mplot3d over to the new matplotlib transformations framework, and it is now available as a toolkit mpl_toolkits.mplot3d (which now comes standard with all mpl installs). See *3D plotting* and *The mplot3d toolkit*.

#### axes grid toolkit

Jae-Joon Lee has added a new toolkit to ease displaying multiple images in matplotlib, as well as some support for curvilinear grids to support the world coordinate system. The toolkit is included standard with all new mpl installs. See *Module - axes_grid1*, *Module - axisartist*, *The axes_grid1 toolkit* and *axisartist*

#### Axis spine placement

Andrew Straw has added the ability to place "axis spines" -- the lines that denote the data limits -- in various arbitrary locations. No longer are your axis lines constrained to be a simple rectangle around the figure -- you can turn on or off left, bottom, right and top, as well as "detach" the spine to offset it away from the data. See *Spine placement* and `matplotlib.spines.Spine`.

### 9.20.3 Changes beyond 0.99.x

- The default behavior of `matplotlib.axes.Axes.set_xlim()`, `matplotlib.axes.Axes.set_ylim()`, and `matplotlib.axes.Axes.axis()`, and their corresponding pyplot functions, has been changed: when view limits are set explicitly with one of these methods, autoscaling is turned off for the matching axis. A new *auto* kwarg is available to control this behavior. The limit kwargs have been renamed to *left* and *right* instead of *xmin* and *xmax*, and *bottom* and *top* instead of *ymin* and *ymax*. The old names may still be used, however.
- There are five new Axes methods with corresponding pyplot functions to facilitate autoscaling, tick location, and tick label formatting, and the general appearance of ticks and tick labels:
  - `matplotlib.axes.Axes.autoscale()` turns autoscaling on or off, and applies it.
  - `matplotlib.axes.Axes.margins()` sets margins used to autoscale the `matplotlib.axes.Axes.viewLim` based on the `matplotlib.axes.Axes.dataLim`.
  - `matplotlib.axes.Axes.locator_params()` allows one to adjust axes locator parameters such as *nbins*.
  - `matplotlib.axes.Axes.ticklabel_format()` is a convenience method for controlling the `matplotlib.ticker.ScalarFormatter` that is used by default with linear axes.
  - `matplotlib.axes.Axes.tick_params()` controls direction, size, visibility, and color of ticks and their labels.
- The `matplotlib.axes.Axes.bar()` method accepts a *error_kw* kwarg; it is a dictionary of kwargs to be passed to the errorbar function.
- The `matplotlib.axes.Axes.hist()` *color* kwarg now accepts a sequence of color specs to match a sequence of datasets.
- The `EllipseCollection` has been changed in two ways:
  - There is a new *units* option, 'xy', that scales the ellipse with the data units. This matches the :class:'~matplotlib.patches.Ellipse` scaling.
  - The *height* and *width* kwargs have been changed to specify the height and width, again for consistency with `Ellipse`, and to better match their names; previously they specified the half-height and half-width.
- There is a new rc parameter `axes.color_cycle`, and the color cycle is now independent of the rc parameter `lines.color`. `matplotlib.Axes.set_default_color_cycle` is deprecated.

- You can now print several figures to one pdf file and modify the document information dictionary of a pdf file. See the docstrings of the class *matplotlib.backends.backend_pdf.PdfPages* for more information.
- Removed configobj and enthought.traits packages, which are only required by the experimental traited config and are somewhat out of date. If needed, install them independently.
- The new rc parameter `savefig.extension` sets the filename extension that is used by *matplotlib.figure.Figure.savefig()* if its *fname* argument lacks an extension.
- In an effort to simplify the backend API, all clipping rectangles and paths are now passed in using GraphicsContext objects, even on collections and images. Therefore:

```
draw_path_collection(self, master_transform, cliprect, clippath,
                     clippath_trans, paths, all_transforms, offsets,
                     offsetTrans, facecolors, edgecolors, linewidths,
                     linestyles, antialiaseds, urls)

# is now

draw_path_collection(self, gc, master_transform, paths, all_transforms,
                     offsets, offsetTrans, facecolors, edgecolors,
                     linewidths, linestyles, antialiaseds, urls)


draw_quad_mesh(self, master_transform, cliprect, clippath,
               clippath_trans, meshWidth, meshHeight, coordinates,
               offsets, offsetTrans, facecolors, antialiased,
               showedges)

# is now

draw_quad_mesh(self, gc, master_transform, meshWidth, meshHeight,
               coordinates, offsets, offsetTrans, facecolors,
               antialiased, showedges)


draw_image(self, x, y, im, bbox, clippath=None, clippath_trans=None)

# is now

draw_image(self, gc, x, y, im)
```

- There are four new Axes methods with corresponding pyplot functions that deal with unstructured triangular grids:
  - *matplotlib.axes.Axes.tricontour()* draws contour lines on a triangular grid.
  - *matplotlib.axes.Axes.tricontourf()* draws filled contours on a triangular grid.
  - *matplotlib.axes.Axes.tripcolor()* draws a pseudocolor plot on a triangular grid.
  - *matplotlib.axes.Axes.triplot()* draws a triangular grid as lines and/or markers.

### 9.20.4 Changes in 0.99

- pylab no longer provides a load and save function. These are available in matplotlib.mlab, or you can use numpy.loadtxt and numpy.savetxt for text files, or np.save and np.load for binary NumPy arrays.
- User-generated colormaps can now be added to the set recognized by `matplotlib.cm.get_cmap`. Colormaps can be made the default and applied to the current image using *`matplotlib.pyplot.set_cmap()`*.
- changed use_mrecords default to False in mlab.csv2rec since this is partially broken
- Axes instances no longer have a "frame" attribute. Instead, use the new "spines" attribute. Spines is a dictionary where the keys are the names of the spines (e.g., 'left','right' and so on) and the values are the artists that draw the spines. For normal (rectilinear) axes, these artists are Line2D instances. For other axes (such as polar axes), these artists may be Patch instances.
- Polar plots no longer accept a resolution kwarg. Instead, each Path must specify its own number of interpolation steps. This is unlikely to be a user-visible change -- if interpolation of data is required, that should be done before passing it to Matplotlib.

### 9.20.5 What's new in Matplotlib 0.98.4

**Table of Contents**


- *What's new in Matplotlib 0.98.4*
  - *Legend enhancements*
  - *Fancy annotations and arrows*
  - *Native OS X backend*
  - *psd amplitude scaling*
  - *Fill between*
  - *Lots more*


It's been four months since the last matplotlib release, and there are a lot of new features and bug-fixes.

Thanks to Charlie Moad for testing and preparing the source release, including binaries for OS X and Windows for python 2.4 and 2.5 (2.6 and 3.0 will not be available until numpy is available on those releases). Thanks to the many developers who contributed to this release, with contributions from Jae-Joon Lee, Michael Droettboom, Ryan May, Eric Firing, Manuel Metz, Jouni K. Seppänen, Jeff Whitaker, Darren Dale, David Kaplan, Michiel de Hoon and many others who submitted patches

### Legend enhancements

Jae-Joon has rewritten the legend class, and added support for multiple columns and rows, as well as fancy box drawing. See *legend()* and *matplotlib.legend.Legend*.

### Fancy annotations and arrows

Jae-Joon has added lots of support to annotations for drawing fancy boxes and connectors in annotations. See *annotate()* and *BoxStyle*, *ArrowStyle*, and *ConnectionStyle*.

### Native OS X backend

Michiel de Hoon has provided a native Mac OSX backend that is almost completely implemented in C. The backend can therefore use Quartz directly and, depending on the application, can be orders of magnitude faster than the existing backends. In addition, no third-party libraries are needed other than Python and NumPy. The backend is interactive from the usual terminal application on Mac using regular Python. It hasn't been tested with ipython yet, but in principle it should to work there as well. Set 'backend : macosx' in your matplotlibrc file, or run your script with:

```
> python myfile.py -dmacosx
```

### psd amplitude scaling

Ryan May did a lot of work to rationalize the amplitude scaling of *psd()* and friends. See *Power spectral density (PSD)*. The changes should increase MATLAB compatibility and increase scaling options.

### Fill between

Added a *fill_between()* function to make it easier to do shaded region plots in the presence of masked data. You can pass an *x* array and a *ylower* and *yupper* array to fill between, and an optional *where* argument which is a logical mask where you want to do the filling.

### Lots more

Here are the 0.98.4 notes from the CHANGELOG:

```
Added mdehoon's native macosx backend from sf patch 2179017 - JDH

Removed the prints in the set_*style commands.  Return the list of
pretty-printed strings instead - JDH
```

(continues on next page)

(continued from previous page)

```
Some of the changes Michael made to improve the output of the
property tables in the rest docs broke of made difficult to use
some of the interactive doc helpers, e.g., setp and getp.  Having all
the rest markup in the ipython shell also confused the docstrings.
I added a new rc param docstring.harcopy, to format the docstrings
differently for hardcopy and other use.  The ArtistInspector
could use a little refactoring now since there is duplication of
effort between the rest out put and the non-rest output - JDH

Updated spectral methods (psd, csd, etc.) to scale one-sided
densities by a factor of 2 and, optionally, scale all densities by
the sampling frequency.  This gives better MATLAB
compatibility. -RM

Fixed alignment of ticks in colorbars. -MGD

drop the deprecated "new" keyword of np.histogram() for numpy 1.2
or later.  -JJL

Fixed a bug in svg backend that new_figure_manager() ignores
keywords arguments such as figsize, etc. -JJL

Fixed a bug that the handlelength of the new legend class set too
short when numpoints=1 -JJL

Added support for data with units (e.g., dates) to
Axes.fill_between. -RM

Added fancybox keyword to legend. Also applied some changes for
better look, including baseline adjustment of the multiline texts
so that it is center aligned. -JJL

The transmuter classes in the patches.py are reorganized as
subclasses of the Style classes. A few more box and arrow styles
are added. -JJL

Fixed a bug in the new legend class that didn't allowed a tuple of
coordinate values as loc. -JJL

Improve checks for external dependencies, using subprocess
(instead of deprecated popen*) and distutils (for version
checking) - DSD

Reimplementation of the legend which supports baseline alignment,
multi-column, and expand mode. - JJL

Fixed histogram autoscaling bug when bins or range are given
explicitly (fixes Debian bug 503148) - MM

Added rcParam axes.unicode_minus which allows plain hyphen for
minus when False - JDH
```

(continues on next page)

(continued from previous page)

```
Added scatterpoints support in Legend. patch by Erik Tollerud -
JJL

Fix crash in log ticking. - MGD

Added static helper method BrokenHBarCollection.span_where and
Axes/pyplot method fill_between.  See
examples/pylab/fill_between.py - JDH

Add x_isdata and y_isdata attributes to Artist instances, and use
them to determine whether either or both coordinates are used when
updating dataLim.  This is used to fix autoscaling problems that
had been triggered by axhline, axhspan, axvline, axvspan. - EF

Update the psd(), csd(), cohere(), and specgram() methods of Axes
and the csd() cohere(), and specgram() functions in mlab to be in
sync with the changes to psd().  In fact, under the hood, these
all call the same core to do computations. - RM

Add 'pad_to' and 'sides' parameters to mlab.psd() to allow
controlling of zero padding and returning of negative frequency
components, respectively.  These are added in a way that does not
change the API. - RM

Fix handling of c kwarg by scatter; generalize is_string_like to
accept numpy and numpy.ma string array scalars. - RM and EF

Fix a possible EINTR problem in dviread, which might help when
saving pdf files from the qt backend. - JKS

Fix bug with zoom to rectangle and twin axes - MGD

Added Jae Joon's fancy arrow, box and annotation enhancements --
see examples/pylab_examples/annotation_demo2.py

Autoscaling is now supported with shared axes - EF

Fixed exception in dviread that happened with Minion - JKS

set_xlim, ylim now return a copy of the viewlim array to avoid
modify inplace surprises

Added image thumbnail generating function
matplotlib.image.thumbnail.  See examples/misc/image_thumbnail.py
- JDH

Applied scatleg patch based on ideas and work by Erik Tollerud and
Jae-Joon Lee. - MM

Fixed bug in pdf backend: if you pass a file object for output
instead of a filename, e.g., in a wep app, we now flush the object
```

(continues on next page)

(continued from previous page)

```
at the end. - JKS

Add path simplification support to paths with gaps. - EF

Fix problem with AFM files that don't specify the font's full name
or family name. - JKS

Added 'scilimits' kwarg to Axes.ticklabel_format() method, for
easy access to the set_powerlimits method of the major
ScalarFormatter. - EF

Experimental new kwarg borderpad to replace pad in legend, based
on suggestion by Jae-Joon Lee.  - EF

Allow spy to ignore zero values in sparse arrays, based on patch
by Tony Yu.  Also fixed plot to handle empty data arrays, and
fixed handling of markers in figlegend. - EF

Introduce drawstyles for lines. Transparently split linestyles
like 'steps--' into drawstyle 'steps' and linestyle '--'.  Legends
always use drawstyle 'default'. - MM

Fixed quiver and quiverkey bugs (failure to scale properly when
resizing) and added additional methods for determining the arrow
angles - EF

Fix polar interpolation to handle negative values of theta - MGD

Reorganized cbook and mlab methods related to numerical
calculations that have little to do with the goals of those two
modules into a separate module numerical_methods.py Also, added
ability to select points and stop point selection with keyboard in
ginput and manual contour labeling code.  Finally, fixed contour
labeling bug. - DMK

Fix backtick in Postscript output. - MGD

[ 2089958 ] Path simplification for vector output backends
Leverage the simplification code exposed through path_to_polygons
to simplify certain well-behaved paths in the vector backends
(PDF, PS and SVG).  "path.simplify" must be set to True in
matplotlibrc for this to work.  - MGD

Add "filled" kwarg to Path.intersects_path and
Path.intersects_bbox. - MGD

Changed full arrows slightly to avoid an xpdf rendering problem
reported by Friedrich Hagedorn. - JKS

Fix conversion of quadratic to cubic Bezier curves in PDF and PS
backends. Patch by Jae-Joon Lee. - JKS
```

(continues on next page)

(continued from previous page)

```
Added 5-point star marker to plot command q- EF

Fix hatching in PS backend - MGD

Fix log with base 2 - MGD

Added support for bilinear interpolation in
NonUniformImage; patch by Gregory Lielens. - EF

Added support for multiple histograms with data of
different length - MM

Fix step plots with log scale - MGD

Fix masked arrays with markers in non-Agg backends - MGD

Fix clip_on kwarg so it actually works correctly - MGD

Fix locale problems in SVG backend - MGD

fix quiver so masked values are not plotted - JSW

improve interactive pan/zoom in qt4 backend on windows - DSD

Fix more bugs in NaN/inf handling.  In particular, path
simplification (which does not handle NaNs or infs) will be turned
off automatically when infs or NaNs are present.  Also masked
arrays are now converted to arrays with NaNs for consistent
handling of masks and NaNs - MGD and EF

Added support for arbitrary rasterization resolutions to the SVG
backend. - MW
```

### 9.20.6 Changes for 0.98.x

- `psd()`, `csd()`, and `cohere()` will now automatically wrap negative frequency components to the beginning of the returned arrays. This is much more sensible behavior and makes them consistent with `specgram()`. The previous behavior was more of an oversight than a design decision.
- Added new keyword parameters *nonposx*, *nonposy* to *matplotlib.axes.Axes* methods that set log scale parameters. The default is still to mask out non-positive values, but the kwargs accept 'clip', which causes non-positive values to be replaced with a very small positive value.
- Added new *matplotlib.pyplot.fignum_exists()* and *matplotlib.pyplot.get_fignums()*; they merely expose information that had been hidden in `matplotlib._pylab_helpers`.
- Deprecated numerix package.
- Added new *matplotlib.image.imsave()* and exposed it to the *matplotlib.pyplot* interface.

- Remove support for pyExcelerator in exceltools -- use xlwt instead
- Changed the defaults of acorr and xcorr to use usevlines=True, maxlags=10 and normed=True since these are the best defaults
- Following keyword parameters for *matplotlib.legend.Legend* are now deprecated and new set of parameters are introduced. The new parameters are given as a fraction of the font-size. Also, *scatteryoffsets*, *fancybox* and *columnspacing* are added as keyword parameters.

| Deprecated | New |
|---|---|
| pad | borderpad |
| labelsep | labelspacing |
| handlelen | handlelength |
| handlestextsep | handletextpad |
| axespad | borderaxespad |

- Removed the configobj and experimental traits rc support
- Modified *matplotlib.mlab.psd()*, *matplotlib.mlab.csd()*, *matplotlib.mlab.cohere()*, and *matplotlib.mlab.specgram()* to scale one-sided densities by a factor of 2. Also, optionally scale the densities by the sampling frequency, which gives true values of densities that can be integrated by the returned frequency values. This also gives better MATLAB compatibility. The corresponding *matplotlib.axes.Axes* methods and *matplotlib.pyplot* functions were updated as well.
- Font lookup now uses a nearest-neighbor approach rather than an exact match. Some fonts may be different in plots, but should be closer to what was requested.
- *matplotlib.axes.Axes.set_xlim()*, *matplotlib.axes.Axes.set_ylim()* now return a copy of the `viewlim` array to avoid modify-in-place surprises.
- `matplotlib.afm.AFM.get_fullname` and `matplotlib.afm.AFM.get_familyname` no longer raise an exception if the AFM file does not specify these optional attributes, but returns a guess based on the required FontName attribute.
- Changed precision kwarg in *matplotlib.pyplot.spy()*; default is 0, and the string value 'present' is used for sparse arrays only to show filled locations.
- *matplotlib.collections.EllipseCollection* added.
- Added `angles` kwarg to *matplotlib.pyplot.quiver()* for more flexible specification of the arrow angles.
- Deprecated (raise NotImplementedError) all the mlab2 functions from *matplotlib.mlab* out of concern that some of them were not clean room implementations.
- Methods *matplotlib.collections.Collection.get_offsets()* and *matplotlib.collections.Collection.set_offsets()* added to *Collection* base class.
- `matplotlib.figure.Figure.figurePatch` renamed `matplotlib.figure.Figure.patch`; `matplotlib.axes.Axes.axesPatch` renamed `matplotlib.axes.Axes.patch`; `matplotlib.axes.Axes.axesFrame` renamed `matplotlib.axes.Axes.`

frame. matplotlib.axes.Axes.get_frame, which returns matplotlib.axes.Axes.patch, is deprecated.

- Changes in the *matplotlib.contour.ContourLabeler* attributes (*matplotlib.pyplot.clabel()* function) so that they all have a form like .labelAttribute. The three attributes that are most likely to be used by end users, .cl, .cl_xy and .cl_cvalues have been maintained for the moment (in addition to their renamed versions), but they are deprecated and will eventually be removed.
- Moved several functions in *matplotlib.mlab* and *matplotlib.cbook* into a separate module matplotlib.numerical_methods because they were unrelated to the initial purpose of mlab or cbook and appeared more coherent elsewhere.

### 9.20.7 Changes for 0.98.1

- Removed broken matplotlib.axes3d support and replaced it with a non-implemented error pointing to 0.91.x

### 9.20.8 Changes for 0.98.0

- *matplotlib.image.imread()* now no longer always returns RGBA data---if the image is luminance or RGB, it will return a MxN or MxNx3 array if possible. Also uint8 is no longer always forced to float.
- Rewrote the *matplotlib.cm.ScalarMappable* callback infrastructure to use *matplotlib.cbook.CallbackRegistry* rather than custom callback handling. Any users of matplotlib.cm.ScalarMappable.add_observer of the *ScalarMappable* should use the matplotlib.cm.ScalarMappable.callbacksSM *CallbackRegistry* instead.
- New axes function and Axes method provide control over the plot color cycle: matplotlib.axes.set_default_color_cycle and matplotlib.axes.Axes.set_color_cycle.
- Matplotlib now requires Python 2.4, so *matplotlib.cbook* will no longer provide set, enumerate(), reversed() or izip compatibility functions.
- In Numpy 1.0, bins are specified by the left edges only. The axes method *matplotlib.axes.Axes.hist()* now uses future Numpy 1.3 semantics for histograms. Providing binedges, the last value gives the upper-right edge now, which was implicitly set to +infinity in Numpy 1.0. This also means that the last bin doesn't contain upper outliers any more by default.
- New axes method and pyplot function, *hexbin()*, is an alternative to *scatter()* for large datasets. It makes something like a *pcolor()* of a 2-D histogram, but uses hexagonal bins.
- New kwarg, symmetric, in *matplotlib.ticker.MaxNLocator* allows one require an axis to be centered around zero.
- Toolkits must now be imported from mpl_toolkits (not matplotlib.toolkits)

### Notes about the transforms refactoring

A major new feature of the 0.98 series is a more flexible and extensible transformation infrastructure, written in Python/Numpy rather than a custom C extension.

The primary goal of this refactoring was to make it easier to extend matplotlib to support new kinds of projections. This is mostly an internal improvement, and the possible user-visible changes it allows are yet to come.

See *matplotlib.transforms* for a description of the design of the new transformation framework.

For efficiency, many of these functions return views into Numpy arrays. This means that if you hold on to a reference to them, their contents may change. If you want to store a snapshot of their current values, use the Numpy array method copy().

The view intervals are now stored only in one place -- in the *matplotlib.axes.Axes* instance, not in the locator instances as well. This means locators must get their limits from their *matplotlib.axis.Axis*, which in turn looks up its limits from the *Axes*. If a locator is used temporarily and not assigned to an Axis or Axes, (e.g., in *matplotlib.contour*), a dummy axis must be created to store its bounds. Call *matplotlib.ticker.TickHelper.create_dummy_axis()* to do so.

The functionality of `Pbox` has been merged with *Bbox*. Its methods now all return copies rather than modifying in place.

The following lists many of the simple changes necessary to update code from the old transformation framework to the new one. In particular, methods that return a copy are named with a verb in the past tense, whereas methods that alter an object in place are named with a verb in the present tense.

### matplotlib.transforms

| Old method | New method |
|---|---|
| `Bbox.get_bounds` | *transforms.Bbox.bounds* |
| `Bbox.width` | *transforms.Bbox.width* |
| `Bbox.height` | *transforms.Bbox.height* |
| `Bbox.intervalx().get_bounds()` `Bbox.intervalx().set_bounds()` | *transforms.Bbox.intervalx* [It is now a property.] |
| `Bbox.intervaly().get_bounds()` `Bbox.intervaly().set_bounds()` | *transforms.Bbox.intervaly* [It is now a property.] |
| `Bbox.xmin` | *transforms.Bbox.x0* or *transforms.Bbox.xmin*[1] |
| `Bbox.ymin` | *transforms.Bbox.y0* or *transforms.Bbox.ymin*[Page 4611, 1] |
| `Bbox.xmax` | *transforms.Bbox.x1* or *transforms.Bbox.xmax*[1] |
| `Bbox.ymax` | *transforms.Bbox.y1* or *transforms.Bbox.ymax*[1] |
| `Bbox.overlaps(bboxes)` | *Bbox.count_overlaps(bboxes)* |
| `bbox_all(bboxes)` | *Bbox.union(bboxes)* [It is a staticmethod.] |
| `lbwh_to_bbox(l, b, w, h)` | *Bbox.from_bounds(x0, y0, w, h)* [It is a staticmethod.] |
| `inverse_transform_bbox(trans, bbox)` | `bbox.inverse_transformed(trans)` |
| `Interval.contains_open(v)` | *interval_contains_open(tuple, v)* |
| `Interval.contains(v)` | *interval_contains(tuple, v)* |
| `identity_transform()` | *transforms.IdentityTransform* |
| `blend_xy_sep_transform(xtrans, ytrans)` | *blended_transform_factory(xtrans, ytrans)* |
| `scale_transform(xs, ys)` | *Affine2D().scale(xs[, ys])* |
| `get_bbox_transform(boxin, boxout)` | *BboxTransform(boxin, boxout)* or *BboxTransformFrom(boxin)* or *BboxTransformTo(boxout)* |
| `Transform.seq_xy_tup(points)` | *Transform.transform(points)* |
| `Transform.inverse_xy_tup(points)` | *Transform.inverted()*.transform(points) |

[1] The *Bbox* is bound by the points (x0, y0) to (x1, y1) and there is no defined order to these points, that is, x0 is not necessarily the left edge of the box. To get the left edge of the *Bbox*, use the read-only property *xmin*.

### matplotlib.axes

| Old method | New method |
|---|---|
| `Axes.get_position()` | *matplotlib.axes.Axes.get_position()*[2] |
| `Axes.set_position()` | *matplotlib.axes.Axes.set_position()*[3] |
| `Axes.toggle_log_lineary()` | *matplotlib.axes.Axes.set_yscale()*[4] |
| `Subplot` class | removed |

The `Polar` class has moved to *matplotlib.projections.polar*.

### matplotlib.artist

| Old method | New method |
|---|---|
| `Artist.set_clip_path(path)` | `Artist.set_clip_path(path, transform)`[5] |

### matplotlib.collections

| Old method | New method |
|---|---|
| *linestyle* | *linestyles*[6] |

### matplotlib.colors

| Old method | New method |
|---|---|
| `ColorConvertor.to_rgba_list(c)` | `colors.to_rgba_array(c)` [*matplotlib.colors.to_rgba_array()* returns an Nx4 NumPy array of RGBA color quadruples.] |

[2] *matplotlib.axes.Axes.get_position()* used to return a list of points, now it returns a *matplotlib.transforms.Bbox* instance.

[3] *matplotlib.axes.Axes.set_position()* now accepts either four scalars or a *matplotlib.transforms.Bbox* instance.

[4] Since the refactoring allows for more than two scale types ('log' or 'linear'), it no longer makes sense to have a toggle. `Axes.toggle_log_lineary()` has been removed.

[5] *matplotlib.artist.Artist.set_clip_path()* now accepts a *matplotlib.path.Path* instance and a *matplotlib.transforms.Transform* that will be applied to the path immediately before clipping.

[6] Linestyles are now treated like all other collection attributes, i.e. a single value or multiple values may be provided.

### matplotlib.contour

| Old method | New method |
|---|---|
| `Contour._segments` | `matplotlib.contour.Contour.get_paths` [Returns a list of *matplotlib.path.Path* instances.] |

### matplotlib.figure

| Old method | New method |
|---|---|
| `Figure.dpi.get()` `Figure.dpi.set()` | *matplotlib.figure.Figure.dpi (a property)* |

### matplotlib.patches

| Old method | New method |
|---|---|
| `Patch.get_verts()` | *matplotlib.patches.Patch.get_path()* [Returns a *matplotlib.path.Path* instance] |

### matplotlib.backend_bases

| Old method | New method |
|---|---|
| `GraphicsContext.set_clip_rectangle(tuple)` | *GraphicsContext.set_clip_rectangle(bbox)* |
| `GraphicsContext.get_clip_path()` | *GraphicsContext.get_clip_path()*[7] |
| `GraphicsContext.set_clip_path()` | *GraphicsContext.set_clip_path()*[8] |

[7] *matplotlib.backend_bases.GraphicsContextBase.get_clip_path()* returns a tuple of the form (*path*, *affine_transform*), where *path* is a *matplotlib.path.Path* instance and *affine_transform* is a *matplotlib.transforms.Affine2D* instance.

[8] *matplotlib.backend_bases.GraphicsContextBase.set_clip_path()* now only accepts a *matplotlib.transforms.TransformedPath* instance.

#### RendererBase

New methods:

- *draw_path(self, gc, path, transform, rgbFace)*
- *draw_markers(self, gc, marker_path, marker_trans, path, trans, rgbFace)*
- *draw_path_collection(self, master_transform, cliprect, clippath, clippath_trans, paths, all_transforms, offsets, offsetTrans, facecolors, edgecolors, linewidths, linestyles, antialiaseds)* *[optional]*

Changed methods:

- `draw_image(self, x, y, im, bbox)` is now *draw_image(self, x, y, im, bbox, clippath, clippath_trans)*

Removed methods:

- `draw_arc`
- `draw_line_collection`
- `draw_line`
- `draw_lines`
- `draw_point`
- `draw_quad_mesh`
- `draw_poly_collection`
- `draw_polygon`
- `draw_rectangle`
- `draw_regpoly_collection`

### 9.20.9 Changes for 0.91.2

- For `csv2rec`, checkrows=0 is the new default indicating all rows will be checked for type inference
- A warning is issued when an image is drawn on log-scaled axes, since it will not log-scale the image data.
- Moved `rec2gtk` to `matplotlib.toolkits.gtktools`
- Moved `rec2excel` to `matplotlib.toolkits.exceltools`
- Removed, dead/experimental ExampleInfo, Namespace and Importer code from *matplotlib*

### 9.20.10 Changes for 0.91.0

- Changed `cbook.is_file_like` to `cbook.is_writable_file_like` and corrected behavior.
- Added *ax* keyword argument to *`pyplot.colorbar()`* and *`Figure.colorbar()`* so that one can specify the axes object from which space for the colorbar is to be taken, if one does not want to make the colorbar axes manually.
- Changed `cbook.reversed` so it yields a tuple rather than a (index, tuple). This agrees with the Python reversed builtin, and cbook only defines reversed if Python doesn't provide the builtin.
- Made skiprows=1 the default on `csv2rec`
- The gd and paint backends have been deleted.
- The errorbar method and function now accept additional kwargs so that upper and lower limits can be indicated by capping the bar with a caret instead of a straight line segment.
- The *`matplotlib.dviread`* file now has a parser for files like psfonts.map and pdftex.map, to map TeX font names to external files.
- The file `matplotlib.type1font` contains a new class for Type 1 fonts. Currently it simply reads pfa and pfb format files and stores the data in a way that is suitable for embedding in pdf files. In the future the class might actually parse the font to allow e.g., subsetting.
- `matplotlib.ft2font` now supports `FT_Attach_File`. In practice this can be used to read an afm file in addition to a pfa/pfb file, to get metrics and kerning information for a Type 1 font.
- The `AFM` class now supports querying CapHeight and stem widths. The get_name_char method now has an isord kwarg like get_width_char.
- Changed *`pcolor()`* default to `shading='flat'`; but as noted now in the docstring, it is preferable to simply use the *edgecolor* keyword argument.
- The mathtext font commands (`\cal`, `\rm`, `\it`, `\tt`) now behave as TeX does: they are in effect until the next font change command or the end of the grouping. Therefore uses of `$\cal{R}$` should be changed to `${\cal R}$`. Alternatively, you may use the new LaTeX-style font commands (`\mathcal`, `\mathrm`, `\mathit`, `\mathtt`) which do affect the following group, e.g., `$\mathcal{R}$`.
- Text creation commands have a new default linespacing and a new `linespacing` kwarg, which is a multiple of the maximum vertical extent of a line of ordinary text. The default is 1.2; `linespacing=2` would be like ordinary double spacing, for example.
- Changed default kwarg in *`matplotlib.colors.Normalize`* to `clip=False`; clipping silently defeats the purpose of the special over, under, and bad values in the colormap, thereby leading to unexpected behavior. The new default should reduce such surprises.
- Made the emit property of *`set_xlim()`* and *`set_ylim()`* `True` by default; removed the Axes custom callback handling into a 'callbacks' attribute which is a *`CallbackRegistry`* instance. This now supports the 'xlim_changed' and 'ylim_changed' Axes events.

### 9.20.11 Changes for 0.90.1

```
The file dviread.py has a (very limited and fragile) dvi reader
for usetex support. The API might change in the future so don't
depend on it yet.

Removed deprecated support for a float value as a gray-scale;
now it must be a string, like '0.5'.  Added alpha kwarg to
ColorConverter.to_rgba_list.

New method set_bounds(vmin, vmax) for formatters, locators sets
the viewInterval and dataInterval from floats.

Removed deprecated colorbar_classic.

Line2D.get_xdata and get_ydata valid_only=False kwarg is replaced
by orig=True.  When True, it returns the original data, otherwise
the processed data (masked, converted)

Some modifications to the units interface.
units.ConversionInterface.tickers renamed to
units.ConversionInterface.axisinfo and it now returns a
units.AxisInfo object rather than a tuple.  This will make it
easier to add axis info functionality (e.g., I added a default label
on this iteration) w/o having to change the tuple length and hence
the API of the client code every time new functionality is added.
Also, units.ConversionInterface.convert_to_value is now simply
named units.ConversionInterface.convert.

Axes.errorbar uses Axes.vlines and Axes.hlines to draw its error
limits int he vertical and horizontal direction.  As you'll see
in the changes below, these functions now return a LineCollection
rather than a list of lines.  The new return signature for
errorbar is  ylins, caplines, errorcollections where
errorcollections is a xerrcollection, yerrcollection

Axes.vlines and Axes.hlines now create and returns a LineCollection, not a␣
 ↪list
of lines.  This is much faster.  The kwarg signature has changed,
so consult the docs

MaxNLocator accepts a new Boolean kwarg ('integer') to force
ticks to integer locations.

Commands that pass an argument to the Text constructor or to
Text.set_text() now accept any object that can be converted
with '%s'.  This affects xlabel(), title(), etc.

Barh now takes a **kwargs dict instead of most of the old
arguments. This helps ensure that bar and barh are kept in sync,
but as a side effect you can no longer pass e.g., color as a
positional argument.
```

(continues on next page)

(continued from previous page)

```
ft2font.get_charmap() now returns a dict that maps character codes
to glyph indices (until now it was reversed)

Moved data files into lib/matplotlib so that setuptools' develop
mode works. Re-organized the mpl-data layout so that this source
structure is maintained in the installation. (i.e., the 'fonts' and
'images' sub-directories are maintained in site-packages.).
Suggest removing site-packages/matplotlib/mpl-data and
~/.matplotlib/ttffont.cache before installing
```

### 9.20.12 Changes for 0.90.0

```
All artists now implement a "pick" method which users should not
call.  Rather, set the "picker" property of any artist you want to
pick on (the epsilon distance in points for a hit test) and
register with the "pick_event" callback.  See
examples/pick_event_demo.py for details

Bar, barh, and hist have "log" binary kwarg: log=True
sets the ordinate to a log scale.

Boxplot can handle a list of vectors instead of just
an array, so vectors can have different lengths.

Plot can handle 2-D x and/or y; it plots the columns.

Added linewidth kwarg to bar and barh.

Made the default Artist._transform None (rather than invoking
identity_transform for each artist only to have it overridden
later).  Use artist.get_transform() rather than artist._transform,
even in derived classes, so that the default transform will be
created lazily as needed

New LogNorm subclass of Normalize added to colors.py.
All Normalize subclasses have new inverse() method, and
the __call__() method has a new clip kwarg.

Changed class names in colors.py to match convention:
normalize -> Normalize, no_norm -> NoNorm.  Old names
are still available for now.

Removed obsolete pcolor_classic command and method.

Removed lineprops and markerprops from the Annotation code and
replaced them with an arrow configurable with kwarg arrowprops.
See examples/annotation_demo.py - JDH
```

### 9.20.13 Changes for 0.87.7

```
Completely reworked the annotations API because I found the old
API cumbersome.  The new design is much more legible and easy to
read.  See matplotlib.text.Annotation and
examples/annotation_demo.py

markeredgecolor and markerfacecolor cannot be configured in
matplotlibrc any more. Instead, markers are generally colored
automatically based on the color of the line, unless marker colors
are explicitly set as kwargs - NN

Changed default comment character for load to '#' - JDH

math_parse_s_ft2font_svg from mathtext.py & mathtext2.py now returns
width, height, svg_elements. svg_elements is an instance of Bunch (
cmbook.py) and has the attributes svg_glyphs and svg_lines, which are both
lists.

Renderer.draw_arc now takes an additional parameter, rotation.
It specifies to draw the artist rotated in degrees anti-
clockwise.  It was added for rotated ellipses.

Renamed Figure.set_figsize_inches to Figure.set_size_inches to
better match the get method, Figure.get_size_inches.

Removed the copy_bbox_transform from transforms.py; added
shallowcopy methods to all transforms.  All transforms already
had deepcopy methods.

FigureManager.resize(width, height): resize the window
specified in pixels

barh: x and y args have been renamed to width and bottom
respectively, and their order has been swapped to maintain
a (position, value) order.

bar and barh: now accept kwarg 'edgecolor'.

bar and barh: The left, height, width and bottom args can
now all be scalars or sequences; see docstring.

barh: now defaults to edge aligned instead of center
aligned bars

bar, barh and hist: Added a keyword arg 'align' that
controls between edge or center bar alignment.

Collections: PolyCollection and LineCollection now accept
vertices or segments either in the original form [(x,y),
(x,y), ...] or as a 2D numerix array, with X as the first column
and Y as the second. Contour and quiver output the numerix
```

(continues on next page)

(continued from previous page)

```
form.  The transforms methods Bbox.update() and
Transformation.seq_xy_tups() now accept either form.

Collections: LineCollection is now a ScalarMappable like
PolyCollection, etc.

Specifying a grayscale color as a float is deprecated; use
a string instead, e.g., 0.75 -> '0.75'.

Collections: initializers now accept any mpl color arg, or
sequence of such args; previously only a sequence of rgba
tuples was accepted.

Colorbar: completely new version and api; see docstring.  The
original version is still accessible as colorbar_classic, but
is deprecated.

Contourf: "extend" kwarg replaces "clip_ends"; see docstring.
Masked array support added to pcolormesh.

Modified aspect-ratio handling:
    Removed aspect kwarg from imshow
    Axes methods:
        set_aspect(self, aspect, adjustable=None, anchor=None)
        set_adjustable(self, adjustable)
        set_anchor(self, anchor)
    Pylab interface:
        axis('image')

 Backend developers: ft2font's load_char now takes a flags
 argument, which you can OR together from the LOAD_XXX
 constants.
```

### 9.20.14 Changes for 0.86

```
Matplotlib data is installed into the matplotlib module.
This is similar to package_data.  This should get rid of
having to check for many possibilities in _get_data_path().
The MATPLOTLIBDATA env key is still checked first to allow
for flexibility.

1) Separated the color table data from cm.py out into
a new file, _cm.py, to make it easier to find the actual
code in cm.py and to add new colormaps. Everything
from _cm.py is imported by cm.py, so the split should be
transparent.
2) Enabled automatic generation of a colormap from
a list of colors in contour; see modified
examples/contour_demo.py.
3) Support for imshow of a masked array, with the
```

(continues on next page)

(continued from previous page)

```
ability to specify colors (or no color at all) for
masked regions, and for regions that are above or
below the normally mapped region.  See
examples/image_masked.py.
4) In support of the above, added two new classes,
ListedColormap, and no_norm, to colors.py, and modified
the Colormap class to include common functionality. Added
a clip kwarg to the normalize class.
```

### 9.20.15 Changes for 0.85

```
Made xtick and ytick separate props in rc

made pos=None the default for tick formatters rather than 0 to
indicate "not supplied"

Removed "feature" of minor ticks which prevents them from
overlapping major ticks.  Often you want major and minor ticks at
the same place, and can offset the major ticks with the pad.  This
could be made configurable

Changed the internal structure of contour.py to a more OO style.
Calls to contour or contourf in axes.py or pylab.py now return
a ContourSet object which contains references to the
LineCollections or PolyCollections created by the call,
as well as the configuration variables that were used.
The ContourSet object is a "mappable" if a colormap was used.

Added a clip_ends kwarg to contourf. From the docstring:
         * clip_ends = True
           If False, the limits for color scaling are set to the
           minimum and maximum contour levels.
           True (default) clips the scaling limits.  Example:
           if the contour boundaries are V = [-100, 2, 1, 0, 1, 2, 100],
           then the scaling limits will be [-100, 100] if clip_ends
           is False, and [-3, 3] if clip_ends is True.
Added kwargs linewidths, antialiased, and nchunk to contourf.  These
are experimental; see the docstring.

Changed Figure.colorbar():
    kw argument order changed;
    if mappable arg is a non-filled ContourSet, colorbar() shows
            lines instead hof polygons.
    if mappable arg is a filled ContourSet with clip_ends=True,
            the endpoints are not labelled, so as to give the
            correct impression of open-endedness.

Changed LineCollection.get_linewidths to get_linewidth, for
consistency.
```

### 9.20.16 Changes for 0.84

```
Unified argument handling between hlines and vlines.  Both now
take optionally a fmt argument (as in plot) and a keyword args
that can be passed onto Line2D.

Removed all references to "data clipping" in rc and lines.py since
these were not used and not optimized.  I'm sure they'll be
resurrected later with a better implementation when needed.

'set' removed - no more deprecation warnings.  Use 'setp' instead.

Backend developers: Added flipud method to image and removed it
from to_str.  Removed origin kwarg from backend.draw_image.
origin is handled entirely by the frontend now.
```

### 9.20.17 Changes for 0.83

```
- Made HOME/.matplotlib the new config dir where the matplotlibrc
  file, the ttf.cache, and the tex.cache live.  The new default
  filenames in .matplotlib have no leading dot and are not hidden.
  e.g., the new names are matplotlibrc, tex.cache, and ttffont.cache.
  This is how ipython does it so it must be right.

  If old files are found, a warning is issued and they are moved to
  the new location.

- backends/__init__.py no longer imports new_figure_manager,
  draw_if_interactive and show from the default backend, but puts
  these imports into a call to pylab_setup.  Also, the Toolbar is no
  longer imported from WX/WXAgg.  New usage:

    from backends import pylab_setup
    new_figure_manager, draw_if_interactive, show = pylab_setup()

- Moved Figure.get_width_height() to FigureCanvasBase. It now
  returns int instead of float.
```

### 9.20.18 Changes for 0.82

```
- toolbar import change in GTKAgg, GTKCairo and WXAgg

- Added subplot config tool to GTK* backends -- note you must now
  import the NavigationToolbar2 from your backend of choice rather
  than from backend_gtk because it needs to know about the backend
  specific canvas -- see examples/embedding_in_gtk2.py.  Ditto for
  wx backend -- see examples/embedding_in_wxagg.py
```

(continues on next page)

(continued from previous page)

```
- hist bin change

    Sean Richards notes there was a problem in the way we created
    the binning for histogram, which made the last bin
    underrepresented.  From his post:

      I see that hist uses the linspace function to create the bins
      and then uses searchsorted to put the values in their correct
      bin. That's all good but I am confused over the use of linspace
      for the bin creation. I wouldn't have thought that it does
      what is needed, to quote the docstring it creates a "Linear
      spaced array from min to max". For it to work correctly
      shouldn't the values in the bins array be the same bound for
      each bin? (i.e. each value should be the lower bound of a
      bin). To provide the correct bins for hist would it not be
      something like

      def bins(xmin, xmax, N):
        if N==1: return xmax
        dx = (xmax-xmin)/N # instead of N-1
        return xmin + dx*arange(N)


     This suggestion is implemented in 0.81.  My test script with these
     changes does not reveal any bias in the binning

      from matplotlib.numerix.mlab import randn, rand, zeros, Float
      from matplotlib.mlab import hist, mean

      Nbins = 50
      Ntests = 200
      results = zeros((Ntests,Nbins), typecode=Float)
      for i in range(Ntests):
          print 'computing', i
          x = rand(10000)
          n, bins = hist(x, Nbins)
          results[i] = n
      print mean(results)
```

### 9.20.19 Changes for 0.81

```
- pylab and artist "set" functions renamed to setp to avoid clash
  with python2.4 built-in set.  Current version will issue a
  deprecation warning which will be removed in future versions

- imshow interpolation arguments changes for advanced interpolation
  schemes.  See help imshow, particularly the interpolation,
  filternorm and filterrad kwargs
```

(continues on next page)

(continued from previous page)

```
- Support for masked arrays has been added to the plot command and
  to the Line2D object.  Only the valid points are plotted.  A
  "valid_only" kwarg was added to the get_xdata() and get_ydata()
  methods of Line2D; by default it is False, so that the original
  data arrays are returned. Setting it to True returns the plottable
  points.

- contour changes:

  Masked arrays: contour and contourf now accept masked arrays as
    the variable to be contoured.  Masking works correctly for
    contour, but a bug remains to be fixed before it will work for
    contourf.  The "badmask" kwarg has been removed from both
    functions.

   Level argument changes:

     Old version: a list of levels as one of the positional
     arguments specified the lower bound of each filled region; the
     upper bound of the last region was taken as a very large
     number.  Hence, it was not possible to specify that z values
     between 0 and 1, for example, be filled, and that values
     outside that range remain unfilled.

     New version: a list of N levels is taken as specifying the
     boundaries of N-1 z ranges.  Now the user has more control over
     what is colored and what is not.  Repeated calls to contourf
     (with different colormaps or color specifications, for example)
     can be used to color different ranges of z.  Values of z
     outside an expected range are left uncolored.

     Example:
       Old: contourf(z, [0, 1, 2]) would yield 3 regions: 0-1, 1-2, and >2.
       New: it would yield 2 regions: 0-1, 1-2.  If the same 3 regions were
       desired, the equivalent list of levels would be [0, 1, 2,
       1e38].
```

### 9.20.20 Changes for 0.80

```
- xlim/ylim/axis always return the new limits regardless of
  arguments.  They now take kwargs which allow you to selectively
  change the upper or lower limits while leaving unnamed limits
  unchanged.  See help(xlim) for example
```

### 9.20.21 Changes for 0.73

```
- Removed deprecated ColormapJet and friends

- Removed all error handling from the verbose object

- figure num of zero is now allowed
```

### 9.20.22 Changes for 0.72

```
- Line2D, Text, and Patch copy_properties renamed update_from and
  moved into artist base class

- LineCollections.color renamed to LineCollections.set_color for
  consistency with set/get introspection mechanism,

- pylab figure now defaults to num=None, which creates a new figure
  with a guaranteed unique number

- contour method syntax changed - now it is MATLAB compatible

    unchanged: contour(Z)
    old: contour(Z, x=Y, y=Y)
    new: contour(X, Y, Z)

  see http://matplotlib.sf.net/matplotlib.pylab.html#-contour


 - Increased the default resolution for save command.

 - Renamed the base attribute of the ticker classes to _base to avoid conflict
   with the base method.  Sitt for subs

 - subs=none now does autosubbing in the tick locator.

 - New subplots that overlap old will delete the old axes.  If you
   do not want this behavior, use fig.add_subplot or the axes
   command
```

### 9.20.23 Changes for 0.71

```
Significant numerix namespace changes, introduced to resolve
namespace clashes between python built-ins and mlab names.
Refactored numerix to maintain separate modules, rather than
folding all these names into a single namespace.  See the following
mailing list threads for more information and background

  http://sourceforge.net/mailarchive/forum.php?thread_id=6398890&forum_
```

(continues on next page)

(continued from previous page)

```
 ↪id=36187
  http://sourceforge.net/mailarchive/forum.php?thread_id=6323208&forum_
 ↪id=36187


OLD usage

  from matplotlib.numerix import array, mean, fft

NEW usage

  from matplotlib.numerix import array
  from matplotlib.numerix.mlab import mean
  from matplotlib.numerix.fft import fft

numerix dir structure mirrors numarray (though it is an incomplete
implementation)

  numerix
  numerix/mlab
  numerix/linear_algebra
  numerix/fft
  numerix/random_array

but of course you can use 'numerix : Numeric' and still get the
symbols.

pylab still imports most of the symbols from Numerix, MLab, fft,
etc, but is more cautious.  For names that clash with python names
(min, max, sum), pylab keeps the builtins and provides the numeric
versions with an a* prefix, e.g., (amin, amax, asum)
```

### 9.20.24 Changes for 0.70

```
MplEvent factored into a base class Event and derived classes
MouseEvent and KeyEvent

Removed defunct set_measurement in wx toolbar
```

### 9.20.25 Changes for 0.65.1

```
removed add_axes and add_subplot from backend_bases.  Use
figure.add_axes and add_subplot instead.  The figure now manages the
current axes with gca and sca for get and set current axes.  If you
have code you are porting which called, e.g., figmanager.add_axes, you
can now simply do figmanager.canvas.figure.add_axes.
```

### 9.20.26 Changes for 0.65

```
mpl_connect and mpl_disconnect in the MATLAB interface renamed to
connect and disconnect

Did away with the text methods for angle since they were ambiguous.
fontangle could mean fontstyle (oblique, etc) or the rotation of the
text.  Use style and rotation instead.
```

### 9.20.27 Changes for 0.63

```
Dates are now represented internally as float days since 0001-01-01,
UTC.

All date tickers and formatters are now in matplotlib.dates, rather
than matplotlib.tickers

converters have been abolished from all functions and classes.
num2date and date2num are now the converter functions for all date
plots

Most of the date tick locators have a different meaning in their
constructors.  In the prior implementation, the first argument was a
base and multiples of the base were ticked.  e.g.,

  HourLocator(5)  # old: tick every 5 minutes

In the new implementation, the explicit points you want to tick are
provided as a number or sequence

   HourLocator(range(0,5,61))  # new: tick every 5 minutes

This gives much greater flexibility.  I have tried to make the
default constructors (no args) behave similarly, where possible.

Note that YearLocator still works under the base/multiple scheme.
The difference between the YearLocator and the other locators is
that years are not recurrent.


Financial functions:

  matplotlib.finance.quotes_historical_yahoo(ticker, date1, date2)

   date1, date2 are now datetime instances.  Return value is a list
   of quotes where the quote time is a float - days since gregorian
   start, as returned by date2num

   See examples/finance_demo.py for example usage of new API
```

### 9.20.28 Changes for 0.61

```
canvas.connect is now deprecated for event handling.  use
mpl_connect and mpl_disconnect instead.  The callback signature is
func(event) rather than func(widget, event)
```

### 9.20.29 Changes for 0.60

```
ColormapJet and Grayscale are deprecated.  For backwards
compatibility, they can be obtained either by doing

  from matplotlib.cm import ColormapJet

or

  from matplotlib.matlab import *

They are replaced by cm.jet and cm.grey
```

### 9.20.30 Changes for 0.54.3

```
removed the set_default_font / get_default_font scheme from the
font_manager to unify customization of font defaults with the rest of
the rc scheme.  See examples/font_properties_demo.py and help(rc) in
matplotlib.matlab.
```

### 9.20.31 Changes for 0.54

#### MATLAB interface

#### dpi

Several of the backends used a PIXELS_PER_INCH hack that I added to try and make images render consistently across backends. This just complicated matters. So you may find that some font sizes and line widths appear different than before. Apologies for the inconvenience. You should set the dpi to an accurate value for your screen to get true sizes.

### pcolor and scatter

There are two changes to the MATLAB interface API, both involving the patch drawing commands. For efficiency, pcolor and scatter have been rewritten to use polygon collections, which are a new set of objects from matplotlib.collections designed to enable efficient handling of large collections of objects. These new collections make it possible to build large scatter plots or pcolor plots with no loops at the python level, and are significantly faster than their predecessors. The original pcolor and scatter functions are retained as pcolor_classic and scatter_classic.

The return value from pcolor is a PolyCollection. Most of the properties that are available on rectangles or other patches are also available on PolyCollections, e.g., you can say:

```
c = scatter(blah, blah)
c.set_linewidth(1.0)
c.set_facecolor('r')
c.set_alpha(0.5)
```

or:

```
c = scatter(blah, blah)
set(c, 'linewidth', 1.0, 'facecolor', 'r', 'alpha', 0.5)
```

Because the collection is a single object, you no longer need to loop over the return value of scatter or pcolor to set properties for the entire list.

If you want the different elements of a collection to vary on a property, e.g., to have different line widths, see matplotlib.collections for a discussion on how to set the properties as a sequence.

For scatter, the size argument is now in points^2 (the area of the symbol in points) as in MATLAB and is not in data coords as before. Using sizes in data coords caused several problems. So you will need to adjust your size arguments accordingly or use scatter_classic.

### mathtext spacing

For reasons not clear to me (and which I'll eventually fix) spacing no longer works in font groups. However, I added three new spacing commands which compensate for this " (regular space), '/' (small space) and 'hspace{frac}' where frac is a fraction of fontsize in points. You will need to quote spaces in font strings, is:

```
title(r'$\rm{Histogram\ of\ IQ:}\ \mu=100,\ \sigma=15$')
```

### Object interface - Application programmers

#### Autoscaling

The x and y axis instances no longer have autoscale view. These are handled by axes.autoscale_view

#### Axes creation

You should not instantiate your own Axes any more using the OO API. Rather, create a Figure as before and in place of:

```
f = Figure(figsize=(5,4), dpi=100)
a = Subplot(f, 111)
f.add_axis(a)
```

use:

```
f = Figure(figsize=(5,4), dpi=100)
a = f.add_subplot(111)
```

That is, add_axis no longer exists and is replaced by:

```
add_axes(rect, axisbg=defaultcolor, frameon=True)
add_subplot(num, axisbg=defaultcolor, frameon=True)
```

#### Artist methods

If you define your own Artists, you need to rename the _draw method to draw

#### Bounding boxes

matplotlib.transforms.Bound2D is replaced by matplotlib.transforms.Bbox. If you want to construct a bbox from left, bottom, width, height (the signature for Bound2D), use matplotlib.transforms.lbwh_to_bbox, as in:

```
bbox = clickBBox = lbwh_to_bbox(left, bottom, width, height)
```

The Bbox has a different API than the Bound2D. e.g., if you want to get the width and height of the bbox

**OLD**:

```
width  = fig.bbox.x.interval()
height = fig.bbox.y.interval()
```

**NEW**:

```
width  = fig.bbox.width()
height = fig.bbox.height()
```

### Object constructors

You no longer pass the bbox, dpi, or transforms to the various Artist constructors. The old way or creating lines and rectangles was cumbersome because you had to pass so many attributes to the Line2D and Rectangle classes not related directly to the geometry and properties of the object. Now default values are added to the object when you call axes.add_line or axes.add_patch, so they are hidden from the user.

If you want to define a custom transformation on these objects, call o.set_transform(trans) where trans is a Transformation instance.

In prior versions of you wanted to add a custom line in data coords, you would have to do:

```
l = Line2D(dpi, bbox, x, y,
           color = color,
           transx = transx,
           transy = transy,
           )
```

now all you need is:

```
l = Line2D(x, y, color=color)
```

and the axes will set the transformation for you (unless you have set your own already, in which case it will eave it unchanged)

### Transformations

The entire transformation architecture has been rewritten. Previously the x and y transformations where stored in the xaxis and yaxis instances. The problem with this approach is it only allows for separable transforms (where the x and y transformations don't depend on one another). But for cases like polar, they do. Now transformations operate on x,y together. There is a new base class matplotlib.transforms.Transformation and two concrete implementations, matplotlib.transforms.SeparableTransformation and matplotlib.transforms.Affine. The SeparableTransformation is constructed with the bounding box of the input (this determines the rectangular coordinate system of the input, i.e., the x and y view limits), the bounding box of the display, and possibly nonlinear transformations of x and y. The 2 most frequently used transformations, data coordinates -> display and axes coordinates -> display are available as ax.transData and ax.transAxes. See alignment_demo.py which uses axes coords.

Also, the transformations should be much faster now, for two reasons

- they are written entirely in extension code
- because they operate on x and y together, they can do the entire transformation in one loop. Earlier I did something along the lines of:

```
xt = sx*func(x) + tx
yt = sy*func(y) + ty
```

Although this was done in numerix, it still involves 6 length(x) for-loops (the multiply, add, and function evaluation each for x and y). Now all of that is done in a single pass.

If you are using transformations and bounding boxes to get the cursor position in data coordinates, the method calls are a little different now. See the updated examples/coords_demo.py which shows you how to do this.

Likewise, if you are using the artist bounding boxes to pick items on the canvas with the GUI, the bbox methods are somewhat different. You will need to see the updated examples/object_picker.py.

See unit/transforms_unit.py for many examples using the new transformations.

### 9.20.32 Changes for 0.50

```
* refactored Figure class so it is no longer backend dependent.
  FigureCanvasBackend takes over the backend specific duties of the
  Figure.  matplotlib.backend_bases.FigureBase moved to
  matplotlib.figure.Figure.

* backends must implement FigureCanvasBackend (the thing that
  controls the figure and handles the events if any) and
  FigureManagerBackend (wraps the canvas and the window for MATLAB
  interface).  FigureCanvasBase implements a backend switching
  mechanism

* Figure is now an Artist (like everything else in the figure) and
  is totally backend independent

* GDFONTPATH renamed to TTFPATH

* backend faceColor argument changed to rgbFace

* colormap stuff moved to colors.py

* arg_to_rgb in backend_bases moved to class ColorConverter in
  colors.py

* GD users must upgrade to gd-2.0.22 and gdmodule-0.52 since new gd
  features (clipping, antialiased lines) are now used.

* Renderer must implement points_to_pixels

Migrating code:

MATLAB interface:

  The only API change for those using the MATLAB interface is in how
  you call figure redraws for dynamically updating figures.  In the
  old API, you did
```

(continues on next page)

(continued from previous page)

```
    fig.draw()

  In the new API, you do

    manager = get_current_fig_manager()
    manager.canvas.draw()

  See the examples system_monitor.py, dynamic_demo.py, and anim.py

API

  There is one important API change for application developers.
  Figure instances used subclass GUI widgets that enabled them to be
  placed directly into figures.  e.g., FigureGTK subclassed
  gtk.DrawingArea.  Now the Figure class is independent of the
  backend, and FigureCanvas takes over the functionality formerly
  handled by Figure.  In order to include figures into your apps,
  you now need to do, for example

    # gtk example
    fig = Figure(figsize=(5,4), dpi=100)
    canvas = FigureCanvasGTK(fig)  # a gtk.DrawingArea
    canvas.show()
    vbox.pack_start(canvas)

  If you use the NavigationToolbar, this in now initialized with a
  FigureCanvas, not a Figure.  The examples embedding_in_gtk.py,
  embedding_in_gtk2.py, and mpl_with_glade.py all reflect the new
  API so use these as a guide.

  All prior calls to

   figure.draw()  and
   figure.print_figure(args)

  should now be

   canvas.draw()  and
   canvas.print_figure(args)

  Apologies for the inconvenience.  This refactorization brings
  significant more freedom in developing matplotlib and should bring
  better plotting capabilities, so I hope the inconvenience is worth
  it.
```

### 9.20.33 Changes for 0.42

```
* Refactoring AxisText to be backend independent.  Text drawing and
  get_window_extent functionality will be moved to the Renderer.

* backend_bases.AxisTextBase is now text.Text module

* All the erase and reset functionality removed from AxisText - not
  needed with double buffered drawing.  Ditto with state change.
  Text instances have a get_prop_tup method that returns a hashable
  tuple of text properties which you can use to see if text props
  have changed, e.g., by caching a font or layout instance in a dict
  with the prop tup as a key -- see RendererGTK.get_pango_layout in
  backend_gtk for an example.

* Text._get_xy_display renamed Text.get_xy_display

* Artist set_renderer and wash_brushes methods removed

* Moved Legend class from matplotlib.axes into matplotlib.legend

* Moved Tick, XTick, YTick, Axis, XAxis, YAxis from matplotlib.axes
  to matplotlib.axis

* moved process_text_args to matplotlib.text

* After getting Text handled in a backend independent fashion, the
  import process is much cleaner since there are no longer cyclic
  dependencies

* matplotlib.matlab._get_current_fig_manager renamed to
  matplotlib.matlab.get_current_fig_manager to allow user access to
  the GUI window attribute, e.g., figManager.window for GTK and
  figManager.frame for wx
```

### 9.20.34 Changes for 0.40

```
- Artist
    * __init__ takes a DPI instance and a Bound2D instance which is
      the bounding box of the artist in display coords
    * get_window_extent returns a Bound2D instance
    * set_size is removed; replaced by bbox and dpi
    * the clip_gc method is removed.  Artists now clip themselves with
      their box
    * added _clipOn boolean attribute.  If True, gc clip to bbox.

- AxisTextBase
    * Initialized with a transx, transy which are Transform instances
    * set_drawing_area removed
    * get_left_right and get_top_bottom are replaced by get_window_extent
```

(continues on next page)

(continued from previous page)

```
- Line2D Patches now take transx, transy
    * Initialized with a transx, transy which are Transform instances

- Patches
   * Initialized with a transx, transy which are Transform instances

- FigureBase attributes dpi is a DPI instance rather than scalar and
  new attribute bbox is a Bound2D in display coords, and I got rid
  of the left, width, height, etc... attributes.  These are now
  accessible as, for example, bbox.x.min is left, bbox.x.interval()
  is width, bbox.y.max is top, etc...

- GcfBase attribute pagesize renamed to figsize

- Axes
    * removed figbg attribute
    * added fig instance to __init__
    * resizing is handled by figure call to resize.

- Subplot
    * added fig instance to __init__

- Renderer methods for patches now take gcEdge and gcFace instances.
  gcFace=None takes the place of filled=False

- True and False symbols provided by cbook in a python2.3 compatible
  way

- new module transforms supplies Bound1D, Bound2D and Transform
  instances and more

- Changes to the MATLAB helpers API

  * _matlab_helpers.GcfBase is renamed by Gcf.  Backends no longer
    need to derive from this class.  Instead, they provide a factory
    function new_figure_manager(num, figsize, dpi).  The destroy
    method of the GcfDerived from the backends is moved to the derived
    FigureManager.

  * FigureManagerBase moved to backend_bases

  * Gcf.get_all_figwins renamed to Gcf.get_all_fig_managers

Jeremy:

  Make sure to self._reset = False in AxisTextWX._set_font.  This was
  something missing in my backend code.
```

# Part V

# Contribute

Matplotlib is a community project maintained for and by its users.

There are many ways you can help!

CHAPTER

TEN

# CONTRIBUTE

**Important:** If you plan to contribute to Matplotlib, please read the development version of this document as it will have the most up to date installation instructions, workflow process, and contributing guidelines.

Thank you for your interest in helping to improve Matplotlib!

There are various ways to contribute: optimizing and refactoring code, detailing unclear documentation and writing new examples, helping the community, reporting and fixing bugs and requesting and implementing new features...

**Submit a bug report**

We have preloaded the issue creation page with a Markdown form that you can use to provide relevant context. Thank you for your help in keeping bug reports complete, targeted and descriptive.

Report a bug

**Request a new feature**

We will give feedback on the feature proposal. Since Matplotlib is an open source project with limited resources, we encourage users to then also *participate in the implementation*.

Request a feature

We welcome you to get more involved with the Matplotlib project! If you are new to contributing, we recommend that you first read our *contributing guide*. If you are contributing code or documentation, please follow our guides for setting up and managing a *development environment and workflow*. For code, documentation, or triage, please follow the corresponding *contribution guidelines*.

## 10.1 New contributors

### 10.1.1 Contribute

You've discovered a bug or something else you want to change in Matplotlib — excellent!

You've worked out a way to fix it — even better!

You want to tell us about it — best of all!

This project is a community effort, and everyone is welcome to contribute. Everyone within the community is expected to abide by our code of conduct.

Below, you can find a number of ways to contribute, and how to connect with the Matplotlib community.

### Get started

There is no pre-defined pathway for new contributors -- we recommend looking at existing issue and pull request discussions, and following the conversations during pull request reviews to get context. Or you can deep-dive into a subset of the code-base to understand what is going on.

### Do I really have something to contribute to Matplotlib?

100% yes! There are so many ways to contribute to our community. Take a look at the following sections to learn more.

There are a few typical new contributor profiles:

- **You are a Matplotlib user, and you see a bug, a potential improvement, or something that annoys you, and you can fix it.**

  You can search our issue tracker for an existing issue that describes your problem or open a new issue to inform us of the problem you observed and discuss the best approach to fix it. If your contributions would not be captured on GitHub (social media, communication, educational content), you can also reach out to us on gitter, Discourse or attend any of our community meetings.

- **You are not a regular Matplotlib user but a domain expert: you know about visualization, 3D plotting, design, technical writing, statistics, or some other field where Matplotlib could be improved.**

  Awesome -- you have a focus on a specific application and domain and can start there. In this case, maintainers can help you figure out the best implementation; open an issue or pull request with a starting point, and we'll be happy to discuss technical approaches.

  If you prefer, you can use the GitHub functionality for "draft" pull requests and request early feedback on whatever you are working on, but you should be aware that maintainers may not review your contribution unless it has the "Ready to review" state on GitHub.

- **You are new to Matplotlib, both as a user and contributor, and want to start contributing but have yet to develop a particular interest.**

  Having some previous experience or relationship with the library can be very helpful when making open-source contributions. It helps you understand why things are the way they are and how they *should* be. Having first-hand experience and context is valuable both for what you can bring to the conversation (and given the breadth of Matplotlib's usage, there is a good chance it is a unique context in any given conversation) and make it easier to understand where other people are coming from.

  Understanding the entire codebase is a long-term project, and nobody expects you to do this right away. If you are determined to get started with Matplotlib and want to learn, going through the basic

functionality, choosing something to focus on (3d, testing, documentation, animations, etc.) and gaining context on this area by reading the issues and pull requests touching these subjects is a reasonable approach.

### Code

You want to implement a feature or fix a bug or help with maintenance - much appreciated! Our library source code is found in:

- Python library code: `lib/`
- C-extension code: `src/`
- Tests: `lib/matplotlib/tests/`

Because many people use and work on Matplotlib, we have guidelines for keeping our code consistent and mitigating the impact of changes.

- *Coding guidelines*
- *API guidelines*
- *Pull request guidelines*

Code is contributed through pull requests, so we recommend that you start at *Start a pull request* If you get stuck, please reach out on the *Contributor incubator*

### Documentation

You as an end-user of Matplotlib can make a valuable contribution because you more clearly see the potential for improvement than a core developer. For example, you can:

- Fix a typo
- Clarify a docstring
- Write or update an *example plot*
- Write or update a comprehensive *tutorial*

Our code is documented inline in the source code files in `matplotlib/lib`. Our website structure mirrors our folder structure, meaning that a narrative document's URL roughly corresponds to its location in our folder structure:

using the library

- `galleries/plot_types/`
- `users/getting_started/`
- `galleries/user_explain/`
- `galleries/tutorials/`
- `galleries/examples/`

- `doc/api/`

information about the library

- `doc/install/`
- `doc/project/`
- `doc/devel/`
- `doc/users/resources/index.rst`
- `doc/users/faq.rst`

Other documentation is generated from the following external sources:

- matplotlib.org homepage: https://github.com/matplotlib/mpl-brochure-site
- cheat sheets: https://github.com/matplotlib/cheatsheets
- third party packages: https://github.com/matplotlib/mpl-third-party

Instructions and guidelines for contributing documentation are found in:

- *Write documentation*
- *Documentation style guide*
- *Guidelines for assigning tags to gallery examples*

Documentation is contributed through pull requests, so we recommend that you start at *Start a pull request*. If that feels intimidating, we encourage you to open an issue describing what improvements you would make. If you get stuck, please reach out on the *Contributor incubator*

### Community

Matplotlib's community is built by its members, if you would like to help out see our *Community management guide*.

It helps us if you spread the word: reference the project from your blog and articles or link to it from your website!

If Matplotlib contributes to a project that leads to a scientific publication, please cite us following the *Citing Matplotlib* guidelines.

If you have developed an extension to Matplotlib, please consider adding it to our third party package list.

### Get connected

When in doubt, we recommend going together! Get connected with our community of active contributors, many of whom felt just like you when they started out and are happy to welcome you and support you as you get to know how we work, and where things are.

### Contributor incubator

The incubator is our non-public communication channel for new contributors. It is a private gitter (chat) room moderated by core Matplotlib developers where you can get guidance and support for your first few PRs. It's a place where you can ask questions about anything: how to use git, GitHub, how our PR review process works, technical questions about the code, what makes for good documentation or a blog post, how to get involved in community work, or get a "pre-review" on your PR.

To join, please go to our public community channel, and ask to be added to `#incubator`. One of our core developers will see your message and will add you.

### New Contributors Meeting

Once a month, we host a meeting to discuss topics that interest new contributors. Anyone can attend, present, or sit in and listen to the call. Among our attendees are fellow new contributors, as well as maintainers, and veteran contributors, who are keen to support onboarding of new folks and share their experience. You can find our community calendar link at the Scientific Python website, and you can browse previous meeting notes on GitHub. We recommend joining the meeting to clarify any doubts, or lingering questions you might have, and to get to know a few of the people behind the GitHub handles □. You can reach out to us on gitter for any clarifications or suggestions. We □ feedback!

### Work on an issue

In general, the Matplotlib project does not assign issues. Issues are "assigned" or "claimed" by opening a PR; there is no other assignment mechanism. If you have opened such a PR, please comment on the issue thread to avoid duplication of work. Please check if there is an existing PR for the issue you are addressing. If there is, try to work with the author by submitting reviews of their code or commenting on the PR rather than opening a new PR; duplicate PRs are subject to being closed. However, if the existing PR is an outline, unlikely to work, or stalled, and the original author is unresponsive, feel free to open a new PR referencing the old one.

### Good first issues

While any contributions are welcome, we have marked some issues as particularly suited for new contributors by the label good first issue. These are well documented issues, that do not require a deep understanding of the internals of Matplotlib. The issues may additionally be tagged with a difficulty. `Difficulty: Easy` is suited for people with little Python experience. `Difficulty: Medium` and `Difficulty: Hard` require more programming experience. This could be for a variety of reasons, among them, though not necessarily all at the same time:

- The issue is in areas of the code base which have more interdependencies, or legacy code.
- It has less clearly defined tasks, which require some independent exploration, making suggestions, or follow-up discussions to clarify a good path to resolve the issue.
- It involves Python features such as decorators and context managers, which have subtleties due to our implementation decisions.

### Start a pull request

The preferred way to contribute to Matplotlib is to fork the main repository on GitHub, then submit a "pull request" (PR). You can do this by cloning a copy of the Maplotlib repository to your own computer, or alternatively using GitHub Codespaces, a cloud-based in-browser development environment that comes with the appropriated setup to contribute to Matplotlib.

### Workflow overview

A brief overview of the workflow is as follows.

1. Create an account on GitHub if you do not already have one.
2. Fork the project repository by clicking on the **Fork** button near the top of the page. This creates a copy of the code under your account on the GitHub server.
3. Set up a development environment:

#### Local development

Clone this copy to your local disk:

```
git clone https://github.com/<YOUR GITHUB USERNAME>/matplotlib.git
```

### Using GitHub Codespaces

Check out the Matplotlib repository and activate your development environment:

1. Open codespaces on your fork by clicking on the green "Code" button on the GitHub web interface and selecting the "Codespaces" tab.
2. Next, click on "Open codespaces on <your branch name>". You will be able to change branches later, so you can select the default `main` branch.
3. After the codespace is created, you will be taken to a new browser tab where you can use the terminal to activate a pre-defined conda environment called `mpl-dev`:

   ```
   conda activate mpl-dev
   ```

4. Install the local version of Matplotlib with:

   ```
   python -m pip install --no-build-isolation --editable .[dev]
   ```

   See *Setting up Matplotlib for development* for detailed instructions.

5. Create a branch to hold your changes:

   ```
   git checkout -b my-feature origin/main
   ```

   and start making changes. Never work in the `main` branch!

6. Work on this task using Git to do the version control. Codespaces persist for some time (check the documentation for details) and can be managed on https://github.com/codespaces. When you're done editing e.g., `lib/matplotlib/collections.py`, do:

   ```
   git add lib/matplotlib/collections.py
   git commit
   ```

   to record your changes in Git, then push them to your GitHub fork with:

   ```
   git push -u origin my-feature
   ```

### GitHub Codespaces workflows

If you need to open a GUI window with Matplotlib output on Codespaces, our configuration includes a lightweight Fluxbox-based desktop. You can use it by connecting to this desktop via your web browser. To do this:

1. **Press `F1` or `Ctrl/Cmd+Shift+P` and select**

   `Ports: Focus on Ports View` in the VSCode session to bring it into focus. Open the ports view in your tool, select the `noVNC` port, and click the Globe icon.

2. **In the browser that appears, click the Connect button and enter the desktop**

   password (`vscode` by default).

Check the GitHub instructions for more details on connecting to the desktop.

#### View documentation

If you also built the documentation pages, you can view them using Codespaces. Use the "Extensions" icon in the activity bar to install the "Live Server" extension. Locate the `doc/build/html` folder in the Explorer, right click the file you want to open and select "Open with Live Server."

#### Open a pull request on Matplotlib

Finally, go to the web page of *your fork* of the Matplotlib repo, and click **Compare & pull request** to send your changes to the maintainers for review. The base repository is `matplotlib/matplotlib` and the base branch is generally `main`. For more guidance, see GitHub's pull request tutorial.

For more detailed instructions on how to set up Matplotlib for development and best practices for contribution, see *Setting up Matplotlib for development* and *Development workflow*.

*Where should I start?*

*Where should I ask questions?*

*How do I work on an issue?*

*How do I start a pull request?*

Contribute code

Write documentation

Build community

## 10.2 Development environment

**Install**

### 10.2.1 Setting up Matplotlib for development

To set up Matplotlib for development follow these steps:

- *Fork the Matplotlib repository*
- *Retrieve the latest version of the code*
- *Create a dedicated environment*
- *Install Dependencies*

- *Install Matplotlib in editable mode*
  - *Build options*
- *Verify the Installation*
- *Install pre-commit hooks*


### Fork the Matplotlib repository

Matplotlib is hosted at https://github.com/matplotlib/matplotlib.git. If you plan on solving issues or submitting pull requests to the main Matplotlib repository, you should first *fork* this repository by visiting https://github.com/matplotlib/matplotlib.git and clicking on the `Fork` button on the top right of the page. See the GitHub documentation for more details.

### Retrieve the latest version of the code

Now that your fork of the repository lives under your GitHub username, you can retrieve the most recent version of the source code with one of the following commands (replace `<your-username>` with your GitHub username):

#### https

```
git clone https://github.com/<your-username>/matplotlib.git
```

#### ssh

```
git clone git@github.com:<your-username>/matplotlib.git
```

This requires you to setup an SSH key in advance, but saves you from typing your password at every connection.

This will place the sources in a directory `matplotlib` below your current working directory and set the remote name `origin` to point to your fork. Change into this directory before continuing:

```
cd matplotlib
```

Now set the remote name `upstream` to point to the Matplotlib main repository:

#### https

```
git remote add upstream https://github.com/matplotlib/matplotlib.git
```

#### ssh

```
git remote add upstream git@github.com:matplotlib/matplotlib.git
```

You can now use `upstream` to retrieve the most current snapshot of the source code, as described in *Development workflow*.

### Additional `git` and `GitHub` resources

For more information on `git` and `GitHub`, see:

- Git documentation
- GitHub-Contributing to a Project
- GitHub Skills
- Git for development
- Additional Git Resources
- Installing git
- Managing remote repositories
- https://tacaswell.github.io/think-like-git.html
- https://tom.preston-werner.com/2009/05/19/the-git-parable.html

### Create a dedicated environment

You should set up a dedicated environment to decouple your Matplotlib development from other Python and Matplotlib installations on your system.

The simplest way to do this is to use either Python's virtual environment venv or conda.

#### venv environment

Create a new venv environment with

```
python -m venv <file folder location>
```

and activate it with one of the following

```
source <file folder location>/bin/activate  # Linux/macOS
<file folder location>\Scripts\activate.bat  # Windows cmd.exe
<file folder location>\Scripts\Activate.ps1  # Windows PowerShell
```

On some systems, you may need to type `python3` instead of `python`. For a discussion of the technical reasons, see PEP-394.

Install the Python dependencies with

```
pip install -r requirements/dev/dev-requirements.txt
```

### conda environment

Create a new conda environment and install the Python dependencies with

```
conda env create -f environment.yml
```

You can use `mamba` instead of `conda` in the above command if you have mamba installed.

Activate the environment using

```
conda activate mpl-dev
```

Remember to activate the environment whenever you start working on Matplotlib.

### Install Dependencies

Most Python dependencies will be installed when *setting up the environment* but non-Python dependencies like C++ compilers, LaTeX, and other system applications must be installed separately.

### Install Matplotlib in editable mode

Install Matplotlib in editable mode from the `matplotlib` directory using the command

```
python -m pip install --verbose --no-build-isolation --editable ".[dev]"
```

The 'editable/develop mode' builds everything and places links in your Python environment so that Python will be able to import Matplotlib from your development source directory. This allows you to import your modified version of Matplotlib without having to re-install after changing a `.py` or compiled extension file.

When working on a branch that does not have Meson enabled, meaning it does not have PR #26621 in its history (log), you will have to reinstall from source each time you change any compiled extension code.

If the installation is not working, please consult the *troubleshooting guide*. If the guide does not offer a solution, please reach out via chat or *open an issue*.

### Build options

If you are working heavily with files that need to be compiled, you may want to inspect the compilation log. This can be enabled by setting the environment variable `MESONPY_EDITABLE_VERBOSE` or by setting the `editable-verbose` config during installation

```
python -m pip install --no-build-isolation --config-settings=editable-
 ↪verbose=true --editable .
```

For more information on installation and other configuration options, see the Meson Python editable installs guide.

For a list of the other environment variables you can set before install, see *Environment variables*.

### Verify the Installation

Run the following command to make sure you have correctly installed Matplotlib in editable mode. The command should be run when the virtual environment is activated:

```
python -c "import matplotlib; print(matplotlib.__file__)"
```

This command should return : `<matplotlib_local_repo>\lib\matplotlib\__init__.py`

We encourage you to run tests and build docs to verify that the code installed correctly and that the docs build cleanly, so that when you make code or document related changes you are aware of the existing issues beforehand.

- Run test cases to verify installation *Testing*
- Verify documentation build *Write documentation*

### Install pre-commit hooks

pre-commit hooks save time in the review process by identifying issues with the code before a pull request is formally opened. Most hooks can also aide in fixing the errors, and the checks should have corresponding *development workflow* and *pull request* guidelines. Hooks are configured in .pre-commit-config.yaml and include checks for spelling and formatting, flake 8 conformity, accidentally committed files, import order, and incorrect branching.

Install pre-commit hooks

```
python -m pip install pre-commit
pre-commit install
```

Hooks are run automatically after the `git commit` stage of the *editing workflow*. When a hook has found and fixed an error in a file, that file must be *staged and committed* again.

Hooks can also be run manually. All the hooks can be run, in order as listed in `.pre-commit-config.yaml`, against the full codebase with

```
pre-commit run --all-files
```

To run a particular hook manually, run `pre-commit run` with the hook id

```
pre-commit run <hook id> --all-files
```

**Workflow**

### 10.2.2 Development workflow

#### Workflow summary

To keep your work well organized, with readable history, and in turn make it easier for project maintainers (that might be you) to see what you've done, and why you did it, we recommend the following:

- Don't make changes in your local `main` branch!
- Before starting a new set of changes, fetch all changes from `upstream/main`, and start a new *feature branch* from that.
- Make a new branch for each feature or bug fix — "one task, one branch".
- Name your branch for the purpose of the changes - e.g. `bugfix-for-issue-14` or `refactor-database-code`.
- If you get stuck, reach out on Gitter or discourse.
- When you're ready or need feedback on your code, open a pull request so that the Matplotlib developers can give feedback and eventually include your suggested code into the `main` branch.

#### Update the `main` branch

First make sure you have followed *Setting up Matplotlib for development*.

From time to time you should fetch the upstream changes from GitHub:

```
git fetch upstream
```

This will pull down any commits you don't have, and set the remote branches to point to the right commit.

#### Make a new feature branch

When you are ready to make some changes to the code, you should start a new branch. Branches that are for a collection of related edits are often called 'feature branches'.

Making a new branch for each set of related changes will make it easier for someone reviewing your branch to see what you are doing.

Choose an informative name for the branch to remind yourself and the rest of us what the changes in the branch are for. For example `add-ability-to-fly`, or `bugfix-for-issue-42`.

```
# Update the main branch
git fetch upstream
# Make new feature branch starting at current main
git branch my-new-feature upstream/main
git checkout my-new-feature
```

If you started making changes on your local `main` branch, you can convert the branch to a feature branch by renaming it:

```
git branch -m <newname>
```

Generally, you will want to keep your feature branches on your public GitHub fork of Matplotlib. To do this, you `git push` this new branch up to your GitHub repo. Generally, if you followed the instructions in these pages, and by default, git will have a link to your fork of the GitHub repo, called `origin`. You push up to your own fork with:

```
git push origin my-new-feature
```

In git >= 1.7 you can ensure that the link is correctly set by using the `--set-upstream` option:

```
git push --set-upstream origin my-new-feature
```

From now on git will know that `my-new-feature` is related to the `my-new-feature` branch in the GitHub repo.

If you first opened the pull request from your `main` branch and then converted it to a feature branch, you will need to close the original pull request and open a new pull request from the renamed branch. See GitHub: working with branches.

### The editing workflow

1. Make some changes
2. Save the changes
3. See which files have changed with `git status`. You'll see a listing like this one:

```
# On branch ny-new-feature
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working␣
 ↪directory)
#
#  modified:   README
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#  INSTALL
no changes added to commit (use "git add" and/or "git commit -a")
```

4. Check what the actual changes are with `git diff`.
5. Add any new files to version control `git add new_file_name`.
6. To commit **all** modified files into the local copy of your repo, type:

   ```
   git commit -am 'A commit message'
   ```

   Note the `-am` options to `commit`. The `m` flag signals that you are going to type a message on the command line. The `a` flag stages every file that has been modified, except files listed in `.gitignore`. For more information, see why the -a flag? and the git commit manual page.
7. To push the changes up to your forked repo on GitHub, do a `git push`.

### Open a pull request

When you are ready to ask for someone to review your code and consider a merge, submit your Pull Request (PR).

Enter a title for the set of changes with some explanation of what you've done. Mention anything you'd like particular attention for - such as a complicated change or some code you are not happy with.

If you don't think your request is ready to be merged, just say so in your pull request message and use the "Draft PR" feature of GitHub. This is a good way of getting some preliminary code review.

### Update a pull request

When updating your pull request after making revisions, instead of adding new commits, please consider amending your initial commit(s) to keep the commit history clean.

You can achieve this by using

```
git commit -a --amend --no-edit
git push [your-remote-repo] [your-branch] --force-with-lease
```

### Manage commit history

### Explore your repository

To see a graphical representation of the repository branches and commits:

```
gitk --all
```

To see a linear list of commits for this branch:

```
git log
```

### Recover from mistakes

Sometimes, you mess up merges or rebases. Luckily, in git it is relatively straightforward to recover from such mistakes.

If you mess up during a rebase:

```
git rebase --abort
```

If you notice you messed up after the rebase:

```
# reset branch back to the saved point
git reset --hard tmp
```

If you forgot to make a backup branch:

```
# look at the reflog of the branch
git reflog show cool-feature

8630830 cool-feature@{0}: commit: BUG: io: close file handles immediately
278dd2a cool-feature@{1}: rebase finished: refs/heads/my-feature-branch onto␣
 ↪11ee694744f2552d
26aa21a cool-feature@{2}: commit: BUG: lib: make seek_gzip_factory not leak␣
 ↪gzip obj
...

# reset the branch to where it was before the botched rebase
git reset --hard cool-feature@{2}
```

### Rewrite commit history

---

**Note:** Do this only for your own feature branches.

---

Is there an embarrassing typo in a commit you made? Or perhaps you made several false starts you don't want posterity to see.

This can be done via *interactive rebasing*.

Suppose that the commit history looks like this:

```
git log --oneline
eadc391 Fix some remaining bugs
a815645 Modify it so that it works
2dec1ac Fix a few bugs + disable
13d7934 First implementation
6ad92e5 * masked is now an instance of a new object, MaskedConstant
29001ed Add pre-nep for a copule of structured_array_extensions.
...
```

and `6ad92e5` is the last commit in the `cool-feature` branch. Suppose we want to make the following changes:

- Rewrite the commit message for `13d7934` to something more sensible.
- Combine the commits `2dec1ac`, `a815645`, `eadc391` into a single one.

We do as follows:

```
# make a backup of the current state
git branch tmp HEAD
# interactive rebase
git rebase -i 6ad92e5
```

This will open an editor with the following text in it:

```
pick 13d7934 First implementation
pick 2dec1ac Fix a few bugs + disable
pick a815645 Modify it so that it works
pick eadc391 Fix some remaining bugs

# Rebase 6ad92e5..eadc391 onto 6ad92e5
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#
# If you remove a line here THAT COMMIT WILL BE LOST.
# However, if you remove everything, the rebase will be aborted.
#
```

To achieve what we want, we will make the following changes to it:

```
r 13d7934 First implementation
pick 2dec1ac Fix a few bugs + disable
f a815645 Modify it so that it works
f eadc391 Fix some remaining bugs
```

This means that (i) we want to edit the commit message for `13d7934`, and (ii) collapse the last three commits into one. Now we save and quit the editor.

Git will then immediately bring up an editor for editing the commit message. After revising it, we get the output:

```
[detached HEAD 721fc64] FOO: First implementation
 2 files changed, 199 insertions(+), 66 deletions(-)
[detached HEAD 0f22701] Fix a few bugs + disable
 1 files changed, 79 insertions(+), 61 deletions(-)
Successfully rebased and updated refs/heads/my-feature-branch.
```

and now, the history looks like this:

```
0f22701 Fix a few bugs + disable
721fc64 ENH: Sophisticated feature
6ad92e5 * masked is now an instance of a new object, MaskedConstant
```

If it went wrong, recovery is again possible as explained *above*.

If you have not yet pushed this branch to github, you can carry on as normal, however if you *have* already pushed this commit see *Push with force* for how to replace your already published commits with the new ones.

### Rebase onto `upstream/main`

Let's say you thought of some work you'd like to do. You *Update the main branch* and *Make a new feature branch* called `cool-feature`. At this stage, `main` is at some commit, let's call it E. Now you make some new commits on your `cool-feature` branch, let's call them A, B, C. Maybe your changes take a while, or you come back to them after a while. In the meantime, `main` has progressed from commit E to commit (say) G:

```
      A---B---C cool-feature
     /
D---E---F---G main
```

At this stage you consider merging `main` into your feature branch, and you remember that this page sternly advises you not to do that, because the history will get messy. Most of the time, you can just ask for a review without worrying about whether `main` has got a little ahead; however sometimes, the changes in `main` might affect your changes, and you need to harmonize them. In this situation you may prefer to do a rebase.

`rebase` takes your changes (A, B, C) and replays them as if they had been made to the current state of `main`. In other words, in this case, it takes the changes represented by A, B, C and replays them on top of G. After the rebase, your history will look like this:

```
              A'--B'--C' cool-feature
             /
D---E---F---G main
```

See rebase without tears for more detail.

To do a rebase on `upstream/main`:

```
# Fetch changes from upstream/main
git fetch upstream
# go to the feature branch
git checkout cool-feature
# make a backup in case you mess up
git branch tmp cool-feature
# rebase cool-feature onto main
git rebase --onto upstream/main upstream/main cool-feature
```

In this situation, where you are already on branch `cool-feature`, the last command can be written more succinctly as:

```
git rebase upstream/main
```

When all looks good, you can delete your backup branch:

```
git branch -D tmp
```

If it doesn't look good you may need to have a look at *Recover from mistakes*.

If you have made changes to files that have also changed in `main`, this may generate merge conflicts that you need to resolve - see the git rebase man page for some instructions at the end of the "Description" section. There is some related help on merging in the git user manual - see resolving a merge.

If you have not yet pushed this branch to github, you can carry on as normal, however if you *have* already pushed this commit see *Push with force* for how to replace your already published commits with the new ones.

### Push with force

If you have in some way re-written already pushed history (e.g. via *Rewrite commit history* or *Rebase onto upstream/main*) leaving you with a git history that looks something like

```
  A'--E cool-feature
 /
D---A---B---C origin/cool-feature
```

where you have pushed the commits `A,B,C` to your fork on GitHub (under the remote name *origin*) but now have the commits `A'` and `E` on your local branch *cool-feature*. If you try to push the new commits to GitHub, it will fail and show an error that looks like

```
$ git push
Pushing to github.com:origin/matplotlib.git
To github.com:origin/matplotlib.git
 ! [rejected]              cool_feature -> cool_feature (non-fast-forward)
error: failed to push some refs to 'github.com:origin/matplotlib.git'
hint: Updates were rejected because the tip of your current branch is behind
hint: its remote counterpart. Integrate the remote changes (e.g.
hint: 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

If this push had succeeded, the commits `A`, `B`, and `C` would no longer be referenced by any branch and they would be discarded:

```
D---A'---E cool-feature, origin/cool-feature
```

By default `git push` helpfully tries to protect you from accidentally discarding commits by rejecting the push to the remote. When this happens, GitHub also adds the helpful suggestion to pull the remote changes and then try pushing again. In some cases, such as if you and a colleague are both committing and pushing to the same branch, this is a correct course of action.

However, in the case of having intentionally re-written history, we *want* to discard the commits on the remote and replace them with the new-and-improved versions from our local branch. In this case, what we want to do is

```
$ git push --force-with-lease
```

which tells git you are aware of the risks and want to do the push anyway. We recommend using `--force-with-lease` over the `--force` flag. The `--force` will do the push no matter what, whereas `--force-with-lease` will only do the push if the remote branch is where the local `git` client thought it was.

Be judicious with force-pushing. It is effectively re-writing published history, and if anyone has fetched the old commits, it will have a different view of history which can cause confusion.

### Automated tests

Whenever a pull request is created or updated, various automated test tools will run on all supported platforms and versions of Python.

- tox is not used in the automated testing. It is supported for testing locally.
- Codecov and CodeQL are currently for information only. Their failure is not necessarily a blocker.

Make sure the Linting, GitHub Actions, AppVeyor, CircleCI, and Azure pipelines are passing before merging. All checks are listed at the bottom of the GitHub page of your pull request.

| Name | Check | Tips for finding cause of failure |
|---|---|---|
| Linting | *code style* | Errors are displayed as annotations on the pull request diff. |
| Mypy | *static type hints* | Errors are displayed as annotations on the pull request diff. |
| Stubtest | | |
| CircleCI | *documentation build* | Search the CircleCI log for `WARNING`. |
| | *tests* | |
| GitHub Actions<br>AppVeyor<br>Azure pipelines | | Search the log for `FAILURES`. Subsequent section should contain information on failed tests.<br><br>On Azure, find the images as *artifacts* of the Azure job:<br>1. Click *Details* on the check on the GitHub PR page.<br>2. Click *View more details on Azure Pipelines* to go to Azure.<br>3. On the overview page *artifacts* are listed in the section *Related*. |

#### Skip CI checks

If you know only a subset of CI checks need to be run, you can skip unneeded CI checks on individual commits by including the following strings in the commit message:

| String | Effect | Notes |
|---|---|---|
| `[ci doc]` | Only run documentation checks. | For when you have only changed documentation. `[ci doc]` is applied automatically when the changes are only to files in `doc/**/` or `galleries/**/` |
| `[skip doc]` | Skip documentation checks. | For when you didn't change documentation. |
| `[skip appveyor]` | Skip AppVeyor run. | Substring must be in first line of commit message. |
| `[skip azp]` | Skip Azure Pipelines. | |
| `[skip actions]` | Skip GitHub Actions. | |
| `[skip ci]` | Skip all CI checks. | Use only for changes where documentation checks and unit tests do not apply. |

`[skip actions]` and `[skip ci]` only skip Github Actions CI workflows that are triggered on `on: push` and `on: pull_request` events. For more information, see Skipping workflow runs.

### 10.2.3 Troubleshooting

For guidance on debugging an installation, see *Frequently asked questions*.

#### Problems with git

First, make sure you have a clean build and install (see *How to completely remove Matplotlib*), get the latest git update, install it and run a simple test script in debug mode:

```
rm -rf /path/to/site-packages/matplotlib*
git clean -xfd
git pull
python -m pip install -v . > build.out
python -c "from pylab import *; set_loglevel('debug'); plot(); show()" > run.
↪out
```

and post `build.out` and `run.out` to the matplotlib-devel mailing list (please do not post git problems to the users list).

Of course, you will want to clearly describe your problem, what you are expecting and what you are getting, but often a clean build and install will help. See also *Get help*.

#### Unlink of file `*/_c_internal_utils.cp311-win_amd64.pyd` failed

The DLL files may be loaded by multiple running instances of Matplotlib; therefore check that Matplotlib is not running in any other application before trying to unlink this file. Multiple versions of Matplotlib can be linked to the same DLL, for example a development version installed in a development conda environment and a stable version running in a Jupyter notebook. To resolve this error, fully close all running instances of Matplotlib.

#### Windows compilation errors

If the compiled extensions are not building on Windows due to errors in linking to Windows' header files, for example `../../src/_tkagg.cpp:133:10: error: 'WM_DPICHANGED' was not declared in this scope`, you should check which compiler Meson is using:

```
Build type: native build
Project name: matplotlib
Project version: 3.9.0.dev0
C compiler for the host machine: cc (gcc 7.2.0 "cc (Rev1, Built by MSYS2␣
↪project) 7.2.0")
C linker for the host machine: cc ld.bfd 2.29.1
C++ compiler for the host machine: c++ (gcc 7.2.0 "c++ (Rev1, Built by MSYS2␣
↪project) 7.2.0")
C++ linker for the host machine: c++ ld.bfd 2.29.1
```

Our *dependencies* documentation lists the minimum header version if you intended to use `MSYS2`. If you intended to use `MSVC` then you may need to force Meson to use MSVC.

## 10.3 Policies and guidelines

**Code**

### 10.3.1 Coding guidelines

We appreciate these guidelines being followed because it improves the readability, consistency, and maintainability of the code base.

---

**API guidelines**

If adding new features, changing behavior or function signatures, or removing public interfaces, please consult the *API guidelines*.

---

### PEP8, as enforced by flake8

Formatting should follow the recommendations of PEP8, as enforced by flake8. Matplotlib modifies PEP8 to extend the maximum line length to 88 characters. You can check flake8 compliance from the command line with

```
python -m pip install flake8
flake8 /path/to/module.py
```

or your editor may provide integration with it. Note that Matplotlib intentionally does not use the black auto-formatter (1), in particular due to its inability to understand the semantics of mathematical expressions (2, 3).

### Package imports

Import the following modules using the standard scipy conventions:

```
import numpy as np
import numpy.ma as ma
import matplotlib as mpl
import matplotlib.pyplot as plt
import matplotlib.cbook as cbook
import matplotlib.patches as mpatches
```

In general, Matplotlib modules should **not** import *rcParams* using `from matplotlib import rcParams`, but rather access it as `mpl.rcParams`. This is because some modules are imported very early, before the *rcParams* singleton is constructed.

### Variable names

When feasible, please use our internal variable naming convention for objects of a given class and objects of any child class:

| base class | variable | multiples |
|---|---|---|
| *FigureBase* | fig | |
| *Axes* | ax | |
| *Transform* | trans | trans_<source>_<target><br>trans_<source> when target is screen |

Generally, denote more than one instance of the same class by adding suffixes to the variable names. If a format isn't specified in the table, use numbers or letters as appropriate.

### Type hints

If you add new public API or change public API, update or add the corresponding mypy type hints. We generally use stub files (`*.pyi`) to store the type information; for example `colors.pyi` contains the type information for `colors.py`. A notable exception is `pyplot.py`, which is type hinted inline.

Type hints are checked by the mypy *pre-commit hook*, can often be verified by running `tox -e stubtest`.

### New modules and files: installation

- If you have added new files or directories, or reorganized existing ones, make sure the new files are included in the `meson.build` in the corresponding directories.
- New modules *may* be typed inline or using parallel stub file like existing modules.

### C/C++ extensions

- Extensions may be written in C or C++.
- Code style should conform to PEP7 (understanding that PEP7 doesn't address C++, but most of its admonitions still apply).
- Python/C interface code should be kept separate from the core C/C++ code. The interface code should be named `FOO_wrap.cpp` or `FOO_wrapper.cpp`.
- Header file documentation (aka docstrings) should be in Numpydoc format. We don't plan on using automated tools for these docstrings, and the Numpydoc format is well understood in the scientific Python community.
- C/C++ code in the `extern/` directory is vendored, and should be kept close to upstream whenever possible. It can be modified to fix bugs or implement new features only if the required changes cannot be made elsewhere in the codebase. In particular, avoid making style fixes to it.

### Keyword argument processing

Matplotlib makes extensive use of `**kwargs` for pass-through customizations from one function to another. A typical example is *`text`*. The definition of *`matplotlib.pyplot.text`* is a simple pass-through to *`matplotlib.axes.Axes.text`*:

```
# in pyplot.py
def text(x, y, s, fontdict=None, **kwargs):
    return gca().text(x, y, s, fontdict=fontdict, **kwargs)
```

*`matplotlib.axes.Axes.text`* (simplified for illustration) just passes all `args` and `kwargs` on to `matplotlib.text.Text.__init__`:

```
# in axes/_axes.py
def text(self, x, y, s, fontdict=None, **kwargs):
    t = Text(x=x, y=y, text=s, **kwargs)
```

and `matplotlib.text.Text.__init__` (again, simplified) just passes them on to the *`matplotlib.artist.Artist.update`* method:

```
# in text.py
def __init__(self, x=0, y=0, text='', **kwargs):
    super().__init__()
    self.update(kwargs)
```

`update` does the work looking for methods named like `set_property` if `property` is a keyword argument. i.e., no one looks at the keywords, they just get passed through the API to the artist constructor which looks for suitably named methods and calls them with the value.

As a general rule, the use of `**kwargs` should be reserved for pass-through keyword arguments, as in the example above. If all the keyword args are to be used in the function, and not passed on, use the key/value keyword args in the function definition rather than the `**kwargs` idiom.

In some cases, you may want to consume some keys in the local function, and let others pass through. Instead of popping arguments to use off `**kwargs`, specify them as keyword-only arguments to the local function. This makes it obvious at a glance which arguments will be consumed in the function. For example, in *`plot()`*, `scalex` and `scaley` are local arguments and the rest are passed on as *`Line2D()`* keyword arguments:

```
# in axes/_axes.py
def plot(self, *args, scalex=True, scaley=True, **kwargs):
    lines = []
    for line in self._get_lines(*args, **kwargs):
        self.add_line(line)
        lines.append(line)
```

### Using logging for debug messages

Matplotlib uses the standard Python `logging` library to write verbose warnings, information, and debug messages. Please use it! In all those places you write `print` calls to do your debugging, try using `logging.debug` instead!

To include `logging` in your module, at the top of the module, you need to `import logging`. Then calls in your code like:

```
_log = logging.getLogger(__name__)  # right after the imports

# code
# more code
_log.info('Here is some information')
_log.debug('Here is some more detailed information')
```

will log to a logger named `matplotlib.yourmodulename`.

If an end-user of Matplotlib sets up `logging` to display at levels more verbose than `logging.WARNING` in their code with the Matplotlib-provided helper:

```
plt.set_loglevel("debug")
```

or manually with

```
import logging
logging.basicConfig(level=logging.DEBUG)
import matplotlib.pyplot as plt
```

Then they will receive messages like

```
DEBUG:matplotlib.backends:backend MacOSX version unknown
DEBUG:matplotlib.yourmodulename:Here is some information
DEBUG:matplotlib.yourmodulename:Here is some more detailed information
```

Avoid using pre-computed strings (`f-strings`, `str.format`,etc.) for logging because of security and performance issues, and because they interfere with style handlers. For example, use `_log.error('hello %s', 'world')` rather than `_log.error('hello {}'.format('world'))` or `_log.error(f'hello {s}')`.

### Which logging level to use?

There are five levels at which you can emit messages.

- `logging.critical` and `logging.error` are really only there for errors that will end the use of the library but not kill the interpreter.
- `logging.warning` and *`_api.warn_external`* are used to warn the user, see below.
- `logging.info` is for information that the user may want to know if the program behaves oddly. They are not displayed by default. For instance, if an object isn't drawn because its position is `NaN`, that can usually be ignored, but a mystified user could call `logging.basicConfig(level=logging.INFO)` and get an error message that says why.
- `logging.debug` is the least likely to be displayed, and hence can be the most verbose. "Expected" code paths (e.g., reporting normal intermediate steps of layouting or rendering) should only log at this level.

By default, `logging` displays all log messages at levels higher than `logging.WARNING` to `sys.stderr`.

The logging tutorial suggests that the difference between `logging.warning` and *`_api.warn_external`* (which uses `warnings.warn`) is that *`_api.warn_external`* should be used for things the user must change to stop the warning (typically in the source), whereas `logging.warning` can be more persistent. Moreover, note that *`_api.warn_external`* will by default only emit a given warning *once* for each line of user code, whereas `logging.warning` will display the message every time it is called.

By default, `warnings.warn` displays the line of code that has the `warn` call. This usually isn't more informative than the warning message itself. Therefore, Matplotlib uses *`_api.warn_external`* which uses `warnings.warn`, but goes up the stack and displays the first line of code outside of Matplotlib. For example, for the module:

```
# in my_matplotlib_module.py
import warnings

def set_range(bottom, top):
    if bottom == top:
        warnings.warn('Attempting to set identical bottom==top')
```

running the script:

```
from matplotlib import my_matplotlib_module
my_matplotlib_module.set_range(0, 0)  # set range
```

will display

```
UserWarning: Attempting to set identical bottom==top
warnings.warn('Attempting to set identical bottom==top')
```

Modifying the module to use *_api.warn_external*:

```
from matplotlib import _api

def set_range(bottom, top):
    if bottom == top:
        _api.warn_external('Attempting to set identical bottom==top')
```

and running the same script will display

```
UserWarning: Attempting to set identical bottom==top
my_matplotlib_module.set_range(0, 0)  # set range
```

### Licenses for contributed code

Matplotlib only uses BSD compatible code. If you bring in code from another project make sure it has a PSF, BSD, MIT or compatible license (see the Open Source Initiative licenses page for details on individual licenses). If it doesn't, you may consider contacting the author and asking them to relicense it. GPL and LGPL code are not acceptable in the main code base, though we are considering an alternative way of distributing L/GPL code through an separate channel, possibly a toolkit. If you include code, make sure you include a copy of that code's license in the license directory if the code's license requires you to distribute the license with it. Non-BSD compatible licenses are acceptable in Matplotlib toolkits (e.g., basemap), but make sure you clearly state the licenses you are using.

### Why BSD compatible?

The two dominant license variants in the wild are GPL-style and BSD-style. There are countless other licenses that place specific restrictions on code reuse, but there is an important difference to be considered in the GPL and BSD variants. The best known and perhaps most widely used license is the GPL, which in addition to granting you full rights to the source code including redistribution, carries with it an extra obligation. If you use GPL code in your own code, or link with it, your product must be released under a GPL compatible license. i.e., you are required to give the source code to other people and give them the right to redistribute it as well. Many of the most famous and widely used open source projects are released under the GPL, including linux, gcc, emacs and sage.

The second major class are the BSD-style licenses (which includes MIT and the python PSF license). These basically allow you to do whatever you want with the code: ignore it, include it in your own open source project, include it in your proprietary product, sell it, whatever. python itself is released under a BSD compatible license, in the sense that, quoting from the PSF license page:

```
There is no GPL-like "copyleft" restriction. Distributing
binary-only versions of Python, modified or not, is allowed. There
is no requirement to release any of your source code. You can also
write extension modules for Python and provide them only in binary
form.
```

Famous projects released under a BSD-style license in the permissive sense of the last paragraph are the BSD operating system, python and TeX.

There are several reasons why early Matplotlib developers selected a BSD compatible license. Matplotlib is a python extension, and we choose a license that was based on the python license (BSD compatible). Also, we wanted to attract as many users and developers as possible, and many software companies will not use GPL code in software they plan to distribute, even those that are highly committed to open source development, such as enthought, out of legitimate concern that use of the GPL will "infect" their code base by its viral nature. In effect, they want to retain the right to release some proprietary code. Companies and institutions who use Matplotlib often make significant contributions, because they have the resources to get a job done, even a boring one. Two of the Matplotlib backends (FLTK and WX) were contributed by private companies. The final reason behind the licensing choice is compatibility with the other python extensions for scientific computing: ipython, numpy, scipy, the enthought tool suite and python itself are all distributed under BSD compatible licenses.

### Licenses for contributed code

Matplotlib only uses BSD compatible code. If you bring in code from another project make sure it has a PSF, BSD, MIT or compatible license (see the Open Source Initiative licenses page for details on individual licenses). If it doesn't, you may consider contacting the author and asking them to relicense it. GPL and LGPL code are not acceptable in the main code base, though we are considering an alternative way of distributing L/GPL code through an separate channel, possibly a toolkit. If you include code, make sure you include a copy of that code's license in the license directory if the code's license requires you to distribute the license with it. Non-BSD compatible licenses are acceptable in Matplotlib toolkits (e.g., basemap), but make sure you clearly state the licenses you are using.

#### Why BSD compatible?

The two dominant license variants in the wild are GPL-style and BSD-style. There are countless other licenses that place specific restrictions on code reuse, but there is an important difference to be considered in the GPL and BSD variants. The best known and perhaps most widely used license is the GPL, which in addition to granting you full rights to the source code including redistribution, carries with it an extra obligation. If you use GPL code in your own code, or link with it, your product must be released under a GPL compatible license. i.e., you are required to give the source code to other people and give them the right to redistribute it as well. Many of the most famous and widely used open source projects are released under the GPL, including linux, gcc, emacs and sage.

The second major class are the BSD-style licenses (which includes MIT and the python PSF license). These basically allow you to do whatever you want with the code: ignore it, include it in your own open source project, include it in your proprietary product, sell it, whatever. python itself is released under a BSD compatible license, in the sense that, quoting from the PSF license page:

```
There is no GPL-like "copyleft" restriction. Distributing
binary-only versions of Python, modified or not, is allowed. There
is no requirement to release any of your source code. You can also
write extension modules for Python and provide them only in binary
form.
```

Famous projects released under a BSD-style license in the permissive sense of the last paragraph are the BSD operating system, python and TeX.

There are several reasons why early Matplotlib developers selected a BSD compatible license. Matplotlib is a python extension, and we choose a license that was based on the python license (BSD compatible). Also, we wanted to attract as many users and developers as possible, and many software companies will not use GPL code in software they plan to distribute, even those that are highly committed to open source development, such as enthought, out of legitimate concern that use of the GPL will "infect" their code base by its viral nature. In effect, they want to retain the right to release some proprietary code. Companies and institutions who use Matplotlib often make significant contributions, because they have the resources to get a job done, even a boring one. Two of the Matplotlib backends (FLTK and WX) were contributed by private companies. The final reason behind the licensing choice is compatibility with the other python extensions for scientific computing: ipython, numpy, scipy, the enthought tool suite and python itself are all distributed under BSD compatible licenses.

### 10.3.2 API guidelines

API consistency and stability are of great value; Therefore, API changes (e.g. signature changes, behavior changes, removals) will only be conducted if the added benefit is worth the effort of adapting existing code.

Because we are a visualization library, our primary output is the final visualization the user sees; therefore, the appearance of the figure is part of the API and any changes, either semantic or *aesthetic*, are backwards-incompatible API changes.

### Default color changes

As discussed at length elsewhere , `jet` is an empirically bad colormap and should not be the default colormap. Due to the position that changing the appearance of the plot breaks backward compatibility, this change has been put off for far longer than it should have been. In addition to changing the default color map we plan to take the chance to change the default color-cycle on plots and to adopt a different colormap for filled plots (`imshow`, `pcolor`, `contourf`, etc) and for scatter like plots.

### Default heat map colormap

The choice of a new colormap is fertile ground to bike-shedding ("No, it should be _this_ color") so we have a proposed set criteria (via Nathaniel Smith) to evaluate proposed colormaps.

- it should be a sequential colormap, because diverging colormaps are really misleading unless you know where the "center" of the data is, and for a default colormap we generally won't.
- it should be perceptually uniform, i.e., human subjective judgments of how far apart nearby colors are should correspond as linearly as possible to the difference between the numerical values they represent, at least locally.
- it should have a perceptually uniform luminance ramp, i.e. if you convert to greyscale it should still be uniform. This is useful both in practical terms (greyscale printers are still a thing!) and because luminance is a very strong and natural cue to magnitude.
- it should also have some kind of variation in hue, because hue variation is a really helpful additional cue to perception, having two cues is better than one, and there's no reason not to do it.
- the hue variation should be chosen to produce reasonable results even for viewers with the more common types of colorblindness. (Which rules out things like red-to-green.)
- For bonus points, it would be nice to choose a hue ramp that still works if you throw away the luminance variation, because then we could use the version with varying luminance for 2d plots, and the version with just hue variation for 3d plots. (In 3d plots you really want to reserve the luminance channel for lighting/shading, because your brain is *really* good at extracting 3d shape from luminance variation. If the 3d surface itself has massively varying luminance then this screws up the ability to see shape.)
- Not infringe any existing IP

### Example script

### Proposed colormaps

### Default scatter colormap

For heat-map like applications it can be desirable to cover as much of the luminance scale as possible, however when colormapping markers, having markers too close to white can be a problem. For that reason we propose using a different (but maybe related) colormap to the heat map for marker-based. The design parameters are the same as above, only with a more limited luminance variation.

### Example script

```
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(1234)

fig, (ax1, ax2) = plt.subplots(1, 2)

N = 50
x = np.random.rand(N)
y = np.random.rand(N)
colors = np.random.rand(N)
area = np.pi * (15 * np.random.rand(N))**2  # 0 to 15 point radiuses

ax1.scatter(x, y, s=area, c=colors, alpha=0.5)


X,Y = np.meshgrid(np.arange(0, 2*np.pi, .2),
                  np.arange(0, 2*np.pi, .2))
U = np.cos(X)
V = np.sin(Y)
Q = ax2.quiver(X, Y, U, V, units='width')
qd = np.random.rand(np.prod(X.shape))
Q.set_array(qd)
```

## Proposed colormaps

### Color cycle / qualitative colormap

When plotting lines it is frequently desirable to plot multiple lines or artists which need to be distinguishable, but there is no inherent ordering.

### Example script

```
import numpy as np
import matplotlib.pyplot as plt

fig, (ax1, ax2) = plt.subplots(1, 2)

x = np.linspace(0, 1, 10)

for j in range(10):
    ax1.plot(x, x * j)


th = np.linspace(0, 2*np.pi, 1024)
```

(continues on next page)

(continued from previous page)

```
for j in np.linspace(0, np.pi, 10):
    ax2.plot(th, np.sin(th + j))

ax2.set_xlim(0, 2*np.pi)
```

### Proposed color cycle

### Add new API and features

Every new function, parameter and attribute that is not explicitly marked as private (i.e., starts with an underscore) becomes part of Matplotlib's public API. As discussed above, changing the existing API is cumbersome. Therefore, take particular care when adding new API:

- Mark helper functions and internal attributes as private by prefixing them with an underscore.
- Carefully think about good names for your functions and variables.
- Try to adopt patterns and naming conventions from existing parts of the Matplotlib API.
- Consider making as many arguments keyword-only as possible. See also API Evolution the Right Way -- Add Parameters Compatibly.

### Deprecate API

API changes in Matplotlib have to be performed following the deprecation process below, except in very rare circumstances as deemed necessary by the development team. Generally API deprecation happens in two stages:

- **introduce:** warn users that the API *will* change
- **expire:** API *is* changed as described in the introduction period

This ensures that users are notified before the change will take effect and thus prevents unexpected breaking of code.

### Rules

- Deprecations are targeted at the next *meso release* (e.g. 3.x)
- Deprecated API is generally removed (expired) two point-releases after introduction of the deprecation. Longer deprecations can be imposed by core developers on a case-by-case basis to give more time for the transition
- The old API must remain fully functional during the deprecation period
- If alternatives to the deprecated API exist, they should be available during the deprecation period
- If in doubt, decisions about API changes are finally made by the API consistency lead developer.

### Introduce deprecation

1. Create *deprecation notice*
2. If possible, issue a `MatplotlibDeprecationWarning` when the deprecated API is used. There are a number of helper tools for this:
   - Use `_api.warn_deprecated()` for general deprecation warnings
   - Use the decorator `@_api.deprecated` to deprecate classes, functions, methods, or properties
   - Use `@_api.deprecate_privatize_attribute` to annotate deprecation of attributes while keeping the internal private version.
   - To warn on changes of the function signature, use the decorators `@_api.delete_parameter`, `@_api.rename_parameter`, and `@_api.make_keyword_only`

   All these helpers take a first parameter *since*, which should be set to the next point release, e.g. "3.x".

   You can use standard rst cross references in *alternative*.
3. Make appropriate changes to the type hints in the associated `.pyi` file. The general guideline is to match runtime reported behavior.
   - Items marked with `@_api.deprecated` or `@_api.deprecate_privatize_attribute` are generally kept during the expiry period, and thus no changes are needed on introduction.
   - Items decorated with `@_api.rename_parameter` or `@_api.make_keyword_only` report the *new* (post deprecation) signature at runtime, and thus *should* be updated on introduction.
   - Items decorated with `@_api.delete_parameter` should include a default value hint for the deleted parameter, even if it did not previously have one (e.g. `param: <type> = ...`).

### Expire deprecation

1. Create *deprecation announcement*. For the content, you can usually copy the deprecation notice and adapt it slightly.
2. Change the code functionality and remove any related deprecation warnings.
3. Make appropriate changes to the type hints in the associated `.pyi` file.
   - Items marked with `@_api.deprecated` or `@_api.deprecate_privatize_attribute` are to be removed on expiry.
   - Items decorated with `@_api.rename_parameter` or `@_api.make_keyword_only` will have been updated at introduction, and require no change now.
   - Items decorated with `@_api.delete_parameter` will need to be updated to the final signature, in the same way as the `.py` file signature is updated.
   - Any entries in `ci/mypy-stubtest-allowlist.txt` which indicate a deprecation version should be double checked. In most cases this is not needed, though some items were never type

hinted in the first place and were added to this file instead. For removed items that were not in the stub file, only deleting from the allowlist is required.

### Announce new and deprecated API

When adding or changing the API in a backward in-compatible way, please add the appropriate *versioning directive* and document it for the release notes and add the entry to the appropriate folder:

| | versioning directive | announcement folder |
|---|---|---|
| new feature | `.. versionadded:: 3.N` | `doc/users/next_whats_new/` |
| API change | `.. versionchanged:: 3.N` | `doc/api/next_api_changes/[kind]` |

When deprecating API, please add a notice as described in the *deprecation guidelines* and summarized here:

| stage | announcement folder |
|---|---|
| *introduce deprecation* | `doc/api/next_api_changes/deprecation` |
| *expire deprecation* | `doc/api/next_api_changes/[kind]` |

Generally the introduction notices can be repurposed for the expiration notice as they are expected to be describing the same API changes and removals.

### Versioning directives

When making a backward incompatible change, please add a versioning directive in the docstring. The directives should be placed at the end of a description block. For example:

```
class Foo:
    """
    This is the summary.

    Followed by a longer description block.

    Consisting of multiple lines and paragraphs.

    .. versionadded:: 3.5

    Parameters
    ----------
    a : int
        The first parameter.
    b: bool, default: False
        This was added later.

        .. versionadded:: 3.6
    """
```

(continues on next page)

(continued from previous page)

```
    def set_b(b):
        """
        Set b.

        .. versionadded:: 3.6

        Parameters
        ----------
        b: bool
```

For classes and functions, the directive should be placed before the *Parameters* section. For parameters, the directive should be placed at the end of the parameter description. The micro release version is omitted and the directive should not be added to entire modules.

### Release notes

For both change notes and what's new, please avoid using cross-references in section titles as it causes links to be confusing in the table of contents. Instead, ensure that a cross-reference is included in the descriptive text.

### API change notes

API change notes for future releases are collected in `next_api_changes`. They are divided into four subdirectories:

- **Deprecations**: Announcements of future changes. Typically, these will raise a deprecation warning and users of this API should change their code to stay compatible with future releases of Matplotlib. If possible, state what should be used instead.
- **Removals**: Parts of the API that got removed. If possible, state what should be used instead.
- **Behaviour changes**: API that stays valid but will yield a different result.
- **Development changes**: Changes to the build process, dependencies, etc.

Please place new entries in these directories with a new file named `99999-ABC.rst`, where `99999` would be the PR number, and `ABC` the author's initials. Typically, each change will get its own file, but you may also amend existing files when suitable. The overall goal is a comprehensible documentation of the changes.

A typical entry could look like this:

```
Locators
~~~~~~~~
The unused `Locator.autoscale()` method is deprecated (pass the axis
limits to `Locator.view_limits()` instead).
```

#### What's new notes

Please place new portions of `whats_new.rst` in the `next_whats_new` directory.

When adding an entry please look at the currently existing files to see if you can extend any of them. If you create a file, name it something like `cool_new_feature.rst` if you have added a brand new feature or something like `updated_feature.rst` for extensions of existing features.

Include contents of the form:

```
Section title for feature
-------------------------

A bunch of text about how awesome the new feature is and examples of how
to use it.

A sub-section
~~~~~~~~~~~~~
```

### 10.3.3 Testing

Matplotlib uses the pytest framework.

The tests are in `lib/matplotlib/tests`, and customizations to the pytest testing infrastructure are in *`matplotlib.testing`*.

#### Requirements

To run the tests you will need to *set up Matplotlib for development*. Note in particular the *additional dependencies* for testing.

---

**Note:** We will assume that you want to run the tests in a development setup.

While you can run the tests against a regular installed version of Matplotlib, this is a far less common use case. You still need the *additional dependencies* for testing. You have to additionally get the reference images from the repository, because they are not distributed with pre-built Matplotlib packages.

---

#### Running the tests

In the root directory of your development repository run:

```
pytest
```

`pytest` can be configured via many command-line parameters. Some particularly useful ones are:

| | |
|---|---|
| `-v` or `--verbose` | Be more verbose |
| `-n NUM` | Run tests in parallel over NUM processes (requires pytest-xdist) |
| `--capture=no` or `-s` | Do not capture stdout |

To run a single test from the command line, you can provide a file path, optionally followed by the function separated by two colons, e.g., (tests do not need to be installed, but Matplotlib should be):

```
pytest lib/matplotlib/tests/test_simplification.py::test_clipping
```

If you want to use `pytest` as a module (via `python -m pytest`), then you will need to avoid clashes between `pytest`'s import mode and Python's search path:

- On more recent Python, you may `disable "unsafe import paths"` (i.e., stop adding the current directory to the import path) with the `-P` argument:

  ```
  python -P -m pytest
  ```

- On older Python, you may enable `isolated mode` (which stops adding the current directory to the import path, but has other repercussions):

  ```
  python -I -m pytest
  ```

- On any Python, set `pytest`'s import mode to the older `prepend` mode (but note that this will break `pytest`'s assert rewriting):

  ```
  python -m pytest --import-mode prepend
  ```

### Viewing image test output

The output of *image-based* tests is stored in a `result_images` directory. These images can be compiled into one HTML page, containing hundreds of images, using the `visualize_tests` tool:

```
python tools/visualize_tests.py
```

Image test failures can also be analysed using the `triage_tests` tool:

```
python tools/triage_tests.py
```

The triage tool allows you to accept or reject test failures and will copy the new image to the folder where the baseline test images are stored. The triage tool requires that *QT* is installed.

### Writing a simple test

Many elements of Matplotlib can be tested using standard tests. For example, here is a test from `matplotlib/tests/test_basic.py`:

```
def test_simple():
    """
    very simple example test
    """
    assert 1 + 1 == 2
```

Pytest determines which functions are tests by searching for files whose names begin with `"test_"` and then within those files for functions beginning with `"test"` or classes beginning with `"Test"`.

Some tests have internal side effects that need to be cleaned up after their execution (such as created figures or modified *rcParams*). The pytest fixture `matplotlib.testing.conftest.mpl_test_settings` will automatically clean these up; there is no need to do anything further.

### Random data in tests

Random data is a very convenient way to generate data for examples, however the randomness is problematic for testing (as the tests must be deterministic!). To work around this set the seed in each test. For numpy's default random number generator use:

```
import numpy as np
rng = np.random.default_rng(19680801)
```

and then use `rng` when generating the random numbers.

The seed is *John Hunter's* birthday.

### Writing an image comparison test

Writing an image-based test is only slightly more difficult than a simple test. The main consideration is that you must specify the "baseline", or expected, images in the *image_comparison* decorator. For example, this test generates a single image and automatically tests it:

```
from matplotlib.testing.decorators import image_comparison
import matplotlib.pyplot as plt

@image_comparison(baseline_images=['line_dashes'], remove_text=True,
                  extensions=['png'], style='mpl20')
def test_line_dashes():
    fig, ax = plt.subplots()
    ax.plot(range(10), linestyle=(0, (3, 3)), lw=5)
```

The first time this test is run, there will be no baseline image to compare against, so the test will fail. Copy the output images (in this case `result_images/test_lines/test_line_dashes.png`) to the

correct subdirectory of `baseline_images` tree in the source directory (in this case `lib/matplotlib/tests/baseline_images/test_lines`). Put this new file under source code revision control (with `git add`). When rerunning the tests, they should now pass.

It is preferred that new tests use `style='mpl20'` as this leads to smaller figures and reflects the newer look of default Matplotlib plots. Also, if the texts (labels, tick labels, etc) are not really part of what is tested, use `remove_text=True` as this will lead to smaller figures and reduce possible issues with font mismatch on different platforms.

### Compare two methods of creating an image

Baseline images take a lot of space in the Matplotlib repository. An alternative approach for image comparison tests is to use the *check_figures_equal* decorator, which should be used to decorate a function taking two *Figure* parameters and draws the same images on the figures using two different methods (the tested method and the baseline method). The decorator will arrange for setting up the figures and then collect the drawn results and compare them.

For example, this test compares two different methods to draw the same circle: plotting a circle using a *matplotlib.patches.Circle* patch vs plotting the circle using the parametric equation of a circle

```
from matplotlib.testing.decorators import check_figures_equal
import matplotib.patches as mpatches
import matplotlib.pyplot as plt
import numpy as np

@check_figures_equal(extensions=['png'], tol=100)
def test_parametric_circle_plot(fig_test, fig_ref):
    red_circle_ref = mpatches.Circle((0, 0), 0.2, color='r', clip_on=False)
    fig_ref.add_artist(red_circle_ref)
    theta = np.linspace(0, 2 * np.pi, 150)
    radius = 0.4
    fig_test.plot(radius * np.cos(theta), radius * np.sin(theta), color='r')
```

Both comparison decorators have a tolerance argument `tol` that is used to specify the tolerance for difference in color value between the two images, where 255 is the maximal difference. The test fails if the average pixel difference is greater than this value.

See the documentation of *image_comparison* and *check_figures_equal* for additional information about their use.

### Creating a new module in matplotlib.tests

We try to keep the tests categorized by the primary module they are testing. For example, the tests related to the `mathtext.py` module are in `test_mathtext.py`.

### Using GitHub Actions for CI

GitHub Actions is a hosted CI system "in the cloud".

GitHub Actions is configured to receive notifications of new commits to GitHub repos and to run builds or tests when it sees these new commits. It looks for a YAML files in `.github/workflows` to see how to test the project.

GitHub Actions is already enabled for the main Matplotlib GitHub repository -- for example, see the Tests workflows.

GitHub Actions should be automatically enabled for your personal Matplotlib fork once the YAML workflow files are in it. It generally isn't necessary to look at these workflows, since any pull request submitted against the main Matplotlib repository will be tested. The Tests workflow is skipped in forked repositories but you can trigger a run manually from the GitHub web interface.

You can see the GitHub Actions results at https://github.com/your_GitHub_user_name/matplotlib/actions -- here's an example.

### Using tox

Tox is a tool for running tests against multiple Python environments, including multiple versions of Python (e.g., 3.7, 3.8) and even different Python implementations altogether (e.g., CPython, PyPy, Jython, etc.), as long as all these versions are available on your system's $PATH (consider using your system package manager, e.g. apt-get, yum, or Homebrew, to install them).

tox makes it easy to determine if your working copy introduced any regressions before submitting a pull request. Here's how to use it:

```
$ pip install tox
$ tox
```

You can also run tox on a subset of environments:

```
$ tox -e py38,py39
```

Tox processes everything serially so it can take a long time to test several environments. To speed it up, you might try using a new, parallelized version of tox called `detox`. Give this a try:

```
$ pip install -U -i http://pypi.testrun.org detox
$ detox
```

Tox is configured using a file called `tox.ini`. You may need to edit this file if you want to add new environments to test (e.g., `py33`) or if you want to tweak the dependencies or the way the tests are run. For more info on the `tox.ini` file, see the Tox Configuration Specification.

### Building old versions of Matplotlib

When running a `git bisect` to see which commit introduced a certain bug, you may (rarely) need to build very old versions of Matplotlib. The following constraints need to be taken into account:

- Matplotlib 1.3 (or earlier) requires numpy 1.8 (or earlier).

### Testing released versions of Matplotlib

Running the tests on an installation of a released version (e.g. PyPI package or conda package) also requires additional setup.

---

**Note:** For an end-user, there is usually no need to run the tests on released versions of Matplotlib. Official releases are tested before publishing.

---

### Install additional dependencies

Install the *additional dependencies for testing*.

### Obtain the reference images

Many tests compare the plot result against reference images. The reference images are not part of the regular packaged versions (pip wheels or conda packages). If you want to run tests with reference images, you need to obtain the reference images matching the version of Matplotlib you want to test.

To do so, either download the matching source distribution `matplotlib-X.Y.Z.tar.gz` from PyPI or alternatively, clone the git repository and `git checkout vX.Y.Z`. Copy the folder `lib/matplotlib/tests/baseline_images` to the folder `matplotlib/tests` of your the matplotlib installation to test. The correct target folder can be found using:

```
python -c "import matplotlib.tests; print(matplotlib.tests.__file__.rsplit('/
↪', 1)[0])"
```

An analogous copying of `lib/mpl_toolkits/*/tests/baseline_images` is necessary for testing `mpl_toolkits`.

#### Run the tests

To run all the tests on your installed version of Matplotlib:

```
pytest --pyargs matplotlib.tests
```

The test discovery scope can be narrowed to single test modules or even single functions:

```
pytest --pyargs matplotlib.tests.test_simplification.py::test_clipping
```

**Documentation**

### 10.3.4 Write documentation

#### Getting started

#### General file structure

All documentation is built from the `doc/`. The `doc/` directory contains configuration files for Sphinx and reStructuredText (ReST; `.rst`) files that are rendered to documentation pages.

Documentation is created in three ways. First, API documentation (`doc/api`) is created by Sphinx from the docstrings of the classes in the Matplotlib library. Except for `doc/api/api_changes/`, `.rst` files in `doc/api` are created when the documentation is built. See *Write docstrings* below.

Second, our example pages, tutorials, and some of the narrative documentation are created by Sphinx Gallery. Sphinx Gallery converts example Python files to `*.rst` files with the result of Matplotlib plot calls as embedded images. See *Write examples and tutorials* below.

Third, Matplotlib has narrative docs written in ReST in subdirectories of `doc/users/`. If you would like to add new documentation that is suited to an `.rst` file rather than a gallery or tutorial example, choose an appropriate subdirectory to put it in, and add the file to the table of contents of `index.rst` of the subdirectory. See *Write ReST pages* below.

---

**Note:** Don't directly edit the `.rst` files in `doc/plot_types`, `doc/gallery`, `doc/tutorials`, and `doc/api` (excepting `doc/api/api_changes/`). Sphinx regenerates files in these directories when building documentation.

---

### Set up the build

The documentation for Matplotlib is generated from reStructuredText (ReST) using the Sphinx documentation generation tool.

To build the documentation you will need to *set up Matplotlib for development*. Note in particular the *additional dependencies* required to build the documentation.

### Build the docs

The documentation sources are found in the `doc/` directory. The configuration file for Sphinx is `doc/conf.py`. It controls which directories Sphinx parses, how the docs are built, and how the extensions are used. To build the documentation in html format, cd into `doc/` and run:

```
make html
```

**Note:** Since the documentation is very large, the first build may take 10-20 minutes, depending on your machine. Subsequent builds will be faster.

Other useful invocations include

```
# Build the html documentation, but skip generation of the gallery images to
# save time.
make html-noplot

# Build the html documentation, but skip specific subdirectories.  If a␣
↪gallery
# directory is skipped, the gallery images are not generated.  The first
# time this is run, it creates ``.mpl_skip_subdirs.yaml`` which can be edited
# to add or remove subdirectories
make html-skip-subdirs

# Delete built files.  May help if you get errors about missing paths or
# broken links.
make clean

# Build pdf docs.
make latexpdf
```

The `SPHINXOPTS` variable is set to `-W --keep-going` by default to build the complete docs but exit with exit status 1 if there are warnings. To unset it, use

```
make SPHINXOPTS= html
```

You can use the `O` variable to set additional options:

- `make O=-j4 html` runs a parallel build with 4 processes.
- `make O=-Dplot_formats=png:100 html` saves figures in low resolution.

Multiple options can be combined, e.g. `make O='-j4 -Dplot_formats=png:100' html`.

On Windows, set the options as environment variables, e.g.:

```
set SPHINXOPTS= & set O=-j4 -Dplot_formats=png:100 & make html
```

### Show locally built docs

The built docs are available in the folder `build/html`. A shortcut for opening them in your default browser is:

```
make show
```

### Write ReST pages

Most documentation is either in the docstrings of individual classes and methods, in explicit `.rst` files, or in examples and tutorials. All of these use the ReST syntax and are processed by Sphinx.

The Sphinx reStructuredText Primer is a good introduction into using ReST. More complete information is available in the reStructuredText reference documentation.

This section contains additional information and conventions how ReST is used in the Matplotlib documentation.

### Formatting and style conventions

It is useful to strive for consistency in the Matplotlib documentation. Here are some formatting and style conventions that are used.

### Section formatting

Use sentence case `Upper lower` for section titles, e.g., `Possible hangups` rather than `Possible Hangups`.

We aim to follow the recommendations from the Python documentation and the Sphinx reStructuredText documentation for section markup characters, i.e.:

- `#` with overline, for parts. This is reserved for the main title in `index.rst`. All other pages should start with "chapter" or lower.
- `*` with overline, for chapters
- =, for sections
- −, for subsections
- `^`, for subsubsections
- `"`, for paragraphs

This may not yet be applied consistently in existing docs.

### Table formatting

Given the size of the table and length of each entry, use:

| | small table | large table |
|---|---|---|
| short entry | simple or grid table | grid table |
| long entry | list table | csv table |

For more information, see rst tables.

### Function arguments

Function arguments and keywords within docstrings should be referred to using the `*emphasis*` role. This will keep Matplotlib's documentation consistent with Python's documentation:

```
Here is a description of *argument*
```

Do not use the `` `default role` ``:

```
Do not describe `argument` like this.  As per the next section,
this syntax will (unsuccessfully) attempt to resolve the argument as a
link to a class or method in the library.
```

nor the ``` ``literal`` ``` role:

```
Do not describe ``argument`` like this.
```

### Refer to other documents and sections

Sphinx supports internal references:

| Role | Links target | Representation in rendered HTML |
|---|---|---|
| `:doc:` | document | link to a page |
| `:ref:` | reference label | link to an anchor associated with a heading |

Examples:

```
See the :doc:`/install/index`

See the tutorial :ref:`quick_start`

See the example :doc:`/gallery/lines_bars_and_markers/simple_plot`
```

will render as:

> See the *Installation*
>
> See the tutorial *Quick start guide*
>
> See the example *Simple Plot*

Sections can also be given reference labels. For instance from the *Installation* link:

```
.. _clean-install:

How to completely remove Matplotlib
===================================

Occasionally, problems with Matplotlib can be solved with a clean...
```

and refer to it using the standard reference syntax:

```
See :ref:`clean-install`
```

will give the following link: *How to completely remove Matplotlib*

To maximize internal consistency in section labeling and references, use hyphen separated, descriptive labels for section references. Keep in mind that contents may be reorganized later, so avoid top level names in references like `user` or `devel` or `faq` unless necessary, because for example the FAQ "what is a backend?" could later become part of the users guide, so the label:

```
.. _what-is-a-backend:
```

is better than:

```
.. _faq-backend:
```

In addition, since underscores are widely used by Sphinx itself, use hyphens to separate words.

### Refer to other code

To link to other methods, classes, or modules in Matplotlib you can use back ticks, for example:

```
`matplotlib.collections.LineCollection`
```

generates a link like this: *matplotlib.collections.LineCollection*.

*Note:* We use the sphinx setting `default_role = 'obj'` so that you don't have to use qualifiers like `:class:`, `:func:`, `:meth:` and the likes.

Often, you don't want to show the full package and module name. As long as the target is unambiguous you can simply leave them out:

```
`.LineCollection`
```

and the link still works: *LineCollection*. Note that you should typically include the leading dot. It tells Sphinx to look for the given name in the whole project. See also the explanation at Sphinx: Cross-referencing Python objects.

If there are multiple code elements with the same name (e.g. `plot()` is a method in multiple classes), you'll have to extend the definition:

```
`.pyplot.plot` or `.Axes.plot`
```

These will show up as *pyplot.plot* or *Axes.plot*. To still show only the last segment you can add a tilde as prefix:

```
`~.pyplot.plot` or `~.Axes.plot`
```

will render as *plot* or *plot*.

Other packages can also be linked via intersphinx:

```
`numpy.mean`
```

will return this link: `numpy.mean`. This works for Python, Numpy, Scipy, and Pandas (full list is in `doc/conf.py`). If external linking fails, you can check the full list of referenceable objects with the following commands:

```
python -m sphinx.ext.intersphinx 'https://docs.python.org/3/objects.inv'
python -m sphinx.ext.intersphinx 'https://numpy.org/doc/stable/objects.inv'
python -m sphinx.ext.intersphinx 'https://docs.scipy.org/doc/scipy/objects.inv
↪'
python -m sphinx.ext.intersphinx 'https://pandas.pydata.org/pandas-docs/
↪stable/objects.inv'
```

### Include figures and files

Image files can directly included in pages with the `image::` directive. e.g., `tutorials/intermediate/constrainedlayout_guide.py` displays a couple of static images:

```
# .. image:: /_static/constrained_layout_1b.png
#    :align: center
```

Files can be included verbatim. For instance the `LICENSE` file is included at *License agreement* using

```
.. literalinclude:: ../../LICENSE/LICENSE
```

The examples directory is copied to `doc/gallery` by sphinx-gallery, so plots from the examples directory can be included using

```
.. plot:: gallery/lines_bars_and_markers/simple_plot.py
```

Note that the python script that generates the plot is referred to, rather than any plot that is created. Sphinx-gallery will provide the correct reference when the documentation is built.

### Tools for writing mathematical expressions

In most cases, you will likely want to use one of Sphinx's builtin Math extensions. In rare cases we want the rendering of the mathematical text in the documentation html to exactly match with the rendering of the mathematical expression in the Matplotlib figure. In these cases, you can use the `matplotlib.sphinxext.mathmpl` Sphinx extension (See also the *Writing mathematical expressions* tutorial.)

### Write docstrings

Most of the API documentation is written in docstrings. These are comment blocks in source code that explain how the code works.

---

**Note:** Some parts of the documentation do not yet conform to the current documentation style. If in doubt, follow the rules given here and not what you may see in the source code. Pull requests updating docstrings to the current style are very welcome.

---

All new or edited docstrings should conform to the numpydoc docstring guide. Much of the ReST syntax discussed above (*Write ReST pages*) can be used for links and references. These docstrings eventually populate the `doc/api` directory and form the reference documentation for the library.

### Example docstring

An example docstring looks like:

```
def hlines(self, y, xmin, xmax, colors=None, linestyles='solid',
           label='', **kwargs):
    """
    Plot horizontal lines at each *y* from *xmin* to *xmax*.

    Parameters
    ----------
    y : float or array-like
        y-indexes where to plot the lines.

    xmin, xmax : float or array-like
        Respective beginning and end of each line. If scalars are
        provided, all lines will have the same length.

    colors : list of colors, default: :rc:`lines.color`

    linestyles : {'solid', 'dashed', 'dashdot', 'dotted'}, optional

    label : str, default: ''

    Returns
    -------
    `~matplotlib.collections.LineCollection`
```

(continues on next page)

(continued from previous page)

```
    Other Parameters
    ----------------
    data : indexable object, optional
        DATA_PARAMETER_PLACEHOLDER
    **kwargs :  `~matplotlib.collections.LineCollection` properties.

    See Also
    --------
    vlines : vertical lines
    axhline : horizontal line across the Axes
    """
```

See the *hlines* documentation for how this renders.

The Sphinx website also contains plenty of documentation concerning ReST markup and working with Sphinx in general.

### Formatting conventions

The basic docstring conventions are covered in the numpydoc docstring guide and the Sphinx documentation. Some Matplotlib-specific formatting conventions to keep in mind:

### Quote positions

The quotes for single line docstrings are on the same line (pydocstyle D200):

```
def get_linewidth(self):
    """Return the line width in points."""
```

The quotes for multi-line docstrings are on separate lines (pydocstyle D213):

```
def set_linestyle(self, ls):
"""
Set the linestyle of the line.

[...]
"""
```

### Function arguments

Function arguments and keywords within docstrings should be referred to using the `*emphasis*` role. This will keep Matplotlib's documentation consistent with Python's documentation:

```
If *linestyles* is *None*, the default is 'solid'.
```

Do not use the `` `default role` `` or the ``` ``literal`` ``` role:

```
Neither `argument` nor ``argument`` should be used.
```

### Quotes for strings

Matplotlib does not have a convention whether to use single-quotes or double-quotes. There is a mixture of both in the current code.

Use simple single or double quotes when giving string values, e.g.

```
If 'tight', try to figure out the tight bbox of the figure.

No ``'extra'`` literal quotes.
```

The use of extra literal quotes around the text is discouraged. While they slightly improve the rendered docs, they are cumbersome to type and difficult to read in plain-text docs.

### Parameter type descriptions

The main goal for parameter type descriptions is to be readable and understandable by humans. If the possible types are too complex use a simplification for the type description and explain the type more precisely in the text.

Generally, the numpydoc docstring guide conventions apply. The following rules expand on them where the numpydoc conventions are not specific.

Use `float` for a type that can be any number.

Use `(float, float)` to describe a 2D position. The parentheses should be included to make the tupleness more obvious.

Use `array-like` for homogeneous numeric sequences, which could typically be a numpy.array. Dimensionality may be specified using `2D`, `3D`, `n-dimensional`. If you need to have variables denoting the sizes of the dimensions, use capital letters in brackets (`(M, N) array-like`). When referring to them in the text they are easier read and no special formatting is needed. Use `array` instead of `array-like` for return types if the returned object is indeed a numpy array.

`float` is the implicit default dtype for array-likes. For other dtypes use `array-like of int`.

Some possible uses:

```
2D array-like
(N,) array-like
(M, N) array-like
(M, N, 3) array-like
array-like of int
```

Non-numeric homogeneous sequences are described as lists, e.g.:

```
list of str
list of `.Artist`
```

### Reference types

Generally, the rules from *referring-to-other-code* apply. More specifically:

Use full references `~matplotlib.colors.Normalize` with an abbreviation tilde in parameter types. While the full name helps the reader of plain text docstrings, the HTML does not need to show the full name as it links to it. Hence, the ~-shortening keeps it more readable.

Use abbreviated links `.Normalize` in the text.

```
norm : `~matplotlib.colors.Normalize`, optional
    A `.Normalize` instance is used to scale luminance data to 0, 1.
```

### Default values

As opposed to the numpydoc guide, parameters need not be marked as *optional* if they have a simple default:

- use `{name} : {type}, default: {val}` when possible.
- use `{name} : {type}, optional` and describe the default in the text if it cannot be explained sufficiently in the recommended manner.

The default value should provide semantic information targeted at a human reader. In simple cases, it restates the value in the function signature. If applicable, units should be added.

```
Prefer:
    interval : int, default: 1000ms
over:
    interval : int, default: 1000
```

If *None* is only used as a sentinel value for "parameter not specified", do not document it as the default. Depending on the context, give the actual default, or mark the parameter as optional if not specifying has no particular effect.

```
Prefer:
    dpi : float, default: :rc:`figure.dpi`
over:
    dpi : float, default: None
```

(continues on next page)

(continued from previous page)

```
Prefer:
    textprops : dict, optional
        Dictionary of keyword parameters to be passed to the
        `~matplotlib.text.Text` instance contained inside TextArea.
over:
    textprops : dict, default: None
        Dictionary of keyword parameters to be passed to the
        `~matplotlib.text.Text` instance contained inside TextArea.
```

### `See also` sections

Sphinx automatically links code elements in the definition blocks of `See also` sections. No need to use backticks there:

```
See Also
--------
vlines : vertical lines
axhline : horizontal line across the Axes
```

### Wrap parameter lists

Long parameter lists should be wrapped using a \ for continuation and starting on the new line without any indent (no indent because pydoc will parse the docstring and strip the line continuation so that indent would result in a lot of whitespace within the line):

```
def add_axes(self, *args, **kwargs):
    """
    ...

    Parameters
    ----------
    projection : {'aitoff', 'hammer', 'lambert', 'mollweide', 'polar', \
'rectilinear'}, optional
        The projection type of the axes.

    ...
    """
```

Alternatively, you can describe the valid parameter values in a dedicated section of the docstring.

### rcParams

rcParams can be referenced with the custom `:rc:` role: `:rc:`foo`` yields `rcParams["foo"] = 'default'`, which is a link to the `matplotlibrc` file description.

### Setters and getters

Artist properties are implemented using setter and getter methods (because Matplotlib predates the Python `property` decorator). By convention, these setters and getters are named `set_PROPERTYNAME` and `get_PROPERTYNAME`; the list of properties thusly defined on an artist and their values can be listed by the *setp* and *getp* functions.

The Parameters block of property setter methods is parsed to document the accepted values, e.g. the docstring of *Line2D.set_linestyle* starts with

```
def set_linestyle(self, ls):
    """
    Set the linestyle of the line.

    Parameters
    ----------
    ls : {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
        etc.
    """
```

which results in the following line in the output of `plt.setp(line)` or `plt.setp(line, "linestyle")`:

```
linestyle or ls: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
```

In some rare cases (mostly, setters which accept both a single tuple and an unpacked tuple), the accepted values cannot be documented in such a fashion; in that case, they can be documented as an `.. ACCEPTS:` block, e.g. for *axes.Axes.set_xlim*:

```
def set_xlim(self, left=None, right=None):
    """
    Set the x-axis view limits.

    Parameters
    ----------
    left : float, optional
        The left xlim in data coordinates. Passing *None* leaves the
        limit unchanged.

        The left and right xlims may also be passed as the tuple
        (*left*, *right*) as the first positional argument (or as
        the *left* keyword argument).

        .. ACCEPTS: (bottom: float, top: float)
```

(continues on next page)

(continued from previous page)

```
    right : float, optional
        etc.
    """
```

Note that the leading `..` makes the `.. ACCEPTS:` block a reST comment, hiding it from the rendered docs.

### Keyword arguments

**Note:** The information in this section is being actively discussed by the development team, so use the docstring interpolation only if necessary. This section has been left in place for now because this interpolation is part of the existing documentation.

Since Matplotlib uses a lot of pass-through `kwargs`, e.g., in every function that creates a line (*plot*, *semilogx*, *semilogy*, etc.), it can be difficult for the new user to know which `kwargs` are supported. Matplotlib uses a docstring interpolation scheme to support documentation of every function that takes a `**kwargs`. The requirements are:

1. single point of configuration so changes to the properties don't require multiple docstring edits.
2. as automated as possible so that as properties change, the docs are updated automatically.

The `@_docstring.interpd` decorator implements this. Any function accepting *Line2D* pass-through `kwargs`, e.g., *matplotlib.axes.Axes.plot*, can list a summary of the *Line2D* properties, as follows:

```
# in axes.py
@_docstring.interpd
def plot(self, *args, **kwargs):
    """
    Some stuff omitted

    Other Parameters
    ----------------
    scalex, scaley : bool, default: True
        These parameters determine if the view limits are adapted to the
        data limits. The values are passed on to `autoscale_view`.

    **kwargs : `.Line2D` properties, optional
        *kwargs* are used to specify properties like a line label (for
        auto legends), linewidth, antialiasing, marker face color.
        Example::

        >>> plot([1, 2, 3], [1, 2, 3], 'go-', label='line 1', linewidth=2)
        >>> plot([1, 2, 3], [1, 4, 9], 'rs', label='line 2')

        If you specify multiple lines with one plot call, the kwargs apply
        to all those lines. In case the label object is iterable, each
        element is used as labels for each set of data.
```

(continues on next page)

(continued from previous page)

```
        Here is a list of available `.Line2D` properties:

        %(Line2D:kwdoc)s
    """
```

The `%(Line2D:kwdoc)` syntax makes `interpd` lookup an *Artist* subclass named `Line2D`, and call *artist.kwdoc* on that class. *artist.kwdoc* introspects the subclass and summarizes its properties as a substring, which gets interpolated into the docstring.

Note that this scheme does not work for decorating an Artist's `__init__`, as the subclass and its properties are not defined yet at that point. Instead, `@_docstring.interpd` can be used to decorate the class itself -- at that point, *kwdoc* can list the properties and interpolate them into `__init__.__doc__`.

### Inherit docstrings

If a subclass overrides a method but does not change the semantics, we can reuse the parent docstring for the method of the child class. Python does this automatically, if the subclass method does not have a docstring.

Use a plain comment `# docstring inherited` to denote the intention to reuse the parent docstring. That way we do not accidentally create a docstring in the future:

```
class A:
    def foo():
        """The parent docstring."""
        pass

class B(A):
    def foo():
        # docstring inherited
        pass
```

### Add figures

As above (see *Include figures and files*), figures in the examples gallery can be referenced with a `.. plot::` directive pointing to the python script that created the figure. For instance the *legend* docstring references the file `examples/text_labels_and_annotations/legend.py`:

```
"""
...

Examples
--------

.. plot:: gallery/text_labels_and_annotations/legend.py
"""
```

Note that `examples/text_labels_and_annotations/legend.py` has been mapped to `gallery/text_labels_and_annotations/legend.py`, a redirection that may be fixed in future re-organization of the docs.

Plots can also be directly placed inside docstrings. Details are in *matplotlib.sphinxext.plot_directive*. A short example is:

```
"""
...

Examples
--------

.. plot::
   import matplotlib.image as mpimg
   img = mpimg.imread('_static/stinkbug.png')
   imgplot = plt.imshow(img)
"""
```

An advantage of this style over referencing an example script is that the code will also appear in interactive docstrings.

### Write examples and tutorials

Examples and tutorials are Python scripts that are run by Sphinx Gallery. Sphinx Gallery finds `*.py` files in source directories and runs the files to create images and narrative that are embedded in `*.rst` files in a build location of the `doc/` directory. Files in the build location should not be directly edited as they will be overwritten by Sphinx gallery. Currently Matplotlib has four galleries as follows:

| Source location | Build location |
|---|---|
| `galleries/plot_types` | `doc/plot_types` |
| `galleries/examples` | `doc/gallery` |
| `galleries/tutorials` | `doc/tutorials` |
| `galleries/users_explain` | `doc/users/explain` |

The first three are traditional galleries. The last, `galleries/users_explain`, is a mixed gallery where some of the files are raw `*.rst` files and some are `*.py` files; Sphinx Gallery just copies these `*.rst` files from the source location to the build location (see *Raw restructured text files in the gallery*, below).

In the Python files, to exclude an example from having a plot generated, insert "sgskip" somewhere in the filename.

### Format examples

The format of these files is relatively straightforward. Properly formatted comment blocks are treated as ReST text, the code is displayed, and figures are put into the built page. Matplotlib uses the `# %%` section separator so that IDEs will identify "code cells" to make it easy to re-run sub-sections of the example.

For instance the example *Simple Plot* example is generated from `/galleries/examples/lines_bars_and_markers/simple_plot.py`, which looks like:

```
"""
===========
Simple Plot
===========

Create a simple plot.
"""
import matplotlib.pyplot as plt
import numpy as np

# Data for plotting
t = np.arange(0.0, 2.0, 0.01)
s = 1 + np.sin(2 * np.pi * t)

# Note that using plt.subplots below is equivalent to using
# fig = plt.figure and then ax = fig.add_subplot(111)
fig, ax = plt.subplots()
ax.plot(t, s)

ax.set(xlabel='time (s)', ylabel='voltage (mV)',
       title='About as simple as it gets, folks')
ax.grid()
plt.show()
```

The first comment block is treated as ReST text. The other comment blocks render as comments in *Simple Plot*.

Tutorials are made with the exact same mechanism, except they are longer, and typically have more than one comment block (i.e. *Quick start guide*). The first comment block can be the same as the example above. Subsequent blocks of ReST text are delimited by the line `# %%` :

```
"""
===========
Simple Plot
===========

Create a simple plot.
"""
...
ax.grid()
plt.show()

# %%
```

(continues on next page)

(continued from previous page)

```
# Second plot
# ===========
#
# This is a second plot that is very nice

fig, ax = plt.subplots()
ax.plot(np.sin(range(50)))
```

In this way text, code, and figures are output in a "notebook" style.

### Sample data

When sample data comes from a public dataset, please cite the source of the data. Sample data should be written out in the code. When this is not feasible, the data can be loaded using *cbook.get_sample_data*.

```
import matplotlib.cbook as cbook
fh = cbook.get_sample_data('mydata.dat')
```

If the data is too large to be included in the code, it should be added to `lib/matplotlib/mpl-data/sample_data/`

### Create mini-gallery

The showcased Matplotlib functions should be listed in an admonition at the bottom as follows

```
# %%
#
# .. admonition:: References
#
#    The use of the following functions, methods, classes and modules is shown
#    in this example:
#
#    - `matplotlib.axes.Axes.fill` / `matplotlib.pyplot.fill`
#    - `matplotlib.axes.Axes.axis` / `matplotlib.pyplot.axis`
```

This allows sphinx-gallery to place an entry to the example in the mini-gallery of the mentioned functions. Whether or not a function is mentioned here should be decided depending on if a mini-gallery link prominently helps to illustrate that function; e.g. mention `matplotlib.pyplot.subplots` only in examples that are about laying out subplots, not in every example that uses it.

Functions that exist in `pyplot` as well as in Axes or Figure should mention both references no matter which one is used in the example code. The `pyplot` reference should always be the second to mention; see the example above.

### Order examples

The order of the sections of the *Tutorials* and the *Examples*, as well as the order of the examples within each section are determined in a two step process from within the `/doc/sphinxext/gallery_order.py`:

- *Explicit order*: This file contains a list of folders for the section order and a list of examples for the subsection order. The order of the items shown in the doc pages is the order those items appear in those lists.
- *Implicit order*: If a folder or example is not in those lists, it will be appended after the explicitly ordered items and all of those additional items will be ordered by pathname (for the sections) or by filename (for the subsections).

As a consequence, if you want to let your example appear in a certain position in the gallery, extend those lists with your example. In case no explicit order is desired or necessary, still make sure to name your example consistently, i.e. use the main function or subject of the example as first word in the filename; e.g. an image example should ideally be named similar to `imshow_mynewexample.py`.

### Raw restructured text files in the gallery

Sphinx Gallery folders usually consist of a `README.txt` and a series of Python source files that are then translated to an `index.rst` file and a series of `example_name.rst` files in the `doc/` subdirectories. However, Sphinx Gallery also allows raw `*.rst` files to be passed through a gallery (see Manually passing files in the Sphinx Gallery documentation). We use this feature in `galleries/users_explain`, where, for instance, `galleries/users_explain/colors` is a regular Sphinx Gallery subdirectory, but `galleries/users_explain/artists` has a mix of `*.rst` and `*py` files. For mixed subdirectories like this, we must add any `*.rst` files to a `:toctree:`, either in the `README.txt` or in a manual `index.rst`.

### Miscellaneous

### Move documentation

Sometimes it is desirable to move or consolidate documentation. With no action this will lead to links either going dead (404) or pointing to old versions of the documentation. Preferable is to replace the old page with an html refresh that immediately redirects the viewer to the new page. So, for example we move `/doc/topic/old_info.rst` to `/doc/topic/new_info.rst`. We remove `/doc/topic/old_info.rst` and in `/doc/topic/new_info.rst` we insert a `redirect-from` directive that tells sphinx to still make the old file with the html refresh/redirect in it (probably near the top of the file to make it noticeable)

```
.. redirect-from:: /topic/old_info
```

In the built docs this will yield an html file `/build/html/topic/old_info.html` that has a refresh to `new_info.html`. If the two files are in different subdirectories:

```
.. redirect-from:: /old_topic/old_info2
```

will yield an html file `/build/html/old_topic/old_info2.html` that has a (relative) refresh to `../topic/new_info.html`.

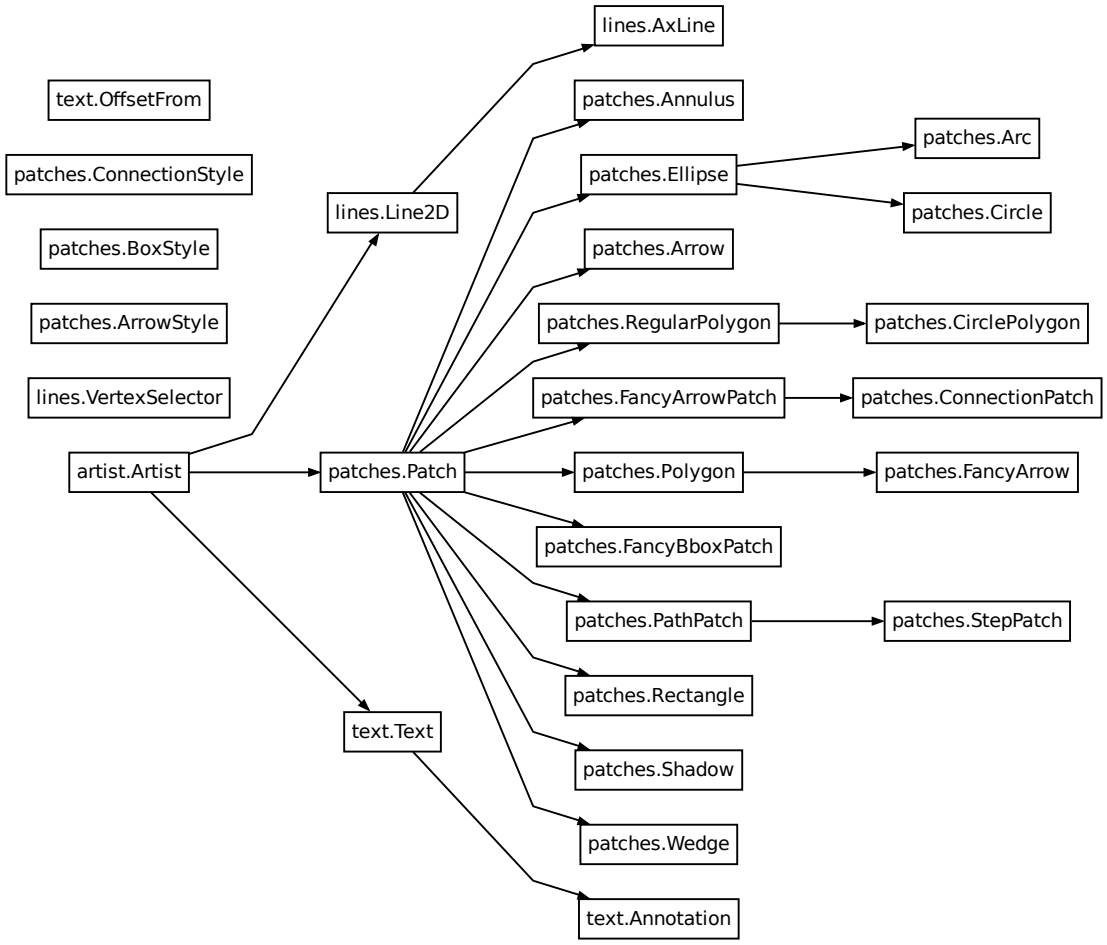
Use the full path for this directive, relative to the doc root at `https://matplotlib.org/stable/`. So `/old_topic/old_info2` would be found by users at `http://matplotlib.org/stable/old_topic/old_info2`. For clarity, do not use relative links.

### Generate inheritance diagrams

Class inheritance diagrams can be generated with the Sphinx inheritance-diagram directive.

Example:

```
.. inheritance-diagram:: matplotlib.patches matplotlib.lines matplotlib.text
   :parts: 2
```



#### Navbar and style

Matplotlib has a few subprojects that share the same navbar and style, so these are centralized as a sphinx theme at mpl_sphinx_theme. Changes to the style or topbar should be made there to propagate across all subprojects.

### 10.3.5 Documentation style guide

This guide contains best practices for the language and formatting of Matplotlib documentation.

**See also:**

For more information about contributing, see the *Write documentation* section.

#### Expository language

For explanatory writing, the following guidelines are for clear and concise language use.

#### Terminology

There are several key terms in Matplotlib that are standards for reliability and consistency in documentation. They are not interchangeable.

| Term | Description | Correct | Incorrect |
|---|---|---|---|
| `Figure` | Matplotlib working space for programming. | • *For Matplotlib objects*: Figure, "The Figure is the working space for the visual.<br>• *Referring to class*: `Figure`, "Methods within the `Figure` provide the visuals."<br>• *General language*: figure, "Michelle Kwan is a famous figure skater." | • "The figure is the working space for visuals."<br>• "Methods in the figure provide the visuals."<br>• "The `Figure` Four leglock is a wrestling move." |
| `Axes` | Subplots within Figure. Contains plot elements and is responsible for plotting and configuring additional details. | • *For Matplotlib objects*: Axes, "An Axes is a subplot within the Figure."<br>• *Referring to class*: `Axes`, "Each `Axes` is specific to one Figure."<br>• *General language*: axes, "Both loggers and lumberjacks use axes to chop wood." OR "There are no standard names for the coordinates in the three axes." (Plural of axis) | • "The axes methods transform the data."<br>• "Each `Axes` is specific to a Figure."<br>• "The musicians on stage call their guitars Axes."<br>• "The point where the Axes meet is the origin of the coordinate system." |
| `Artist` | Broad variety of Matplotlib objects that display visuals. | • *For Matplotlib objects*: Artist, "Artists display visuals and are the visible elements when rendering a Figure."<br>• *Referring to class*: `Artist`, "Each `Artist` has respective methods and functions."<br>• *General language*: artist, "The artist in the museum is from France." | • "Configure the legend artist with its respective method."<br>• "There is an `Artist` for that visual in the graph."<br>• "Some Artists became famous only by accident." |
| `Axis` | Human-readable single dimensional object of reference marks containing ticks, tick labels, spines, and edges. | • *For Matplotlib objects*: Axis, "The Axis for the bar chart is a separate Artist." (plural, Axis objects)<br>• *Referring to class*: `Axis`, "The `Axis` contains respective XAxis and YAxis objects."<br>• *General language*: axis, "Rotation around a fixed axis is a special case of rotational motion." | • "Plot the graph onto the axis."<br>• "Each Axis is usually named after the coordinate which is measured along it."<br>• "In some computer graphics contexts, the ordinate `Axis` may be oriented downwards." |

### Grammar

#### Subject

Use second-person imperative sentences for directed instructions specifying an action. Second-person pronouns are for individual-specific contexts and possessive reference.

| Correct | Incorrect |
|---|---|
| Install Matplotlib from the source directory using the Python `pip` installer program. Depending on your operating system, you may need additional support. | You can install Matplotlib from the source directory. You can find additional support if you are having trouble with your installation. |

#### Tense

Use present simple tense for explanations. Avoid future tense and other modal or auxiliary verbs when possible.

| Correct | Incorrect |
|---|---|
| The fundamental ideas behind Matplotlib for visualization involve taking data and transforming it through functions and methods. | Matplotlib will take data and transform it through functions and methods. They can generate many kinds of visuals. These will be the fundamentals for using Matplotlib. |

#### Voice

Write in active sentences. Passive voice is best for situations or conditions related to warning prompts.

| Correct | Incorrect |
|---|---|
| The function `plot` generates the graph. | The graph is generated by the `plot` function. |
| An error message is returned by the function if there are no arguments. | You will see an error message from the function if there are no arguments. |

### Sentence structure

Write with short sentences using Subject-Verb-Object order regularly. Limit coordinating conjunctions in sentences. Avoid pronoun references and subordinating conjunctive phrases.

| Correct | Incorrect |
|---|---|
| The `pyplot` module in Matplotlib is a collection of functions. These functions create, manage, and manipulate the current Figure and plotting area. | The `pyplot` module in Matplotlib is a collection of functions which create, manage, and manipulate the current Figure and plotting area. |
| The `plot` function plots data to the respective Axes. The Axes corresponds to the respective Figure. | The `plot` function plots data within its respective Axes for its respective Figure. |
| The implicit approach is a convenient shortcut for generating simple plots. | Users that wish to have convenient shortcuts for generating plots use the implicit approach. |

### Formatting

The following guidelines specify how to incorporate code and use appropriate formatting for Matplotlib documentation.

### Code

Matplotlib is a Python library and follows the same standards for documentation.

### Comments

Examples of Python code have comments before or on the same line.

| Correct | Incorrect |
|---|---|
| `# Data`<br>`years = [2006, 2007, 2008]` | `years = [2006, 2007, 2008]`<br>`# Data` |
| `years = [2006, 2007, 2008]  # Data` | |

### Outputs

When generating visuals with Matplotlib using `.py` files in examples, display the visual with *`matplotlib.pyplot.show`* to display the visual. Keep the documentation clear of Python output lines.

| Correct | Incorrect |
| --- | --- |
| `plt.plot([1, 2, 3], [1, 2, 3])`<br>`plt.show()` | `plt.plot([1, 2, 3], [1, 2, 3])` |
| `fig, ax = plt.subplots()`<br>`ax.plot([1, 2, 3], [1, 2, 3])`<br>`fig.show()` | `fig, ax = plt.subplots()`<br>`ax.plot([1, 2, 3], [1, 2, 3])` |

### reStructuredText

Matplotlib uses reStructuredText Markup for documentation. Sphinx helps to transform these documents into appropriate formats for accessibility and visibility.

- reStructuredText Specifications
- Quick Reference Document

### Lists

Bulleted lists are for items that do not require sequencing. Numbered lists are exclusively for performing actions in a determined order.

| Correct | Incorrect |
| --- | --- |
| The example uses three graphs. | The example uses three graphs. |
| • Bar<br>• Line<br>• Pie | 1. Bar<br>2. Line<br>3. Pie |
| These four steps help to get started using Matplotlib. | The following steps are important to get started using Matplotlib. |
| 1. Import the Matplotlib library.<br>2. Import the necessary modules.<br>3. Set and assign data to work on.<br>4. Transform data with methods and functions. | • Import the Matplotlib library.<br>• Import the necessary modules.<br>• Set and assign data to work on.<br>• Transform data with methods and functions. |

### Tables

Use ASCII tables with reStructuredText standards in organizing content. Markdown tables and the csv-table directive are not accepted.

| Correct | Incorrect |
|---|---|

| Correct | Incorrect |
|---|---|
| OK | Not OK |

```
| Correct | Incorrect |
| ------- | --------- |
| OK      | Not OK    |
```

```
+----------+----------+
| Correct  | Incorrect|
+==========+==========+
| OK       | Not OK   |
+----------+----------+
```

```
.. csv-table::
   :header: "correct", "incorrect"
   :widths: 10, 10

   "OK   ", "Not OK"
```

```
==========  ==========
  Correct     Incorrect
==========  ==========
OK          Not OK
==========  ==========
```

### Additional resources

This style guide is not a comprehensive standard. For a more thorough reference of how to contribute to documentation, see the links below. These resources contain common best practices for writing documentation.

- Python Developer's Guide
- Google Developer Style Guide
- IBM Style Guide
- Red Hat Style Guide

### 10.3.6 Guidelines for assigning tags to gallery examples

#### Why do we need tags?

Tags serve multiple purposes.

Tags have a one-to-many organization (i.e. one example can have several tags), while the gallery structure requires that examples are placed in only one location. This means tags provide a secondary layer of organization and make the gallery of examples more flexible and more user-friendly.

They allow for better discoverability, search, and browse functionality. They are helpful for users struggling to write a search query for what they're looking for.

Hidden tags provide additional functionality for maintainers and contributors.

#### What gets a tag?

Every gallery example should be tagged with:

- 1+ content tags
- structural, domain, or internal tag(s) if helpful

Tags can repeat existing forms of organization (e.g. an example is in the Animation folder and also gets an `animation` tag).

Tags are helpful to denote particularly good "byproduct" examples. E.g. the explicit purpose of a gallery example might be to demonstrate a colormap, but it's also a good demonstration of a legend. Tag `legend` to indicate that, rather than changing the title or the scope of the example.

**Tag Categories** - See Tag Glossary for a complete list of tags.

I. API tags: what content from the API reference is in the example?

II. Structural tags: what format is the example? What context can we provide?

III. Domain tags: what discipline(s) might seek this example consistently?

IV. Internal tags: what information is helpful for maintainers or contributors?

#### Proposing new tags

1. Review existing tag list, looking out for similar entries (i.e. `axes` and `axis`).
2. If a relevant tag or subcategory does not yet exist, propose it. Each tag is two parts: `subcategory:tag`. Tags should be one or two words.
3. New tags should be be added when they are relevant to existing gallery entries too. Avoid tags that will link to only a single gallery entry.
4. Tags can recreate other forms of organization.

Note: Tagging organization aims to work for 80-90% of cases. Some examples fall outside of the tagging structure. Niche or specific examples shouldn't be given standalone tags that won't apply to other examples.

#### How to tag?

Put each tag as a directive at the bottom of the page.

#### Related content

#### What is a gallery example?

The gallery of examples contains visual demonstrations of matplotlib features. Gallery examples exist so that users can scan through visual examples.

Unlike tutorials or user guides, gallery examples teach by demonstration, rather than by explanation or instruction.

Gallery examples should avoid instruction or excessive explanation except for brief clarifying code comments. Instead, they can tag related concepts and/or link to relevant tutorials or user guides.

#### Format

All *Examples* should aim to follow the following format:

- Title: 1-6 words, descriptive of content
- Subtitle: 10-50 words, action-oriented description of the example subject
- Image: a clear demonstration of the subject, showing edge cases and different applications if possible
- Code + Text (optional): code, commented as appropriate + written text to add context if necessary

Example:

The `bbox_intersect` gallery example demonstrates the point of visual examples:

- this example is "messy" in that it's hard to categorize, but the gallery is the right spot for it because it makes sense to find it by visual search
- https://matplotlib.org/devdocs/gallery/misc/bbox_intersect.html#sphx-glr-gallery-misc-bbox-intersect-py

**Triage And Review**

### 10.3.7 Bug triaging and issue curation

The issue tracker is important to communication in the project because it serves as the centralized location for making feature requests, reporting bugs, identifying major projects to work on, and discussing priorities. For this reason, it is important to curate the issue list, adding labels to issues and closing issues that are resolved or unresolvable.

Writing well defined issues increases their chances of being successfully resolved. Guidelines on writing a good issue can be found in *here*. The recommendations in this page are adapted from the scikit learn and Pandas contributing guides.

### Improve issue reports

Triaging issues does not require any particular expertise in the internals of Matplotlib, is extremely valuable to the project, and we welcome anyone to participate in issue triage! However, people who are not part of the Matplotlib organization do not have permissions to change milestones, add labels, or close issue.

If you do not have enough GitHub permissions do something (e.g. add a label, close an issue), please leave a comment with your recommendations!

The following actions are typically useful:

- documenting issues that are missing elements to reproduce the problem such as code samples
- suggesting better use of code formatting (e.g. triple back ticks in the markdown).
- suggesting to reformulate the title and description to make them more explicit about the problem to be solved
- linking to related issues or discussions while briefly describing how they are related, for instance "See also #xyz for a similar attempt at this" or "See also #xyz where the same thing was reported" provides context and helps the discussion
- verifying that the issue is reproducible
- classify the issue as a feature request, a long standing bug or a regression

**Fruitful discussions**

Online discussions may be harder than it seems at first glance, in particular given that a person new to open-source may have a very different understanding of the process than a seasoned maintainer.

Overall, it is useful to stay positive and assume good will. The following article explores how to lead online discussions in the context of open source.

### Maintainers and triage team members

In addition to the above, maintainers and the triage team can do the following important tasks:

- Update labels for issues and PRs: see the list of available GitHub labels.
- Triage issues:
  - **reproduce the issue**, if the posted code is a bug label the issue with "status: confirmed bug".
  - **identify regressions**, determine if the reported bug used to work as expected in a recent version of Matplotlib and if so determine the last working version. Regressions should be milestoned for the next bug-fix release and may be labeled as "Release critical".
  - **close usage questions** and politely point the reporter to use discourse or Stack Overflow instead and label as "community support".
  - **close duplicate issues**, after checking that they are indeed duplicate. Ideally, the original submitter moves the discussion to the older, duplicate issue

- **close issues that cannot be replicated**, after leaving time (at least a week) to add extra information

**Closing issues: a tough call**

When uncertain on whether an issue should be closed or not, it is best to strive for consensus with the original poster, and possibly to seek relevant expertise. However, when the issue is a usage question or has been considered as unclear for many years, then it should be closed.

### Preparing PRs for review

Reviewing code is also encouraged. Contributors and users are welcome to participate to the review process following our *review guidelines*.

### Triage workflow

The following workflow is a good way to approach issue triaging:

1. Thank the reporter for opening an issue

   The issue tracker is many people's first interaction with the Matplotlib project itself, beyond just using the library. As such, we want it to be a welcoming, pleasant experience.

2. Is this a usage question? If so close it with a polite message.

3. Is the necessary information provided?

   Check that the poster has filled in the issue template. If crucial information (the version of Python, the version of Matplotlib used, the OS, and the backend), is missing politely ask the original poster to provide the information.

4. Is the issue minimal and reproducible?

   For bug reports, we ask that the reporter provide a minimal reproducible example. See this useful post by Matthew Rocklin for a good explanation. If the example is not reproducible, or if it's clearly not minimal, feel free to ask the reporter if they can provide an example or simplify the provided one. Do acknowledge that writing minimal reproducible examples is hard work. If the reporter is struggling, you can try to write one yourself.

   If a reproducible example is provided, but you see a simplification, add your simpler reproducible example.

   If you cannot reproduce the issue, please report that along with your OS, Python, and Matplotlib versions.

   If we need more information from either this or the previous step please label the issue with "status: needs clarification".

5. Is this a regression?

While we strive for a bug-free library, regressions are the highest priority. If we have broken user-code that *used to* work, we should fix that in the next micro release!

Try to determine when the regression happened by running the reproduction code against older versions of Matplotlib. This can be done by released versions of Matplotlib (to get the version it last worked in) or by using git bisect to find the first commit where it was broken.

6. Is this a duplicate issue?

   We have many open issues. If a new issue seems to be a duplicate, point to the original issue. If it is a clear duplicate, or consensus is that it is redundant, close it. Make sure to still thank the reporter, and encourage them to chime in on the original issue, and perhaps try to fix it.

   If the new issue provides relevant information, such as a better or slightly different example, add it to the original issue as a comment or an edit to the original post.

   Label the closed issue with "status: duplicate"

7. Make sure that the title accurately reflects the issue. If you have the necessary permissions edit it yourself if it's not clear.

8. Add the relevant labels, such as "Documentation" when the issue is about documentation, "Bug" if it is clearly a bug, "New feature" if it is a new feature request, ...

   If the issue is clearly defined and the fix seems relatively straightforward, label the issue as “Good first issue” (and possibly a description of the fix or a hint as to where in the code base to look to get started).

   An additional useful step can be to tag the corresponding module e.g. the "GUI/Qt" label when relevant.

### Triage team

If you would like to join the triage team:

1. Correctly triage 2-3 issues.
2. Ask someone on in the Matplotlib organization (publicly or privately) to recommend you to the triage team (look for "Member" on the top-right of comments on GitHub). If you worked with someone on the issues triaged, they would be a good person to ask.
3. Responsibly exercise your new power!

Anyone with commit or triage rights may nominate a user to be invited to join the triage team by emailing matplotlib-steering-council@numfocus.org .

### 10.3.8 Pull request guidelines

Pull requests (PRs) on GitHub are the mechanism for contributing to Matplotlib's code and documentation.

We value contributions from people with all levels of experience. In particular, if this is your first PR not everything has to be perfect. We'll guide you through the PR process. Nevertheless, please try to follow our guidelines as well as you can to help make the PR process quick and smooth. If your pull request is incomplete or a work-in-progress, please mark it as a draft pull requests on GitHub and specify what feedback from the developers would be helpful.

Please be patient with reviewers. We try our best to respond quickly, but we have limited bandwidth. If there is no feedback within a couple of days, please ping us by posting a comment to your PR or reaching out on a *communication channel*

#### Summary for pull request authors

We recommend that you check that your contribution complies with the following guidelines before submitting a pull request:

- Changes, both new features and bugfixes, should have good test coverage. See *Testing* for more details.
- Update the *documentation* if necessary.
- All public methods should have informative docstrings with sample usage when appropriate. Use the *docstring standards*.
- For high-level plotting functions, consider adding a small example to the *examples gallery*.
- If you add a major new feature or change the API in a backward-incompatible way, please document it as described in *API guidelines*.
- Code should follow our conventions as documented in our *Coding guidelines*.
- When adding or changing public function signatures, add *type hints*.
- When adding keyword arguments, see our guide to *Keyword argument processing*.

When opening a pull request on Github, please ensure that:

- Changes were made on a *feature branch*.
- *pre-commit* checks for spelling, formatting, etc pass
- The pull request targets the *main branch*
- If your pull request addresses an issue, please use the title to describe the issue (e.g. "Add ability to plot timedeltas") and mention the issue number in the pull request description to ensure that a link is created to the original issue (e.g. "Closes #8869" or "Fixes #8869"). This will ensure the original issue mentioned is automatically closed when your PR is merged. For more details, see linking an issue and pull request.
- *Automated tests* pass

For guidance on creating and managing a pull request, please see our *contributing* and *pull request workflow* guides.

### Summary for pull request reviewers

**Please help review and merge PRs!**

If you have commit rights, then you are trusted to use them. Please be patient and kind with contributors.

When reviewing, please ensure that the pull request satisfies the following requirements before merging it:

### Content

- Is the feature / bugfix reasonable?
- Does the PR conform with the *Coding guidelines*?
- Is the *documentation* (docstrings, examples, what's new, API changes) updated?
- Is the change purely stylistic? Generally, such changes are discouraged when not part of other non-stylistic work because it obscures the git history of functional changes to the code. Reflowing a method or docstring as part of a larger refactor/rewrite is acceptable.

### Workflow

- Make sure all *automated tests* pass.
- The PR should *target the main branch*.
- Tag with descriptive *labels*.
- Set the *milestone*.
- Keep an eye on the *number of commits*.
- Approve if all of the above topics are handled.
- *Merge* if a sufficient number of approvals is reached.

### Detailed guidelines

### Documentation

- Every new feature should be documented. If it's a new module, don't forget to add a new rst file to the API docs.
- Each high-level plotting function should have a small example in the `Examples` section of the docstring. This should be as simple as possible to demonstrate the method. More complex examples should go into a dedicated example file in the `examples` directory, which will be rendered to the examples gallery in the documentation.
- Build the docs and make sure all formatting warnings are addressed.
- See *Write documentation* for our documentation style guide.

### Labels

- If you have the rights to set labels, tag the PR with descriptive labels. See the list of labels.
- If the PR makes changes to the wheel building Action, add the "Run cibuildwheel" label to enable testing wheels.

### Milestones

Set the milestone according to these guidelines:

- *New features and API changes* are milestoned for the next meso release `v3.N.0`.
- *Bugfixes, tests for released code, and docstring changes* may be milestoned for the next micro release `v3.N.M`.
- *Documentation changes* (only .rst files and examples) may be milestoned `v3.N-doc`.

If multiple rules apply, choose the first matching from the above list. See *Backport strategy* for detailed guidance on what should or should not be backported.

The milestone marks the release a PR should go into. It states intent, but can be changed because of release planning or re-evaluation of the PR scope and maturity.

All Pull Requests should target the main branch. The milestone tag triggers an *automatic backport* for milestones which have a corresponding branch.

### Merging

- Documentation and examples may be merged by the first reviewer. Use the threshold "is this better than it was?" as the review criteria.
- For code changes (anything in `src` or `lib`) at least two core developers (those with commit rights) should review all pull requests. If you are the first to review a PR and approve of the changes use the GitHub 'approve review' tool to mark it as such. If you are a subsequent reviewer please approve the review and if you think no more review is needed, merge the PR.

  Ensure that all API changes are documented in a file in one of the subdirectories of `doc/api/next_api_changes`, and significant new features have an entry in `doc/user/whats_new`.

  - If a PR already has a positive review, a core developer (e.g. the first reviewer, but not necessarily) may champion that PR for merging. In order to do so, they should ping all core devs both on GitHub and on the dev mailing list, and label the PR with the "Merge with single review?" label. Other core devs can then either review the PR and merge or reject it, or simply request that it gets a second review before being merged. If no one asks for such a second review within a week, the PR can then be merged on the basis of that single review.

    A core dev should only champion one PR at a time and we should try to keep the flow of championed PRs reasonable.

- Do not self merge, except for 'small' patches to un-break the CI or when another reviewer explicitly allows it (ex, "Approve modulo CI passing, may self merge when green").

### Automated tests

Before being merged, a PR should pass the *Automated tests*. If you are unsure why a test is failing, ask on the PR or in our *Official communication channels*

### Number of commits and squashing

- Squashing is case-by-case. The balance is between burden on the contributor, keeping a relatively clean history, and keeping a history usable for bisecting. The only time we are really strict about it is to eliminate binary files (ex multiple test image re-generations) and to remove upstream merges.
- Do not let perfect be the enemy of the good, particularly for documentation or example PRs. If you find yourself making many small suggestions, either open a PR against the original branch, push changes to the contributor branch, or merge the PR and then open a new PR against upstream.
- If you push to a contributor branch leave a comment explaining what you did, ex "I took the liberty of pushing a small clean-up PR to your branch, thanks for your work.". If you are going to make substantial changes to the code or intent of the PR please check with the contributor first.

### Branches and backports

### Current branches

The current active branches are

***main***

The current development version. Future meso (*v3.N.0*) or macro (*v4.0.0*) will be branched from this.

***v3.N.x***

Maintenance branch for Matplotlib 3.N. Future micro releases will be tagged from this.

***v3.N.M-doc***

Documentation for the current micro release. On a micro release, this will be replaced by a properly named branch for the new release.

### Branch selection for pull requests

Generally, all pull requests should target the main branch.

Other branches are fed through *automatic* or *manual*. Directly targeting other branches is only rarely necessary for special maintenance work.

### Backport strategy

Backports to the micro release branch (*v3.N.x*) are the changes that will be included in the next patch (aka bug-fix) release. The goal of the patch releases is to fix bugs without adding any new regressions or behavior changes. We will always attempt to backport:

- critical bug fixes (segfault, failure to import, things that the user cannot work around)
- fixes for regressions introduced in the last two meso releases

and may attempt to backport fixes for regressions introduced in older releases.

In the case where the backport is not clean, for example if the bug fix is built on top of other code changes we do not want to backport, balance the effort and risk of re-implementing the bug fix vs the severity of the bug. When in doubt, err on the side of not backporting.

When backporting a Pull Request fails or is declined, re-milestone the original PR to the next meso release and leave a comment explaining why.

The only changes backported to the documentation branch (*v3.N.M-doc*) are changes to `doc` or `galleries`. Any changes to `lib` or `src`, including docstring-only changes, must not be backported to this branch.

### Automated backports

We use MeeseeksDev bot to automatically backport merges to the correct maintenance branch base on the milestone. To work properly the milestone must be set before merging. If you have commit rights, the bot can also be manually triggered after a merge by leaving a message `@meeseeksdev backport to BRANCH` on the PR. If there are conflicts MeeseeksDev will inform you that the backport needs to be done manually.

The target branch is configured by putting `on-merge: backport to TARGETBRANCH` in the milestone description on it's own line.

If the bot is not working as expected, please report issues to MeeseeksDev.

### Manual backports

When doing backports please copy the form used by MeeseeksDev, `Backport PR #XXXX: TITLE OF PR`. If you need to manually resolve conflicts make note of them and how you resolved them in the commit message.

We do a backport from main to v2.2.x assuming:

- `matplotlib` is a read-only remote branch of the matplotlib/matplotlib repo

The `TARGET_SHA` is the hash of the merge commit you would like to backport. This can be read off of the GitHub PR page (in the UI with the merge notification) or through the git CLI tools.

Assuming that you already have a local branch `v2.2.x` (if not, then `git checkout -b v2.2.x`), and that your remote pointing to `https://github.com/matplotlib/matplotlib` is called `upstream`:

```
git fetch upstream
git checkout v2.2.x  # or include -b if you don't already have this.
git reset --hard upstream/v2.2.x
git cherry-pick -m 1 TARGET_SHA
# resolve conflicts and commit if required
```

Files with conflicts can be listed by `git status`, and will have to be fixed by hand (search on >>>>>). Once the conflict is resolved, you will have to re-add the file(s) to the branch and then continue the cherry pick:

```
git add lib/matplotlib/conflicted_file.py
git add lib/matplotlib/conflicted_file2.py
git cherry-pick --continue
```

Use your discretion to push directly to upstream or to open a PR; be sure to push or PR against the `v2.2.x` upstream branch, not `main`!

**Maintenance**

## 10.3.9 Release guide

---

**This document is only relevant for Matplotlib release managers.**

A guide for developers who are doing a Matplotlib release.

---

### Versioning Scheme

Maplotlib follows the Intended Effort Versioning (EffVer) versioning scheme: *macro.meso.micro*.

***macro***

A release that we expect a large effort from our users to upgrade to. The v1 to v2 transition included a complete overhaul of the default styles and the v2 to v3 transition involved dropping support for Python 2.

Future macro versions would include changes of a comparable scale that can not be done incrementally in meso releases.

***meso***

A release that we expect some effort from our users to upgrade to. We target a *Meso* release every 6 months. These release are primarily intended to release new features to our users, however they also contain intentional feature deprecations and removals per *our policy*.

***micro***

A release that we expect users to require little to no effort to upgrade to. Per our *Backport strategy* we only backport bug fixes to the maintenance branch. We expect minimal impact on users other than possibly breaking work arounds to a fixed bug or bugs being used as features.

These are released as-needed, but typically every 1-2 months between meso releases.

### Making the release branch

---

**Note:** This assumes that a read-only remote for the canonical repository is `remote` and a read/write remote is `DANGER`

---

When a new meso release (vX.Y.0) is approaching, a new release branch must be made. When precisely this should happen is up to the release manager, but this point is where most new features intended for the meso release are merged and you are entering a feature freeze (i.e. newly implemented features will be going into vX.Y+1). This does not necessarily mean that no further changes will be made prior to release, just that those changes will be made using the backport system.

For an upcoming `v3.7.0` release, first create the branch:

```
git switch main
git pull
git switch -c v3.7.x
git push DANGER v3.7.x
```

Update the `v3.7.0` milestone so that the description reads:

```
New features and API changes

on-merge: backport to v3.7.x
```

Micro versions should instead read:

```
Bugfixes and docstring changes

on-merge: backport to v3.7.x
```

Check all active milestones for consistency. Older milestones should also backport to higher meso versions (e.g. `v3.6.3` and `v3.6-doc` should backport to both `v3.6.x` and `v3.7.x` once the `v3.7.x` branch exists and while PR backports are still targeting `v3.6.x`)

Create the milestone for the next-next meso release (i.e. `v3.9.0`, as `v3.8.0` should already exist). While most active items should go in the next meso release, this milestone can help with longer term planning, especially around deprecation cycles.

### Testing

We use GitHub Actions for continuous integration. When preparing for a release, the final tagged commit should be tested locally before it is uploaded:

```
pytest -n 8 .
```

In addition the following test should be run and manually inspected:

```
python tools/memleak.py agg 1000 agg.pdf
```

Run the User Acceptance Tests for the NBAgg and ipympl backends:

```
jupyter notebook lib/matplotlib/backends/web_backend/nbagg_uat.ipynb
```

For ipympl, restart the kernel, add a cell for `%matplotlib widget` and do not run the cell with `matplotlib.use('nbagg')`. Tests which check `connection_info`, use `reshow`, or test the OO interface are not expected to work for `ipympl`.

### GitHub statistics

We automatically extract GitHub issue, PRs, and authors from GitHub via the API. To prepare this list:

1. Archive the existing GitHub statistics page.
   a. Copy the current `doc/users/github_stats.rst` to `doc/users/prev_whats_new/github_stats_X.Y.Z.rst`.
   b. Change the link target at the top of the file.
   c. Remove the "Previous GitHub Stats" section at the end.

   For example, when updating from v3.7.0 to v3.7.1:

```
cp doc/users/github_stats.rst doc/users/prev_whats_new/github_stats_3.7.
 ↪0.rst
$EDITOR doc/users/prev_whats_new/github_stats_3.7.0.rst
```

(continues on next page)

(continued from previous page)

```
# Change contents as noted above.
git add doc/users/prev_whats_new/github_stats_3.7.0.rst
```

2. Re-generate the updated stats:

```
python tools/github_stats.py --since-tag v3.7.0 --milestone=v3.7.1 \
    --project 'matplotlib/matplotlib' --links > doc/users/github_stats.
↪rst
```

3. Review and commit changes. Some issue/PR titles may not be valid reST (the most common issue is `*` which is interpreted as unclosed markup).

---

**Note:** Make sure you authenticate against the GitHub API. If you do not, you will get blocked by GitHub for going over the API rate limits. You can authenticate in one of two ways:

- using the `keyring` package; `pip install keyring` and then when running the stats script, you will be prompted for user name and password, that will be stored in your system keyring, or,
- using a personal access token; generate a new token on this GitHub page with the `repo:public_repo` scope and place the token in `~/.ghoauth`.

---

### Update and validate the docs

#### Merge `*-doc` branch

Merge the most recent 'doc' branch (e.g., `v3.7.0-doc`) into the branch you are going to tag on and delete the doc branch on GitHub.

### Update supported versions in Security Policy

When making macro or meso releases, update the supported versions in the Security Policy in `SECURITY.md`.

For meso version release update the table in `SECURITY.md` to specify that the two most recent meso releases in the current macro version series are supported.

For a macro version release update the table in `SECURITY.md` to specify that the last meso version in the previous macro version series is still supported. Dropping support for the last version of a macro version series will be handled on an ad-hoc basis.

### Update release notes

#### What's new

*Only needed for macro and meso releases. Bugfix releases should not have new features.*

Merge the contents of all the files in `doc/users/next_whats_new/` into a single file `doc/users/prev_whats_new/whats_new_X.Y.0.rst` and delete the individual files.

#### API changes

*Primarily needed for macro and meso releases. We may sometimes have API changes in micro releases.*

Merge the contents of all the files in `doc/api/next_api_changes/` into a single file `doc/api/prev_api_changes/api_changes_X.Y.Z.rst` and delete the individual files.

#### Release notes TOC

Update `doc/users/release_notes.rst`:

- For macro and meso releases add a new section

```
X.Y
===
.. toctree::
    :maxdepth: 1

    prev_whats_new/whats_new_X.Y.0.rst
    ../api/prev_api_changes/api_changes_X.Y.0.rst
    prev_whats_new/github_stats_X.Y.0.rst
```

- For micro releases add the GitHub stats and (if present) the API changes to the existing X.Y section

```
../api/prev_api_changes/api_changes_X.Y.Z.rst
prev_whats_new/github_stats_X.Y.Z.rst
```

#### Update version switcher

Update `doc/_static/switcher.json`:

- If a micro release, *X.Y.Z*, no changes are needed.
- If a macro release, *X.Y.0*, change the name of `name: X.Y+1 (dev)` and `name: X.Y (stable)` as well as adding a new version for the previous stable (`name: X.Y-1`).

### Verify that docs build

Finally, make sure that the docs build cleanly:

```
make -Cdoc O=-j$(nproc) html latexpdf
```

After the docs are built, check that all of the links, internal and external, are still valid. We use `linkchecker` for this:

```
pip install linkchecker
pushd doc/build/html
linkchecker index.html --check-extern
popd
```

Address any issues which may arise. The internal links are checked on Circle CI, so this should only flag failed external links.

### Create release commit and tag

To create the tag, first create an empty commit with a very terse set of the release notes in the commit message:

```
git commit --allow-empty
```

and then create a signed, annotated tag with the same text in the body message:

```
git tag -a -s v3.7.0
```

which will prompt you for your GPG key password and an annotation. For pre-releases it is important to follow **PEP 440** so that the build artifacts will sort correctly in PyPI.

To prevent issues with any down-stream builders which download the tarball from GitHub it is important to move all branches away from the commit with the tag[1]:

```
git commit --allow-empty
```

Finally, push the tag to GitHub:

```
git push DANGER v3.7.x v3.7.0
```

[1] The tarball that is provided by GitHub is produced using git archive. We use setuptools_scm which uses a format string in `lib/matplotlib/_version.py` to have `git` insert a list of references to exported commit (see `.gitattributes` for the configuration). This string is then used by `setuptools_scm` to produce the correct version, based on the git tag, when users install from the tarball. However, if there is a branch pointed at the tagged commit, then the branch name will also be included in the tarball. When the branch eventually moves, anyone who checked the hash of the tarball before the branch moved will have an incorrect hash.

To generate the file that GitHub does use:

```
git archive v3.7.0 -o matplotlib-3.7.0.tar.gz --prefix=matplotlib-3.7.0/
```

Congratulations, the scariest part is done! This assumes the release branch has already been made. Usually this is done at the time of feature freeze for a meso release (which often coincides with the last micro release of the previous meso version)

If this is a final release, also create a 'doc' branch (this is not done for pre-releases):

```
git branch v3.7.0-doc
git push DANGER v3.7.0-doc
```

Update (or create) the `v3.7-doc` milestone. The description should include the instruction for meeseeksmachine to backport changes with the `v3.7-doc` milestone to both the `v3.7.x` branch and the `v3.7.0-doc` branch:

```
Documentation changes (.rst files and examples)

on-merge: backport to v3.7.x
on-merge: backport to v3.7.0-doc
```

Check all active milestones for consistency. Older doc milestones should also backport to higher meso versions (e.g. `v3.6-doc` should backport to both `v3.6.x` and `v3.7.x` if the `v3.7.x` branch exists)

### Release management / DOI

Via the GitHub UI, turn the newly pushed tag into a release. If this is a pre-release remember to mark it as such.

For final releases, also get the DOI from Zenodo (which will automatically produce one once the tag is pushed). Add the DOI post-fix and version to the dictionary in `tools/cache_zenodo_svg.py` and run the script.

This will download the new SVG to `doc/_static/zenodo_cache/`*`postfix`*`.svg` and edit `doc/project/citing.rst`. Commit the new SVG, the change to `tools/cache_zenodo_svg.py`, and the changes to `doc/project/citing.rst` to the VER-doc branch and push to GitHub.

```
git checkout v3.7.0-doc
$EDITOR tools/cache_zenodo_svg.py
python tools/cache_zenodo_svg.py
git commit -a
git push DANGER v3.7.0-doc:v3.7.0-doc
```

### Building binaries

We distribute macOS, Windows, and many Linux wheels as well as a source tarball via PyPI. Most builders should trigger automatically once the tag is pushed to GitHub:

- Windows, macOS and manylinux wheels are built on GitHub Actions. Builds are triggered by the GitHub Action defined in `.github/workflows/cibuildwheel.yml`, and wheels will be available as artifacts of the build.

- The auto-tick bot should open a pull request into the conda-forge feedstock. Review and merge (if you have the power to).

**Warning:** Because this is automated, it is extremely important to bump all branches away from the tag as discussed in *Create release commit and tag*.

### Make distribution and upload to PyPI

Once you have collected all of the wheels (expect this to take a few hours), generate the tarball:

```
git checkout v3.7.0
git clean -xfd
python -m build --sdist
```

and copy all of the wheels into `dist` directory. First, check that the dist files are OK:

```
twine check dist/*
```

and then use `twine` to upload all of the files to PyPI

```
twine upload -s dist/matplotlib*tar.gz
twine upload dist/*whl
```

Congratulations, you have now done the second scariest part!

### Build and deploy documentation

To build the documentation you must have the tagged version installed, but build the docs from the `ver-doc` branch. An easy way to arrange this is:

```
pip install matplotlib
pip install -r requirements/doc/doc-requirements.txt
git checkout v3.7.0-doc
git clean -xfd
make -Cdoc O="-t release -j$(nproc)" html latexpdf LATEXMKOPTS="-silent -f"
```

which will build both the HTML and PDF version of the documentation.

The built documentation exists in the matplotlib.github.com repository. Pushing changes to main automatically updates the website.

The documentation is organized in subdirectories by version. The latest stable release is symlinked from the `stable` directory. The documentation for current main is built on Circle CI and pushed to the devdocs repository. These are available at matplotlib.org/devdocs.

Assuming you have this repository checked out in the same directory as matplotlib

```
cd ../matplotlib.github.com
cp -a ../matplotlib/doc/build/html 3.7.0
rm 3.7.0/.buildinfo
cp ../matplotlib/doc/build/latex/Matplotlib.pdf 3.7.0
```

which will copy the built docs over. If this is a final release, link the `stable` subdirectory to the newest version:

```
rm stable
ln -s 3.7.0 stable
```

You will also need to edit `sitemap.xml` to include the newly released version. Now commit and push everything to GitHub

```
git add *
git commit -a -m 'Updating docs for v3.7.0'
git push DANGER main
```

Congratulations you have now done the third scariest part!

If you have access, clear the CloudFlare caches.

It typically takes about 5-10 minutes for the website to process the push and update the live web page (remember to clear your browser cache).

### Merge up changes to main

After a release is done, the changes from the release branch should be merged into the `main` branch. This is primarily done so that the released tag is on the main branch so `git describe` (and thus `setuptools-scm`) has the most current tag. Secondarily, changes made during release (including removing individualized release notes, fixing broken links, and updating the version switcher) are bubbled up to `main`.

Git conflicts are very likely to arise, though aside from changes made directly to the release branch (mostly as part of the release), they should be relatively-easily resolved by using the version from `main`. This is not a universal rule, and care should be taken to ensure correctness:

```
git switch main
git pull
git switch -c merge_up_v3.7.0
git merge v3.7.x
# resolve conflicts
git merge --continue
```

Due to branch protections for the `main` branch, this is merged via a standard pull request, though the PR cleanliness status check is expected to fail. The PR should not be squashed because the intent is to merge the branch histories.

#### Publicize this release

After the release is published to PyPI and conda, it should be announced through our communication channels:

- Send a short version of the release notes and acknowledgments to all the *Mailing lists*
- Post highlights and link to *What's new* on the active *social media accounts*
- Add a release announcement to the "News" section of matplotlib.org by editing `docs/body.html`. Link to the auto-generated announcement discourse post, which is in Announcements > matplotlib-announcements.

#### Conda packages

The Matplotlib project itself does not release conda packages. In particular, the Matplotlib release manager is not responsible for conda packaging.

For information on the packaging of Matplotlib for conda-forge see https://github.com/conda-forge/matplotlib-feedstock.

### 10.3.10 Community management guide

These guidelines are applicable when **acting as a representative** of Matplotlib, for example at sprints or when giving official talks or tutorials, and in any community venue managed by Matplotlib.

Our approach to community engagement is foremost guided by our *Mission Statement*:

- We demonstrate that we care about visualization as a practice
- We deepen our practice and the community's capacity to support users, facilitate exploration, produce high quality visualizations, and be understandable and extensible
- We showcase advanced use of the library without adding maintenance burden to the documentation and recognize contributions that happen outside of the github workflow.
- We use communications platforms to maintain relationships with contributors who may no longer be active on GitHub, build relationships with potential contributors, and connect with other projects and communities who use Matplotlib.
- In prioritizing understandability and extensiblity, we recognize that people using Matplotlib, in whatever capacity, are part of our community. Doing so empowers our community members to build community with each other, for example by creating educational resources, building third party tools, and building informal mentoring networks.

### Official communication channels

The Scientific Python community uses various communications platforms to stay updated on new features and projects, to contribute by telling us what is on their mind and suggest issues and bugs, and to showcase their use cases and the tools they have built.

The following venues are managed by Matplotlib maintainers and contributors:

- library and docs: https://github.com/matplotlib/matplotlib
- forum: https://discourse.matplotlib.org/
- chat: https://matrix.to/#/#matplotlib:matrix.org
- blog: https://blog.scientific-python.org/

### Social media

### Active social media

- https://twitter.com/matplotlib
- https://instagram.com/matplotart/

### Official accounts

- https://bsky.app/profile/matplotlib.bsky.social
- https://fosstodon.org/@matplotlib
- https://www.tiktok.com/@matplotart
- https://www.youtube.com/matplotlib

### Mailing lists

- matplotlib-announce@python.org
- matplotlib-users@python.org
- matplotlib-devel@python.org

### Social media coordination

- Team mailing list: matplotlib-social@numfocus.org
- Public chat room: https://matrix.to/#/#matplotlib_community:gitter.im

### Maintenance

If you are interested in moderating the chat or forum or accessing the social media accounts:

- Matplotlib maintainers should reach out to the community-manager.
- Everyone else should send an email to matplotlib-social-admin@numfocus.org:
  - Introduce yourself - github handle and participation in the community.
  - Describe the reason for wanting to moderate or contribute to social.

### Content guidelines

Communication on official channels, such as the Matplotlib homepage or on Matplotlib social accounts, should conform to the following standards. If you are unsure if content that you would like to post or share meets these guidelines, ask on the *Social media coordination* channels before posting.

### General guidelines

- Focus on Matplotlib, 3rd party packages, and visualizations made with Matplotlib.
- These are also acceptable topics:
  - Visualization best practices and libraries.
  - Projects and initiatives by NumFOCUS and Scientific Python.
  - How to contribute to open source projects.
  - Projects, such as scientific papers, that use Matplotlib.
- No gratuitous disparaging of other visualization libraries and tools, but criticism is acceptable so long as it serves a constructive purpose.
- Follow communication best practices:
  - Do not share non-expert visualizations when it could cause harm:
    - e.g. https://twitter.com/matplotlib/status/1244178154618605568
  - Clearly state when the visualization data/conclusions cannot be verified.
  - Do not rely on machine translations for sensitive visualization.
- Verify sourcing of content (especially on instagram & blog):
  - Instagram/blog: ensure mpl has right to repost/share content

  - Make sure content is clearly cited:
    * e.g. a tutorial reworking an example must credit the original example
- Limited self/corporate promotion is acceptable.
  - Should be no more than about a quarter of the content.

### Visual media guidelines

Visual media, such as images and videos, must not violate the *code of conduct*, nor any platform's rules. Specifically:

- Visual media must conform to the guidelines of all sites it may be posted on:
  - https://help.twitter.com/en/rules-and-policies/twitter-rules
  - https://help.instagram.com/477434105621119
- Emphasize the visualization techniques demonstrated by the visual media.
- Clearly state that sharing is not an endorsement of the content.
  - e.g. bitcoin related visualizations

### Accessibility

Visual media in communications should be made as accessible as possible:

- Add alt text to images and videos when the platform allows:
  - alt text for data viz
  - general alt text guide
- Warn on bright, strobing, images & turn off autoplay if possible.
- For images and videos made by the social media team:
  - Make graphic perceivable to people who cannot perceive color well due to color-blindness, low vision, or any other reason.
  - Do not make bright, strobing images.
  - More guidelines at https://webaim.org/techniques/images/.

### Social media

Matplotlib aims for a single voice across all social media platforms to build and maintain a consistent brand identity for Matplotlib as an organization. This depersonalization is the norm on social media platforms because it enables constructive and productive conversations; People generally feel more comfortable giving negative and constructive feedback to a brand than to specific contributors.

The current Matplotlib voice and persona aims to be kind, patient, supportive and educational. This is so that it can de-escalate tensions and facilitate constructive conversations; being perceived as negative or argumentative can escalate very fast into long-lasting brand damage, being perceived as personal leads to aggression and accusations faster than an impersonal account, and being perceived as friendly and approachable leads to higher engagement. Instead of speaking with a directive authority, which can be intimidating and lead to negative engagement, it speaks as a peer or educator to empower participation. The current voice encourages more input from folks we engage with, and also makes it possible for folks who are not in the core team to participate in managing the account.

While the *brand identity* is casual, the showcased content is high quality, peer-led resource building. Please follow these guidelines to maintain a consistent brand identity across platforms.

### Persona

On social media, Matplotlib:

- Acts as a sentient visualization library, so talks about itself as a we, us, our, and it. Avoids talking about itself in the 3rd person. Never uses 1st person.
- Is very earnest, eager to please, and aims to be patient & painfully oblivious to snark and sarcasm.
- Gets over-excited over shiny visualizations - lots of emojis and the like - and encourages folks to share their work.
- Highlights various parts of the library, especially the more obscure bits and bobbles.
- Acknowledges that it is a sometimes frustrating tangle of bits & bobbles that can confuse even the folks who work on it & signal boosts their confuzzlment.

### Behavior

When acting as a representative of the library, keep responses polite and assume user statements are in good faith unless they violate the *code of conduct*.

### Social graph

Only follow **organizations and projects**, do not follow individual accounts for any reason, even maintainers/project leads/famous Python people!

Following these types of accounts is encouraged:

- NumFocus and Scientific Python projects
- 3rd party packages
- Visualization related projects and organizations
- Open Source community projects
- Sponsors

### Recurring campaigns

Typically the social media accounts will promote the following:

- Matplotlib releases:
  - Highlight new features & major deprecations
  - Link to download/install instructions
  - Ask folks to try it out.
- third party packages
- NumFocus/Scientific Python/open source visualization project releases
- GSOC/GSOD recruiting and progress

### Retired campaigns

- John Hunter Excellence in Plotting, submission and winners

### Changing the guidelines

As the person tasked with implementing these guidelines, the community-manager should be alerted to proposed changes. Similarly, specific platform guidelines (e.g. twitter, instagram) should be reviewed by the person responsible for that platform, when different from the community manager. If there is no consensus, decisions about guidelines revert to the community manager.

### 10.3.11 Dependency version policy

For the purpose of this document, 'minor version' is in the sense of SemVer (major, minor, patch) or 'meso version' in the sense of EffVer (macro, meso, micro). It includes both major/macro and minor/meso releases. For projects that use date-based versioning, every release is a 'minor version'.

Matplotlib follows NEP 29.

#### Python and NumPy

Matplotlib supports:

- All minor versions of Python released 42 months prior to the project, and at minimum the two latest minor versions.
- All minor versions of `numpy` released in the 24 months prior to the project, and at minimum the last three minor versions.

In `pyproject.toml`, the `requires-python` variable should be set to the minimum supported version of Python. All supported minor versions of Python should be in the test matrix and have binary artifacts built for the release.

Minimum Python and NumPy version support should be adjusted upward on every major and minor release, but never on a patch release.

See also the *List of dependency versions*.

#### Python dependencies

For Python dependencies we should support at least:

**with compiled extensions**

minor versions initially released in the 24 months prior to our planned release date or the oldest that support our minimum Python + NumPy

**without compiled extensions**

minor versions initially released in the 12 months prior to our planned release date or the oldest that supports our minimum Python.

We will only bump these dependencies as we need new features or the old versions no longer support our minimum NumPy or Python.

### Test and documentation dependencies

As these packages are only needed for testing or building the docs and not needed by end-users, we can be more aggressive about dropping support for old versions. However, we need to be careful to not over-run what down-stream packagers support (as most of the run the tests and build the documentation as part of the packaging process).

We will support at least minor versions of the development dependencies released in the 12 months prior to our planned release.

We will only bump these as needed or versions no longer support our minimum Python and NumPy.

### System and C-dependencies

For system or C-dependencies (FreeType, GUI frameworks, LaTeX, Ghostscript, FFmpeg) support as old as practical. These can be difficult to install for end-users and we want to be usable on as many systems as possible. We will bump these on a case-by-case basis.

In the case of GUI frameworks for which we rely on Python bindings being available, we will also drop support for bindings so old that they don't support any Python version that we support.

### List of dependency versions

The following list shows the minimal versions of Python and NumPy dependencies for different versions of Matplotlib. Follow the links for the full specification of the dependencies.

| Matplotlib | Python | NumPy |
|---|---|---|
| 3.9 | 3.9 | 1.23.0 |
| 3.8 | 3.9 | 1.21.0 |
| 3.7 | 3.8 | 1.20.0 |
| 3.6 | 3.8 | 1.19.0 |
| 3.5 | 3.7 | 1.17.0 |
| 3.4 | 3.7 | 1.16.0 |
| 3.3 | 3.6 | 1.15.0 |
| 3.2 | 3.6 | 1.11.0 |
| 3.1 | 3.6 | 1.11.0 |
| 3.0 | 3.5 | 1.10.0 |
| 2.2 | 2.7, 3.4 | 1.7.1 |
| 2.1 | 2.7, 3.4 | 1.7.1 |
| 2.0 | 2.7, 3.4 | 1.7.1 |
| 1.5 | 2.7, 3.4 | 1.6 |
| 1.4 | 2.6, 3.3 | 1.6 |
| 1.3 | 2.6, 3.3 | 1.5 |
| 1.2 | 2.6, 3.1 | 1.4 |
| 1.1 | 2.4 | 1.1 |
| 1.0 | 2.4 | 1.1 |

### 10.3.12 Matplotlib Enhancement Proposals

Matplotlib Enhancement Proposals (MEP), inspired by cpython's PEP's but less formal, are design documents for large or controversial changes to Matplotilb. These documents should provide a discussion of both why and how the changes should be made.

To create a new MEP open a pull request (PR) adding a file based on *the template* to this the MEP directory. For the initial PR only a rough description is required and it should be merged quickly. Further detailed discussion can happen in follow on PRs.

#### MEP Template


- *Status*
- *Branches and Pull requests*
- *Abstract*
- *Detailed description*
- *Implementation*
- *Backward compatibility*
- *Alternatives*


This MEP template is a guideline of the sections that a MEP should contain. Extra sections may be added if appropriate, and unnecessary sections may be noted as such.

#### Status

MEPs go through a number of phases in their lifetime:

- **Discussion**: The MEP is being actively discussed on the mailing list and it is being improved by its author. The mailing list discussion of the MEP should include the MEP number (MEPxxx) in the subject line so they can be easily related to the MEP.
- **Progress**: Consensus was reached and implementation work has begun.
- **Completed**: The implementation has been merged into main.
- **Superseded**: This MEP has been abandoned in favor of another approach.
- **Rejected**: There are currently no plans to implement the proposal.

### Branches and Pull requests

All development branches containing work on this MEP should be linked to from here.

All pull requests submitted relating to this MEP should be linked to from here. (A MEP does not need to be implemented in a single pull request if it makes sense to implement it in discrete phases).

### Abstract

The abstract should be a short description of what the MEP will achieve.

### Detailed description

This section describes the need for the MEP. It should describe the existing problem that it is trying to solve and why this MEP makes the situation better. It should include examples of how the new functionality would be used and perhaps some use cases.

### Implementation

This section lists the major steps required to implement the MEP. Where possible, it should be noted where one step is dependent on another, and which steps may be optionally omitted. Where it makes sense, each step should include a link related pull requests as the implementation progresses.

### Backward compatibility

This section describes the ways in which the MEP breaks backward incompatibility.

### Alternatives

If there were any alternative solutions to solving the same problem, they should be discussed here, along with a justification for the chosen approach.

### MEP8: PEP8


- *Status*
- *Branches and Pull requests*
- *Abstract*
- *Detailed description*
- *Implementation*

- *Backward compatibility*
- *Alternatives*


## Status

**Superseded**

Current guidelines for style, including usage of pep8 are maintained in our pull request guidelines.

We are currently enforcing a sub-set of pep8 on new code contributions.

## Branches and Pull requests

None so far.

## Abstract

The matplotlib codebase predates PEP8, and therefore is less than consistent style-wise in some areas. Bringing the codebase into compliance with PEP8 would go a long way to improving its legibility.

## Detailed description

Some files use four space indentation, some use three. Some use different levels in the same file.

For the most part, class/function/variable naming follows PEP8, but it wouldn't hurt to fix where necessary.

## Implementation

The implementation should be fairly mechanical: running the pep8 tool over the code and fixing where appropriate.

This should be merged in after the 2.0 release, since the changes will likely make merging any pending pull requests more difficult.

Additionally, and optionally, PEP8 compliance could be tracked by an automated build system.

### Backward compatibility

Public names of classes and functions that require change (there shouldn't be many of these) should first be deprecated and then removed in the next release cycle.

### Alternatives

PEP8 is a popular standard for Python code style, blessed by the Python core developers, making any alternatives less desirable.

## MEP9: Global interaction manager


- *Status*
- *Branches and Pull requests*
- *Abstract*
- *Detailed description*
- *Implementation*
- *Current summary of the mixin*
- *Backward compatibility*
- *Alternatives*


Add a global manager for all user interactivity with artists; make any artist resizeable, moveable, highlightable, and selectable as desired by the user.

### Status

**Discussion**

### Branches and Pull requests

https://github.com/dhyams/matplotlib/tree/MEP9

### Abstract

The goal is to be able to interact with matplotlib artists in a very similar way as drawing programs do. When appropriate, the user should be able to move, resize, or select an artist that is already on the canvas. Of course, the script writer is ultimately in control of whether an artist is able to be interacted with, or whether it is static.

This code to do this has already been privately implemented and tested, and would need to be migrated from its current "mixin" implementation, to a bona-fide part of matplotlib.

The end result would be to have four new keywords available to matplotlib.artist.Artist: _moveable_, _resizeable_, _selectable_, and _highlightable_. Setting any one of these keywords to True would activate interactivity for that artist.

In effect, this MEP is a logical extension of event handling in matplotlib; matplotlib already supports "low level" interactions like left mouse presses, a key press, or similar. The MEP extends the support to the logical level, where callbacks are performed on the artists when certain interactive gestures from the user are detected.

### Detailed description

This new functionality would be used to allow the end-user to better interact with the graph. Many times, a graph is almost what the user wants, but a small repositioning and/or resizing of components is necessary. Rather than force the user to go back to the script to trial-and-error the location, and simple drag and drop would be appropriate.

Also, this would better support applications that use matplotlib; here, the end-user has no reasonable access or desire to edit the underlying source in order to fine-tune a plot. Here, if matplotlib offered the capability, one could move or resize artists on the canvas to suit their needs. Also, the user should be able to highlight (with a mouse over) an artist, and select it with a double-click, if the application supports that sort of thing. In this MEP, we also want to support the highlighting and selection natively; it is up to application to handle what happens when the artist is selected. A typical handling would be to display a dialog to edit the properties of the artist.

In the future, as well (this is not part of this MEP), matplotlib could offer backend-specific property dialogs for each artist, which are raised on artist selection. This MEP would be a necessary stepping stone for that sort of capability.

There are currently a few interactive capabilities in matplotlib (e.g. legend.draggable()), but they tend to be scattered and are not available for all artists. This MEP seeks to unify the interactive interface and make it work for all artists.

The current MEP also includes grab handles for resizing artists, and appropriate boxes drawn when artists are moved or resized.

### Implementation

- Add appropriate methods to the "tree" of artists so that the interactivity manager has a consistent interface for the interactivity manager to deal with. The proposed methods to add to the artists, if they are to support interactivity, are:
  - get_pixel_position_ll(self): get the pixel position of the lower left corner of the artist's bounding box
  - get_pixel_size(self): get the size of the artist's bounding box, in pixels
  - set_pixel_position_and_size(self,x,y,dx,dy): set the new size of the artist, such that it fits within the specified bounding box.
- add capability to the backends to 1) provide cursors, since these are needed for visual indication of moving/resizing, and 2) provide a function that gets the current mouse position
- Implement the manager. This has already been done privately (by dhyams) as a mixin, and has been tested quite a bit. The goal would be to move the functionality of the manager into the artists so that it is in matplotlib properly, and not as a "monkey patch" as I currently have it coded.

### Current summary of the mixin

(Note that this mixin is for now just private code, but can be added to a branch obviously)

InteractiveArtistMixin:

Mixin class to make any generic object that is drawn on a matplotlib canvas moveable and possibly resizeable. The Powerpoint model is followed as closely as possible; not because I'm enamoured with Powerpoint, but because that's what most people understand. An artist can also be selectable, which means that the artist will receive the on_activated() callback when double clicked. Finally, an artist can be highlightable, which means that a highlight is drawn on the artist whenever the mouse passes over. Typically, highlightable artists will also be selectable, but that is left up to the user. So, basically there are four attributes that can be set by the user on a per-artist basis:

- highlightable
- selectable
- moveable
- resizeable

To be moveable (draggable) or resizeable, the object that is the target of the mixin must support the following protocols:

- get_pixel_position_ll(self)
- get_pixel_size(self)
- set_pixel_position_and_size(self,x,y,sx,sy)

Note that nonresizeable objects are free to ignore the sx and sy parameters. To be highlightable, the object that is the target of the mixin must also support the following protocol:

- get_highlight(self)

Which returns a list of artists that will be used to draw the highlight.

If the object that is the target of the mixin is not an matplotlib artist, the following protocols must also be implemented. Doing so is usually fairly trivial, as there has to be an artist *somewhere* that is being drawn. Typically your object would just route these calls to that artist.

- get_figure(self)
- get_axes(self)
- contains(self,event)
- set_animated(self,flag)
- draw(self,renderer)
- get_visible(self)

The following notifications are called on the artist, and the artist can optionally implement these.

- on_select_begin(self)
- on_select_end(self)
- on_drag_begin(self)
- on_drag_end(self)
- on_activated(self)
- on_highlight(self)
- on_right_click(self,event)
- on_left_click(self,event)
- on_middle_click(self,event)
- on_context_click(self,event)
- on_key_up(self,event)
- on_key_down(self,event)

The following notifications are called on the canvas, if no interactive artist handles the event:

- on_press(self,event)
- on_left_click(self,event)
- on_middle_click(self,event)
- on_right_click(self,event)
- on_context_click(self,event)
- on_key_up(self,event)
- on_key_down(self,event)

The following functions, if present, can be used to modify the behavior of the interactive object:

- press_filter(self,event) # determines if the object wants to have the press event routed to it
- handle_unpicked_cursor() # can be used by the object to set a cursor as the cursor passes over the object when it is unpicked.

Supports multiple canvases, maintaining a drag lock, motion notifier, and a global "enabled" flag per canvas. Supports fixed aspect ratio resizings by holding the shift key during the resize.

Known problems:

- Zorder is not obeyed during the selection/drag operations. Because of the blit technique used, I do not believe this can be fixed. The only way I can think of is to search for all artists that have a zorder greater then me, set them all to animated, and then redraw them all on top during each drag refresh. This might be very slow; need to try.
- the mixin only works for wx backends because of two things: 1) the cursors are hardcoded, and 2) there is a call to wx.GetMousePosition() Both of these shortcomings are reasonably fixed by having each backend supply these things.

### Backward compatibility

No problems with backward compatibility, although once this is in place, it would be appropriate to obsolete some of the existing interactive functions (like legend.draggable())

### Alternatives

None that I know of.

### MEP10: Docstring consistency


- *Status*
- *Branches and Pull requests*
- *Abstract*
- *Detailed description*
  - *Numpy docstring format*
  - *Cross references*
  - *Overriding signatures*
  - *Linking rather than duplicating*
  - *autosummary extension*
  - *Examples linking to relevant documentation*
  - *Documentation using help() vs. a browser*

- *Implementation*
- *Backward compatibility*
- *Alternatives*


## Status

### Progress

This is still an on-going effort

## Branches and Pull requests

## Abstract

matplotlib has a great deal of inconsistency between docstrings. This not only makes the docs harder to read, but it is harder on contributors, because they don't know which specifications to follow. There should be a clear docstring convention that is followed consistently.

The organization of the API documentation is difficult to follow. Some pages, such as pyplot and axes, are enormous and hard to browse. There should instead be short summary tables that link to detailed documentation. In addition, some of the docstrings themselves are quite long and contain redundant information.

Building the documentation takes a long time and uses a `make.py` script rather than a Makefile.

## Detailed description

There are number of new tools and conventions available since matplotlib started using Sphinx that make life easier. The following is a list of proposed changes to docstrings, most of which involve these new features.

## Numpy docstring format

Numpy docstring format: This format divides the docstring into clear sections, each having different parsing rules that make the docstring easy to read both as raw text and as HTML. We could consider alternatives, or invent our own, but this is a strong choice, as it's well used and understood in the Numpy/Scipy community.

### Cross references

Most of the docstrings in matplotlib use explicit "roles" when linking to other items, for example: `:func:`myfunction``. As of Sphinx 0.4, there is a "default_role" that can be set to "obj", which will polymorphically link to a Python object of any type. This allows one to write `` `myfunction` `` instead. This makes docstrings much easier to read and edit as raw text. Additionally, Sphinx allows for setting a current module, so links like `` `~matplotlib.axes.Axes.set_xlim` `` could be written as `` `~axes.Axes.set_xlim` ``.

### Overriding signatures

Many methods in matplotlib use the `*args` and `**kwargs` syntax to dynamically handle the keyword arguments that are accepted by the function, or to delegate on to another function. This, however, is often not useful as a signature in the documentation. For this reason, many matplotlib methods include something like:

```
def annotate(self, *args, **kwargs):
    """
    Create an annotation: a piece of text referring to a data
    point.

    Call signature::

      annotate(s, xy, xytext=None, xycoords='data',
               textcoords='data', arrowprops=None, **kwargs)
    """
```

This can't be parsed by Sphinx, and is rather verbose in raw text. As of Sphinx 1.1, if the `autodoc_docstring_signature` config value is set to True, Sphinx will extract a replacement signature from the first line of the docstring, allowing this:

```
def annotate(self, *args, **kwargs):
    """
    annotate(s, xy, xytext=None, xycoords='data',
               textcoords='data', arrowprops=None, **kwargs)

    Create an annotation: a piece of text referring to a data
    point.
    """
```

The explicit signature will replace the actual Python one in the generated documentation.

### Linking rather than duplicating

Many of the docstrings include long lists of accepted keywords by interpolating things into the docstring at load time. This makes the docstrings very long. Also, since these tables are the same across many docstrings, it inserts a lot of redundant information in the docs -- particularly a problem in the printed version.

These tables should be moved to docstrings on functions whose only purpose is for help. The docstrings that refer to these tables should link to them, rather than including them verbatim.

### autosummary extension

The Sphinx autosummary extension should be used to generate summary tables, that link to separate pages of documentation. Some classes that have many methods (e.g. *Axes*) should be documented with one method per page, whereas smaller classes should have all of their methods together.

### Examples linking to relevant documentation

The examples, while helpful at illustrating how to use a feature, do not link back to the relevant docstrings. This could be addressed by adding module-level docstrings to the examples, and then including that docstring in the parsed content on the example page. These docstrings could easily include references to any other part of the documentation.

### Documentation using help() vs. a browser

Using Sphinx markup in the source allows for good-looking docs in your browser, but the markup also makes the raw text returned using help() look terrible. One of the aims of improving the docstrings should be to make both methods of accessing the docs look good.

### Implementation

1. The numpydoc extensions should be turned on for matplotlib. There is an important question as to whether these should be included in the matplotlib source tree, or used as a dependency. Installing Numpy is not sufficient to get the numpydoc extensions -- it's a separate install procedure. In any case, to the extent that they require customization for our needs, we should endeavor to submit those changes upstream and not fork them.
2. Manually go through all of the docstrings and update them to the new format and conventions. Updating the cross references (from `` :func:`myfunc` `` to `` `func` ``) may be able to be semi-automated. This is a lot of busy work, and perhaps this labor should be divided on a per-module basis so no single developer is over-burdened by it.
3. Reorganize the API docs using autosummary and `sphinx-autogen`. This should hopefully have minimal impact on the narrative documentation.

4. Modify the example page generator (`gen_rst.py`) so that it extracts the module docstring from the example and includes it in a non-literal part of the example page.
5. Use `sphinx-quickstart` to generate a new-style Sphinx Makefile. The following features in the current `make.py` will have to be addressed in some other way:
   - Copying of some static content
   - Specifying a "small" build (only low-resolution PNG files for examples)

Steps 1, 2, and 3 are interdependent. 4 and 5 may be done independently, though 5 has some dependency on 3.

### Backward compatibility

As this mainly involves docstrings, there should be minimal impact on backward compatibility.

### Alternatives

None yet discussed.

### MEP11: Third-party dependencies


- *Status*
- *Branches and Pull requests*
- *Abstract*
- *Detailed description*
  - *Current behavior*
  - *Desired behavior*
- *Implementation*
- *Backward compatibility*
- *Alternatives*


This MEP attempts to improve the way in which third-party dependencies in matplotlib are handled.

### Status

**Completed** -- needs to be merged

### Branches and Pull requests

#1157: Use automatic dependency resolution

#1290: Debundle pyparsing

#1261: Update six to 1.2

### Abstract

One of the goals of matplotlib has been to keep it as easy to install as possible. To that end, some third-party dependencies are included in the source tree and, under certain circumstances, installed alongside matplotlib. This MEP aims to resolve some problems with that approach, bring some consistency, while continuing to make installation convenient.

At the time that was initially done, setuptools, easy_install and PyPI were not mature enough to be relied on. However, at present, we should be able to safely leverage the "modern" versions of those tools, distribute and pip.

While matplotlib has dependencies on both Python libraries and C/C++ libraries, this MEP addresses only the Python libraries so as to not confuse the issue. C libraries represent a larger and mostly orthogonal set of problems.

### Detailed description

matplotlib depends on the following third-party Python libraries:

- Numpy
- dateutil (pure Python)
- pytz (pure Python)
- six -- required by dateutil (pure Python)
- pyparsing (pure Python)
- PIL (optional)
- GUI frameworks: pygtk, gobject, tkinter, PySide, PyQt4, wx (all optional, but one is required for an interactive GUI)

### Current behavior

When installing from source, a **`git`** checkout or pip:

- `setup.py` attempts to `import numpy`. If this fails, the installation fails.
- For each of dateutil, pytz and six, `setup.py` attempts to import them (from the top-level namespace). If that fails, matplotlib installs its local copy of the library into the top-level namespace.
- pyparsing is always installed inside of the matplotlib namespace.

This behavior is most surprising when used with pip, because no pip dependency resolution is performed, even though it is likely to work for all of these packages.

The fact that pyparsing is installed in the matplotlib namespace has reportedly (#1290) confused some users into thinking it is a matplotlib-related module and import it from there rather than the top-level.

When installing using the Windows installer, dateutil, pytz and six are installed at the top-level *always*, potentially overwriting already installed copies of those libraries.

TODO: Describe behavior with the OS-X installer.

When installing using a package manager (Debian, RedHat, MacPorts etc.), this behavior actually does the right thing, and there are no special patches in the matplotlib packages to deal with the fact that we handle dateutil, pytz and six in this way. However, care should be taken that whatever approach we move to continues to work in that context.

Maintaining these packages in the matplotlib tree and making sure they are up-to-date is a maintenance burden. Advanced new features that may require a third-party pure Python library have a higher barrier to inclusion because of this burden.

### Desired behavior

Third-party dependencies are downloaded and installed from their canonical locations by leveraging pip, distribute and PyPI.

dateutil, pytz, and pyparsing should be made into optional dependencies -- though obviously some features would fail if they aren't installed. This will allow the user to decide whether they want to bother installing a particular feature.

### Implementation

For installing from source, and assuming the user has all of the C-level compilers and dependencies, this can be accomplished fairly easily using distribute and following the instructions here. The only anticipated change to the matplotlib library code will be to import pyparsing from the top-level namespace rather than from within matplotlib. Note that distribute will also allow us to remove the direct dependency on six, since it is, strictly speaking, only a direct dependency of dateutil.

For binary installations, there are a number of alternatives (here ordered from best/hardest to worst/easiest):

1. The distutils wininst installer allows a post-install script to run. It might be possible to get this script to run pip to install the other dependencies. (See this thread for someone who has trod that ground before).
2. Continue to ship dateutil, pytz, six and pyparsing in our installer, but use the post-install-script to install them *only* if they cannot already be found.
3. Move all of these packages inside a (new) `matplotlib.extern` namespace so it is clear for outside users that these are external packages. Add some conditional imports in the core matplotlib codebase so dateutil (at the top-level) is tried first, and failing that `matplotlib.extern.dateutil` is used.

2 and 3 are undesirable as they still require maintaining copies of these packages in our tree -- and this is exacerbated by the fact that they are used less -- only in the binary installers. None of these 3 approaches address Numpy, which will still have to be manually installed using an installer.

TODO: How does this relate to the Mac OS-X installer?

### Backward compatibility

At present, matplotlib can be installed from source on a machine without the third party dependencies and without an internet connection. After this change, an internet connection (and a working PyPI) will be required to install matplotlib for the first time. (Subsequent matplotlib updates or development work will run without accessing the network).

### Alternatives

Distributing binary eggs doesn't feel like a usable solution. That requires getting easy_install installed first, and Windows users generally prefer the well known `.exe` or `.msi` installer that works out of the box.

## MEP12: Improve Gallery and Examples


- *Status*
- *Branches and Pull requests*
- *Abstract*
- *Detailed description*
- *Implementation*
  - *Gallery sections*
  - *Clean up guidelines*
    - *Additional suggestions*
- *Backward compatibility*

- *Alternatives*
  - *Tags*


## Status

### Progress

Initial changes added in 1.3. Conversion of the gallery is on-going. 29 September 2015 - The last `pylab_examples` where `pylab` is imported has been converted over to use *`matplotlib.pyplot`* and `numpy`.

## Branches and Pull requests

#1623, #1924, #2181

PR #2474 demonstrates a single example being cleaned up and moved to the appropriate section.

## Abstract

Reorganizing the matplotlib plot gallery would greatly simplify navigation of the gallery. In addition, examples should be cleaned-up and simplified for clarity.

## Detailed description

The matplotlib gallery was recently set up to split examples up into sections. As discussed in that PR[1], the current example sections (`api`, `pylab_examples`) aren't terribly useful to users: New sections in the gallery would help users find relevant examples.

These sections would also guide a cleanup of the examples: Initially, all the current examples would remain and be listed under their current directories. Over time, these examples could be cleaned up and moved into one of the new sections.

This process allows users to easily identify examples that need to be cleaned up; i.e. anything in the `api` and `pylab_examples` directories.

[1] https://github.com/matplotlib/matplotlib/pull/714

### Implementation

1. Create new gallery sections. [Done]
2. Clean up examples and move them to the new gallery sections (over the course of many PRs and with the help of many users/developers). [In progress]

### Gallery sections

The naming of sections is critical and will guide the clean-up effort. The current sections are:

- Lines, bars, and markers (more-or-less 1D data)
- Shapes and collections
- Statistical plots
- Images, contours, and fields
- Pie and polar charts: Round things
- Color
- Text, labels, and annotations
- Ticks and spines
- Subplots, axes, and figures
- Specialty plots (e.g., sankey, radar, tornado)
- Showcase (plots with tweaks to make them publication-quality)
- separate sections for toolboxes (already exists: 'mplot3d', 'axes_grid', 'units', 'widgets')

These names are certainly up for debate. As these sections grow, we should reevaluate them and split them up as necessary.

### Clean up guidelines

The current examples in the `api` and `pylab_examples` sections of the gallery would remain in those directories until they are cleaned up. After clean-up, they would be moved to one of the new gallery sections described above. "Clean-up" should involve:

- sphinx-gallery docstrings: a title and a description of the example formatted as follows, at the top of the example:

```
"""
==============================
Colormaps alter your perception
==============================

Here I plot the function
```

(continues on next page)

(continued from previous page)

```
.. math:: f(x, y) = \sin(x) + \cos(y)

with different colormaps. Look at how colormaps alter your perception!
"""
```

- PEP8 clean-ups (running flake8, or a similar checker, is highly recommended)
- Commented-out code should be removed.
- Replace uses of `pylab` interface with `pyplot` (+ `numpy`, etc.). See c25ef1e
- Remove shebang line, e.g.:

```
#!/usr/bin/env python
```

- Use consistent imports. In particular:

```
import numpy as np

import matplotlib.pyplot as plt
```

  Avoid importing specific functions from these modules (e.g. `from numpy import sin`)

- Each example should focus on a specific feature (excluding `showcase` examples, which will show more "polished" plots). Tweaking unrelated to that feature should be removed. See f7b2217, e57b5fc, and 1458aa8

Use of `pylab` should be demonstrated/discussed on a dedicated help page instead of the gallery examples.

**Note:** When moving an existing example, you should search for references to that example. For example, the API documentation for `axes.py` and `pyplot.py` may use these examples to generate plots. Use your favorite search tool (e.g., grep, ack, grin, pss) to search the matplotlib package. See 2dc9a46 and aa6b410

### Additional suggestions

- Provide links (both ways) between examples and API docs for the methods/objects used. (issue #2222)
- Use `plt.subplots` (note trailing "s") in preference over `plt.subplot`.
- Rename the example to clarify it's purpose. For example, the most basic demo of `imshow` might be `imshow_demo.py`, and one demonstrating different interpolation settings would be `imshow_demo_interpolation.py` (*not* `imshow_demo2.py`).
- Split up examples that try to do too much. See 5099675 and fc2ab07
- Delete examples that don't show anything new.
- Some examples exercise esoteric features for unit testing. These tweaks should be moved out of the gallery to an example in the `unit` directory located in the root directory of the package.
- Add plot titles to clarify intent of the example. See bd2b13c

### Backward compatibility

The website for each Matplotlib version is readily accessible, so users who want to refer to old examples can still do so.

### Alternatives

### Tags

Tagging examples will also help users search the example gallery. Although tags would be a big win for users with specific goals, the plot gallery will remain the entry point to these examples, and sections could really help users navigate the gallery. Thus, tags are complementary to this reorganization.

## MEP13: Use properties for Artists


- *Status*
- *Branches and Pull requests*
- *Abstract*
- *Detailed description*
- *Implementation*
- *Backward compatibility*
- *Examples*
  - *axes.Axes.set_axis_off/set_axis_on*
  - *axes.Axes.get_xlim/set_xlim and get_autoscalex_on/set_autoscalex_on*
  - *axes.Axes.get_title/set_title*
  - *axes.Axes.get_xticklabels/set_xticklabels*
- *Alternatives*


### Status

- **Discussion**

### Branches and Pull requests

None

### Abstract

Wrap all of the matplotlib getter and setter methods with python properties, allowing them to be read and written like class attributes.

### Detailed description

Currently matplotlib uses getter and setter functions (usually prefixed with get_ and set_, respectively) for reading and writing data related to classes. However, since 2.6 python supports properties, which allow such setter and getter functions to be accessed as though they were attributes. This proposal would implement all existing setter and getter methods as properties.

### Implementation

1. All existing getter and setter methods will need to have two aliases, one with the get_ or set_ prefix and one without. Getter methods that currently lack prefixes should be recording in a text file.
2. Classes should be reorganized so setter and getter methods are sequential in the code, with getter methods first.
3. Getter and setter methods that provide additional optional arguments should have those arguments accessible in another manner, either as additional getter or setter methods or attributes of other classes. If those classes are not accessible, getters for them should be added.
4. Property decorators will be added to the setter and getter methods without the prefix. Those with the prefix will be marked as deprecated.
5. Docstrings will need to be rewritten so the getter with the prefix has the current docstring and the getter without the prefix has a generic docstring appropriate for an attribute.
6. Automatic alias generation will need to be modified so it will also create aliases for the properties.
7. All instances of getter and setter method calls will need to be changed to attribute access.
8. All setter and getter aliases with prefixes will be removed

The following steps can be done simultaneously: 1, 2, and 3; 4 and 5; 6 and 7.

Only the following steps must be done in the same release: 4, 5, and 6. All other changes can be done in separate releases. 8 should be done several macro releases after everything else.

### Backward compatibility

All existing getter methods that do not have a prefix (such as get_) will need to be changed from function calls to attribute access. In most cases this will only require removing the parenthesis.

setter and getter methods that have additional optional arguments will need to have those arguments implemented in another way, either as a separate property in the same class or as attributes or properties of another class.

Cases where the setter returns a value will need to be changed to using the setter followed by the getter.

Cases where there are set_ATTR_on() and set_ATTR_off() methods will be changed to ATTR_on properties.

### Examples

#### axes.Axes.set_axis_off/set_axis_on

Current implementation:

```
axes.Axes.set_axis_off()
axes.Axes.set_axis_on()
```

New implementation:

```
True = axes.Axes.axis_on
False = axes.Axes.axis_on
axes.Axes.axis_on = True
axes.Axes.axis_on = False
```

#### axes.Axes.get_xlim/set_xlim and get_autoscalex_on/set_autoscalex_on

Current implementation:

```
[left, right] = axes.Axes.get_xlim()
auto = axes.Axes.get_autoscalex_on()

[left, right] = axes.Axes.set_xlim(left=left, right=right, emit=emit,␣
 ↪auto=auto)
[left, right] = axes.Axes.set_xlim(left=left, right=None, emit=emit,␣
 ↪auto=auto)
[left, right] = axes.Axes.set_xlim(left=None, right=right, emit=emit,␣
 ↪auto=auto)
[left, right] = axes.Axes.set_xlim(left=left, emit=emit, auto=auto)
[left, right] = axes.Axes.set_xlim(right=right, emit=emit, auto=auto)

axes.Axes.set_autoscalex_on(auto)
```

New implementation:

```
[left, right] = axes.Axes.axes_xlim
auto = axes.Axes.autoscalex_on

axes.Axes.axes_xlim = [left, right]
axes.Axes.axes_xlim = [left, None]
axes.Axes.axes_xlim = [None, right]
axes.Axes.axes_xlim[0] = left
axes.Axes.axes_xlim[1] = right

axes.Axes.autoscalex_on = auto

axes.Axes.emit_xlim = emit
```

### axes.Axes.get_title/set_title

Current implementation:

```
string = axes.Axes.get_title()
axes.Axes.set_title(string, fontdict=fontdict, **kwargs)
```

New implementation:

```
string = axes.Axes.title
string = axes.Axes.title_text.text

text.Text = axes.Axes.title_text
text.Text.<attribute> = attribute
text.Text.fontdict = fontdict

axes.Axes.title = string
axes.Axes.title = text.Text
axes.Axes.title_text = string
axes.Axes.title_text = text.Text
```

### axes.Axes.get_xticklabels/set_xticklabels

Current implementation:

```
[text.Text] = axes.Axes.get_xticklabels()
[text.Text] = axes.Axes.get_xticklabels(minor=False)
[text.Text] = axes.Axes.get_xticklabels(minor=True)
[text.Text] = axes.Axes.([string], fontdict=None, **kwargs)
[text.Text] = axes.Axes.([string], fontdict=None, minor=False, **kwargs)
[text.Text] = axes.Axes.([string], fontdict=None, minor=True, **kwargs)
```

New implementation:

```
[text.Text] = axes.Axes.xticklabels
[text.Text] = axes.Axes.xminorticklabels
axes.Axes.xticklabels = [string]
axes.Axes.xminorticklabels = [string]
axes.Axes.xticklabels = [text.Text]
axes.Axes.xminorticklabels = [text.Text]
```

### Alternatives

Instead of using decorators, it is also possible to use the property function. This would change the procedure so that all getter methods that lack a prefix will need to be renamed or removed. This makes handling docstrings more difficult and harder to read.

It is not necessary to deprecate the setter and getter methods, but leaving them in will complicate the code.

This could also serve as an opportunity to rewrite or even remove automatic alias generation.

Another alternate proposal:

Convert `set_xlim`, `set_xlabel`, `set_title`, etc. to `xlim`, `xlabel`, `title`,... to make the transition from `plt` functions to `axes` methods significantly simpler. These would still be methods, not properties, but it's still a great usability enhancement while retaining the interface.

## MEP14: Text handling


- *Status*
- *Branches and Pull requests*
- *Abstract*
- *Detailed description*
- *Implementation*
- *Backward compatibility*
- *Alternatives*


### Status

- **Discussion**

### Branches and Pull requests

Issue #253 demonstrates a bug where using the bounding box rather than the advance width of text results in misaligned text. This is a minor point in the grand scheme of things, but it should be addressed as part of this MEP.

### Abstract

By reorganizing how text is handled, this MEP aims to:

- improve support for Unicode and non-ltr languages
- improve text layout (especially multi-line text)
- allow support for more fonts, especially non-Apple-format TrueType fonts and OpenType fonts.
- make the font configuration easier and more transparent

### Detailed description

#### Text layout

At present, matplotlib has two different ways to render text: "built-in" (based on FreeType and our own Python code), and "usetex" (based on calling out to a TeX installation). Adjunct to the "built-in" renderer there is also the Python-based "mathtext" system for rendering mathematical equations using a subset of the TeX language without having a TeX installation available. Support for these two engines in strewn about many source files, including every backend, where one finds clauses like

```
if rcParams['text.usetex']: # do one thing else: # do another
```

Adding a third text rendering approach (more on that later) would require editing all of these places as well, and therefore doesn't scale.

Instead, this MEP proposes adding a concept of "text engines", where the user could select one of many different approaches for rendering text. The implementations of each of these would be localized to their own set of modules, and not have little pieces around the whole source tree.

Why add more text rendering engines? The "built-in" text rendering has a number of shortcomings.

- It only handles right-to-left languages, and doesn't handle many special features of Unicode, such as combining diacriticals.
- The multiline support is imperfect and only supports manual line-breaking -- it cannot break up a paragraph into lines of a certain length.
- It also does not handle inline formatting changes in order to support something like Markdown, re-StructuredText or HTML. (Though rich-text formatting is contemplated in this MEP, since we want to make sure this design allows it, the specifics of a rich-text formatting implementation is outside of the scope of this MEP.)

Supporting these things is difficult, and is the "full-time job" of a number of other projects:

- pango/harfbuzz
- QtTextLayout
- Microsoft DirectWrite
- Apple Core Text

Of the above options, it should be noted that harfbuzz is designed from the start as a cross platform option with minimal dependencies, so therefore is a good candidate for a single option to support.

Additionally, for supporting rich text, we could consider using WebKit, and possibly whether than represents a good single cross-platform option. Again, however, rich text formatting is outside of the scope of this project.

Rather than trying to reinvent the wheel and add these features to matplotlib's "built-in" text renderer, we should provide a way to leverage these projects to get more powerful text layout. The "built-in" renderer will still need to exist for reasons of ease of installation, but its feature set will be more limited compared to the others. [TODO: This MEP should clearly decide what those limited features are, and fix any bugs to bring the implementation into a state of working correctly in all cases that we want it to work. I know @leejjoon has some thoughts on this.]

**Font selection**

Going from an abstract description of a font to a file on disk is the task of the font selection algorithm -- it turns out to be much more complicated than it seems at first.

The "built-in" and "usetex" renderers have very different ways of handling font selection, given their different technologies. TeX requires the installation of TeX-specific font packages, for example, and cannot use TrueType fonts directly. Unfortunately, despite the different semantics for font selection, the same set of font properties are used for each. This is true of both the *FontProperties* class and the font-related *rcParams* (which basically share the same code underneath). Instead, we should define a core set of font selection parameters that will work across all text engines, and have engine-specific configuration to allow the user to do engine-specific things when required. For example, it is possible to directly select a font by name in the "built-in" using *rcParams["font.family"]* (default: `['sans-serif']`), but the same is not possible with "usetex". It may be possible to make it easier to use TrueType fonts by using XeTeX, but users will still want to use the traditional metafonts through TeX font packages. So the issue still stands that different text engines will need engine-specific configuration, and it should be more obvious to the user which configuration will work across text engines and which are engine-specific.

Note that even excluding "usetex", there are different ways to find fonts. The default is to use the font list cache in *font_manager* which matches fonts using our own algorithm based on the CSS font matching algorithm. It doesn't always do the same thing as the native font selection algorithms on Linux (fontconfig), Mac and Windows, and it doesn't always find all of the fonts on the system that the OS would normally pick up. However, it is cross-platform, and always finds the fonts that ship with matplotlib. The Cairo and MacOSX backends (and presumably a future HTML5-based backend) currently bypass this mechanism and use the OS-native ones. The same is true when not embedding fonts in SVG, PS or PDF files and opening them in a third-party viewer. A downside there is that (at least with Cairo, need to confirm with MacOSX) they don't always find the fonts we ship with matplotlib. (It may be possible to add the fonts to their search path, though, or we may need to find a way to install our fonts to a location the OS expects to find them).

There are also special modes in the PS and PDF to only use the core fonts that are always available to those formats. There, the font lookup mechanism must only match against those fonts. It is unclear whether the

OS-native font lookup systems can handle this case.

There is also experimental support for using fontconfig for font selection in matplotlib, turned off by default. fontconfig is the native font selection algorithm on Linux, but is also cross platform and works well on the other platforms (though obviously is an additional dependency there).

Many of the text layout libraries proposed above (pango, QtTextLayout, DirectWrite and CoreText etc.) insist on using the font selection library from their own ecosystem.

All of the above seems to suggest that we should move away from our self-written font selection algorithm and use the native APIs where possible. That's what Cairo and MacOSX backends already want to use, and it will be a requirement of any complex text layout library. On Linux, we already have the bones of a fontconfig implementation (which could also be accessed through pango). On Windows and Mac we may need to write custom wrappers. The nice thing is that the API for font lookup is relatively small, and essentially consist of "given a dictionary of font properties, give me a matching font file".

**Font subsetting**

Font subsetting is currently handled using ttconv. ttconv was a standalone commandline utility for converting TrueType fonts to subsetted Type 3 fonts (among other features) written in 1995, which matplotlib (well, I) forked in order to make it work as a library. It only handles Apple-style TrueType fonts, not ones with the Microsoft (or other vendor) encodings. It doesn't handle OpenType fonts at all. This means that even though the STIX fonts come as .otf files, we have to convert them to .ttf files to ship them with matplotlib. The Linux packagers hate this -- they'd rather just depend on the upstream STIX fonts. ttconv has also been shown to have a few bugs that have been difficult to fix over time.

Instead, we should be able to use FreeType to get the font outlines and write our own code (probably in Python) to output subsetted fonts (Type 3 on PS and PDF and paths on SVG). Freetype, as a popular and well-maintained project, handles a wide variety of fonts in the wild. This would remove a lot of custom C code, and remove some code duplication between backends.

Note that subsetting fonts this way, while the easiest route, does lose the hinting in the font, so we will need to continue, as we do now, provide a way to embed the entire font in the file where possible.

Alternative font subsetting options include using the subsetting built-in to Cairo (not clear if it can be used without the rest of Cairo), or using fontforge (which is a heavy and not terribly cross-platform dependency).

**Freetype wrappers**

Our FreeType wrapper could really use a reworking. It defines its own image buffer class (when a Numpy array would be easier). While FreeType can handle a huge diversity of font files, there are limitations to our wrapper that make it much harder to support non-Apple-vendor TrueType files, and certain features of OpenType files. (See #2088 for a terrible result of this, just to support the fonts that ship with Windows 7 and 8). I think a fresh rewrite of this wrapper would go a long way.

**Text anchoring and alignment and rotation**

The handling of baselines was changed in 1.3.0 such that the backends are now given the location of the baseline of the text, not the bottom of the text. This is probably the correct behavior, and the MEP refactoring should also follow this convention.

In order to support alignment on multi-line text, it should be the responsibility of the (proposed) text engine to handle text alignment. For a given chunk of text, each engine calculates a bounding box for that text and

the offset of the anchor point within that box. Therefore, if the va of a block was "top", the anchor point would be at the top of the box.

Rotating of text should always be around the anchor point. I'm not sure that lines up with current behavior in matplotlib, but it seems like the sanest/least surprising choice. [This could be revisited once we have something working]. Rotation of text should not be handled by the text engine -- that should be handled by a layer between the text engine and the rendering backend so it can be handled in a uniform way. [I don't see any advantage to rotation being handled by the text engines individually...]

There are other problems with text alignment and anchoring that should be resolved as part of this work. [TODO: enumerate these].

**Other minor problems to fix**

The mathtext code has backend-specific code -- it should instead provide its output as just another text engine. However, it's still desirable to have mathtext layout inserted as part of a larger layout performed by another text engine, so it should be possible to do this. It's an open question whether embedding the text layout of an arbitrary text engine in another should be possible.

The text mode is currently set by a global rcParam ("text.usetex") so it's either all on or all off. We should continue to have a global rcParam to choose the text engine ("text.layout_engine"), but it should under the hood be an overridable property on the *Text* object, so the same figure can combine the results of multiple text layout engines if necessary.

### Implementation

A concept of a "text engine" will be introduced. Each text engine will implement a number of abstract classes. The `TextFont` interface will represent text for a given set of font properties. It isn't necessarily limited to a single font file -- if the layout engine supports rich text, it may handle a number of font files in a family. Given a `TextFont` instance, the user can get a `TextLayout` instance, which represents the layout for a given string of text in a given font. From a `TextLayout`, an iterator over `TextSpans` is returned so the engine can output raw editable text using as few spans as possible. If the engine would rather get individual characters, they can be obtained from the `TextSpan` instance:

```
class TextFont(TextFontBase):
    def __init__(self, font_properties):
        """
        Create a new object for rendering text using the given font␣
 ↪properties.
        """
        pass

    def get_layout(self, s, ha, va):
        """
        Get the TextLayout for the given string in the given font and
        the horizontal (left, center, right) and verticalalignment (top,
        center, baseline, bottom)
        """
        pass

class TextLayout(TextLayoutBase):
```

(continues on next page)

(continued from previous page)

```
    def get_metrics(self):
        """
        Return the bounding box of the layout, anchored at (0, 0).
        """
        pass

    def get_spans(self):
        """
        Returns an iterator over the spans of different in the layout.
        This is useful for backends that want to editable raw text as
        individual lines.  For rich text where the font may change,
        each span of different font type will have its own span.
        """
        pass

    def get_image(self):
        """
        Returns a rasterized image of the text.  Useful for raster backends,
        like Agg.

        In all likelihood, this will be overridden in the backend, as it can
        be created from get_layout(), but certain backends may want to
        override it if their library provides it (as freetype does).
        """
        pass

    def get_rectangles(self):
        """
        Returns an iterator over the filled black rectangles in the layout.
        Used by TeX and mathtext for drawing, for example, fraction lines.
        """
        pass

    def get_path(self):
        """
        Returns a single Path object of the entire laid out text.

        [Not strictly necessary, but might be useful for textpath
        functionality]
        """
        pass

class TextSpan(TextSpanBase):
    x, y      # Position of the span -- relative to the text layout as a whole
              # where (0, 0) is the anchor.  y is the baseline of the span.
    fontfile  # The font file to use for the span
    text      # The text content of the span

    def get_path(self):
        pass  # See TextLayout.get_path

    def get_chars(self):
```

(continues on next page)

(continued from previous page)

```
        """
        Returns an iterator over the characters in the span.
        """
        pass

class TextChar(TextCharBase):
    x, y      # Position of the character -- relative to the text layout as
              # a whole, where (0, 0) is the anchor.  y is in the baseline
              # of the character.
    codepoint # The unicode code point of the character -- only for␣
 ↪informational
              # purposes, since the mapping of codepoint to glyph_id may have␣
 ↪been
              # handled in a complex way by the layout engine.  This is an int
              # to avoid problems on narrow Unicode builds.
    glyph_id  # The index of the glyph within the font
    fontfile  # The font file to use for the char

    def get_path(self):
        """
        Get the path for the character.
        """
pass
```

Graphic backends that want to output subset of fonts would likely build up a file-global dictionary of characters where the keys are (fontname, glyph_id) and the values are the paths so that only one copy of the path for each character will be stored in the file.

Special casing: The "usetex" functionality currently is able to get Postscript directly from TeX to insert directly in a Postscript file, but for other backends, parses a DVI file and generates something more abstract. For a case like this, `TextLayout` would implement `get_spans` for most backends, but add `get_ps` for the Postscript backend, which would look for the presence of this method and use it if available, or fall back to `get_spans`. This kind of special casing may also be necessary, for example, when the graphics backend and text engine belong to the same ecosystem, e.g. Cairo and Pango, or MacOSX and CoreText.

There are three main pieces to the implementation:

1) Rewriting the freetype wrapper, and removing ttconv.

   a) Once (1) is done, as a proof of concept, we can move to the upstream STIX .otf fonts

   b) Add support for web fonts loaded from a remote URL. (Enabled by using freetype for font subsetting).

2) Refactoring the existing "builtin" and "usetex" code into separate text engines and to follow the API outlined above.

3) Implementing support for advanced text layout libraries.

(1) and (2) are fairly independent, though having (1) done first will allow (2) to be simpler. (3) is dependent on (1) and (2), but even if it doesn't get done (or is postponed), completing (1) and (2) will make it easier to move forward with improving the "builtin" text engine.

### Backward compatibility

The layout of text with respect to its anchor and rotation will change in hopefully small, but improved, ways. The layout of multiline text will be much better, as it will respect horizontal alignment. The layout of bidirectional text or other advanced Unicode features will now work inherently, which may break some things if users are currently using their own workarounds.

Fonts will be selected differently. Hacks that used to sort of work between the "builtin" and "usetex" text rendering engines may no longer work. Fonts found by the OS that weren't previously found by matplotlib may be selected.

### Alternatives

TBD

## MEP15: Fix axis autoscaling when limits are specified for one axis only


- *Status*
- *Branches and Pull requests*
- *Abstract*
- *Detailed description*
- *Implementation*
- *Backward compatibility*
- *Alternatives*


### Status

**Discussion**

### Branches and Pull requests

None so far.

### Abstract

When one Axis of a 2-dimensional plot is overridden via *set_xlim* or *set_ylim*, automatic scaling of the remaining Axis should be based on the data that falls within the specified limits of the first Axis.

### Detailed description

When axis limits for a 2-D plot are specified for one axis only (via *set_xlim* or *set_ylim*), matplotlib currently does not currently rescale the other axis. The result is that the displayed curves or symbols may be compressed into a tiny portion of the available area, so that the final plot conveys much less information than it would with appropriate axis scaling.

The proposed change of behavior would make matplotlib choose the scale for the remaining axis using only the data that falls within the limits for the axis where limits were specified.

### Implementation

I don't know enough about the internals of matplotlib to be able to suggest an implementation.

### Backward compatibility

From the standpoint of software interfaces, there would be no break in backward compatibility. Some outputs would be different, but if the user truly desires the previous behavior, he/she can achieve this by overriding the axis scaling for both axes.

### Alternatives

The only alternative that I can see is to maintain the status quo.

## MEP19: Continuous Integration

### Status

**Completed**

## Branches and Pull requests

## Abstract

matplotlib could benefit from better and more reliable continuous integration, both for testing and building installers and documentation.

## Detailed description

## Current state-of-the-art

### Testing

matplotlib currently uses Travis-CI for automated tests. While Travis-CI should be praised for how much it does as a free service, it has a number of shortcomings:

- It often fails due to network timeouts when installing dependencies.
- It often fails for inexplicable reasons.
- build or test products can only be saved from build off of branches on the main repo, not pull requests, so it is often difficult to "post mortem" analyse what went wrong. This is particularly frustrating when the failure cannot be subsequently reproduced locally.
- It is not extremely fast. matplotlib's cpu and memory requirements for testing are much higher than the average Python project.
- It only tests on Ubuntu Linux, and we have only minimal control over the specifics of the platform. It can be upgraded at any time outside of our control, causing unexpected delays at times that may not be convenient in our release schedule.

On the plus side, Travis-CI's integration with github -- automatically testing all pending pull requests -- is exceptional.

### Builds

There is no centralized effort for automated binary builds for matplotlib. However, the following disparate things are being done [If the authors mentioned here could fill in detail, that would be great!]:

- @sandrotosi: builds Debian packages
- @takluyver: Has automated Ubuntu builds on Launchpad
- @cgohlke: Makes Windows builds (don't know how automated that is)
- @r-owen: Makes OS-X builds (don't know how automated that is)

### Documentation

Documentation of main is now built by travis and uploaded to https://matplotlib.org/devdocs/index.html

@NelleV, I believe, generates the docs automatically and posts them on the web to chart MEP10 progress.

### Peculiarities of matplotlib

matplotlib has complex requirements that make testing and building more taxing than many other Python projects.

- The CPU time to run the tests is quite high. It puts us beyond the free accounts of many CI services (e.g. ShiningPanda)
- It has a large number of dependencies, and testing the full matrix of all combinations is impractical. We need to be clever about what space we test and guarantee to support.

### Requirements

This section outlines the requirements that we would like to have.

1. Testing all pull requests by hooking into the GitHub API, as Travis-CI does
2. Testing on all major platforms: Linux, Mac OS-X, MS Windows (in that order of priority, based on user survey)
3. Retain the last n days worth of build and test products, to aid in post-mortem debugging.
4. Automated nightly binary builds, so that users can test the bleeding edge without installing a complete compilation environment.
5. Automated benchmarking. It would be nice to have a standard benchmark suite (separate from the tests) whose performance could be tracked over time, in different backends and platforms. While this is separate from building and testing, ideally it would run on the same infrastructure.
6. Automated nightly building and publishing of documentation (or as part of testing, to ensure PRs don't introduce documentation bugs). (This would not replace the static documentation for stable releases as a default).
7. The test systems should be manageable by multiple developers, so that no single person becomes a bottleneck. (Travis-CI's design does this well -- storing build configuration in the git repository, rather than elsewhere, is a very good design.)
8. Make it easy to test a large but sparse matrix of different versions of matplotlib's dependencies. The matplotlib user survey provides some good data as to where to focus our efforts: https://docs.google.com/spreadsheets/d/1jbK0J4cIkyBNncnS-gP7pINSliNy9lI-N4JHwxlNSXE/edit
9. Nice to have: A decentralized design so that those with more obscure platforms can publish build results to a central dashboard.

### Implementation

This part is yet-to-be-written.

However, ideally, the implementation would be a third-party service, to avoid adding system administration to our already stretched time. As we have some donated funds, this service may be a paid one if it offers significant time-saving advantages over free offerings.

### Backward compatibility

Backward compatibility is not a major concern for this MEP. We will replace current tools and procedures with something better and throw out the old.

### Alternatives

### Hangout Notes

### CI Infrastructure

- We like Travis and it will probably remain part of our arsenal in any event. The reliability issues are being looked into.
- Enable Amazon S3 uploads of testing products on Travis. This will help with post-mortem of failures (@mdboom is looking into this now).
- We want Mac coverage. The best bet is probably to push Travis to enable it for our project by paying them for a Pro account (since they don't otherwise allow testing on both Linux and Mac).
- We want Windows coverage. Shining Panda is an option there.
- Investigate finding or building a tool that would collect and synthesize test results from a number of sources and post it to GitHub using the GitHub API. This may be of general use to the Scipy community.
- For both Windows and Mac, we should document (or better yet, script) the process of setting up the machine for a build, and how to build binaries and installers. This may require getting information from Russel Owen and Christoph Gohlke. This is a necessary step for doing automated builds, but would also be valuable for a number of other reasons.

### The test framework itself

- We should investigate ways to make it take less time
  - Eliminating redundant tests, if possible
  - General performance improvements to matplotlib will help
- We should be covering more things, particularly more backends
- We should have more unit tests, fewer integration tests, if possible

### MEP21: color and cm refactor


- *Status*
- *Branches and Pull requests*
- *Abstract*
- *Detailed description*
- *Implementation*
- *Backward compatibility*
- *Alternatives*


#### Status

- **Discussion**: This MEP has not commenced yet, but here are some ongoing ideas which may become a part of this MEP:

#### Branches and Pull requests

#### Abstract

- color
  - tidy up the namespace
  - Define a "Color" class
  - make it easy to convert from one color type to another `` `hex -> RGB` ``, `` `RGB -> hex` ``, `` `HSV -> RGB` `` etc.
  - improve the construction of a colormap - the dictionary approach is archaic and overly complex (though incredibly powerful)
  - make it possible to interpolate between two or more color types in different modes, especially useful for construction of colormaps in HSV space for instance
- cm
  - rename the module to something more descriptive - mappables?

Overall, there are a lot of improvements that can be made with matplotlib color handling - managing backwards compatibility will be difficult as there are some badly named variables/modules which really shouldn't exist - but a clear path and message for migration should be available, with a large amount of focus on this in the API changes documentation.

### Detailed description

### Implementation

### Backward compatibility

### Alternatives

### MEP22: Toolbar rewrite


- *Status*
- *Branches and Pull requests*
- *Abstract*
- *Detailed description*
- *Implementation*
  - *ToolBase(object)*
    - *Methods*
    - *Available Tools*
  - *ToolToggleBase(ToolBase)*
    - *Methods*
    - *Available Tools*
  - *NavigationBase*
    - *Methods (intended for the end user)*
  - *ToolbarBase*
    - *Methods (for backend implementation)*
- *Backward compatibility*

### Status

**Progress**

### Branches and Pull requests

Previous work:

- https://github.com/matplotlib/matplotlib/pull/1849
- https://github.com/matplotlib/matplotlib/pull/2557
- https://github.com/matplotlib/matplotlib/pull/2465

Pull Requests:

- Removing the NavigationToolbar classes https://github.com/matplotlib/matplotlib/pull/2740 **CLOSED**
- Keeping the NavigationToolbar classes https://github.com/matplotlib/matplotlib/pull/2759 **CLOSED**
- Navigation by events: https://github.com/matplotlib/matplotlib/pull/3652

### Abstract

The main goal of this MEP is to make it easier to modify (add, change, remove) the way the user interacts with the figures.

The user interaction with the figure is deeply integrated within the Canvas and Toolbar. Making extremely difficult to do any modification.

This MEP proposes the separation of this interaction into Toolbar, Navigation and Tools to provide independent access and reconfiguration.

This approach will make easier to create and share tools among users. In the far future, we can even foresee a kind of Marketplace for `Tools` where the most popular can be added into the main distribution.

### Detailed description

The reconfiguration of the Toolbar is complex, most of the time it requires a custom backend.

The creation of custom Tools sometimes interferes with the Toolbar, as example see https://github.com/matplotlib/matplotlib/issues/2694 also the shortcuts are hardcoded and again not easily modifiable https://github.com/matplotlib/matplotlib/issues/2699

The proposed solution is to take the actions out of the `Toolbar` and the shortcuts out of the `Canvas`. The actions and shortcuts will be in the form of `Tools`.

A new class `Navigation` will be the bridge between the events from the `Canvas` and `Toolbar` and redirect them to the appropriate `Tool`.

At the end the user interaction will be divided into three classes:

- NavigationBase: This class is instantiated for each FigureManager and connect the all user interactions with the Tools
- ToolbarBase: This existing class is relegated only as a GUI access to Tools.
- ToolBase: Is the basic definition of Tools.

### Implementation

### ToolBase(object)

Tools can have a graphical representation as the `SubplotTool` or not even be present in the Toolbar as `Quit`.

The *`ToolBase`* has the following class attributes for configuration at definition time

- keymap = None: Key(s) to be used to trigger the tool
- description = '': Small description of the tool
- image = None: Image that is used in the toolbar

The following instance attributes are set at instantiation:

- name
- navigation

### Methods

- `trigger(self, event)`: This is the main method of the Tool, it is called when the Tool is triggered by:
  - Toolbar button click
  - keypress associated with the Tool Keymap
  - Call to navigation.trigger_tool(name)
- `set_figure(self, figure)`: Set the figure and navigation attributes
- `destroy(self, *args)`: Destroy the `Tool` graphical interface (if exists)

### Available Tools

- ToolQuit
- ToolEnableAllNavigation
- ToolEnableNavigation
- ToolToggleGrid
- ToolToggleFullScreen
- ToolToggleYScale
- ToolToggleXScale
- ToolHome
- ToolBack
- ToolForward
- SaveFigureBase
- ConfigureSubplotsBase

### ToolToggleBase(ToolBase)

The *ToolToggleBase* has the following class attributes for configuration at definition time

- radio_group = None: Attribute to group 'radio' like tools (mutually exclusive)
- cursor = None: Cursor to use when the tool is active

The **Toggleable** Tools, can capture keypress, mouse moves, and mouse button press

### Methods

- `enable(self, event)`: Called by *ToolToggleBase.trigger* method
- `disable(self, event)`: Called when the tool is untoggled
- `toggled`: **Property** True or False

### Available Tools

- ToolZoom
- ToolPan

### NavigationBase

Defines the following attributes:

- canvas:
- keypresslock: Lock to know if the `canvas` `key_press_event` is available and process it
- messagelock: Lock to know if the message is available to write

### Methods (intended for the end user)

- `nav_connect(self, s, func)`: Connect to navigation for events
- `nav_disconnect(self, cid)`: Disconnect from navigation event
- `message_event(self, message, sender=None)`: Emit a tool_message_event event
- `active_toggle(self)`: **Property** The currently toggled tools or None
- `get_tool_keymap(self, name)`: Return a list of keys that are associated with the tool
- `set_tool_keymap(self, name, ``*keys)``: Set the keys for the given tool
- `remove_tool(self, name)`: Removes tool from the navigation control.
- `add_tools(self, tools)`: Add multiple tools to `Navigation`
- `add_tool(self, name, tool, group=None, position=None)`: Add a tool to the `Navigation`
- `tool_trigger_event(self, name, sender=None, canvasevent=None, data=None)`: Trigger a tool and fire the event
- `tools`: **Property** A dict with available tools with corresponding keymaps, descriptions and objects
- `get_tool(self, name)`: Return the tool object

### ToolbarBase

### Methods (for backend implementation)

- `add_toolitem(self, name, group, position, image, description, toggle)`: Add a toolitem to the toolbar. This method is a callback from `tool_added_event` (emitted by navigation)
- `set_message(self, s)`: Display a message on toolbar or in status bar
- `toggle_toolitem(self, name)`: Toggle the toolitem without firing event.
- `remove_toolitem(self, name)`: Remove a toolitem from the `Toolbar`

### Backward compatibility

For backward compatibility added 'navigation' to the list of values supported by *rcParams["toolbar"]* (default: `'toolbar2'`), that is used for `Navigation` classes instantiation instead of the NavigationToolbar classes

With this parameter, it makes it transparent to anyone using the existing backends.

[@pelson comment: This also gives us an opportunity to avoid needing to implement all of this in the same PR - some backends can potentially exist without the new functionality for a short while (but it must be done at some point).]

### MEP23: Multiple Figures per GUI window


- *Status*
- *Branches and Pull requests*
- *Abstract*
- *Detailed description*
- *Implementation*
  - *FigureManagerBase*
  - *new_figure_manager*
  - *new_figure_manager_given_figure*
  - *NavigationBase*
- *Backward compatibility*
- *Alternatives*


### Status

**Discussion**

### Branches and Pull requests

**Previous work** - https://github.com/matplotlib/matplotlib/pull/2465 **To-delete**

### Abstract

Add the possibility to have multiple figures grouped under the same *FigureManager*

### Detailed description

Under the current structure, every canvas has its own window.

This is and may continue to be the desired method of operation for most use cases.

Sometimes when there are too many figures open at the same time, it is desirable to be able to group these under the same window. See PR #2194.

The proposed solution modifies *FigureManagerBase* to contain and manage more than one `Canvas`. The settings parameter *rcParams["backend.multifigure"]* control when the **MultiFigure** behaviour is desired.

**Note**

It is important to note, that the proposed solution, assumes that the MEP22. is already in place. This is simply because the actual implementation of the `Toolbar` makes it pretty hard to switch between canvases.

### Implementation

The first implementation will be done in GTK3 using a Notebook as canvas container.

#### `FigureManagerBase`

will add the following new methods

- `add_canvas`: To add a canvas to an existing *FigureManager* object
- `remove_canvas`: To remove a canvas from a *FigureManager* object, if it is the last one, it will be destroyed
- `move_canvas`: To move a canvas from one *FigureManager* to another.
- `set_canvas_title`: To change the title associated with a specific canvas container
- `get_canvas_title`: To get the title associated with a specific canvas container
- `get_active_canvas`: To get the canvas that is in the foreground and is subject to the gui events. There is no `set_active_canvas` because the active canvas, is defined when `show` is called on a `Canvas` object.

#### new_figure_manager

To control which *FigureManager* will contain the new figures, an extra optional parameter *figuremanager* will be added, this parameter value will be passed to `new_figure_manager_given_figure`.

#### new_figure_manager_given_figure

- If *figuremanager* parameter is given, this *FigureManager* object will be used instead of creating a new one.
- If `rcParams['backend.multifigure']` is True: The last *FigureManager* object will be used instead of creating a new one.

#### NavigationBase

Modifies the `NavigationBase` to keep a list of canvases, directing the actions to the active one.

### Backward compatibility

For the **MultiFigure** properties to be visible, the user has to activate them directly setting `rcParams['backend.multifigure'] = True`

It should be backwards compatible for backends that adhere to the current *FigureManagerBase* structure even if they have not implemented the **MultiFigure** magic yet.

### Alternatives

Instead of modifying the *FigureManagerBase* it could be possible to add a parallel class, that handles the cases where `rcParams['backend.multifigure'] = True`. This will warranty that there won't be any problems with custom made backends, but also makes bigger the code, and more things to maintain.

## MEP24: Negative radius in polar plots


- *Status*
- *Branches and Pull requests*
- *Abstract*
- *Detailed description*
- *Implementation*
- *Related Issues*

- *Backward compatibility*
- *Alternatives*


### Status

*Discussion*

### Branches and Pull requests

None

### Abstract

It is clear that polar plots need to be able to gracefully handle negative r values (not by clipping or reflection).

### Detailed description

One obvious application that we should support is bB plots (see https://github.com/matplotlib/matplotlib/issues/1730#issuecomment-40815837), but this seems more generally useful (for example growth rate as a function of angle). The assumption in the current code (as I understand it) is that the center of the graph is `r==0`, however it would be good to be able to set the center to be at any `r` (with any value less than the offset clipped).

### Implementation

### Related Issues

#1730, #1603, #2203, #2133

### Backward compatibility

### Alternatives

### MEP25: Serialization


- *Status*
- *Branches and Pull requests*

- *Abstract*
- *Detailed description*
- *Examples*
- *Implementation*
- *Backward compatibility*
- *Alternatives*


## Status

**Rejected**

This work is important, but this particular effort has stalled.

## Branches and Pull requests

- development branches:
- related pull requests:

## Abstract

This MEP aims at adding a serializable `Controller` objects to act as an `Artist` managers. Users would then communicate changes to an `Artist` via a `Controller`. In this way, functionality of the `Controller` objects may be added incrementally since each `Artist` is still responsible for drawing everything. The goal is to create an API that is usable both by graphing libraries requiring high-level descriptions of figures and libraries requiring low-level interpretations.

## Detailed description

Matplotlib is a core plotting engine with an API that many users already understand. It's difficult/impossible for other graphing libraries to (1) get a complete figure description, (2) output raw data from the figure object as the user has provided it, (3) understand the semantics of the figure objects without heuristics, and (4) give matplotlib a complete figure description to visualize. In addition, because an `Artist` has no conception of its own semantics within the figure, it's difficult to interact with them in a natural way.

In this sense, matplotlib will adopt a standard Model-View-Controller (MVC) framework. The *Model* will be the user defined data, style, and semantics. The *Views* are the ensemble of each individual `Artist`, which are responsible for producing the final image based on the *model*. The *Controller* will be the `Controller` object managing its set of `Artist` objects.

The `Controller` must be able to export the information that it's carrying about the figure on command, perhaps via a `to_json` method or similar. Because it would be extremely extraneous to duplicate all of

the information in the model with the controller, only user-specified information (data + style) are explicitly kept. If a user wants more information (defaults) from the view/model, it should be able to query for it.

- This might be annoying to do, non-specified kwargs are pulled from the rcParams object which is in turn created from reading a user specified file and can be dynamically changed at run time. I suppose we could keep a dict of default defaults and compare against that. Not clear how this will interact with the style sheet [[MEP26]] - @tacaswell

Additional Notes:

- The "raw data" does not necessarily need to be a `list`, `ndarray`, etc. Rather, it can more abstractly just have a method to yield data when needed.
- Because the `Controller` will contain extra information that users may not want to keep around, it should *not* be created by default. You should be able to both (a) instantiate a `Controller` with a figure and (b) build a figure with a `Controller`.

Use Cases:

- Export all necessary informat
- Serializing a matplotlib figure, saving it, and being able to rerun later.
- Any other source sending an appropriately formatted representation to matplotlib to open

### Examples

Here are some examples of what the controllers should be able to do.

1. Instantiate a matplotlib figure from a serialized representation (e.g., JSON):

```
import json
from matplotlib.controllers import Controller
with open('my_figure') as f:
    o = json.load(f)
c = Controller(o)
fig = c.figure
```

2. Manage artists from the controller (e.g., Line2D):

```
# not really sure how this should look
c.axes[0].lines[0].color = 'b'
# ?
```

3. Export serializable figure representation:

```
o = c.to_json()
# or... we should be able to throw a figure object in there too
o = Controller.to_json(mpl_fig)
```

### Implementation

1. Create base `Controller` objects that are able to manage `Artist` objects (e.g., `Hist`)

   Comments:

   - initialization should happen via unpacking `**`, so we need a copy of call signature parameter for the `Artist` we're ultimately trying to control. Unfortunate hard-coded repetition...
   - should the additional `**kwargs` accepted by each `Artist` be tracked at the `Controller`
   - how does a `Controller` know which artist belongs where? E.g., do we need to pass `axes` references?

   Progress:

   - A simple NB demonstrating some functionality for `Line2DController` objects: https://nbviewer.jupyter.org/gist/theengineear/f0aa8d79f64325e767c0

2. Write in protocols for the `Controller` to *update* the model.

   Comments:

   - how should containers be dealt with? E.g., what happens to old patches when we re-bin a histogram?
   - in the link from (1), the old line is completely destroyed and redrawn, what if something is referencing it?

3. Create method by which a json object can be assembled from the `Controllers`
4. Deal with serializing the unserializable aspects of a figure (e.g., non-affine transforms?)
5. Be able to instantiate from a serialized representation
6. Reimplement the existing pyplot and Axes method, e.g. `pyplot.hist` and `Axes.hist` in terms of the new controller class.

> @theengineer: in #2 above, what do you mean by *get updates* from each `Artist`?

^ Yup. The `Controller` *shouldn't* need to get updated. This just happens in #3. Delete comments when you see this.

### Backward compatibility

- pickling will change
- non-affine transformations will require a defined pickling method

### Alternatives

PR #3150 suggested adding semantics by parasitically attaching extra containers to axes objects. This is a more complete solution with what should be a more developed/flexible/powerful framework.

### MEP26: Artist styling


- *Status*
- *Branches and Pull requests*
- *Abstract*
- *Detailed description*
- *Implementation*
  - *BNF Grammar*
  - *Syntax*
    - *Selectors*
    - *GID selector*
    - *Attributes and values*
  - *Parsing*
  - *Visitor pattern for matplotlib figure*
- *Backward compatibility*
- *Alternatives*
- *Appendix*
  - *Matplotlib primitives*

### Status

**Rejected**

### Branches and Pull requests

### Abstract

This MEP proposes a new stylesheet implementation to allow more comprehensive and dynamic styling of artists.

The current version of matplotlib (1.4.0) allows stylesheets based on the rcParams syntax to be applied before creation of a plot. The methodology below proposes a new syntax, based on CSS, which would allow styling of individual artists and properties, which can be applied dynamically to existing objects.

This is related to (and makes steps toward) the overall goal of moving to a DOM/tree-like architecture.

### Detailed description

Currently, the look and appearance of existing artist objects (figure, axes, Line2D, etc.) can only be updated via `set_` and `get_` methods on the artist object, which is quite laborious, especially if no reference to the artist(s) has been stored. The new style sheets introduced in 1.4 allow styling before a plot is created, but do not offer any means to dynamically update plots or distinguish between artists of the same type (i.e. to specify the `line color` and `line style` separately for differing *Line2D* objects).

The initial development should concentrate on allowing styling of artist primitives (those *Artist*s that do not contain other *Artist*s), and further development could expand the CSS syntax rules and parser to allow more complex styling. See the appendix for a list of primitives.

The new methodology would require development of a number of steps:

- A new stylesheet syntax (likely based on CSS) to allow selection of artists by type, class, id, etc.
- A mechanism by which to parse a stylesheet into a tree
- A mechanism by which to translate the parse-tree into something which can be used to update the properties of relevant artists. Ideally this would implement a method by which to traverse the artists in a tree-like structure.
- A mechanism by which to generate a stylesheet from existing artist properties. This would be useful to allow a user to export a stylesheet from an existing figure (where the appearance may have been set using the matplotlib API)...

### Implementation

It will be easiest to allow a '3rd party' to modify/set the style of an artist if the 'style' is created as a separate class and store against the artist as a property. The *GraphicsContextBase* class already provides a the basis of a `Style` class and an artist's *draw* method can be refactored to use the `Style` class rather than setting up its own *GraphicsContextBase* and transferring its style-related properties to it. A minimal example of how this could be implemented is shown here: https://github.com/JamesRamm/mpl_experiment

IMO, this will also make the API and code base much neater as individual get/set methods for artist style properties are now redundant... Indirectly related would be a general drive to replace get/set methods with properties. Implementing the style class with properties would be a big stride toward this...

For initial development, I suggest developing a syntax based on a much (much much) simplified version of CSS. I am in favour of dubbing this Artist Style Sheets :+1: :

### BNF Grammar

I propose a very simple syntax to implement initially (like a proof of concept), which can be expanded upon in the future. The BNF form of the syntax is given below and then explained

```
RuleSet ::= SelectorSequence "{"Declaration"}"

SelectorSequence :: = Selector {"," Selector}

Declaration ::= propName":" propValue";"

Selector ::= ArtistIdent{"#"Ident}

propName ::= Ident

propValue ::= Ident | Number | Colour | "None"
```

`ArtistIdent`, `Ident`, `Number` and `Colour` are tokens (the basic building blocks of the expression) which are defined by regular expressions.

### Syntax

A CSS stylesheet consists of a series of **rule sets** in hierarchical order (rules are applied from top to bottom). Each rule follows the syntax

```
selector {attribute: value;}
```

Each rule can have any number of `attribute: value` pairs, and a stylesheet can have any number of rules.

The initial syntax is designed only for *Artist* primitives. It does not address the question of how to set properties on *Container* types (whose properties may themselves be *Artist*s with settable properties), however, a future solution to this could simply be nested `RuleSets`

### Selectors

Selectors define the object to which the attribute updates should be applied. As a starting point, I propose just 2 selectors to use in initial development:

Artist Type Selector

Select an *Artist* by it's type. E.g *Line2D* or *Text*:

```
Line2D {attribute: value}
```

The regex for matching the artist type selector (ArtistIdent in the BNF grammar) would be:

```
ArtistIdent = r'(?P<ArtistIdent>\bLine2D\b|\bText\b|\bAxesImage\b|\
 ↪bFigureImage\b|\bPatch\b)'
```

### GID selector

Select an *Artist* by its gid:

```
Line2D#myGID {attribute: value}
```

A gid can be any string, so the regex could be as follows:

```
Ident = r'(?P<Ident>[a-zA-Z_][a-zA-Z_0-9]*)'
```

The above selectors roughly correspond to their CSS counterparts (http://www.w3.org/TR/CSS21/selector.html)

### Attributes and values

- Attributes are any valid (settable) property for the *Artist* in question.
- Values are any valid value for the property (Usually a string, or number).

### Parsing

Parsing would consist of breaking the stylesheet into tokens (the python cookbook gives a nice tokenizing recipe on page 66), applying the syntax rules and constructing a Tree. This requires defining the grammar of the stylesheet (again, we can borrow from CSS) and writing a parser. Happily, there is a recipe for this in the python cookbook as well.

### Visitor pattern for matplotlib figure

In order to apply the stylesheet rules to the relevant artists, we need to 'visit' each artist in a figure and apply the relevant rule. Here is a visitor class (again, thanks to python cookbook), where each `node` would be an artist in the figure. A `visit_` method would need to be implemented for each mpl artist, to handle the different properties for each

```
class Visitor:
    def visit(self, node):
        name = 'visit_' + type(node).__name__
        meth = getattr(self, name, None)
        if meth is None:
            raise NotImplementedError
        return meth(node)
```

An `evaluator` class would then take the stylesheet rules and implement the visitor on each one of them.

### Backward compatibility

Implementing a separate `Style` class would break backward compatibility as many get/set methods on an artist would become redundant. While it would be possible to alter these methods to hook into the `Style` class (stored as a property against the artist), I would be in favor of simply removing them to both neaten/simplify the codebase and to provide a simple, uncluttered API...

### Alternatives

No alternatives, but some of the ground covered here overlaps with MEP25, which may assist in this development

### Appendix

### Matplotlib primitives

This will form the initial selectors which stylesheets can use.

- Line2D
- Text
- AxesImage
- FigureImage
- Patch

### MEP27: Decouple pyplot from backends


- *Status*
- *Branches and Pull requests*
- *Abstract*
- *Detailed description*
- *Implementation*
- *Future compatibility*
- *Backward compatibility*
- *Alternatives*
- *Questions*


## Status

**Progress**

## Branches and Pull requests

Main PR (including GTK3):

- https://github.com/matplotlib/matplotlib/pull/4143

Backend specific branch diffs:

- https://github.com/OceanWolf/matplotlib/compare/backend-refactor...OceanWolf:backend-refactor-tkagg
- https://github.com/OceanWolf/matplotlib/compare/backend-refactor...OceanWolf:backend-refactor-qt
- https://github.com/OceanWolf/matplotlib/compare/backend-refactor...backend-refactor-wx

## Abstract

This MEP refactors the backends to give a more structured and consistent API, removing generic code and consolidate existing code. To do this we propose splitting:

1. `FigureManagerBase` and its derived classes into the core functionality class `FigureManager` and a backend specific class `WindowBase` and
2. `ShowBase` and its derived classes into `Gcf.show_all` and `MainLoopBase`.

### Detailed description

This MEP aims to consolidate the backends API into one single uniform API, removing generic code out of the backend (which includes `_pylab_helpers` and `Gcf`), and push code to a more appropriate level in matplotlib. With this we automatically remove inconsistencies that appear in the backends, such as `FigureManagerBase.resize(w, h)` which sometimes sets the canvas, and other times set the entire window to the dimensions given, depending on the backend.

Two main places for generic code appear in the classes derived from `FigureManagerBase` and `ShowBase`.

1. `FigureManagerBase` has **three** jobs at the moment:
    1. The documentation describes it as a *Helper class for pyplot mode, wraps everything up into a neat bundle*
    2. But it doesn't just wrap the canvas and toolbar, it also does all of the windowing tasks itself. The conflation of these two tasks gets seen the best in the following line: `self.set_window_title("Figure %d" % num)` This combines backend specific code `self.set_window_title(title)` with matplotlib generic code `title = "Figure %d" % num`.
    3. Currently the backend specific subclass of `FigureManager` decides when to end the mainloop. This also seems very wrong as the figure should have no control over the other figures.
2. `ShowBase` has two jobs:
    1. It has the job of going through all figure managers registered in `_pylab_helpers.Gcf` and telling them to show themselves.
    2. And secondly it has the job of performing the backend specific `mainloop` to block the main programme and thus keep the figures from dying.

### Implementation

The description of this MEP gives us most of the solution:

1. To remove the windowing aspect out of `FigureManagerBase` letting it simply wrap this new class along with the other backend classes. Create a new `WindowBase` class that can handle this functionality, with pass-through methods (:arrow_right:) to `WindowBase`. Classes that subclass `WindowBase` should also subclass the GUI specific window class to ensure backward compatibility (`manager.window == manager.window`).
2. Refactor the mainloop of `ShowBase` into `MainLoopBase`, which encapsulates the end of the loop as well. We give an instance of `MainLoop` to `FigureManager` as a key unlock the exit method (requiring all keys returned before the loop can die). Note this opens the possibility for multiple backends to run concurrently.
3. Now that `FigureManagerBase` has no backend specifics in it, to rename it to `FigureManager`, and move to a new file `backend_managers.py` noting that:

1. This allows us to break up the conversion of backends into separate PRs as we can keep the existing `FigureManagerBase` class and its dependencies intact.
2. And this also anticipates MEP22 where the new `NavigationBase` has morphed into a backend independent `ToolManager`.

| FigureManagerBase(canvas, num) | FigureManager(figure, num) | `WindowBase(title)` | Notes |
|---|---|---|---|
| show | | show | |
| destroy | calls destroy on all components | destroy | |
| full_screen_toggle | handles logic | set_fullscreen | |
| resize | | resize | |
| key_press | key_press | | |
| get_window_title | | get_window_title | |
| set_window_title | | set_window_title | |
| | _get_toolbar | | A common method to all subclasses of FigureManagerBase |
| | | set_default_size | |
| | | add_element_to_ | |

| ShowBase | MainLoopBase | Notes |
|---|---|---|
| mainloop | begin | |
| | end | Gets called automagically when no more instances of the subclass exist |
| __call__ | | Method moved to Gcf.show_all |

### Future compatibility

As eluded to above when discussing MEP 22, this refactor makes it easy to add in new generic features. At the moment, MEP 22 has to make ugly hacks to each class extending from `FigureManagerBase`. With this code, this only needs to get made in the single `FigureManager` class. This also makes the later deprecation of `NavigationToolbar2` very straightforward, only needing to touch the single `FigureManager` class

MEP 23 makes for another use case where this refactored code will come in very handy.

### Backward compatibility

As we leave all backend code intact, only adding missing methods to existing classes, this should work seamlessly for all use cases. The only difference will lie for backends that used `FigureManager.resize` to resize the canvas and not the window, due to the standardisation of the API.

I would envision that the classes made obsolete by this refactor get deprecated and removed on the same timetable as `NavigationToolbar2`, also note that the change in call signature to the `FigureCanvasWx` constructor, while backward compatible, I think the old (imho ugly style) signature should get deprecated and removed in the same manner as everything else.

| backend | manager.resize(w,h) | Extra |
|---|---|---|
| gtk3 | window | |
| Tk | canvas | |
| Qt | window | |
| Wx | canvas | FigureManagerWx had `frame` as an alias to window, so this also breaks BC. |

### Alternatives

If there were any alternative solutions to solving the same problem, they should be discussed here, along with a justification for the chosen approach.

### Questions

Mdehoon: Can you elaborate on how to run multiple backends concurrently?

OceanWolf: @mdehoon, as I say, not for this MEP, but I see this MEP opens it up as a future possibility. Basically the `MainLoopBase` class acts a per backend Gcf, in this MEP it tracks the number of figures open per backend, and manages the mainloops for those backends. It closes the backend specific mainloop when it detects that no figures remain open for that backend. Because of this I imagine that with only a small amount of tweaking that we can do full-multi-backend matplotlib. No idea yet why one would want to, but I leave the possibility there in MainLoopBase. With all the backend-code specifics refactored out of `FigureManager` also aids in this, one manager to rule them (the backends) all.

Mdehoon: @OceanWolf, OK, thanks for the explanation. Having a uniform API for the backends is very important for the maintainability of matplotlib. I think this MEP is a step in the right direction.

## MEP28: Remove Complexity from Axes.boxplot


- *Status*
- *Branches and Pull requests*
- *Abstract*
- *Detailed description*
  - *Importance*
- *Implementation*
  - *Passing transform functions to `cbook.boxplots_stats`*
  - *Simplifications to the `Axes.boxplot` API and other functions*
- *Backward compatibility*
  - *Schedule*
  - *Anticipated Impacts to Users*
  - *Anticipated Impacts to Downstream Libraries*
- *Alternatives*
  - *Variations on the theme*
  - *Doing less*
  - *Doing nothing*


### Status

**Discussion**

### Branches and Pull requests

The following lists any open PRs or branches related to this MEP:

1. Deprecate redundant statistical kwargs in `Axes.boxplot`: https://github.com/phobson/matplotlib/tree/MEP28-initial-deprecations
2. Deprecate redundant style options in `Axes.boxplot`: https://github.com/phobson/matplotlib/tree/MEP28-initial-deprecations
3. Deprecate passings 2D NumPy arrays as input: None
4. Add pre- & post-processing options to `cbook.boxplot_stats`: https://github.com/phobson/matplotlib/tree/boxplot-stat-transforms
5. Exposing `cbook.boxplot_stats` through `Axes.boxplot` kwargs: None

6. Remove redundant statistical kwargs in `Axes.boxplot`: None
7. Remove redundant style options in `Axes.boxplot`: None
8. Remaining items that arise through discussion: None

### Abstract

Over the past few releases, the `Axes.boxplot` method has grown in complexity to support fully customizable artist styling and statistical computation. This lead to `Axes.boxplot` being split off into multiple parts. The statistics needed to draw a boxplot are computed in `cbook.boxplot_stats`, while the actual artists are drawn by `Axes.bxp`. The original method, `Axes.boxplot` remains as the most public API that handles passing the user-supplied data to `cbook.boxplot_stats`, feeding the results to `Axes.bxp`, and pre-processing style information for each facet of the boxplot plots.

This MEP will outline a path forward to rollback the added complexity and simplify the API while maintaining reasonable backwards compatibility.

### Detailed description

Currently, the `Axes.boxplot` method accepts parameters that allow the users to specify medians and confidence intervals for each box that will be drawn in the plot. These were provided so that advanced users could provide statistics computed in a different fashion that the simple method provided by matplotlib. However, handling this input requires complex logic to make sure that the forms of the data structure match what needs to be drawn. At the moment, that logic contains 9 separate if/else statements nested up to 5 levels deep with a for loop, and may raise up to 2 errors. These parameters were added prior to the creation of the `Axes.bxp` method, which draws boxplots from a list of dictionaries containing the relevant statistics. Matplotlib also provides a function that computes these statistics via `cbook.boxplot_stats`. Note that advanced users can now either a) write their own function to compute the stats required by `Axes.bxp`, or b) modify the output returned by `cbook.boxplots_stats` to fully customize the position of the artists of the plots. With this flexibility, the parameters to manually specify only the medians and their confidences intervals remain for backwards compatibility.

Around the same time that the two roles of `Axes.boxplot` were split into `cbook.boxplot_stats` for computation and `Axes.bxp` for drawing, both `Axes.boxplot` and `Axes.bxp` were written to accept parameters that individually toggle the drawing of all components of the boxplots, and parameters that individually configure the style of those artists. However, to maintain backwards compatibility, the `sym` parameter (previously used to specify the symbol of the fliers) was retained. This parameter itself requires fairly complex logic to reconcile the `sym` parameters with the newer `flierprops` parameter at the default style specified by `matplotlibrc`.

This MEP seeks to dramatically simplify the creation of boxplots for novice and advanced users alike. Importantly, the changes proposed here will also be available to downstream packages like seaborn, as seaborn smartly allows users to pass arbitrary dictionaries of parameters through the seaborn API to the underlying matplotlib functions.

This will be achieved in the following way:

1. `cbook.boxplot_stats` will be modified to allow pre- and post- computation transformation functions to be passed in (e.g., `np.log` and `np.exp` for lognormally distributed data)
2. `Axes.boxplot` will be modified to also accept and naïvely pass them to `cbook.boxplots_stats` (Alt: pass the stat function and a dict of its optional parameters).
3. Outdated parameters from `Axes.boxplot` will be deprecated and later removed.

### Importance

Since the limits of the whiskers are computed arithmetically, there is an implicit assumption of normality in box and whisker plots. This primarily affects which data points are classified as outliers.

Allowing transformations to the data and the results used to draw boxplots will allow users to opt-out of that assumption if the data are known to not fit a normal distribution.

Below is an example of how `Axes.boxplot` classifies outliers of lognormal data differently depending one these types of transforms.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cbook
np.random.seed(0)

fig, ax = plt.subplots(figsize=(4, 6))
ax.set_yscale('log')
data = np.random.lognormal(-1.75, 2.75, size=37)

stats = cbook.boxplot_stats(data, labels=['arithmetic'])
logstats = cbook.boxplot_stats(np.log(data), labels=['log-transformed'])

for lsdict in logstats:
    for key, value in lsdict.items():
        if key != 'label':
            lsdict[key] = np.exp(value)

stats.extend(logstats)
ax.bxp(stats)
fig.show()
```

### Implementation

#### Passing transform functions to `cbook.boxplots_stats`

This MEP proposes that two parameters (e.g., `transform_in` and `transform_out` be added to the cookbook function that computes the statistics for the boxplot function. These will be optional keyword-only arguments and can easily be set to `lambda x: x` as a no-op when omitted by the user. The `transform_in` function will be applied to the data as the `boxplot_stats` function loops through each subset

of the data passed to it. After the list of statistics dictionaries are computed the `transform_out` function is applied to each value in the dictionaries.

These transformations can then be added to the call signature of `Axes.boxplot` with little impact to that method's complexity. This is because they can be directly passed to `cbook.boxplot_stats`. Alternatively, `Axes.boxplot` could be modified to accept an optional statistical function kwarg and a dictionary of parameters to be directly passed to it.

At this point in the implementation users and external libraries like seaborn would have complete control via the `Axes.boxplot` method. More importantly, at the very least, seaborn would require no changes to its API to allow users to take advantage of these new options.

### Simplifications to the `Axes.boxplot` API and other functions

Simplifying the boxplot method consists primarily of deprecating and then removing the redundant parameters. Optionally, a next step would include rectifying minor terminological inconsistencies between `Axes.boxplot` and `Axes.bxp`.

The parameters to be deprecated and removed include:

1. `usermedians` - processed by 10 SLOC, 3 `if` blocks, a `for` loop
2. `conf_intervals` - handled by 15 SLOC, 6 `if` blocks, a `for` loop
3. `sym` - processed by 12 SLOC, 4 `if` blocks

Removing the `sym` option allows all code in handling the remaining styling parameters to be moved to `Axes.bxp`. This doesn't remove any complexity, but does reinforce the single responsibility principle among `Axes.bxp`, `cbook.boxplot_stats`, and `Axes.boxplot`.

Additionally, the `notch` parameter could be renamed `shownotches` to be consistent with `Axes.bxp`. This kind of cleanup could be taken a step further and the `whis`, `bootstrap`, `autorange` could be rolled into the kwargs passed to the new `statfxn` parameter.

### Backward compatibility

Implementation of this MEP would eventually result in the backwards incompatible deprecation and then removal of the keyword parameters `usermedians`, `conf_intervals`, and `sym`. Cursory searches on GitHub indicated that `usermedians`, `conf_intervals` are used by few users, who all seem to have a very strong knowledge of matplotlib. A robust deprecation cycle should provide sufficient time for these users to migrate to a new API.

Deprecation of `sym` however, may have a much broader reach into the matplotlib userbase.

### Schedule

An accelerated timeline could look like the following:

1. v2.0.1 add transforms to `cbook.boxplots_stats`, expose in `Axes.boxplot`
2. v2.1.0 Initial Deprecations , and using 2D NumPy arrays as input
    a. Using 2D NumPy arrays as input. The semantics around 2D arrays are generally confusing.
    b. `usermedians`, `conf_intervals`, `sym` parameters
3. v2.2.0
    a. remove `usermedians`, `conf_intervals`, `sym` parameters
    b. deprecate `notch` in favor of `shownotches` to be consistent with other parameters and `Axes.bxp`
4. v2.3.0
    a. remove `notch` parameter
    b. move all style and artist toggling logic to `Axes.bxp` such `Axes.boxplot` is little more than a broker between `Axes.bxp` and `cbook.boxplots_stats`

### Anticipated Impacts to Users

As described above deprecating `usermedians` and `conf_intervals` will likely impact few users. Those who will be impacted are almost certainly advanced users who will be able to adapt to the change.

Deprecating the `sym` option may import more users and effort should be taken to collect community feedback on this.

### Anticipated Impacts to Downstream Libraries

The source code (GitHub master as of 2016-10-17) was inspected for seaborn and python-ggplot to see if these changes would impact their use. None of the parameters nominated for removal in this MEP are used by seaborn. The seaborn APIs that use matplotlib's boxplot function allow user's to pass arbitrary `**kwargs` through to matplotlib's API. Thus seaborn users with modern matplotlib installations will be able to take full advantage of any new features added as a result of this MEP.

Python-ggplot has implemented its own function to draw boxplots. Therefore, no impact can come to it as a result of implementing this MEP.

### Alternatives

### Variations on the theme

This MEP can be divided into a few loosely coupled components:

1. Allowing pre- and post-computation transformation function in `cbook.boxplot_stats`
2. Exposing that transformation in the `Axes.boxplot` API
3. Removing redundant statistical options in `Axes.boxplot`
4. Shifting all styling parameter processing from `Axes.boxplot` to `Axes.bxp`.

With this approach, #2 depends and #1, and #4 depends on #3.

There are two possible approaches to #2. The first and most direct would be to mirror the new `transform_in` and `transform_out` parameters of `cbook.boxplot_stats` in `Axes.boxplot` and pass them directly.

The second approach would be to add `statfxn` and `statfxn_args` parameters to `Axes.boxplot`. Under this implementation, the default value of `statfxn` would be `cbook.boxplot_stats`, but users could pass their own function. Then `transform_in` and `transform_out` would then be passed as elements of the `statfxn_args` parameter.

```
def boxplot_stats(data, ..., transform_in=None, transform_out=None):
    if transform_in is None:
        transform_in = lambda x: x

    if transform_out is None:
        transform_out = lambda x: x

    output = []
    for _d in data:
        d = transform_in(_d)
        stat_dict = do_stats(d)
        for key, value in stat_dict.item():
            if key != 'label':
                stat_dict[key] = transform_out(value)
        output.append(d)
    return output


 class Axes(...):
     def boxplot_option1(data, ..., transform_in=None, transform_out=None):
         stats = cbook.boxplot_stats(data, ...,
                                     transform_in=transform_in,
                                     transform_out=transform_out)
         return self.bxp(stats, ...)

     def boxplot_option2(data, ..., statfxn=None, **statopts):
         if statfxn is None:
             statfxn = boxplot_stats
```

(continues on next page)

(continued from previous page)

```
        stats = statfxn(data, **statopts)
        return self.bxp(stats, ...)
```

Both cases would allow users to do the following:

```
fig, ax1 = plt.subplots()
artists1 = ax1.boxplot_optionX(data, transform_in=np.log,
                               transform_out=np.exp)
```

But Option Two lets a user write a completely custom stat function (e.g., `my_box_stats`) with fancy BCA confidence intervals and the whiskers set differently depending on some attribute of the data.

This is available under the current API:

```
fig, ax1 = plt.subplots()
my_stats = my_box_stats(data, bootstrap_method='BCA',
                        whisker_method='dynamic')
ax1.bxp(my_stats)
```

And would be more concise with Option Two

```
fig, ax = plt.subplots()
statopts = dict(transform_in=np.log, transform_out=np.exp)
ax.boxplot(data, ..., **statopts)
```

Users could also pass their own function to compute the stats:

```
fig, ax1 = plt.subplots()
ax1.boxplot(data, statfxn=my_box_stats, bootstrap_method='BCA',
            whisker_method='dynamic')
```

From the examples above, Option Two seems to have only marginal benefit, but in the context of downstream libraries like seaborn, its advantage is more apparent as the following would be possible without any patches to seaborn:

```
import seaborn
tips = seaborn.load_data('tips')
g = seaborn.factorplot(x="day", y="total_bill", hue="sex", data=tips,
                       kind='box', palette="PRGn", shownotches=True,
                       statfxn=my_box_stats, bootstrap_method='BCA',
                       whisker_method='dynamic')
```

This type of flexibility was the intention behind splitting the overall boxplot API in the current three functions. In practice however, downstream libraries like seaborn support versions of matplotlib dating back well before the split. Thus, adding just a bit more flexibility to the `Axes.boxplot` could expose all the functionality to users of the downstream libraries with modern matplotlib installation without intervention from the downstream library maintainers.

### Doing less

Another obvious alternative would be to omit the added pre- and post- computation transform functionality in `cbook.boxplot_stats` and `Axes.boxplot`, and simply remove the redundant statistical and style parameters as described above.

### Doing nothing

As with many things in life, doing nothing is an option here. This means we simply advocate for users and downstream libraries to take advantage of the split between `cbook.boxplot_stats` and `Axes.bxp` and let them decide how to provide an interface to that.

## MEP29: Text light markup


- *Status*
- *Branches and Pull requests*
- *Abstract*
- *Detailed description*
- *Implementation*
  - *Improvements*
  - *Problems*
- *Backward compatibility*
- *Alternatives*


### Status

Discussion

### Branches and Pull requests

None at the moment, proof of concept only.

### Abstract

This MEP proposes to add lightweight markup to the text artist.

### Detailed description

Using different size/color/family in a text annotation is difficult because the `text` method accepts argument for size/color/family/weight/etc. that are used for the whole text. But, if one wants, for example, to have different colors, one has to look at the gallery where one such example is provided: *Concatenating text objects with different properties*

This example takes a list of strings as well as a list of colors which makes it cumbersome to use. An alternative would be to use a restricted set of pango-like markup and to interpret this markup.

Some markup examples:

```
Hello <b>world!</b>`
Hello <span color="blue">world!</span>
```

### Implementation

A proof of concept is provided in markup_example.py but it currently only handles the horizontal direction.

### Improvements

- This proof of concept uses regex to parse the text but it may be better to use the html.parser from the standard library.
- Computation of text fragment positions could benefit from the OffsetFrom class. See for example item 5 in Using Complex Coordinates with Annotations

### Problems

- One serious problem is how to deal with text having both LaTeX and HTML-like tags. For example, consider the following:

  ```
  $<b>Bold$</b>
  ```

  Recommendation would be to have mutual exclusion.

### Backward compatibility

None at the moment since it is only a proof of concept

### Alternatives

As proposed by @anntzer, this could be also implemented as improvements to mathtext. For example:

```
r"$\text{Hello \textbf{world}}$"
r"$\text{Hello \textcolor{blue}{world}}$"
r"$\text{Hello \textsf{\small world}}$"
```

# Part VI

# About us

Matplotlib was created by neurobiologist John Hunter to work with EEG data. It grew to be used and developed by many people in many different fields. John's goal was that Matplotlib make easy things easy and hard things possible.

CHAPTER

ELEVEN

# PROJECT INFORMATION

## 11.1 Mission Statement

The Matplotlib developer community develops, maintains, and supports Matplotlib and its extensions to provide data visualization tools for the Scientific Python Ecosystem.

Adapting the requirements *laid out by John Hunter* Matplotlib should:

- Support users of the Scientific Python ecosystem;
- Facilitate interactive data exploration;
- Produce high-quality raster and vector format outputs suitable for publication;
- Provide a simple graphical user interface and support embedding in applications;
- Be understandable and extensible by people familiar with data processing in Python;
- Make common plots easy, and novel or complex visualizations possible.

We believe that a diverse developer community creates the best software, and we welcome anyone who shares our mission, and our values described in the code of conduct.

## 11.2 History

**Note:** The following introductory text was written in 2008 by John D. Hunter (1968-2012), the original author of Matplotlib.

Matplotlib is a library for making 2D plots of arrays in Python. Although it has its origins in emulating the MATLAB graphics commands, it is independent of MATLAB, and can be used in a Pythonic, object-oriented way. Although Matplotlib is written primarily in pure Python, it makes heavy use of NumPy and other extension code to provide good performance even for large arrays.

Matplotlib is designed with the philosophy that you should be able to create simple plots with just a few commands, or just one! If you want to see a histogram of your data, you shouldn't need to instantiate objects, call methods, set properties, and so on; it should just work.

For years, I used to use MATLAB exclusively for data analysis and visualization. MATLAB excels at making nice looking plots easy. When I began working with EEG data, I found that I needed to write applications to interact with my data, and developed an EEG analysis application in MATLAB. As the application grew in complexity, interacting with databases, http servers, manipulating complex data structures, I began to strain against the limitations of MATLAB as a programming language, and decided to start over in Python. Python more than makes up for all of MATLAB's deficiencies as a programming language, but I was having difficulty finding a 2D plotting package (for 3D VTK more than exceeds all of my needs).

When I went searching for a Python plotting package, I had several requirements:

- Plots should look great - publication quality. One important requirement for me is that the text looks good (antialiased, etc.)
- Postscript output for inclusion with TeX documents
- Embeddable in a graphical user interface for application development
- Code should be easy enough that I can understand it and extend it
- Making plots should be easy

Finding no package that suited me just right, I did what any self-respecting Python programmer would do: rolled up my sleeves and dived in. Not having any real experience with computer graphics, I decided to emulate MATLAB's plotting capabilities because that is something MATLAB does very well. This had the added advantage that many people have a lot of MATLAB experience, and thus they can quickly get up to steam plotting in python. From a developer's perspective, having a fixed user interface (the pylab interface) has been very useful, because the guts of the code base can be redesigned without affecting user code.

The Matplotlib code is conceptually divided into three parts: the *pylab interface* is the set of functions provided by `pylab` which allow the user to create plots with code quite similar to MATLAB figure generating code (*Pyplot tutorial*). The *Matplotlib frontend* or *Matplotlib API* is the set of classes that do the heavy lifting, creating and managing figures, text, lines, plots and so on (*Artist tutorial*). This is an abstract interface that knows nothing about output. The *backends* are device-dependent drawing devices, aka renderers, that transform the frontend representation to hardcopy or a display device (*What is a backend?*). Example backends: PS creates PostScript® hardcopy, SVG creates Scalable Vector Graphics hardcopy, Agg creates PNG output using the high quality Anti-Grain Geometry library that ships with Matplotlib, GTK embeds Matplotlib in a Gtk+ application, GTKAgg uses the Anti-Grain renderer to create a figure and embed it in a Gtk+ application, and so on for PDF, WxWidgets, Tkinter, etc.

Matplotlib is used by many people in many different contexts. Some people want to automatically generate PostScript files to send to a printer or publishers. Others deploy Matplotlib on a web application server to generate PNG output for inclusion in dynamically-generated web pages. Some use Matplotlib interactively from the Python shell in Tkinter on Windows. My primary use is to embed Matplotlib in a Gtk+ EEG application that runs on Windows, Linux and Macintosh OS X.

---

Matplotlib's original logo (2003 -- 2008).

Matplotlib logo (2008 - 2015).

## 11.3 Contributor Covenant Code of Conduct

### 11.3.1 Our Pledge

We as members, contributors, and leaders pledge to make participation in our community a harassment-free experience for everyone, regardless of age, body size, visible or invisible disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

We pledge to act and interact in ways that contribute to an open, welcoming, diverse, inclusive, and healthy community.

### 11.3.2 Our Standards

Examples of behavior that contributes to a positive environment for our community include:

- Demonstrating empathy and kindness toward other people
- Being respectful of differing opinions, viewpoints, and experiences
- Giving and gracefully accepting constructive feedback
- Accepting responsibility and apologizing to those affected by our mistakes, and learning from the experience
- Focusing on what is best not just for us as individuals, but for the overall community

Examples of unacceptable behavior include:

- The use of sexualized language or imagery, and sexual attention or advances of any kind
- Trolling, insulting or derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or email address, without their explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

### 11.3.3 Enforcement Responsibilities

Community leaders are responsible for clarifying and enforcing our standards of acceptable behavior and will take appropriate and fair corrective action in response to any behavior that they deem inappropriate, threatening, offensive, or harmful.

Community leaders have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, and will communicate reasons for moderation decisions when appropriate.

### 11.3.4 Scope

This Code of Conduct applies within all community spaces, and also applies when an individual is officially representing the community in public spaces. Examples of representing our community include using an official e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event.

### 11.3.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported to the community leaders responsible for enforcement at matplotlib-coc@numfocus.org which is monitored by the CoC subcommittee or a report can be made using the NumFOCUS Code of Conduct report form. If community leaders cannot come to a resolution about enforcement, reports will be escalated to the NumFocus Code of Conduct committee (conduct@numfocus.org). All complaints will be reviewed and investigated promptly and fairly.

All community leaders are obligated to respect the privacy and security of the reporter of any incident.

### 11.3.6 Enforcement Guidelines

Community leaders will follow these Community Impact Guidelines in determining the consequences for any action they deem in violation of this Code of Conduct:

#### 1. Correction

**Community Impact**: Use of inappropriate language or other behavior deemed unprofessional or unwelcome in the community.

**Consequence**: A private, written warning from community leaders, providing clarity around the nature of the violation and an explanation of why the behavior was inappropriate. A public apology may be requested.

#### 2. Warning

**Community Impact**: A violation through a single incident or series of actions.

**Consequence**: A warning with consequences for continued behavior. No interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, for a specified period of time. This includes avoiding interactions in community spaces as well as external channels like social media. Violating these terms may lead to a temporary or permanent ban.

#### 3. Temporary Ban

**Community Impact**: A serious violation of community standards, including sustained inappropriate behavior.

**Consequence**: A temporary ban from any sort of interaction or public communication with the community for a specified period of time. No public or private interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, is allowed during this period. Violating these terms may lead to a permanent ban.

#### 4. Permanent Ban

**Community Impact**: Demonstrating a pattern of violation of community standards, including sustained inappropriate behavior, harassment of an individual, or aggression toward or disparagement of classes of individuals.

**Consequence**: A permanent ban from any sort of public interaction within the community.

### 11.3.7 Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 2.0, available at https://www.contributor-covenant.org/version/2/0/code_of_conduct.html.

Community Impact Guidelines were inspired by Mozilla's code of conduct enforcement ladder.

For answers to common questions about this code of conduct, see the FAQ at https://www.contributor-covenant.org/faq. Translations are available at https://www.contributor-covenant.org/translations.

## 11.4 Citing Matplotlib

If Matplotlib contributes to a project that leads to a scientific publication, please acknowledge this fact by citing J. D. Hunter, "Matplotlib: A 2D Graphics Environment", Computing in Science & Engineering, vol. 9, no. 3, pp. 90-95, 2007.

```
@Article{Hunter:2007,
  Author    = {Hunter, J. D.},
  Title     = {Matplotlib: A 2D graphics environment},
  Journal   = {Computing in Science \& Engineering},
  Volume    = {9},
  Number    = {3},
  Pages     = {90--95},
  abstract  = {Matplotlib is a 2D graphics package used for Python for
  application development, interactive scripting, and publication-quality
  image generation across user interfaces and operating systems.},
  publisher = {IEEE COMPUTER SOC},
  doi       = {10.1109/MCSE.2007.55},
```

(continues on next page)

(continued from previous page)

```
  year      = 2007
}
```

Download BibTeX bibliography file: CITATION.bib

### 11.4.1 DOIs

The following DOI represents *all* Matplotlib versions. Please select a more specific DOI from the list below, referring to the version used for your publication.

DOI 10.5281/zenodo.592536

#### By version

**v3.9.0**

DOI 10.5281/zenodo.11201097

**v3.8.4**

DOI 10.5281/zenodo.10916799

**v3.8.3**

DOI 10.5281/zenodo.10661079

**v3.8.2**

DOI 10.5281/zenodo.10150955

**v3.8.1**

DOI 10.5281/zenodo.10059757

**v3.8.0**

DOI 10.5281/zenodo.8347255

**v3.7.3**

DOI 10.5281/zenodo.8336761

**v3.7.2**

DOI 10.5281/zenodo.8118151

**v3.7.1**

DOI 10.5281/zenodo.7697899

**v3.7.0**

DOI 10.5281/zenodo.7637593

**v3.6.3**

DOI 10.5281/zenodo.7527665

**v3.6.2**

DOI 10.5281/zenodo.7275322

**v3.6.1**

DOI 10.5281/zenodo.7162185

**v3.6.0**

DOI 10.5281/zenodo.7084615

**v3.5.3**

DOI 10.5281/zenodo.6982547

**v3.5.2**

DOI 10.5281/zenodo.6513224

**v3.5.1**

DOI 10.5281/zenodo.5773480

**v3.5.0**

DOI 10.5281/zenodo.5706396

**v3.4.3**

DOI 10.5281/zenodo.5194481

**v3.4.2**

DOI 10.5281/zenodo.4743323

**v3.4.1**

DOI 10.5281/zenodo.4649959

**v3.4.0**

DOI 10.5281/zenodo.4638398

**v3.3.4**

DOI 10.5281/zenodo.4475376

**v3.3.3**

DOI 10.5281/zenodo.4268928

**v3.3.2**

DOI 10.5281/zenodo.4030140

**v3.3.1**

DOI 10.5281/zenodo.3984190

**v3.3.0**

DOI 10.5281/zenodo.3948793

**v3.2.2**

DOI 10.5281/zenodo.3898017

**v3.2.1**

DOI 10.5281/zenodo.3714460

**v3.2.0**

DOI 10.5281/zenodo.3695547

**v3.1.3**

DOI 10.5281/zenodo.3633844

**v3.1.2**

DOI 10.5281/zenodo.3563226

**v3.1.1**

DOI 10.5281/zenodo.3264781

**v3.1.0**

DOI 10.5281/zenodo.2893252

**v3.0.3**

DOI 10.5281/zenodo.2577644

**v3.0.2**

DOI 10.5281/zenodo.1482099

**v3.0.1**

**v2.2.5**

**v3.0.0**

**v2.2.4**

**v2.2.3**

**v2.2.2**

**v2.2.1**

**v2.2.0**

**v2.1.2**

**v2.1.1**

**v2.1.0**

**v2.0.2**

**v2.0.1**

**v2.0.0**

DOI 10.5281/zenodo.248351

**v1.5.3**

DOI 10.5281/zenodo.61948

**v1.5.2**

DOI 10.5281/zenodo.56926

**v1.5.1**

DOI 10.5281/zenodo.44579

**v1.5.0**

DOI 10.5281/zenodo.32914

**v1.4.3**

DOI 10.5281/zenodo.15423

**v1.4.2**

DOI 10.5281/zenodo.12400

**v1.4.1**

DOI 10.5281/zenodo.12287

**v1.4.0**

DOI 10.5281/zenodo.11451

## 11.5 License

Matplotlib only uses BSD compatible code, and its license is based on the PSF license. See the Open Source Initiative licenses page for details on individual licenses. Non-BSD compatible licenses (e.g., LGPL) are acceptable in matplotlib toolkits. For a discussion of the motivations behind the licencing choice, see *Licenses for contributed code*.

### 11.5.1 Copyright policy

John Hunter began Matplotlib around 2003. Since shortly before his passing in 2012, Michael Droettboom has been the lead maintainer of Matplotlib, but, as has always been the case, Matplotlib is the work of many.

Prior to July of 2013, and the 1.3.0 release, the copyright of the source code was held by John Hunter. As of July 2013, and the 1.3.0 release, matplotlib has moved to a shared copyright model.

Matplotlib uses a shared copyright model. Each contributor maintains copyright over their contributions to Matplotlib. But, it is important to note that these contributions are typically only changes to the repositories. Thus, the Matplotlib source code, in its entirety, is not the copyright of any single person or institution. Instead, it is the collective copyright of the entire Matplotlib Development Team. If individual contributors want to maintain a record of what changes/contributions they have specific copyright on, they should indicate their copyright in the commit message of the change, when they commit the change to one of the matplotlib repositories.

The Matplotlib Development Team is the set of all contributors to the matplotlib project. A full list can be obtained from the git version control logs.

### 11.5.2 License agreement

#### License agreement for Matplotlib versions 1.3.0 and later

```
License agreement for matplotlib versions 1.3.0 and later
=========================================================

1. This LICENSE AGREEMENT is between the Matplotlib Development Team
("MDT"), and the Individual or Organization ("Licensee") accessing and
otherwise using matplotlib software in source or binary form and its
associated documentation.

2. Subject to the terms and conditions of this License Agreement, MDT
hereby grants Licensee a nonexclusive, royalty-free, world-wide license
to reproduce, analyze, test, perform and/or display publicly, prepare
derivative works, distribute, and otherwise use matplotlib
alone or in any derivative version, provided, however, that MDT's
License Agreement and MDT's notice of copyright, i.e., "Copyright (c)
2012- Matplotlib Development Team; All Rights Reserved" are retained in
matplotlib  alone or in any derivative version prepared by
Licensee.

3. In the event Licensee prepares a derivative work that is based on or
incorporates matplotlib or any part thereof, and wants to
make the derivative work available to others as provided herein, then
Licensee hereby agrees to include in any such work a brief summary of
the changes made to matplotlib .

4. MDT is making matplotlib available to Licensee on an "AS
IS" basis.  MDT MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR
IMPLIED.  BY WAY OF EXAMPLE, BUT NOT LIMITATION, MDT MAKES NO AND
```

(continues on next page)

(continued from previous page)

```
DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS
FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF MATPLOTLIB
WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. MDT SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF MATPLOTLIB
 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR
LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING
MATPLOTLIB , OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF
THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material
breach of its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any
relationship of agency, partnership, or joint venture between MDT and
Licensee.  This License Agreement does not grant permission to use MDT
trademarks or trade name in a trademark sense to endorse or promote
products or services of Licensee, or any third party.

8. By copying, installing or otherwise using matplotlib ,
Licensee agrees to be bound by the terms and conditions of this License
Agreement.

License agreement for matplotlib versions prior to 1.3.0
========================================================

1. This LICENSE AGREEMENT is between John D. Hunter ("JDH"), and the
Individual or Organization ("Licensee") accessing and otherwise using
matplotlib software in source or binary form and its associated
documentation.

2. Subject to the terms and conditions of this License Agreement, JDH
hereby grants Licensee a nonexclusive, royalty-free, world-wide license
to reproduce, analyze, test, perform and/or display publicly, prepare
derivative works, distribute, and otherwise use matplotlib
alone or in any derivative version, provided, however, that JDH's
License Agreement and JDH's notice of copyright, i.e., "Copyright (c)
2002-2011 John D. Hunter; All Rights Reserved" are retained in
matplotlib  alone or in any derivative version prepared by
Licensee.

3. In the event Licensee prepares a derivative work that is based on or
incorporates matplotlib  or any part thereof, and wants to
make the derivative work available to others as provided herein, then
Licensee hereby agrees to include in any such work a brief summary of
the changes made to matplotlib.

4. JDH is making matplotlib  available to Licensee on an "AS
IS" basis.  JDH MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR
IMPLIED.  BY WAY OF EXAMPLE, BUT NOT LIMITATION, JDH MAKES NO AND
DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS
FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF MATPLOTLIB
```

(continues on next page)

(continued from previous page)

```
WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. JDH SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF MATPLOTLIB
 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR
LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING
MATPLOTLIB , OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF
THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material
breach of its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any
relationship of agency, partnership, or joint venture between JDH and
Licensee.  This License Agreement does not grant permission to use JDH
trademarks or trade name in a trademark sense to endorse or promote
products or services of Licensee, or any third party.

8. By copying, installing or otherwise using matplotlib,
Licensee agrees to be bound by the terms and conditions of this License
Agreement.
```

### 11.5.3 Bundled software

#### JSX Tools Resize Observer

```
# CC0 1.0 Universal

## Statement of Purpose

The laws of most jurisdictions throughout the world automatically confer
exclusive Copyright and Related Rights (defined below) upon the creator and
subsequent owner(s) (each and all, an “owner”) of an original work of
authorship and/or a database (each, a “Work”).

Certain owners wish to permanently relinquish those rights to a Work for the
purpose of contributing to a commons of creative, cultural and scientific␣
 ↪works
(“Commons”) that the public can reliably and without fear of later claims of
infringement build upon, modify, incorporate in other works, reuse and
redistribute as freely as possible in any form whatsoever and for any␣
 ↪purposes,
including without limitation commercial purposes. These owners may contribute
to the Commons to promote the ideal of a free culture and the further
production of creative, cultural and scientific works, or to gain reputation␣
 ↪or
greater distribution for their Work in part through the use and efforts of
others.

For these and/or other purposes and motivations, and without any expectation␣
```

(continues on next page)

(continued from previous page)

```
 ↪of
additional consideration or compensation, the person associating CC0 with a
Work (the “Affirmer”), to the extent that he or she is an owner of Copyright
and Related Rights in the Work, voluntarily elects to apply CC0 to the Work␣
 ↪and
publicly distribute the Work under its terms, with knowledge of his or her
Copyright and Related Rights in the Work and the meaning and intended legal
effect of CC0 on those rights.

1. Copyright and Related Rights. A Work made available under CC0 may be
   protected by copyright and related or neighboring rights (“Copyright and
   Related Rights”). Copyright and Related Rights include, but are not limited
   to, the following:
   1. the right to reproduce, adapt, distribute, perform, display,␣
 ↪communicate,
      and translate a Work;
   2. moral rights retained by the original author(s) and/or performer(s);
   3. publicity and privacy rights pertaining to a person’s image or likeness
      depicted in a Work;
   4. rights protecting against unfair competition in regards to a Work,
      subject to the limitations in paragraph 4(i), below;
   5. rights protecting the extraction, dissemination, use and reuse of data␣
 ↪in
      a Work;
   6. database rights (such as those arising under Directive 96/9/EC of the
      European Parliament and of the Council of 11 March 1996 on the legal
      protection of databases, and under any national implementation thereof,
      including any amended or successor version of such directive); and
   7. other similar, equivalent or corresponding rights throughout the world
      based on applicable law or treaty, and any national implementations
      thereof.

2. Waiver. To the greatest extent permitted by, but not in contravention of,
   applicable law, Affirmer hereby overtly, fully, permanently, irrevocably␣
 ↪and
   unconditionally waives, abandons, and surrenders all of Affirmer’s␣
 ↪Copyright
   and Related Rights and associated claims and causes of action, whether now
   known or unknown (including existing as well as future claims and causes of
   action), in the Work (i) in all territories worldwide, (ii) for the maximum
   duration provided by applicable law or treaty (including future time
   extensions), (iii) in any current or future medium and for any number of
   copies, and (iv) for any purpose whatsoever, including without limitation
   commercial, advertising or promotional purposes (the “Waiver”). Affirmer
   makes the Waiver for the benefit of each member of the public at large and
   to the detriment of Affirmer’s heirs and successors, fully intending that
   such Waiver shall not be subject to revocation, rescission, cancellation,
   termination, or any other legal or equitable action to disrupt the quiet
   enjoyment of the Work by the public as contemplated by Affirmer’s express
   Statement of Purpose.

3. Public License Fallback. Should any part of the Waiver for any reason be
```

(continues on next page)

(continued from previous page)

```
   judged legally invalid or ineffective under applicable law, then the Waiver
   shall be preserved to the maximum extent permitted taking into account
   Affirmer's express Statement of Purpose. In addition, to the extent the
   Waiver is so judged Affirmer hereby grants to each affected person a
   royalty-free, non transferable, non sublicensable, non exclusive,
   irrevocable and unconditional license to exercise Affirmer's Copyright and
   Related Rights in the Work (i) in all territories worldwide, (ii) for the
   maximum duration provided by applicable law or treaty (including future␣
 ↪time
   extensions), (iii) in any current or future medium and for any number of
   copies, and (iv) for any purpose whatsoever, including without limitation
   commercial, advertising or promotional purposes (the "License"). The␣
 ↪License
   shall be deemed effective as of the date CC0 was applied by Affirmer to the
   Work. Should any part of the License for any reason be judged legally
   invalid or ineffective under applicable law, such partial invalidity or
   ineffectiveness shall not invalidate the remainder of the License, and in
   such case Affirmer hereby affirms that he or she will not (i) exercise any
   of his or her remaining Copyright and Related Rights in the Work or (ii)
   assert any associated claims and causes of action with respect to the Work,
   in either case contrary to Affirmer's express Statement of Purpose.

4. Limitations and Disclaimers.
   1. No trademark or patent rights held by Affirmer are waived, abandoned,
      surrendered, licensed or otherwise affected by this document.
   2. Affirmer offers the Work as-is and makes no representations or␣
 ↪warranties
      of any kind concerning the Work, express, implied, statutory or
      otherwise, including without limitation warranties of title,
      merchantability, fitness for a particular purpose, non infringement, or
      the absence of latent or other defects, accuracy, or the present or
      absence of errors, whether or not discoverable, all to the greatest
      extent permissible under applicable law.
   3. Affirmer disclaims responsibility for clearing rights of other persons
      that may apply to the Work or any use thereof, including without
      limitation any person's Copyright and Related Rights in the Work.
      Further, Affirmer disclaims responsibility for obtaining any necessary
      consents, permissions or other rights required for any use of the Work.
   4. Affirmer understands and acknowledges that Creative Commons is not a
      party to this document and has no duty or obligation with respect to␣
 ↪this
      CC0 or use of the Work.

For more information, please see
http://creativecommons.org/publicdomain/zero/1.0/.
```

### QT4 Editor

```
Module creating PyQt4 form dialogs/layouts to edit various type of parameters


formlayout License Agreement (MIT License)
------------------------------------------

Copyright (c) 2009 Pierre Raybaut

Permission is hereby granted, free of charge, to any person
obtaining a copy of this software and associated documentation
files (the "Software"), to deal in the Software without
restriction, including without limitation the rights to use,
copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the
Software is furnished to do so, subject to the following
conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
OTHER DEALINGS IN THE SOFTWARE.
"""
```

### Colormaps and themes

### ColorBrewer

```
Apache-Style Software License for ColorBrewer software and ColorBrewer Color␣
 ↪Schemes

Copyright (c) 2002 Cynthia Brewer, Mark Harrower, and The Pennsylvania State␣
 ↪University.

Licensed under the Apache License, Version 2.0 (the "License"); you may not␣
 ↪use this file except in compliance with the License.
You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software␣
```

(continues on next page)

(continued from previous page)

```
 ↪distributed
under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
CONDITIONS OF ANY KIND, either express or implied. See the License for the
specific language governing permissions and limitations under the License.
```

### Solarized

```
https://github.com/altercation/solarized/blob/master/LICENSE
Copyright (c) 2011 Ethan Schoonover

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
THE SOFTWARE.
```

### Yorick

```
BSD-style license for gist/yorick colormaps.

Copyright:

  Copyright (c) 1996.  The Regents of the University of California.
                         All rights reserved.

Permission to use, copy, modify, and distribute this software for any
purpose without fee is hereby granted, provided that this entire
notice is included in all copies of any software which is or includes
a copy or modification of this software and in all copies of the
supporting documentation for such software.

This work was produced at the University of California, Lawrence
Livermore National Laboratory under contract no. W-7405-ENG-48 between
the U.S. Department of Energy and The Regents of the University of
California for the operation of UC LLNL.
```

(continues on next page)

(continued from previous page)

```
                                DISCLAIMER

This software was prepared as an account of work sponsored by an
agency of the United States Government.  Neither the United States
Government nor the University of California nor any of their
employees, makes any warranty, express or implied, or assumes any
liability or responsibility for the accuracy, completeness, or
usefulness of any information, apparatus, product, or process
disclosed, or represents that its use would not infringe
privately-owned rights.  Reference herein to any specific commercial
products, process, or service by trade name, trademark, manufacturer,
or otherwise, does not necessarily constitute or imply its
endorsement, recommendation, or favoring by the United States
Government or the University of California.  The views and opinions of
authors expressed herein do not necessarily state or reflect those of
the United States Government or the University of California, and
shall not be used for advertising or product endorsement purposes.


                                  AUTHOR

David H. Munro wrote Yorick and Gist.  Berkeley Yacc (byacc) generated
the Yorick parser.  The routines in Math are from LAPACK and FFTPACK;
MathC contains C translations by David H. Munro.  The algorithms for
Yorick's random number generator and several special functions in
Yorick/include were taken from Numerical Recipes by Press, et. al.,
although the Yorick implementations are unrelated to those in
Numerical Recipes.  A small amount of code in Gist was adapted from
the X11R4 release, copyright M.I.T. -- the complete copyright notice
may be found in the (unused) file Gist/host.c.
```

## Fonts

### American Mathematical Society (AMS) fonts

```
The cmr10.pfb file is a Type-1 version of one of Knuth's Computer Modern␣
↪fonts.
It is included here as test data only, but the following license applies.

Copyright (c) 1997, 2009, American Mathematical Society (http://www.ams.org).
All Rights Reserved.

"cmb10" is a Reserved Font Name for this Font Software.
"cmbsy10" is a Reserved Font Name for this Font Software.
"cmbsy5" is a Reserved Font Name for this Font Software.
"cmbsy6" is a Reserved Font Name for this Font Software.
"cmbsy7" is a Reserved Font Name for this Font Software.
"cmbsy8" is a Reserved Font Name for this Font Software.
```

(continues on next page)

(continued from previous page)

```
"cmbsy9" is a Reserved Font Name for this Font Software.
"cmbx10" is a Reserved Font Name for this Font Software.
"cmbx12" is a Reserved Font Name for this Font Software.
"cmbx5" is a Reserved Font Name for this Font Software.
"cmbx6" is a Reserved Font Name for this Font Software.
"cmbx7" is a Reserved Font Name for this Font Software.
"cmbx8" is a Reserved Font Name for this Font Software.
"cmbx9" is a Reserved Font Name for this Font Software.
"cmbxsl10" is a Reserved Font Name for this Font Software.
"cmbxti10" is a Reserved Font Name for this Font Software.
"cmcsc10" is a Reserved Font Name for this Font Software.
"cmcsc8" is a Reserved Font Name for this Font Software.
"cmcsc9" is a Reserved Font Name for this Font Software.
"cmdunh10" is a Reserved Font Name for this Font Software.
"cmex10" is a Reserved Font Name for this Font Software.
"cmex7" is a Reserved Font Name for this Font Software.
"cmex8" is a Reserved Font Name for this Font Software.
"cmex9" is a Reserved Font Name for this Font Software.
"cmff10" is a Reserved Font Name for this Font Software.
"cmfi10" is a Reserved Font Name for this Font Software.
"cmfib8" is a Reserved Font Name for this Font Software.
"cminch" is a Reserved Font Name for this Font Software.
"cmitt10" is a Reserved Font Name for this Font Software.
"cmmi10" is a Reserved Font Name for this Font Software.
"cmmi12" is a Reserved Font Name for this Font Software.
"cmmi5" is a Reserved Font Name for this Font Software.
"cmmi6" is a Reserved Font Name for this Font Software.
"cmmi7" is a Reserved Font Name for this Font Software.
"cmmi8" is a Reserved Font Name for this Font Software.
"cmmi9" is a Reserved Font Name for this Font Software.
"cmmib10" is a Reserved Font Name for this Font Software.
"cmmib5" is a Reserved Font Name for this Font Software.
"cmmib6" is a Reserved Font Name for this Font Software.
"cmmib7" is a Reserved Font Name for this Font Software.
"cmmib8" is a Reserved Font Name for this Font Software.
"cmmib9" is a Reserved Font Name for this Font Software.
"cmr10" is a Reserved Font Name for this Font Software.
"cmr12" is a Reserved Font Name for this Font Software.
"cmr17" is a Reserved Font Name for this Font Software.
"cmr5" is a Reserved Font Name for this Font Software.
"cmr6" is a Reserved Font Name for this Font Software.
"cmr7" is a Reserved Font Name for this Font Software.
"cmr8" is a Reserved Font Name for this Font Software.
"cmr9" is a Reserved Font Name for this Font Software.
"cmsl10" is a Reserved Font Name for this Font Software.
"cmsl12" is a Reserved Font Name for this Font Software.
"cmsl8" is a Reserved Font Name for this Font Software.
"cmsl9" is a Reserved Font Name for this Font Software.
"cmsltt10" is a Reserved Font Name for this Font Software.
"cmss10" is a Reserved Font Name for this Font Software.
"cmss12" is a Reserved Font Name for this Font Software.
"cmss17" is a Reserved Font Name for this Font Software.
```

(continues on next page)

(continued from previous page)

```
"cmss8" is a Reserved Font Name for this Font Software.
"cmss9" is a Reserved Font Name for this Font Software.
"cmssbx10" is a Reserved Font Name for this Font Software.
"cmssdc10" is a Reserved Font Name for this Font Software.
"cmssi10" is a Reserved Font Name for this Font Software.
"cmssi12" is a Reserved Font Name for this Font Software.
"cmssi17" is a Reserved Font Name for this Font Software.
"cmssi8" is a Reserved Font Name for this Font Software.
"cmssi9" is a Reserved Font Name for this Font Software.
"cmssq8" is a Reserved Font Name for this Font Software.
"cmssqi8" is a Reserved Font Name for this Font Software.
"cmsy10" is a Reserved Font Name for this Font Software.
"cmsy5" is a Reserved Font Name for this Font Software.
"cmsy6" is a Reserved Font Name for this Font Software.
"cmsy7" is a Reserved Font Name for this Font Software.
"cmsy8" is a Reserved Font Name for this Font Software.
"cmsy9" is a Reserved Font Name for this Font Software.
"cmtcsc10" is a Reserved Font Name for this Font Software.
"cmtex10" is a Reserved Font Name for this Font Software.
"cmtex8" is a Reserved Font Name for this Font Software.
"cmtex9" is a Reserved Font Name for this Font Software.
"cmti10" is a Reserved Font Name for this Font Software.
"cmti12" is a Reserved Font Name for this Font Software.
"cmti7" is a Reserved Font Name for this Font Software.
"cmti8" is a Reserved Font Name for this Font Software.
"cmti9" is a Reserved Font Name for this Font Software.
"cmtt10" is a Reserved Font Name for this Font Software.
"cmtt12" is a Reserved Font Name for this Font Software.
"cmtt8" is a Reserved Font Name for this Font Software.
"cmtt9" is a Reserved Font Name for this Font Software.
"cmu10" is a Reserved Font Name for this Font Software.
"cmvtt10" is a Reserved Font Name for this Font Software.
"euex10" is a Reserved Font Name for this Font Software.
"euex7" is a Reserved Font Name for this Font Software.
"euex8" is a Reserved Font Name for this Font Software.
"euex9" is a Reserved Font Name for this Font Software.
"eufb10" is a Reserved Font Name for this Font Software.
"eufb5" is a Reserved Font Name for this Font Software.
"eufb7" is a Reserved Font Name for this Font Software.
"eufm10" is a Reserved Font Name for this Font Software.
"eufm5" is a Reserved Font Name for this Font Software.
"eufm7" is a Reserved Font Name for this Font Software.
"eurb10" is a Reserved Font Name for this Font Software.
"eurb5" is a Reserved Font Name for this Font Software.
"eurb7" is a Reserved Font Name for this Font Software.
"eurm10" is a Reserved Font Name for this Font Software.
"eurm5" is a Reserved Font Name for this Font Software.
"eurm7" is a Reserved Font Name for this Font Software.
"eusb10" is a Reserved Font Name for this Font Software.
"eusb5" is a Reserved Font Name for this Font Software.
"eusb7" is a Reserved Font Name for this Font Software.
"eusm10" is a Reserved Font Name for this Font Software.
```

(continues on next page)

(continued from previous page)

```
"eusm5" is a Reserved Font Name for this Font Software.
"eusm7" is a Reserved Font Name for this Font Software.
"lasy10" is a Reserved Font Name for this Font Software.
"lasy5" is a Reserved Font Name for this Font Software.
"lasy6" is a Reserved Font Name for this Font Software.
"lasy7" is a Reserved Font Name for this Font Software.
"lasy8" is a Reserved Font Name for this Font Software.
"lasy9" is a Reserved Font Name for this Font Software.
"lasyb10" is a Reserved Font Name for this Font Software.
"lcircle1" is a Reserved Font Name for this Font Software.
"lcirclew" is a Reserved Font Name for this Font Software.
"lcmss8" is a Reserved Font Name for this Font Software.
"lcmssb8" is a Reserved Font Name for this Font Software.
"lcmssi8" is a Reserved Font Name for this Font Software.
"line10" is a Reserved Font Name for this Font Software.
"linew10" is a Reserved Font Name for this Font Software.
"msam10" is a Reserved Font Name for this Font Software.
"msam5" is a Reserved Font Name for this Font Software.
"msam6" is a Reserved Font Name for this Font Software.
"msam7" is a Reserved Font Name for this Font Software.
"msam8" is a Reserved Font Name for this Font Software.
"msam9" is a Reserved Font Name for this Font Software.
"msbm10" is a Reserved Font Name for this Font Software.
"msbm5" is a Reserved Font Name for this Font Software.
"msbm6" is a Reserved Font Name for this Font Software.
"msbm7" is a Reserved Font Name for this Font Software.
"msbm8" is a Reserved Font Name for this Font Software.
"msbm9" is a Reserved Font Name for this Font Software.
"wncyb10" is a Reserved Font Name for this Font Software.
"wncyi10" is a Reserved Font Name for this Font Software.
"wncyr10" is a Reserved Font Name for this Font Software.
"wncysc10" is a Reserved Font Name for this Font Software.
"wncyss10" is a Reserved Font Name for this Font Software.

This Font Software is licensed under the SIL Open Font License, Version 1.1.
This license is copied below, and is also available with a FAQ at:
http://scripts.sil.org/OFL

-----------------------------------------------------------
SIL OPEN FONT LICENSE Version 1.1 - 26 February 2007
-----------------------------------------------------------

PREAMBLE
The goals of the Open Font License (OFL) are to stimulate worldwide
development of collaborative font projects, to support the font creation
efforts of academic and linguistic communities, and to provide a free and
open framework in which fonts may be shared and improved in partnership
with others.

The OFL allows the licensed fonts to be used, studied, modified and
redistributed freely as long as they are not sold by themselves. The
fonts, including any derivative works, can be bundled, embedded,
```

(continues on next page)

(continued from previous page)

```
redistributed and/or sold with any software provided that any reserved
names are not used by derivative works. The fonts and derivatives,
however, cannot be released under any other type of license. The
requirement for fonts to remain under this license does not apply
to any document created using the fonts or their derivatives.

DEFINITIONS
"Font Software" refers to the set of files released by the Copyright
Holder(s) under this license and clearly marked as such. This may
include source files, build scripts and documentation.

"Reserved Font Name" refers to any names specified as such after the
copyright statement(s).

"Original Version" refers to the collection of Font Software components as
distributed by the Copyright Holder(s).

"Modified Version" refers to any derivative made by adding to, deleting,
or substituting -- in part or in whole -- any of the components of the
Original Version, by changing formats or by porting the Font Software to a
new environment.

"Author" refers to any designer, engineer, programmer, technical
writer or other person who contributed to the Font Software.

PERMISSION & CONDITIONS
Permission is hereby granted, free of charge, to any person obtaining
a copy of the Font Software, to use, study, copy, merge, embed, modify,
redistribute, and sell modified and unmodified copies of the Font
Software, subject to the following conditions:

1) Neither the Font Software nor any of its individual components,
in Original or Modified Versions, may be sold by itself.

2) Original or Modified Versions of the Font Software may be bundled,
redistributed and/or sold with any software, provided that each copy
contains the above copyright notice and this license. These can be
included either as stand-alone text files, human-readable headers or
in the appropriate machine-readable metadata fields within text or
binary files as long as those fields can be easily viewed by the user.

3) No Modified Version of the Font Software may use the Reserved Font
Name(s) unless explicit written permission is granted by the corresponding
Copyright Holder. This restriction only applies to the primary font name as
presented to the users.

4) The name(s) of the Copyright Holder(s) or the Author(s) of the Font
Software shall not be used to promote, endorse or advertise any
Modified Version, except to acknowledge the contribution(s) of the
Copyright Holder(s) and the Author(s) or with their explicit written
permission.
```

(continues on next page)

(continued from previous page)

```
5) The Font Software, modified or unmodified, in part or in whole,
must be distributed entirely under this license, and must not be
distributed under any other license. The requirement for fonts to
remain under this license does not apply to any document created
using the Font Software.

TERMINATION
This license becomes null and void if any of the above conditions are
not met.

DISCLAIMER
THE FONT SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT
OF COPYRIGHT, PATENT, TRADEMARK, OR OTHER RIGHT. IN NO EVENT SHALL THE
COPYRIGHT HOLDER BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
INCLUDING ANY GENERAL, SPECIAL, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL
DAMAGES, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
FROM, OUT OF THE USE OR INABILITY TO USE THE FONT SOFTWARE OR FROM
OTHER DEALINGS IN THE FONT SOFTWARE.
```

### BaKoMa

```
                    BaKoMa Fonts Licence
                    --------------------

  This licence covers two font packs (known as BaKoMa Fonts Collection,
  which is available at `CTAN:fonts/cm/ps-type1/bakoma/'):

    1) BaKoMa-CM (1.1/12-Nov-94)
       Computer Modern Fonts in PostScript Type 1 and TrueType font formats.

    2) BaKoMa-AMS (1.2/19-Jan-95)
       AMS TeX fonts in PostScript Type 1 and TrueType font formats.

  Copyright (C) 1994, 1995, Basil K. Malyshev. All Rights Reserved.

  Permission to copy and distribute these fonts for any purpose is
  hereby granted without fee, provided that the above copyright notice,
  author statement and this permission notice appear in all copies of
  these fonts and related documentation.

  Permission to modify and distribute modified fonts for any purpose is
  hereby granted without fee, provided that the copyright notice,
  author statement, this permission notice and location of original
  fonts (http://www.ctan.org/tex-archive/fonts/cm/ps-type1/bakoma)
  appear in all copies of modified fonts and related documentation.

  Permission to use these fonts (embedding into PostScript, PDF, SVG
```

(continues on next page)

(continued from previous page)

```
  and printing by using any software) is hereby granted without fee.
  It is not required to provide any notices about using these fonts.

 Basil K. Malyshev
 INSTITUTE FOR HIGH ENERGY PHYSICS
 IHEP, OMVT
 Moscow Region
 142281 PROTVINO
 RUSSIA

 E-Mail:         bakoma@mail.ru
      or         malyshev@mail.ihep.ru
```

### Carlogo

```
----> we renamed carlito -> carlogo to comply with the terms <----

Copyright (c) 2010-2013 by tyPoland Lukasz Dziedzic with Reserved Font Name
 ↪"Carlito".

This Font Software is licensed under the SIL Open Font License, Version 1.1.
This license is copied below, and is also available with a FAQ at: http://
 ↪scripts.sil.org/OFL

-----------------------------------------------------------
SIL OPEN FONT LICENSE Version 1.1 - 26 February 2007
-----------------------------------------------------------

PREAMBLE
The goals of the Open Font License (OFL) are to stimulate worldwide␣
 ↪development of collaborative font projects, to support the font creation␣
 ↪efforts of academic and linguistic communities, and to provide a free and␣
 ↪open framework in which fonts may be shared and improved in partnership␣
 ↪with others.

The OFL allows the licensed fonts to be used, studied, modified and␣
 ↪redistributed freely as long as they are not sold by themselves. The fonts,␣
 ↪including any derivative works, can be bundled, embedded, redistributed and/
 ↪or sold with any software provided that any reserved names are not used by␣
 ↪derivative works. The fonts and derivatives, however, cannot be released␣
 ↪under any other type of license. The requirement for fonts to remain under␣
 ↪this license does not apply to any document created using the fonts or␣
 ↪their derivatives.

DEFINITIONS
"Font Software" refers to the set of files released by the Copyright␣
 ↪Holder(s) under this license and clearly marked as such. This may include␣
 ↪source files, build scripts and documentation.
```

(continues on next page)

(continued from previous page)

```
"Reserved Font Name" refers to any names specified as such after the␣
 ↪copyright statement(s).

"Original Version" refers to the collection of Font Software components as␣
 ↪distributed by the Copyright Holder(s).

"Modified Version" refers to any derivative made by adding to, deleting, or␣
 ↪substituting -- in part or in whole -- any of the components of the␣
 ↪Original Version, by changing formats or by porting the Font Software to a␣
 ↪new environment.

"Author" refers to any designer, engineer, programmer, technical writer or␣
 ↪other person who contributed to the Font Software.

PERMISSION & CONDITIONS
Permission is hereby granted, free of charge, to any person obtaining a copy␣
 ↪of the Font Software, to use, study, copy, merge, embed, modify,␣
 ↪redistribute, and sell modified and unmodified copies of the Font Software,␣
 ↪subject to the following conditions:

1) Neither the Font Software nor any of its individual components, in␣
 ↪Original or Modified Versions, may be sold by itself.

2) Original or Modified Versions of the Font Software may be bundled,␣
 ↪redistributed and/or sold with any software, provided that each copy␣
 ↪contains the above copyright notice and this license. These can be included␣
 ↪either as stand-alone text files, human-readable headers or in the␣
 ↪appropriate machine-readable metadata fields within text or binary files as␣
 ↪long as those fields can be easily viewed by the user.

3) No Modified Version of the Font Software may use the Reserved Font Name(s)␣
 ↪unless explicit written permission is granted by the corresponding␣
 ↪Copyright Holder. This restriction only applies to the primary font name as␣
 ↪presented to the users.

4) The name(s) of the Copyright Holder(s) or the Author(s) of the Font␣
 ↪Software shall not be used to promote, endorse or advertise any Modified␣
 ↪Version, except to acknowledge the contribution(s) of the Copyright␣
 ↪Holder(s) and the Author(s) or with their explicit written permission.

5) The Font Software, modified or unmodified, in part or in whole, must be␣
 ↪distributed entirely under this license, and must not be distributed under␣
 ↪any other license. The requirement for fonts to remain under this license␣
 ↪does not apply to any document created using the Font Software.

TERMINATION
This license becomes null and void if any of the above conditions are not met.

DISCLAIMER
THE FONT SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS␣
 ↪OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OF MERCHANTABILITY,␣
 ↪FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF COPYRIGHT, PATENT,␣
```

(continues on next page)

(continued from previous page)

```
↪TRADEMARK, OR OTHER RIGHT. IN NO EVENT SHALL THE COPYRIGHT HOLDER BE LIABLE␣
↪FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, INCLUDING ANY GENERAL, SPECIAL,␣
↪INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, WHETHER IN AN ACTION OF␣
↪CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF THE USE OR INABILITY TO␣
↪USE THE FONT SOFTWARE OR FROM OTHER DEALINGS IN THE FONT SOFTWARE.
```

### Courier 10

```
The Courier10PitchBT-Bold.pfb file is a Type-1 version of
Courier 10 Pitch BT Bold by Bitstream, obtained from
<https://ctan.org/tex-archive/fonts/courierten>. It is included
here as test data only, but the following license applies.


(c) Copyright 1989-1992, Bitstream Inc., Cambridge, MA.

You are hereby granted permission under all Bitstream propriety rights
to use, copy, modify, sublicense, sell, and redistribute the 4 Bitstream
Charter (r) Type 1 outline fonts and the 4 Courier Type 1 outline fonts
for any purpose and without restriction; provided, that this notice is
left intact on all copies of such fonts and that Bitstream's trademark
is acknowledged as shown below on all unmodified copies of the 4 Charter
Type 1 fonts.

BITSTREAM CHARTER is a registered trademark of Bitstream Inc.
```

### STIX

```
The STIX fonts distributed with matplotlib have been modified from
their canonical form.  They have been converted from OTF to TTF format
using Fontforge and this script:

  #!/usr/bin/env fontforge
  i=1
  while ( i<$argc )
    Open($argv[i])
    Generate($argv[i]:r + ".ttf")
    i = i+1
  endloop

The original STIX Font License begins below.

----------------------------------------------------------

STIX Font License

24 May 2010
```

(continues on next page)

(continued from previous page)

```
Copyright (c) 2001-2010 by the STI Pub Companies, consisting of the American
Institute of Physics, the American Chemical Society, the American Mathematical
Society, the American Physical Society, Elsevier, Inc., and The Institute of
Electrical and Electronic Engineers, Inc. (www.stixfonts.org), with Reserved
Font Name STIX Fonts, STIX Fonts (TM) is a  trademark of The Institute of
Electrical and Electronics Engineers, Inc.

Portions copyright (c) 1998-2003 by MicroPress, Inc. (www.micropress-inc.com),
with Reserved Font Name TM Math. To obtain additional mathematical fonts,␣
 ↪please
contact MicroPress, Inc., 68-30 Harrow Street, Forest Hills, NY 11375, USA,
Phone: (718) 575-1816.

Portions copyright (c) 1990 by Elsevier, Inc.

This Font Software is licensed under the SIL Open Font License, Version 1.1.
This license is copied below, and is also available with a FAQ at:
https://scripts.sil.org/OFL

-----------------------------------------------------------
SIL OPEN FONT LICENSE Version 1.1 - 26 February 2007
-----------------------------------------------------------

PREAMBLE
The goals of the Open Font License (OFL) are to stimulate worldwide
development of collaborative font projects, to support the font creation
efforts of academic and linguistic communities, and to provide a free and
open framework in which fonts may be shared and improved in partnership
with others.

The OFL allows the licensed fonts to be used, studied, modified and
redistributed freely as long as they are not sold by themselves. The
fonts, including any derivative works, can be bundled, embedded,
redistributed and/or sold with any software provided that any reserved
names are not used by derivative works. The fonts and derivatives,
however, cannot be released under any other type of license. The
requirement for fonts to remain under this license does not apply
to any document created using the fonts or their derivatives.

DEFINITIONS
"Font Software" refers to the set of files released by the Copyright
Holder(s) under this license and clearly marked as such. This may
include source files, build scripts and documentation.

"Reserved Font Name" refers to any names specified as such after the
copyright statement(s).

"Original Version" refers to the collection of Font Software components as
distributed by the Copyright Holder(s).

"Modified Version" refers to any derivative made by adding to, deleting,
```

(continues on next page)

(continued from previous page)

```
or substituting -- in part or in whole -- any of the components of the
Original Version, by changing formats or by porting the Font Software to a
new environment.

"Author" refers to any designer, engineer, programmer, technical
writer or other person who contributed to the Font Software.

PERMISSION & CONDITIONS
Permission is hereby granted, free of charge, to any person obtaining
a copy of the Font Software, to use, study, copy, merge, embed, modify,
redistribute, and sell modified and unmodified copies of the Font
Software, subject to the following conditions:

1) Neither the Font Software nor any of its individual components,
in Original or Modified Versions, may be sold by itself.

2) Original or Modified Versions of the Font Software may be bundled,
redistributed and/or sold with any software, provided that each copy
contains the above copyright notice and this license. These can be
included either as stand-alone text files, human-readable headers or
in the appropriate machine-readable metadata fields within text or
binary files as long as those fields can be easily viewed by the user.

3) No Modified Version of the Font Software may use the Reserved Font
Name(s) unless explicit written permission is granted by the corresponding
Copyright Holder. This restriction only applies to the primary font name as
presented to the users.

4) The name(s) of the Copyright Holder(s) or the Author(s) of the Font
Software shall not be used to promote, endorse or advertise any
Modified Version, except to acknowledge the contribution(s) of the
Copyright Holder(s) and the Author(s) or with their explicit written
permission.

5) The Font Software, modified or unmodified, in part or in whole,
must be distributed entirely under this license, and must not be
distributed under any other license. The requirement for fonts to
remain under this license does not apply to any document created
using the Font Software.

TERMINATION
This license becomes null and void if any of the above conditions are
not met.

DISCLAIMER
THE FONT SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT
OF COPYRIGHT, PATENT, TRADEMARK, OR OTHER RIGHT. IN NO EVENT SHALL THE
COPYRIGHT HOLDER BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
INCLUDING ANY GENERAL, SPECIAL, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL
DAMAGES, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
```

(continues on next page)

(continued from previous page)

```
FROM, OUT OF THE USE OR INABILITY TO USE THE FONT SOFTWARE OR FROM
OTHER DEALINGS IN THE FONT SOFTWARE.
```

## 11.6 Credits

Matplotlib was written by John D. Hunter, with contributions from an ever-increasing number of users and developers. The current lead developer is Thomas A. Caswell, who is assisted by many active developers. Please also see our instructions on *Citing Matplotlib*.

The following is a list of contributors extracted from the git revision control history of the project:

4over7, 816-8055, Aaron Boushley, Aashil Patel, AbdealiJK, Abhinav Sagar, Abhinuv Nitin Pitale, Acanthostega, Adam, Adam Ginsburg, Adam Gomaa, Adam Heck, Adam J. Stewart, Adam Ortiz, Adam Paszke, Adam Ruszkowski, Adam Williamson, Adrian Price-Whelan, Adrien Chardon, Adrien F. Vincent, Ahmet Bakan, Akshay Nair, Alan Bernstein, Alan Du, Alberto, Alejandro Dubrovsky, Aleksey Bilogur, Alex C. Szatmary, Alex Loew, Alex Rothberg, Alex Rudy, AlexCav, Alexander Buchkovsky, Alexander Harnisch, Alexander Rudy, Alexander Taylor, Alexei Colin, Alexis Bienvenüe, Ali Mehdi, Ali Uneri, Alistair Muldal, Allan Haldane, Allen Downey, Alon Hershenhorn, Alvaro Sanchez, Amit Aronovitch, Amy, Amy Roberts, AmyTeegarden, AndersonDaniel, Andras Deak, Andrea Bedini, Andreas Gustafsson, Andreas Hilboll, Andreas Mayer, Andreas Mueller, Andreas Wallner, Andrew Dawson, Andrew Merrill, Andrew Nelson, Andrew Straw, Andy Mastbaum, Andy Zhu, Ankur Dedania, Anthony Scopatz, Anton Akhmerov, Antony Lee, Anubhav Shrimal, Ao Liu (frankliuao), Ardie Orden, Arie, Ariel Hernán Curiale, Arnaud Gardelein, Arpad Horvath, Arthur Paulino, Arvind, Aseem Bansal, Ashley Whetter, Atharva Khare, Avinash Sharma, Ayappan P, BHT, BTWS, Bas van Schaik, Bastian Bechtold, Behram Mistree, Ben, Ben Cohen, Ben Gamari, Ben Keller, Ben Root, Benedikt Daurer, Benjamin Bengfort, Benjamin Berg, Benjamin Congdon, Benjamin Reedlunn, Bernhard M. Wiedemann, Bharat123rox, Bianca Gibson, Binglin Chang, Bingyao Liu, Björn Dahlgren, Blaise Thompson, Boaz Mohar, Bradley M. Froehle, Brandon Liu, Brendan Zhang, Brennan Magee, Brett Cannon, Brett Graham, Brian Mattern, Brian McLaughlin, Brigitta Sipocz, Bruno Beltran, Bruno Pagani, Bruno Zohreh, CJ Carey, Cameron Bates, Cameron Davidson-Pilon, Cameron Fackler, Carissa Brittain, Carl Michal, Carsten Schelp, Carwyn Pelley, Casey Webster, Casper van der Wel, Charles Moad, Charles Ruan, Chen Karako, Cho Yin Yong, Chris, Chris Barnes, Chris Beaumont, Chris G, Chris Holdgraf, Chris Zimmerman, Christer Jensen, Christian Brodbeck, Christian Brueffer, Christian Stade-Schuldt, Christoph Dann, Christoph Deil, Christoph Gohlke, Christoph Hoffmann, Christoph Pohl, Christoph Reiter, Christopher Bradshaw, Cimarron Mittelsteadt, Clemens Brunner, Cody Scot, Colin, Colin Carroll, Cong Ma, Conner R. Phillips, Corey Farwell, Craig Citro, Craig M, Craig Tenney, DaCoEx, Dakota Blair, Damian, Damon McDougall, Dan Hickstein, Dana, Daniel C. Marcu, Daniel Hyams, Daniel Laidig, Daniel O'Connor, DanielMatu, Daniele Nicolodi, Danny Hermes, Dara Adib, Darren Dale, DaveL17, David A, David Anderson, David Chudzicki, David Haberthür, David Huard, David Kaplan, David Kent, David Kua, David Stansby, David Trémouilles, Dean Malmgren, Deng Tian, Derek Kim, Derek Tropf, Devashish Deshpande, Diego Mora Cespedes, Dietmar Schwertberger, Dietrich Brunn, Divyam Madaan, Dmitry Lupyan, Dmitry Mottl, Dmitry Shachnev, Dominik Schmidt, DonaldSeo, Dora Fraeman Caswell, DoriekeMG, Dorota Jarecka, Doug Blank, Drew J. Sonne, Duncan Macleod, Dylan Evans, E. G. Patrick Bos, Edin Salkovic, Edoardo Pizzigoni, Egor Panfilov, Elan Ernest, Elena Glassman, Elias Pipping, Elijah Schutz, Elizabeth Seiver, Elliott Sales de Andrade, Elvis Stansvik, Emil Mikulic, Emlyn Price, Eric Dill, Eric Firing, Eric Larson, Eric Ma, Eric O. LEBIGOT (EOL), Eric Relson, Eric Wieser, Erik Bray, Erik M. Bray, Erin Pintozzi, Eugen Beck, Eugene Yurtsev, Evan Davey, Ezra Peisach, Fabian Kloosterman,

Fabian-Robert Stöter, Fabien Maussion, Fabio Zanini, FedeMiorelli, Federico Ariza, Felipe, Felix, Felix Kohlgrüber, Felix Yan, Fernando Perez, Filip Dimitrovski, Filipe Fernandes, Florencia Noriega, Florian Le Bourdais, Florian Rhiem, Francesco Montesano, Francis Colas, Franco Vaccari, Francoise Provencher, Frank Sauerburger, Frank Yu, François Magimel, Gabe, Gabriel Munteanu, Gal Avineri, Galen Lynch, Gauravjeet, Gaute Hope, Gazing, Gellule Xg, Geoffrey Spear, Geoffroy Billotey, Georg Raiser, Gerald Storer, Gina, Giovanni, Graeme Smecher, Graham Poulter, Greg Lucas, Gregory Ashton, Gregory R. Lee, Grillard, Grégory Lielens, Guillaume Gay, Guillermo Breto, Gustavo Braganca, Gustavo Goretkin, HHest, Hajoon Choi, Hakan Kucukdereli, Hanno Rein, Hans Dembinski, Hans Meine, Hans Moritz Günther, Harnesser, Harshal Prakash Patankar, Harshit Patni, Hassan Kibirige, Hastings Greer, Heath Henley, Heiko Oberdiek, Helder, Henning Pohl, Herbert Kruitbosch, Holger Peters, Hubert Holin, Hugo van Kemenade, Ian Hincks, Ian Thomas, Ida Hjorth, Ignas Anikevicius (gns_ank), Ildar Akhmetgaleev, Ilia Kurenkov, Ilya Flyamer, ImSoErgodic, ImportanceOfBeingErnest, Inception95, Ingo Fründ, Ioannis Filippidis, Isa Hassen, Isaac Schwabacher, Isaac Slavitt, Ismo Toijala, J Alammar, J. Goutin, Jaap Versteegh, Jack Kelly, Jacob McDonald, Jacobson Okoro, Jae-Joon Lee, Jaime Fernandez, Jake Lee, Jake Vanderplas, James A. Bednar, James Adams, James Pallister, James R. Evans, JamesMakela, Jamie Nunez, Jan S. (Milania1), Jan Schlüter, Jan Schulz, Jan-Philip Gehrcke, Jan-willem De Bleser, Jarrod Millman, Jascha Ulrich, Jason Grout, Jason King, Jason Liw Yan Chong, Jason Miller, Jason Neal, Jason Zheng, Javad, JayP16, Jean-Benoist Leger, Jeff Lutgen, Jeff Whitaker, Jeffrey Bingham, Jeffrey Hokanson @ Loki, JelsB, Jens Hedegaard Nielsen, Jeremy Fix, Jeremy O'Donoghue, Jeremy Thurgood, Jeroonk, Jessica B. Hamrick, Jiahao Chen, Jim Radford, Jochen Voss, Jody Klymak, Joe Kington, Joel B. Mohler, Joel Frederico, Joel Wanner, Johannes H. Jensen, Johannes Wienke, John Hoffman, John Hunter, John Vandenberg, Johnny Gill, JojoBoulix, Jon Haitz Legarreta Gorroño, Jonas Camillus Jeppesen, Jonathan Waltman, Jorge Moraleda, Jorrit Wronski, Joscha Reimer, Josef Heinen, Joseph Albert, Joseph Fox-Rabinovitz, Joseph Jon Booker, Joseph Martinot-Lagarde, Joshua Taillon, José Ricardo, Jouni K. Seppänen, Joy Bhalla, Juan Nunez-Iglesias, Juanjo Bazán, Julia Sprenger, Julian Mehne, Julian Taylor, Julian V. Modesto, JulianCienfuegos, Julien Lhermitte, Julien Schueller, Julien Woillez, Julien-Charles Lévesque, Jun Tan, Justin Cai, Jörg Dietrich, Kacper Kowalik (Xarthisius), Kai Muehlbauer, Kanchana Ranasinghe, Kanwar245, Katrin Leinweber, Katy Huff, Kayla Ngan, Keerysanth Sribaskaran, Ken McIvor, Kenneth Ma, Kevin Chan, Kevin Davies, Kevin Ji, Kevin Keating, Kevin Mader, Kevin Rose, Kexuan Sun, Kieran Ramos, Kimmo Palin, Kjartan Myrdal, Kjell Le, Klara Gerlei, Konrad Förstner, Konstantin Tretyakov, Kristen M. Thyng, Kyle Bridgemohansingh, Kyle Sunden, Kyler Brown, Lance Hepler, Laptop11_ASPP2016, Larry Bradley, Laurent Thomas, Lawrence D'Anna, Leeonadoh, Lennart Fricke, Leo Singer, Leon Loopik, Leon Yin, LevN0, Levi Kilcher, Liam Brannigan, Lion Krischer, Lionel Miller, Lodato Luciano, Lori J, Loïc Estève, Loïc Séguin-C, Luca Verginer, Luis Pedro Coelho, Luke Davis, Maarten Baert, Maciej Dems, Magnus Nord, Maik Riechert, Majid alDosari, Maksym P, Manan, Manan Kevadiya, Manish Devgan, Manuel GOACOLOU, Manuel Jung, Manuel Metz, Manuel Nuno Melo, Maoz Gelbart, Marat K, Marc Abramowitz, Marcel Martin, Marco Gorelli, Marco-Gorelli, Marcos Duarte, Marek Rudnicki, Marianne Corvellec, Marin Gilles, Mark Harfouche, Mark Wolf, Marko Baštovanović, Markus Roth, Markus Rothe, Martin Dengler, Martin Fitzpatrick, Martin Spacek, Martin Teichmann, Martin Thoma, Martin Ueding, Massimo Santini, Masud Rahman, Mathieu Duponchelle, Matt Giuca, Matt Hancock, Matt Klein, Matt Li, Matt Newville, Matt Shen, Matt Terry, Matthew Bell, Matthew Brett, Matthew Emmett, Matthias Bussonnier, Matthias Geier, Matthias Lüthi, Matthieu Caneill, MatthieuDartiailh, Matti Picus, Matěj Týč, Max Chen, Max Humber, Max Shinn, Maximilian Albert, Maximilian Maahn, Maximilian Nöthe, Maximilian Trescher, MeeseeksMachine, Mellissa Cross, Mher Kazandjian, Michael, Michael Droettboom, Michael Jancsy, Michael Sarahan, Michael Scott Cuthbert, Michael Seifert, Michael Welter, Michaël Defferrard, Michele Mastropietro, Michiel de Hoon, Michka Popoff, Mike Henninger, Mike Jarvis, Mike Kaufman, Mikhail Korobov, MinRK, Mingkai Dong, Minty Zhang, MirandaXM, Miriam Sierig, Mitar, Molly Rossow, Moritz Boehle, Mudit Surana, Muhammad Mehdi, MuhammadFarooq1234, Mykola Dvornik, Naoya Kanai, Nathan Goldbaum, Nathan Musoke, Nathaniel M. Beaver,

Neil, Neil Crighton, Nelle Varoquaux, Niall Robinson, Nic Eggert, Nicholas Devenish, Nick Forrington, Nick Garvey, Nick Papior, Nick Pope, Nick Semenkovich, Nico Schlömer, Nicolas Courtemanche, Nicolas P. Rougier, Nicolas Pinto, Nicolas Tessore, Nik Quibin, Nikita Kniazev, Niklas Koep, Nikolay Vyahhi, Nils Werner, Ninad Bhat, Norbert Nemec, Norman Fomferra, O. Castany, OceanWolf, Oleg Selivanov, Olga Botvinnik, Oliver Natt, Oliver Willekens, Olivier, Om Sitapara, Omar Chehab, Oriol Abril, Orso Meneghini, Osarumwense, Pankaj Pandey, Paramonov Andrey, Parfenov Sergey, Pascal Bugnion, Pastafarianist, Patrick Chen, Patrick Feiring, Patrick Marsh, Patrick Shriwise, PatrickFeiring, Paul, Paul Barret, Paul Ganssle, Paul Gierz, Paul Hobson, Paul Hoffman, Paul Ivanov, Paul J. Koprowski, Paul Kirow, Paul Romano, Paul Seyfert, Pauli Virtanen, Pavel Fedin, Pavol Juhas, Per Parker, Perry Greenfield, Pete Bachant, Pete Huang, Pete Peterson, Peter Iannucci, Peter Mackenzie-Helnwein, Peter Mortensen, Peter Schutt, Peter St. John, Peter Würtz, Petr Danecek, Phil Elson, Phil Ruffwind, Philippe Pinard, Pierre Haessig, Pierre Thibault, Pierre de Buyl, Pim Schellart, Piti Ongmongkolkul, Po, Pranav Garg, Przemysław Dąbek, Puneeth Chaganti, QiCuiHub, Qingpeng "Q.P." Zhang, RAKOTOARISON Herilalaina, Ram Rachum, Ramiro Gómez, Randy Olson, Raphael, Rasmus Diederichsen, Ratin_Kumar, Rebecca W Perry, Reinier Heeres, Remi Rampin, Ricardo Mendes, Riccardo Di Maio, Richard Gowers, Richard Hattersley, Richard Ji-Cathriner, Richard Trieu, Ricky, Rishikesh, Rob Harrigan, Robert Johansson, Robin Dunn, Robin Neatherway, Robin Wilson, Rohan Walker, Roland Wirth, Roman Yurchak, Ronald Hartley-Davies, RoryIAngus, Roy Smith, Rui Lopes, Russell Owen, RutgerK, Ryan, Ryan Blomberg, Ryan D'Souza, Ryan Dale, Ryan May, Ryan Morshead, Ryan Nelson, RyanPan, SBCV, Sairam Pillai, Saket Choudhary, Salganos, Salil Vanvari, Salinder Sidhu, Sam Vaughan, SamSchott, Sameer D'Costa, Samesh Lakhotia, Samson, Samuel St-Jean, Sander, Sandro Tosi, Scott Howard, Scott Lasley, Scott Lawrence, Scott Stevenson, Sean Farley, Sebastian Bullinger, Sebastian Pinnau, Sebastian Raschka, Sebastián Vanrell, Seraphim Alvanides, Sergey B Kirpichev, Sergey Kholodilov, Sergey Koposov, Seunghoon Park, Siddhesh Poyarekar, Sidharth Bansal, Silviu Tantos, Simon Cross, Simon Gibbons, Simon Legner, Skelpdar, Skipper Seabold, Slav Basharov, Snowhite, SojiroFukuda, Sourav Singh, Spencer McIntyre, Stanley, Simon, Stefan Lehmann, Stefan Mitic, Stefan Pfenninger, Stefan van der Walt, Stefano Rivera, Stephan Erb, Stephane Raynaud, Stephen Horst, Stephen-Chilcote, Sterling Smith, Steve Chaplin, Steve Dower, Steven G. Johnson, Steven Munn, Steven Silvester, Steven Tilley, Stuart Mumford, Tadeo Corradi, Taehoon Lee, Takafumi Arakaki, Takeshi Kanmae, Tamas Gal, Tanuj, Taras Kuzyo, Ted Drain, Ted Petrou, Terence Honles, Terrence J. Katzenbaer, Terrence Katzenbaer, The Gitter Badger, Thein Oo, Thomas A Caswell, Thomas Hisch, Thomas Kluyver, Thomas Lake, Thomas Levine, Thomas Mansencal, Thomas Robitaille, Thomas Spura, Thomas VINCENT, Thorsten Liebig, Tian Xia, Till Hoffmann, Till Stensitzki, Tim Hoffmann, Timo Vanwynsberghe, Tobia De Koninck, Tobias Froehlich, Tobias Hoppe, Tobias Megies, Todd Jennings, Todd Miller, Tom, Tom Augspurger, Tom Dupré la Tour, Tom Flannaghan, Tomas Kazmar, Tony S Yu, Tor Colvin, Travis Oliphant, Trevor Bekolay, Trish Gillett-Kawamoto, Truong Pham, Tuan Dung Tran, Tyler Makaro, Tyrone Xiong, Ulrich Dobramysl, Umair Idris, V. Armando Solé, V. R, Vadim Markovtsev, Valentin Haenel, Valentin Schmidt, Vedant Nanda, Venkada, Vidur Satija, Viktor Kerkez, Vincent L.M. Mazoyer, Viraj Mohile, Vitaly Buka, Vlad Seghete, Víctor Terrón, Víctor Zabalza, WANG Aiyong, Warren Weckesser, Wen Li, Wendell Smith, Werner F Bruhin, Wes Campaigne, Wieland Hoffmann, Will Handley, Will Silva, William Granados, William Mallard, William Manley, Wouter Overmeire, Xiaowen Tang, Xufeng Wang, Yann Tambouret, Yao-Yuan Mao, Yaron de Leeuw, Yu Feng, Yue Zhihan, Yunfei Yang, Yuri D'Elia, Yuval Langer, Yuxin Wu, Yuya, Zac Hatfield-Dodds, Zach Pincus, Zair Mubashar, Zbigniew Jędrzejewski-Szmek, Zhili (Jerry) Pan, Zulko, ahed87, akrherz, alcinos, alex, alvarosg, andrzejnovak, aneda, anykraus, aparamon, apodemus, arokem, as691454, aseagram, ash13, aszilagyi, azure-pipelines[bot], bblay, bduick, bev-a-tron, blackw1ng, blah blah, brut, btang02, buefox, burrbull, butterw, cammil, captainwhippet, cclauss, ch3rn0v, chadawagner, chaoyi1, chebee7i, chelseatroy, chuanzhu xu, cknd, cldssty, clintval, dabana, dahlbaek, danielballan, daronjp, davidovitch, daydreamt, deeenes, deepyaman, djdt, dlmccaffrey, domspad, donald, donchanee, drevicko, e-q, elpres, endolith, esvhd, et2010, fardal, ffteja, fgb, fibersnet, fourpoints, fredrik-1, frenchwr, fuzzythecat, fvgoto, gcallah, gitj, gluap,

gnaggnoyil, goir, goldstarwebs, greg-roper, gregorybchris, gwin-zegal, hannah, helmiriawan, henryhu123, hugadams, ilivni, insertroar, itziakos, jacob-on-github, jb-leger, jbbrokaw, jbhopkins, jdollichon, jerrylui803, jess, jfbu, jhelie, jli, joaonsg, joelostblom, jonchar, juan.gonzalez, kcrisman, keithbriggs, kelsiegr, khyox, kikocorreoso, klaus, klonuo, kolibril13, kramer65, krishna katyal, ksafran, kshramt, lboogaard, legitz7, lepuchi, lichri12, limtaesu, lspvic, luftek, luz.paz, lzkelley, mamrehn, marky, masamson, mbyt, mcelrath, mcquin, mdipierro, mikhailov, miquelastein, mitch, mlub, mobando, mromanie, muahah, myyc, nathan78906, navdeep rana, nbrunett, nemanja, neok-m4700, nepix32, nickystringer, njwhite, nmartensen, nwin, ob, pdubcali, pibion, pkienzle, productivememberofsociety666, profholzer, pupssman, rahiel, ranjanm, rebot, rhoef, rsnape, ruin, rvhbooth, s0vereign, s9w, saksmito, scls19fr, scott-vsi, sdementen, serv-inc, settheory, sfroid, shaunwbell, simon-kraeusel, simonpf, sindunuragarp, smheidrich, sohero, spiessbuerger, stahlous, stone, stonebig, switham, sxntxn, syngron, teresy, thoo, thuvejan, tmdavison, tomoemon, tonyyli, torfbolt, u55, ugurthemaster, ultra-andy, vab9, vbr, vishalBindal, vraelvrangr, watkinrt, woclass, xbtsw, xuanyuansen, y1thof, yeo, zhangeugenia, zhoubecky, Élie Gouzien, Андрей Парамонов

Some earlier contributors not included above are (with apologies to any we have missed):

Charles Twardy, Gary Ruben, John Gill, David Moore, Paul Barrett, Jared Wahlstrand, Jim Benson, Paul Mcguire, Andrew Dalke, Nadia Dencheva, Baptiste Carvello, Sigve Tjoraand, Ted Drain, James Amundson, Daishi Harada, Nicolas Young, Paul Kienzle, John Porter, and Jonathon Taylor.

Thanks to Tony Yu for the original logo design.

We also thank all who have reported bugs, commented on proposed changes, or otherwise contributed to Matplotlib's development and usefulness.

# Part VII

# Appendices

# BIBLIOGRAPHY


[Ware] http://ccom.unh.edu/sites/default/files/publications/Ware_1988_CGA_Color_sequences_univariate_maps.pdf

[Moreland] http://www.kennethmoreland.com/color-maps/ColorMapsExpanded.pdf

[list-colormaps] https://gist.github.com/endolith/2719900#id7

[mycarta-banding] https://mycarta.wordpress.com/2012/10/14/the-rainbow-is-deadlong-live-the-rainbow-part-4-cie-lab-he

[mycarta-jet] https://mycarta.wordpress.com/2012/10/06/the-rainbow-is-deadlong-live-the-rainbow-part-3/

[kovesi-colormaps] https://arxiv.org/abs/1509.03700

[bw] https://tannerhelland.com/3643/grayscale-image-algorithm-vb6/

[colorblindness] http://www.color-blindness.com/

[IBM] https://doi.org/10.1109/VISUAL.1995.480803

[turbo] https://ai.googleblog.com/2019/08/turbo-improved-rainbow-colormap-for.html

[1] Michel Bernadou, Kamal Hassan, "Basis functions for general Hsieh-Clough-Tocher triangles, complete or reduced.", International Journal for Numerical Methods in Engineering, 17(5):784 - 789. 2.01.

[2] C.T. Kelley, "Iterative Methods for Optimization".

# PYTHON MODULE INDEX

## m


matplotlib, **??**
matplotlib._afm, 1829
matplotlib._api, 3802
matplotlib._api.deprecation, 3806
matplotlib._docstring, 2787
matplotlib._enums, 3810
matplotlib._tight_bbox, 3716
matplotlib._tight_layout, 3716
matplotlib._type1font, 3766
matplotlib.animation, 1832
matplotlib.artist, 1867
matplotlib.axes, 1903
matplotlib.axis, 2218
matplotlib.backend_bases, 2253
matplotlib.backend_managers, 2284
matplotlib.backend_tools, 2288
matplotlib.backends, 2303
matplotlib.backends.backend_agg, 2308
matplotlib.backends.backend_cairo, 2315
matplotlib.backends.backend_gtk3, 2320
matplotlib.backends.backend_gtk3agg, 2320
matplotlib.backends.backend_gtk3cairo, 2320
matplotlib.backends.backend_gtk4, 2320
matplotlib.backends.backend_gtk4agg, 2320
matplotlib.backends.backend_gtk4cairo, 2320
matplotlib.backends.backend_mixed, 2303
matplotlib.backends.backend_nbagg, 2321
matplotlib.backends.backend_pdf, 2322
matplotlib.backends.backend_pgf, 2335
matplotlib.backends.backend_ps, 2341
matplotlib.backends.backend_qt, 2350
matplotlib.backends.backend_qt5agg, 2350
matplotlib.backends.backend_qt5cairo, 2350
matplotlib.backends.backend_qtagg, 2350
matplotlib.backends.backend_qtcairo, 2350
matplotlib.backends.backend_svg, 2351
matplotlib.backends.backend_template, 2304
matplotlib.backends.backend_tkagg, 2357
matplotlib.backends.backend_tkcairo, 2357
matplotlib.backends.backend_webagg, 2362
matplotlib.backends.backend_webagg_core, 2358
matplotlib.backends.backend_wx, 2364
matplotlib.backends.backend_wxagg, 2364
matplotlib.backends.backend_wxcairo, 2364
matplotlib.backends.qt_compat, 2350
matplotlib.backends.registry, 2346
matplotlib.bezier, 2364
matplotlib.category, 2368
matplotlib.cbook, 2371

matplotlib.cm, 2384
matplotlib.collections, 2388
matplotlib.colorbar, 2704
matplotlib.colors, 2712
matplotlib.container, 2753
matplotlib.contour, 2756
matplotlib.dates, 2768
matplotlib.dviread, 2788
matplotlib.figure, 2792
matplotlib.font_manager, 2941
matplotlib.ft2font, 2951
matplotlib.gridspec, 2956
matplotlib.hatch, 2966
matplotlib.image, 2967
matplotlib.layout_engine, 2985
matplotlib.legend, 2989
matplotlib.legend_handler, 3003
matplotlib.lines, 3015
matplotlib.markers, 3038
matplotlib.mathtext, 3043
matplotlib.mlab, 3046
matplotlib.offsetbox, 3064
matplotlib.patches, 3095
matplotlib.path, 3194
matplotlib.patheffects, 3203
matplotlib.projections, 3526
matplotlib.projections.geo, 3556
matplotlib.projections.polar, 3527
matplotlib.pyplot, 3211
matplotlib.quiver, 3588
matplotlib.rcsetup, 3605
matplotlib.sankey, 3609
matplotlib.scale, 3614
matplotlib.sphinxext.figmpl_directive, 3635
matplotlib.sphinxext.mathmpl, 3630
matplotlib.sphinxext.plot_directive, 3631
matplotlib.spines, 3637
matplotlib.style, 3643
matplotlib.table, 3645
matplotlib.testing, 3657
matplotlib.testing.compare, 3659
matplotlib.testing.decorators, 3660
matplotlib.testing.exceptions, 3662
matplotlib.texmanager, 3688
matplotlib.text, 3662
matplotlib.ticker, 3689
matplotlib.transforms, 3718
matplotlib.tri, 3755
matplotlib.units, 3769
matplotlib.widgets, 3772
mpl_toolkits.axes_grid1, 3910
mpl_toolkits.axes_grid1.anchored_artists, 3911
mpl_toolkits.axes_grid1.axes_divider, 3930
mpl_toolkits.axes_grid1.axes_grid, 3940
mpl_toolkits.axes_grid1.axes_rgb, 3945
mpl_toolkits.axes_grid1.axes_size, 3948
mpl_toolkits.axes_grid1.inset_locator, 3953
mpl_toolkits.axes_grid1.mpl_axes, 3977
mpl_toolkits.axes_grid1.parasite_axes, 3985
mpl_toolkits.axisartist, 3989
mpl_toolkits.axisartist.angle_helper, 3990
mpl_toolkits.axisartist.axes_divider, 3997
mpl_toolkits.axisartist.axes_grid, 3997
mpl_toolkits.axisartist.axes_rgb, 4001
mpl_toolkits.axisartist.axis_artist, 4002
mpl_toolkits.axisartist.axisline_style, 4025
mpl_toolkits.axisartist.axislines, 4027
mpl_toolkits.axisartist.floating_axes, 4041
mpl_toolkits.axisartist.grid_finder, 4045
mpl_toolkits.axisartist.grid_helper_curvelinea 4049
mpl_toolkits.axisartist.parasite_axes, 4051
mpl_toolkits.mplot3d, 3873
mpl_toolkits.mplot3d.art3d, 3879
mpl_toolkits.mplot3d.axes3d, 3874
mpl_toolkits.mplot3d.axis3d, 3874

mpl_toolkits.mplot3d.proj3d, 3907

## p

pylab, 4051

# INDEX

## Non-alphabetical


__add__() (*matplotlib.transforms.Transform method*), 3745
__call__() (*matplotlib.colors.AsinhNorm method*), 2719
__call__() (*matplotlib.colors.BoundaryNorm method*), 2720
__call__() (*matplotlib.colors.Colormap method*), 2732
__call__() (*matplotlib.colors.FuncNorm method*), 2723
__call__() (*matplotlib.colors.LogNorm method*), 2725
__call__() (*matplotlib.colors.NoNorm method*), 2717
__call__() (*matplotlib.colors.Normalize method*), 2715
__call__() (*matplotlib.colors.PowerNorm method*), 2727
__call__() (*matplotlib.colors.SymLogNorm method*), 2729
__call__() (*matplotlib.colors.TwoSlopeNorm method*), 2731
__call__() (*matplotlib.patches.BoxStyle.Circle method*), 3119
__call__() (*matplotlib.patches.BoxStyle.DArrow method*), 3119
__call__() (*matplotlib.patches.BoxStyle.Ellipse method*), 3119
__call__() (*matplotlib.patches.BoxStyle.LArrow method*), 3120
__call__() (*matplotlib.patches.BoxStyle.RArrow method*), 3120
__call__() (*matplotlib.patches.BoxStyle.Round method*), 3120
__call__() (*matplotlib.patches.BoxStyle.Round4 method*), 3121
__call__() (*matplotlib.patches.BoxStyle.Roundtooth method*), 3121
__call__() (*matplotlib.patches.BoxStyle.Sawtooth method*), 3121
__call__() (*matplotlib.patches.BoxStyle.Square method*), 3122
__call__() (*mpl_toolkits.axes_grid1.axes_divider.AxesLocator method*), 3932
__call__() (*mpl_toolkits.axes_grid1.inset_locator.AnchoredLocatorBase method*), 3954
__call__() (*mpl_toolkits.axes_grid1.inset_locator.InsetPosition method*), 3969
__call__() (*mpl_toolkits.axes_grid1.mpl_axes.Axes.AxisDict method*), 3979
__call__() (*mpl_toolkits.axes_grid1.mpl_axes.SimpleChainedObjects method*), 3985
__call__() (*mpl_toolkits.axisartist.angle_helper.ExtremeFinderCycle method*), 3992
__call__() (*mpl_toolkits.axisartist.angle_helper.FormatterDMS method*), 3992
__call__() (*mpl_toolkits.axisartist.angle_helper.FormatterHMS method*), 3993
__call__() (*mpl_toolkits.axisartist.angle_helper.LocatorD method*), 3994
__call__() (*mpl_toolkits.axisartist.angle_helper.LocatorDM method*), 3995
__call__() (*mpl_toolkits.axisartist.angle_helper.LocatorDMS method*), 3995
__call__() (*mpl_toolkits.axisartist.angle_helper.LocatorH method*), 3995
__call__() (*mpl_toolkits.axisartist.angle_helper.LocatorHM method*), 3995
__call__() (*mpl_toolkits.axisartist.angle_helper.LocatorHMS method*), 3996
__call__() (*mpl_toolkits.axisartist.axislines.Axes method*), 4030
__call__() (*mpl_toolkits.axisartist.floating_axes.ExtremeFinderFixed method*), 4041
__call__() (*mpl_toolkits.axisartist.grid_finder.DictFormatter method*), 4045
__call__() (*mpl_toolkits.axisartist.grid_finder.ExtremeFinderSimple method*), 4046
__call__() (*mpl_toolkits.axisartist.grid_finder.FixedLocator method*), 4046
__call__() (*mpl_toolkits.axisartist.grid_finder.FormatterPrettyPrint method*), 4047
__call__()

(*mpl_toolkits.axisartist.grid_finder.MaxNLocator method*), 4048
__init__() (*matplotlib.animation.AbstractMovieWriter method*), 1859
__init__() (*matplotlib.animation.Animation method*), 1833
__init__() (*matplotlib.animation.ArtistAnimation method*), 1840
__init__() (*matplotlib.animation.FFMpegBase method*), 1866
__init__() (*matplotlib.animation.FFMpegFileWriter method*), 1853
__init__() (*matplotlib.animation.FFMpegWriter method*), 1849
__init__() (*matplotlib.animation.FileMovieWriter method*), 1864
__init__() (*matplotlib.animation.FuncAnimation method*), 1839
__init__() (*matplotlib.animation.HTMLWriter method*), 1846
__init__() (*matplotlib.animation.ImageMagickBase method*), 1867
__init__() (*matplotlib.animation.ImageMagickFileWriter method*), 1854
__init__() (*matplotlib.animation.ImageMagickWriter method*), 1851
__init__() (*matplotlib.animation.MovieWriter method*), 1861
__init__() (*matplotlib.animation.MovieWriterRegistry method*), 1857
__init__() (*matplotlib.animation.PillowWriter method*), 1844
__init__() (*matplotlib.animation.TimedAnimation method*), 1856
__sub__() (*matplotlib.transforms.Transform method*), 3746

## A

AbstractMovieWriter (*class in matplotlib.animation*), 1859
AbstractPathEffect (*class in matplotlib.patheffects*), 3203
acorr() (*in module matplotlib.pyplot*), 3314
acorr() (*matplotlib.axes.Axes method*), 1975
active (*matplotlib.widgets.Widget property*), 3802
active_pane() (*mpl_toolkits.mplot3d.axis3d.Axis method*), 3874
active_toggle (*matplotlib.backend_managers.ToolManager property*), 2284
activecolor (*matplotlib.widgets.RadioButtons property*), 3786
Add (*class in mpl_toolkits.axes_grid1.axes_size*), 3949
add() (*matplotlib.sankey.Sankey method*), 3611
add_artist() (*matplotlib.axes.Axes method*), 2185
add_artist() (*matplotlib.figure.Figure method*), 2857
add_artist() (*matplotlib.figure.FigureBase method*), 2904
add_artist() (*matplotlib.figure.SubFigure method*), 2900
add_artist() (*matplotlib.offsetbox.AuxTransformBox method*), 3074
add_artist() (*matplotlib.offsetbox.DrawingArea method*), 3077
add_artist() (*mpl_toolkits.axes_grid1.axes_size.MaxExtent method*), 3951
add_auto_adjustable_area() (*mpl_toolkits.axes_grid1.axes_divider.Divider method*), 3933
add_axes() (*matplotlib.figure.Figure method*), 2796
add_axes() (*matplotlib.figure.FigureBase method*), 2904
add_axes() (*matplotlib.figure.SubFigure method*), 2867
add_axobserver() (*matplotlib.figure.Figure method*), 2854
add_callback() (*matplotlib.artist.Artist method*), 1869
add_callback() (*matplotlib.axes.Axes method*), 2196
add_callback() (*matplotlib.backend_bases.TimerBase method*), 2280
add_callback() (*matplotlib.collections.AsteriskPolygonCollection method*), 2389
add_callback() (*matplotlib.collections.CircleCollection method*), 2411
add_callback() (*matplotlib.collections.Collection method*), 2434
add_callback() (*matplotlib.collections.EllipseCollection method*), 2456
add_callback() (*matplotlib.collections.EventCollection method*), 2478
add_callback() (*matplotlib.collections.LineCollection method*), 2502
add_callback() (*matplotlib.collections.PatchCollection method*), 2524
add_callback() (*matplotlib.collections.PathCollection method*), 2545
add_callback() (*matplotlib.collections.PolyCollection method*), 2568
add_callback() (*matplotlib.collections.PolyQuadMesh method*), 2591
add_callback() (*matplotlib.collections.QuadMesh method*), 2615
add_callback() (*matplotlib.collections.RegularPolyCollection method*), 2637
add_callback() (*matplotlib.collections.StarPolygonCollection method*), 2659
add_callback() (*matplotlib.collections.TriMesh method*), 2682
add_callback() (*matplotlib.container.Container method*), 2753
add_cell() (*matplotlib.table.Table method*), 3651
add_child_axes() (*matplotlib.axes.Axes method*), 2186
add_collection() (*matplotlib.axes.Axes method*), 2186
add_collection3d() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3863
add_container() (*matplotlib.axes.Axes method*), 2186
add_contour_set() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*),

3872
add_contourf_set() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3872
add_figure() (*matplotlib.backend_tools.ToolViewsPositions method*), 2298
add_gridspec() (*matplotlib.figure.Figure method*), 2811
add_gridspec() (*matplotlib.figure.FigureBase method*), 2907
add_gridspec() (*matplotlib.figure.SubFigure method*), 2878
add_image() (*matplotlib.axes.Axes method*), 2186
add_label() (*matplotlib.contour.ContourLabeler method*), 2756
add_label_clabeltext() (*matplotlib.contour.ContourLabeler method*), 2756
add_label_near() (*matplotlib.contour.ContourLabeler method*), 2756
add_line() (*matplotlib.axes.Axes method*), 2187
add_lines() (*matplotlib.colorbar.Colorbar method*), 2706
add_patch() (*matplotlib.axes.Axes method*), 2187
add_positions() (*matplotlib.collections.EventCollection method*), 2479
add_subfigure() (*matplotlib.figure.Figure method*), 2814
add_subfigure() (*matplotlib.figure.FigureBase method*), 2908
add_subfigure() (*matplotlib.figure.SubFigure method*), 2879
add_subplot() (*matplotlib.figure.Figure method*), 2800
add_subplot() (*matplotlib.figure.FigureBase method*), 2908
add_subplot() (*matplotlib.figure.SubFigure method*), 2870
add_table() (*matplotlib.axes.Axes method*), 2189
add_tool() (*matplotlib.backend_bases.ToolContainerBase method*), 2281
add_tool() (*matplotlib.backend_managers.ToolManager method*), 2284
add_toolitem() (*matplotlib.backend_bases.ToolContainerBase method*), 2281
add_tools_to_container() (*in module matplotlib.backend_tools*), 2300
add_tools_to_manager() (*in module matplotlib.backend_tools*), 2301
add_web_socket() (*matplotlib.backends.backend_webagg_core.FigureManagerWebAgg method*), 2360
addfont() (*matplotlib.font_manager.FontManager method*), 2942
addGouraudTriangles() (*matplotlib.backends.backend_pdf.PdfFile method*), 2326
adir (*mpl_toolkits.mplot3d.axis3d.Axis property*), 3874
adjust_axes_lim() (*mpl_toolkits.axisartist.floating_axes.FloatingAxesBase method*), 4042
adjust_bbox() (*in module matplotlib._tight_bbox*), 3716
adjust_compatible (*matplotlib.layout_engine.ConstrainedLayoutEngine property*), 2986
adjust_compatible (*matplotlib.layout_engine.LayoutEngine property*), 2987
adjust_compatible (*matplotlib.layout_engine.PlaceHolderLayoutEngine property*), 2988
adjust_compatible (*matplotlib.layout_engine.TightLayoutEngine property*), 2988
adjust_drawing_area() (*matplotlib.legend_handler.HandlerBase method*), 3003
Affine2D (*class in matplotlib.transforms*), 3720
Affine2DBase (*class in matplotlib.transforms*), 3721
AffineBase (*class in matplotlib.transforms*), 3723
AffineDeltaTransform (*class in matplotlib.transforms*), 3725
AFM (*class in matplotlib._afm*), 1830
afmFontProperty() (*in module matplotlib.font_manager*), 2948
AitoffAxes (*class in matplotlib.projections.geo*), 3556
AitoffAxes.AitoffTransform (*class in matplotlib.projections.geo*), 3559
AitoffAxes.InvertedAitoffTransform (*class in matplotlib.projections.geo*), 3559
aliased_name() (*matplotlib.artist.ArtistInspector method*), 1901
aliased_name_rest() (*matplotlib.artist.ArtistInspector method*), 1901
align_labels() (*matplotlib.figure.Figure method*), 2837
align_labels() (*matplotlib.figure.FigureBase method*), 2911
align_labels() (*matplotlib.figure.SubFigure method*), 2897
align_titles() (*matplotlib.figure.Figure method*), 2840
align_titles() (*matplotlib.figure.FigureBase method*), 2912
align_titles() (*matplotlib.figure.SubFigure method*), 2899
align_xlabels() (*matplotlib.figure.Figure method*), 2838
align_xlabels() (*matplotlib.figure.FigureBase method*), 2912
align_xlabels() (*matplotlib.figure.SubFigure method*), 2897
align_ylabels() (*matplotlib.figure.Figure method*), 2839
align_ylabels() (*matplotlib.figure.FigureBase method*), 2913
align_ylabels() (*matplotlib.figure.SubFigure method*), 2898
allkinds (*matplotlib.contour.ContourSet property*), 2761
allow_rasterization() (*in module matplotlib.artist*), 1897
allsegs (*matplotlib.contour.ContourSet property*), 2761
alpha (*matplotlib.contour.ContourSet property*), 2761
alpha_cmd() (*matplotlib.backends.backend_pdf.GraphicsContextPdf*

*method*), 2323
alphaState() (*matplotlib.backends.backend_pdf.PdfFile method*), 2327
anchored() (*matplotlib.transforms.BboxBase method*), 3731
AnchoredAuxTransformBox (*class in mpl_toolkits.axes_grid1.anchored_artists*), 3912
AnchoredDirectionArrows (*class in mpl_toolkits.axes_grid1.anchored_artists*), 3915
AnchoredDrawingArea (*class in mpl_toolkits.axes_grid1.anchored_artists*), 3920
AnchoredEllipse (*class in mpl_toolkits.axes_grid1.anchored_artists*), 3923
AnchoredLocatorBase (*class in mpl_toolkits.axes_grid1.inset_locator*), 3953
AnchoredOffsetbox (*class in matplotlib.offsetbox*), 3065
AnchoredSizeBar (*class in mpl_toolkits.axes_grid1.anchored_artists*), 3925
AnchoredSizeLocator (*class in mpl_toolkits.axes_grid1.inset_locator*), 3955
AnchoredText (*class in matplotlib.offsetbox*), 3068
AnchoredZoomLocator (*class in mpl_toolkits.axes_grid1.inset_locator*), 3958
angle (*matplotlib.patches.Annulus property*), 3098
angle (*matplotlib.patches.Ellipse property*), 3136
angle_spectrum() (*in module matplotlib.mlab*), 3048
angle_spectrum() (*in module matplotlib.pyplot*), 3316
angle_spectrum() (*matplotlib.axes.Axes method*), 1977
Animation (*class in matplotlib.animation*), 1833
anncoords (*matplotlib.offsetbox.AnnotationBbox property*), 3071
anncoords (*matplotlib.text.Annotation property*), 3681
annotate() (*in module matplotlib.pyplot*), 3407
annotate() (*matplotlib.axes.Axes method*), 2073
Annotation (*class in matplotlib.text*), 3678
AnnotationBbox (*class in matplotlib.offsetbox*), 3069
Annulus (*class in matplotlib.patches*), 3097
antialiased (*matplotlib.contour.ContourSet property*), 2761
append_axes() (*mpl_toolkits.axes_grid1.axes_divider.AxesDivider method*), 3930
append_positions() (*matplotlib.collections.EventCollection method*), 2479
append_size() (*mpl_toolkits.axes_grid1.axes_divider.Divider method*), 3933
apply_aspect() (*matplotlib.axes.Axes method*), 2163
apply_aspect() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3858
Arc (*class in matplotlib.patches*), 3101
arc() (*matplotlib.path.Path class method*), 3196
arc_spine() (*matplotlib.spines.Spine class method*), 3638
Arrow (*class in matplotlib.patches*), 3105
arrow (*matplotlib.patches.ArrowStyle.BarAB attribute*), 3110
arrow (*matplotlib.patches.ArrowStyle.BracketA attribute*), 3110
arrow (*matplotlib.patches.ArrowStyle.BracketAB attribute*), 3111
arrow (*matplotlib.patches.ArrowStyle.BracketB attribute*), 3111
arrow (*matplotlib.patches.ArrowStyle.BracketCurve attribute*), 3111
arrow (*matplotlib.patches.ArrowStyle.CurveA attribute*), 3113
arrow (*matplotlib.patches.ArrowStyle.CurveAB attribute*), 3113
arrow (*matplotlib.patches.ArrowStyle.CurveB attribute*), 3114
arrow (*matplotlib.patches.ArrowStyle.CurveBracket attribute*), 3114
arrow (*matplotlib.patches.ArrowStyle.CurveFilledA attribute*), 3115
arrow (*matplotlib.patches.ArrowStyle.CurveFilledAB attribute*), 3115
arrow (*matplotlib.patches.ArrowStyle.CurveFilledB attribute*), 3116
arrow() (*in module matplotlib.pyplot*), 3419
arrow() (*matplotlib.axes.Axes method*), 2084
ArrowAxisClass (*mpl_toolkits.axisartist.axisline_style.AxislineStyle.FilledArrow attribute*), 4026
ArrowAxisClass (*mpl_toolkits.axisartist.axisline_style.AxislineStyle.SimpleArrow attribute*), 4026
ArrowStyle (*class in matplotlib.patches*), 3108
ArrowStyle.BarAB (*class in matplotlib.patches*), 3110
ArrowStyle.BracketA (*class in matplotlib.patches*), 3110
ArrowStyle.BracketAB (*class in matplotlib.patches*), 3110
ArrowStyle.BracketB (*class in matplotlib.patches*), 3111
ArrowStyle.BracketCurve (*class in matplotlib.patches*), 3111
ArrowStyle.Curve (*class in matplotlib.patches*), 3112
ArrowStyle.CurveA (*class in matplotlib.patches*), 3112
ArrowStyle.CurveAB (*class in matplotlib.patches*), 3113
ArrowStyle.CurveB (*class in matplotlib.patches*), 3113
ArrowStyle.CurveBracket (*class in matplotlib.patches*), 3114
ArrowStyle.CurveFilledA (*class in matplotlib.patches*), 3114
ArrowStyle.CurveFilledAB (*class in matplotlib.patches*), 3115
ArrowStyle.CurveFilledB (*class in matplotlib.patches*), 3115
ArrowStyle.Fancy (*class in matplotlib.patches*), 3116
ArrowStyle.Simple (*class in matplotlib.patches*), 3117
ArrowStyle.Wedge (*class in matplotlib.patches*), 3117
Artist (*class in matplotlib.artist*), 1869
ArtistAnimation (*class in matplotlib.animation*), 1839
ArtistInspector (*class in matplotlib.artist*), 1901
ArtistList (*class in matplotlib.axes.Axes*), 2217
artists (*matplotlib.widgets.ToolHandles property*), 3800
artists (*matplotlib.widgets.ToolLineHandles property*), 3801
ascender (*matplotlib.ft2font.FT2Font attribute*), 2953
AsinhLocator (*class in matplotlib.ticker*), 3692
AsinhNorm (*class in matplotlib.colors*), 2718
AsinhScale (*class in matplotlib.scale*), 3614
AsinhTransform (*class in matplotlib.scale*), 3615

AsteriskPolygonCollection (*class in matplotlib.collections*), 2388
attach_note() (*matplotlib.backends.backend_pdf.PdfPages method*), 2329
AttributeCopier (*class in mpl_toolkits.axisartist.axis_artist*), 4004
auto_scale_xyz() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3856
auto_set_column_width() (*matplotlib.table.Table method*), 3651
auto_set_font_size() (*matplotlib.table.Cell method*), 3646
auto_set_font_size() (*matplotlib.table.Table method*), 3652
auto_tick_multipliers (*matplotlib.scale.AsinhScale attribute*), 3615
AutoDateFormatter (*class in matplotlib.dates*), 2771
AutoDateLocator (*class in matplotlib.dates*), 2772
autofmt_xdate() (*matplotlib.figure.Figure method*), 2841
autofmt_xdate() (*matplotlib.figure.FigureBase method*), 2914
AutoLocator (*class in matplotlib.ticker*), 3693
AutoMinorLocator (*class in matplotlib.ticker*), 3693
autoscale() (*in module matplotlib.pyplot*), 3449
autoscale() (*matplotlib.axes.Axes method*), 2159
autoscale() (*matplotlib.cm.ScalarMappable method*), 2386
autoscale() (*matplotlib.collections.AsteriskPolygonCollection method*), 2390
autoscale() (*matplotlib.collections.CircleCollection method*), 2411
autoscale() (*matplotlib.collections.Collection method*), 2435
autoscale() (*matplotlib.collections.EllipseCollection method*), 2456
autoscale() (*matplotlib.collections.EventCollection method*), 2479
autoscale() (*matplotlib.collections.LineCollection method*), 2502
autoscale() (*matplotlib.collections.PatchCollection method*), 2524
autoscale() (*matplotlib.collections.PathCollection method*), 2545
autoscale() (*matplotlib.collections.PolyCollection method*), 2568
autoscale() (*matplotlib.collections.PolyQuadMesh method*), 2591
autoscale() (*matplotlib.collections.QuadMesh method*), 2616
autoscale() (*matplotlib.collections.RegularPolyCollection method*), 2638
autoscale() (*matplotlib.collections.StarPolygonCollection method*), 2660
autoscale() (*matplotlib.collections.TriMesh method*), 2683
autoscale() (*matplotlib.colors.CenteredNorm method*), 2722
autoscale() (*matplotlib.colors.Normalize method*), 2715
autoscale() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3855
autoscale_None() (*matplotlib.cm.ScalarMappable method*), 2386
autoscale_None() (*matplotlib.collections.AsteriskPolygonCollection method*), 2390
autoscale_None() (*matplotlib.collections.CircleCollection method*), 2411
autoscale_None() (*matplotlib.collections.Collection method*), 2435
autoscale_None() (*matplotlib.collections.EllipseCollection method*), 2456
autoscale_None() (*matplotlib.collections.EventCollection method*), 2479
autoscale_None() (*matplotlib.collections.LineCollection method*), 2502
autoscale_None() (*matplotlib.collections.PatchCollection method*), 2525
autoscale_None() (*matplotlib.collections.PathCollection method*), 2546
autoscale_None() (*matplotlib.collections.PolyCollection method*), 2569
autoscale_None() (*matplotlib.collections.PolyQuadMesh method*), 2591
autoscale_None() (*matplotlib.collections.QuadMesh method*), 2616
autoscale_None() (*matplotlib.collections.RegularPolyCollection method*), 2638
autoscale_None() (*matplotlib.collections.StarPolygonCollection method*), 2660
autoscale_None() (*matplotlib.collections.TriMesh method*), 2683
autoscale_None() (*matplotlib.colors.AsinhNorm method*), 2719
autoscale_None() (*matplotlib.colors.CenteredNorm method*), 2722
autoscale_None() (*matplotlib.colors.FuncNorm method*), 2724
autoscale_None() (*matplotlib.colors.LogNorm method*), 2725
autoscale_None() (*matplotlib.colors.Normalize method*), 2715
autoscale_None() (*matplotlib.colors.SymLogNorm method*), 2730
autoscale_None() (*matplotlib.colors.TwoSlopeNorm method*), 2731
autoscale_view() (*matplotlib.axes.Axes method*), 2160
autoscale_view() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3855
AuxTransformBox (*class in matplotlib.offsetbox*), 3074
available (*in module matplotlib.style*), 3645
available() (*matplotlib.widgets.LockDraw method*), 3782
Axes (*class in matplotlib.axes*), 1903
Axes (*class in mpl_toolkits.axes_grid1.mpl_axes*), 3977

Axes (*class in mpl_toolkits.axisartist.axislines*), 4028
axes (*matplotlib.artist.Artist property*), 1890
axes (*matplotlib.axis.Axis property*), 2246
axes (*matplotlib.collections.AsteriskPolygonCollection property*), 2390
axes (*matplotlib.collections.CircleCollection property*), 2411
axes (*matplotlib.collections.Collection property*), 2435
axes (*matplotlib.collections.EllipseCollection property*), 2457
axes (*matplotlib.collections.EventCollection property*), 2479
axes (*matplotlib.collections.LineCollection property*), 2502
axes (*matplotlib.collections.PatchCollection property*), 2525
axes (*matplotlib.collections.PathCollection property*), 2546
axes (*matplotlib.collections.PolyCollection property*), 2569
axes (*matplotlib.collections.PolyQuadMesh property*), 2591
axes (*matplotlib.collections.QuadMesh property*), 2616
axes (*matplotlib.collections.RegularPolyCollection property*), 2638
axes (*matplotlib.collections.StarPolygonCollection property*), 2660
axes (*matplotlib.collections.TriMesh property*), 2683
axes (*matplotlib.figure.Figure property*), 2812
axes (*matplotlib.offsetbox.OffsetBox property*), 3080
axes() (*in module matplotlib.pyplot*), 3212
Axes3D (*class in mpl_toolkits.mplot3d.axes3d*), 3815
Axes.AxisDict (*class in mpl_toolkits.axes_grid1.mpl_axes*), 3979
AxesDivider (*class in mpl_toolkits.axes_grid1.axes_divider*), 3930
AxesGrid (*in module mpl_toolkits.axes_grid1.axes_grid*), 3940
AxesGrid (*in module mpl_toolkits.axisartist.axes_grid*), 3998
AxesImage (*class in matplotlib.image*), 2967
AxesLocator (*class in mpl_toolkits.axes_grid1.axes_divider*), 3931
AXESPAD (*matplotlib.table.Table attribute*), 3651
AxesWidget (*class in matplotlib.widgets*), 3772
AxesX (*class in mpl_toolkits.axes_grid1.axes_size*), 3949
AxesY (*class in mpl_toolkits.axes_grid1.axes_size*), 3950
AxesZero (*class in mpl_toolkits.axisartist.axislines*), 4034
axhline() (*in module matplotlib.pyplot*), 3302
axhline() (*matplotlib.axes.Axes method*), 1964
axhspan() (*in module matplotlib.pyplot*), 3305
axhspan() (*matplotlib.axes.Axes method*), 1966
Axis (*class in matplotlib.axis*), 2219
Axis (*class in mpl_toolkits.mplot3d.axis3d*), 3874
axis (*matplotlib.ticker.TickHelper attribute*), 3714
axis (*mpl_toolkits.axes_grid1.mpl_axes.Axes property*), 3979
axis (*mpl_toolkits.axisartist.axislines.Axes property*), 4030
axis() (*in module matplotlib.pyplot*), 3449
axis() (*matplotlib.axes.Axes method*), 2108
axis_aligned_extrema() (*matplotlib.bezier.BezierSegment method*), 2364
axis_date() (*matplotlib.axis.Axis method*), 2236
axis_name (*matplotlib.axis.XAxis attribute*), 2242
axis_name (*matplotlib.axis.YAxis attribute*), 2244
axis_name (*matplotlib.projections.polar.RadialAxis attribute*), 3548
axis_name (*matplotlib.projections.polar.ThetaAxis attribute*), 3553
AxisArtist (*class in mpl_toolkits.axisartist.axis_artist*), 4004
AxisArtistHelper (*class in mpl_toolkits.axisartist.axislines*), 4038
AxisArtistHelperRectlinear (*class in mpl_toolkits.axisartist.axislines*), 4038
AxisInfo (*class in matplotlib.units*), 3769
axisinfo() (*matplotlib.category.StrCategoryConverter static method*), 2368
axisinfo() (*matplotlib.dates.ConciseDateConverter method*), 2774
axisinfo() (*matplotlib.dates.DateConverter method*), 2776
axisinfo() (*matplotlib.units.ConversionInterface static method*), 3770
AxisLabel (*class in mpl_toolkits.axisartist.axis_artist*), 4009
AxislineStyle (*class in mpl_toolkits.axisartist.axisline_style*), 4025
AxislineStyle.FilledArrow (*class in mpl_toolkits.axisartist.axisline_style*), 4026
AxislineStyle.SimpleArrow (*class in mpl_toolkits.axisartist.axisline_style*), 4026
AxisScaleBase (*class in matplotlib.backend_tools*), 2288
AxLine (*class in matplotlib.lines*), 3034
axline() (*in module matplotlib.pyplot*), 3311
axline() (*matplotlib.axes.Axes method*), 1973
axvline() (*in module matplotlib.pyplot*), 3307
axvline() (*matplotlib.axes.Axes method*), 1968
axvspan() (*in module matplotlib.pyplot*), 3309
axvspan() (*matplotlib.axes.Axes method*), 1971

## B

BACK (*matplotlib.backend_bases.MouseButton attribute*), 2269
back() (*matplotlib.backend_bases.NavigationToolbar2 method*), 2271
back() (*matplotlib.backend_tools.ToolViewsPositions method*), 2298
back() (*matplotlib.cbook.Stack method*), 2374
backend_for_gui_framework() (*matplotlib.backends.registry.BackendRegistry method*), 2347
BackendFilter (*class in matplotlib.backends.registry*), 2346
BackendRegistry (*class in matplotlib.backends.registry*), 2347
bar() (*in module matplotlib.pyplot*), 3278
bar() (*matplotlib.axes.Axes method*), 1940
bar() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3820
bar3d() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3821
bar_label() (*in module matplotlib.pyplot*), 3286
bar_label() (*matplotlib.axes.Axes method*), 1948
Barbs (*class in matplotlib.quiver*), 3600
barbs() (*in module matplotlib.pyplot*), 3435
barbs() (*matplotlib.axes.Axes method*), 2094
BarContainer (*class in matplotlib.container*), 2753
barh() (*in module matplotlib.pyplot*), 3282
barh() (*matplotlib.axes.Axes method*), 1944
base (*matplotlib.scale.FuncScaleLog property*), 3617

base (*matplotlib.scale.LogScale property*), 3622
base (*matplotlib.scale.SymmetricalLogScale property*), 3628
BASE_COLORS (*in module matplotlib.colors*), 2753
Bbox (*class in matplotlib.transforms*), 3725
bbox (*matplotlib._afm.CharMetrics attribute*), 1831
bbox (*matplotlib.ft2font.FT2Font attribute*), 2953
bbox_artist() (*in module matplotlib.patches*), 3193
BboxBase (*class in matplotlib.transforms*), 3731
BboxConnector (*class in mpl_toolkits.axes_grid1.inset_locator*), 3960
BboxConnectorPatch (*class in mpl_toolkits.axes_grid1.inset_locator*), 3963
BboxImage (*class in matplotlib.image*), 2971
BboxPatch (*class in mpl_toolkits.axes_grid1.inset_locator*), 3966
BboxTransform (*class in matplotlib.transforms*), 3735
BboxTransformFrom (*class in matplotlib.transforms*), 3736
BboxTransformTo (*class in matplotlib.transforms*), 3736
BboxTransformToMaxOnly (*class in matplotlib.transforms*), 3736
begin_text (*matplotlib.backends.backend_pdf.Op attribute*), 2324
begin_typing() (*matplotlib.widgets.TextBox method*), 3799
beginStream() (*matplotlib.backends.backend_pdf.PdfFile method*), 2327
BezierSegment (*class in matplotlib.bezier*), 2364
bin_path() (*matplotlib.animation.ImageMagickBase class method*), 1867
bin_path() (*matplotlib.animation.MovieWriter class method*), 1862
blend_hsv() (*matplotlib.colors.LightSource method*), 2741
blend_overlay() (*matplotlib.colors.LightSource method*), 2742
blend_soft_light() (*matplotlib.colors.LightSource method*), 2742
blended_transform_factory() (*in module matplotlib.transforms*), 3753
BlendedAffine2D (*class in matplotlib.transforms*), 3737
BlendedGenericTransform (*class in matplotlib.transforms*), 3737
blit() (*matplotlib.backend_bases.FigureCanvasBase method*), 2255
blit() (*matplotlib.backends.backend_tkagg.FigureCanvasTkAgg method*), 2357
blit() (*matplotlib.backends.backend_webagg_core.FigureCanvasWebAggCore method*), 2358
blocked() (*matplotlib.cbook.CallbackRegistry method*), 2372
BoundaryNorm (*class in matplotlib.colors*), 2720
bounds (*matplotlib.transforms.Bbox property*), 3727
bounds (*matplotlib.transforms.BboxBase property*), 3732
box() (*in module matplotlib.pyplot*), 3451
boxplot() (*in module matplotlib.pyplot*), 3343
boxplot() (*matplotlib.axes.Axes method*), 2004
boxplot_stats() (*in module matplotlib.cbook*), 2374
BoxStyle (*class in matplotlib.patches*), 3118
BoxStyle.Circle (*class in matplotlib.patches*), 3119
BoxStyle.DArrow (*class in matplotlib.patches*), 3119
BoxStyle.Ellipse (*class in matplotlib.patches*), 3119
BoxStyle.LArrow (*class in matplotlib.patches*), 3119
BoxStyle.RArrow (*class in matplotlib.patches*), 3120
BoxStyle.Round (*class in matplotlib.patches*), 3120
BoxStyle.Round4 (*class in matplotlib.patches*), 3120
BoxStyle.Roundtooth (*class in matplotlib.patches*), 3121
BoxStyle.Sawtooth (*class in matplotlib.patches*), 3121
BoxStyle.Square (*class in matplotlib.patches*), 3121
broken_barh() (*in module matplotlib.pyplot*), 3296
broken_barh() (*matplotlib.axes.Axes method*), 1958
bubble() (*matplotlib.cbook.Stack method*), 2374
buffer_rgba() (*matplotlib.backends.backend_agg.FigureCanvasAgg method*), 2309
buffer_rgba() (*matplotlib.backends.backend_agg.RendererAgg method*), 2312
Button (*class in matplotlib.widgets*), 3773
button_pick_id (*matplotlib.backend_bases.FigureCanvasBase property*), 2255
button_press_handler() (*in module matplotlib.backend_bases*), 2283
bxp() (*matplotlib.axes.Axes method*), 2011

## C

caching_module_getattr() (*in module matplotlib._api*), 3802
calc_label_rot_and_inline() (*matplotlib.contour.ContourLabeler method*), 2756
calculate_plane_coefficients() (*matplotlib.tri.Triangulation method*), 3756
calculate_rms() (*in module matplotlib.testing.compare*), 3659
CallbackRegistry (*class in matplotlib.cbook*), 2371
callbacks (*matplotlib.backend_bases.FigureCanvasBase property*), 2255
can_pan() (*matplotlib.axes.Axes method*), 2198
can_pan() (*matplotlib.projections.geo.GeoAxes method*), 3564
can_pan() (*matplotlib.projections.polar.PolarAxes method*), 3535
can_pan() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3865
can_zoom() (*matplotlib.axes.Axes method*), 2198
can_zoom() (*matplotlib.projections.geo.GeoAxes method*), 3564
can_zoom() (*matplotlib.projections.polar.PolarAxes method*), 3535
can_zoom() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3865
canvas (*matplotlib.backend_managers.ToolManager property*), 2285
canvas (*matplotlib.backend_tools.ToolBase property*), 2290
canvas (*matplotlib.lines.VertexSelector property*), 3034
canvas (*matplotlib.offsetbox.DraggableBase property*), 3076

CapStyle (*class in matplotlib._enums*), 3811
capstyle_cmd() (*matplotlib.backends.backend_pdf.GraphicsContextPdf method*), 2323
capstyles (*matplotlib.backends.backend_pdf.GraphicsContextPdf attribute*), 2323
CapStyleType (*in module matplotlib.typing*), 3768
CbarAxesBase (*class in mpl_toolkits.axes_grid1.axes_grid*), 3940
Cell (*class in matplotlib.table*), 3645
center (*matplotlib.patches.Annulus property*), 3098
center (*matplotlib.patches.Ellipse property*), 3136
center (*matplotlib.widgets.RectangleSelector property*), 3792
CenteredNorm (*class in matplotlib.colors*), 2721
changed() (*matplotlib.cm.ScalarMappable method*), 2386
changed() (*matplotlib.collections.AsteriskPolygonCollection method*), 2390
changed() (*matplotlib.collections.CircleCollection method*), 2411
changed() (*matplotlib.collections.Collection method*), 2435
changed() (*matplotlib.collections.EllipseCollection method*), 2457
changed() (*matplotlib.collections.EventCollection method*), 2479
changed() (*matplotlib.collections.LineCollection method*), 2502
changed() (*matplotlib.collections.PatchCollection method*), 2525
changed() (*matplotlib.collections.PathCollection method*), 2546
changed() (*matplotlib.collections.PolyCollection method*), 2569
changed() (*matplotlib.collections.PolyQuadMesh method*), 2591
changed() (*matplotlib.collections.QuadMesh method*), 2616
changed() (*matplotlib.collections.RegularPolyCollection method*), 2638
changed() (*matplotlib.collections.StarPolygonCollection method*), 2660
changed() (*matplotlib.collections.TriMesh method*), 2683
changed() (*matplotlib.contour.ContourSet method*), 2761
CharMetrics (*class in matplotlib._afm*), 1831
check_figures_equal() (*in module matplotlib.testing.decorators*), 3660
check_gc() (*matplotlib.backends.backend_pdf.RendererPdf method*), 2330
check_getitem() (*in module matplotlib._api*), 3803
check_if_parallel() (*in module matplotlib.bezier*), 2365
check_in_list() (*in module matplotlib._api*), 3803
check_isinstance() (*in module matplotlib._api*), 3804
check_shape() (*in module matplotlib._api*), 3804
CheckButtons (*class in matplotlib.widgets*), 3774
checksum (*matplotlib.dviread.Tfm attribute*), 2791
cids (*matplotlib.offsetbox.DraggableBase property*), 3076
Circle (*class in matplotlib.patches*), 3122
circle() (*matplotlib.path.Path class method*), 3196
circle_ratios() (*matplotlib.tri.TriAnalyzer method*), 3764
CircleCollection (*class in matplotlib.collections*), 2410
CirclePolygon (*class in matplotlib.patches*), 3125
Circles (*class in matplotlib.hatch*), 2966
circular_spine() (*matplotlib.spines.Spine class method*), 3638
cla() (*in module matplotlib.pyplot*), 3215
cla() (*matplotlib.axes.Axes method*), 2107
clabel() (*in module matplotlib.pyplot*), 3364
clabel() (*matplotlib.axes.Axes method*), 2029
clabel() (*matplotlib.contour.ContourLabeler method*), 2757
clabel() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3826
classproperty (*class in matplotlib._api*), 3804
clean() (*matplotlib.cbook.Grouper method*), 2373
cleaned() (*matplotlib.path.Path method*), 3196
clear() (*matplotlib.axes.Axes method*), 2107
clear() (*matplotlib.axis.Axis method*), 2221
clear() (*matplotlib.backend_tools.ToolViewsPositions method*), 2298
clear() (*matplotlib.backends.backend_agg.RendererAgg method*), 2312
clear() (*matplotlib.cbook.Stack method*), 2374
clear() (*matplotlib.figure.Figure method*), 2860
clear() (*matplotlib.figure.FigureBase method*), 2914
clear() (*matplotlib.ft2font.FT2Font method*), 2953
clear() (*matplotlib.projections.geo.GeoAxes method*), 3565
clear() (*matplotlib.projections.geo.LambertAxes method*), 3580
clear() (*matplotlib.projections.polar.PolarAxes method*), 3535
clear() (*matplotlib.projections.polar.RadialAxis method*), 3549
clear() (*matplotlib.projections.polar.ThetaAxis method*), 3553
clear() (*matplotlib.spines.Spine method*), 3638
clear() (*matplotlib.transforms.Affine2D method*), 3720
clear() (*matplotlib.widgets.CheckButtons method*), 3776
clear() (*matplotlib.widgets.Cursor method*), 3778
clear() (*matplotlib.widgets.MultiCursor method*), 3783
clear() (*matplotlib.widgets.RadioButtons method*), 3786
clear() (*mpl_toolkits.axes_grid1.mpl_axes.Axes method*), 3981
clear() (*mpl_toolkits.axes_grid1.parasite_axes.HostAxesBase method*), 3986
clear() (*mpl_toolkits.axes_grid1.parasite_axes.ParasiteAxesBase method*), 3987
clear() (*mpl_toolkits.axisartist.axislines.Axes method*), 4032
clear() (*mpl_toolkits.axisartist.axislines.AxesZero method*), 4036
clear() (*mpl_toolkits.axisartist.floating_axes.FloatingAxesBase method*), 4042
clear() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3838

clearup_closed() (*matplotlib.backends.backend_nbagg.FigureManagerNbAgg method*), 2321
clf() (*in module matplotlib.pyplot*), 3215
clf() (*matplotlib.figure.FigureBase method*), 2915
clim() (*in module matplotlib.pyplot*), 3477
clip (*matplotlib.backends.backend_pdf.Op attribute*), 2324
clip (*matplotlib.colors.Normalize property*), 2715
clip_children (*matplotlib.offsetbox.DrawingArea property*), 3077
clip_cmd() (*matplotlib.backends.backend_pdf.GraphicsContextPdf method*), 2323
clip_to_bbox() (*matplotlib.path.Path method*), 3196
close() (*in module matplotlib.pyplot*), 3216
close() (*matplotlib.backends.backend_pdf.PdfFile method*), 2327
close() (*matplotlib.backends.backend_pdf.PdfPages method*), 2329
close() (*matplotlib.backends.backend_pgf.PdfPages method*), 2337
close() (*matplotlib.backends.backend_svg.XMLWriter method*), 2356
close() (*matplotlib.dviread.Dvi method*), 2788
close_fill_stroke (*matplotlib.backends.backend_pdf.Op attribute*), 2324
close_group() (*matplotlib.backend_bases.RendererBase method*), 2273
close_group() (*matplotlib.backends.backend_svg.RendererSVG method*), 2352
close_group() (*matplotlib.patheffects.PathEffectRenderer method*), 3204
close_stroke (*matplotlib.backends.backend_pdf.Op attribute*), 2324
CloseEvent (*class in matplotlib.backend_bases*), 2253
closepath (*matplotlib.backends.backend_pdf.Op attribute*), 2324
CLOSEPOLY (*matplotlib.path.Path attribute*), 3195
closest() (*matplotlib.widgets.ToolHandles method*), 3800
closest() (*matplotlib.widgets.ToolLineHandles method*), 3801
code_type (*matplotlib.path.Path attribute*), 3197
codes (*matplotlib.legend.Legend attribute*), 2995
codes (*matplotlib.offsetbox.AnchoredOffsetbox attribute*), 3066
codes (*matplotlib.path.Path property*), 3197
codes (*matplotlib.table.Table attribute*), 3652
codes (*matplotlib.text.TextPath property*), 3686
coefs (*matplotlib.transforms.BboxBase attribute*), 3732
cohere() (*in module matplotlib.mlab*), 3050
cohere() (*in module matplotlib.pyplot*), 3319
cohere() (*matplotlib.axes.Axes method*), 1980
Collection (*class in matplotlib.collections*), 2432
collections (*matplotlib.contour.ContourSet property*), 2761
color_sequences (*in module matplotlib*), 1828
color_sequences (*in module matplotlib.pyplot*), 3486
Colorbar (*class in matplotlib.colorbar*), 2704
colorbar (*matplotlib.cm.ScalarMappable attribute*), 2386
colorbar (*matplotlib.collections.AsteriskPolygonCollection attribute*), 2390
colorbar (*matplotlib.collections.CircleCollection attribute*), 2411
colorbar (*matplotlib.collections.Collection attribute*), 2435
colorbar (*matplotlib.collections.EllipseCollection attribute*), 2457
colorbar (*matplotlib.collections.EventCollection attribute*), 2479
colorbar (*matplotlib.collections.LineCollection attribute*), 2502
colorbar (*matplotlib.collections.PatchCollection attribute*), 2525
colorbar (*matplotlib.collections.PathCollection attribute*), 2546
colorbar (*matplotlib.collections.PolyCollection attribute*), 2569
colorbar (*matplotlib.collections.PolyQuadMesh attribute*), 2592
colorbar (*matplotlib.collections.QuadMesh attribute*), 2616
colorbar (*matplotlib.collections.RegularPolyCollection attribute*), 2638
colorbar (*matplotlib.collections.StarPolygonCollection attribute*), 2660
colorbar (*matplotlib.collections.TriMesh attribute*), 2683
colorbar() (*in module matplotlib.pyplot*), 3477
colorbar() (*matplotlib.figure.Figure method*), 2818
colorbar() (*matplotlib.figure.FigureBase method*), 2915
colorbar() (*matplotlib.figure.SubFigure method*), 2881
colorbar() (*mpl_toolkits.axes_grid1.axes_grid.CbarAxesBase method*), 3940
colorbar_extend (*matplotlib.colors.Colormap attribute*), 2732
colorbar_gridspec (*matplotlib.layout_engine.ConstrainedLayoutEngine property*), 2986
colorbar_gridspec (*matplotlib.layout_engine.LayoutEngine property*), 2987
colorbar_gridspec (*matplotlib.layout_engine.PlaceHolderLayoutEngine property*), 2988
colorbar_gridspec (*matplotlib.layout_engine.TightLayoutEngine property*), 2988
ColorbarBase (*in module matplotlib.colorbar*), 2710
Colormap (*class in matplotlib.colors*), 2732
ColormapRegistry (*class in matplotlib.cm*), 2384
colormaps (*in module matplotlib*), 1828
colormaps (*in module matplotlib.pyplot*), 3486
ColorSequenceRegistry (*class in matplotlib.colors*), 2739
ColorType (*in module matplotlib.typing*), 3768
ColourType (*in module matplotlib.typing*), 3768
colspan (*matplotlib.gridspec.SubplotSpec property*), 2960

commands (*matplotlib.backends.backend_pdf.GraphicsContextPdf attribute*), 2323
comment() (*matplotlib.backends.backend_svg.XMLWriter method*), 2356
CommSocket (*class in matplotlib.backends.backend_nbagg*), 2321
comparable_formats() (*in module matplotlib.testing.compare*), 3659
compare_images() (*in module matplotlib.testing.compare*), 3659
complex_spectrum() (*in module matplotlib.mlab*), 3051
composite_images() (*in module matplotlib.image*), 2981
composite_transform_factory() (*in module matplotlib.transforms*), 3753
CompositeAffine2D (*class in matplotlib.transforms*), 3739
CompositeGenericTransform (*class in matplotlib.transforms*), 3739
CompositePart (*class in matplotlib._afm*), 1831
compressobj (*matplotlib.backends.backend_pdf.Stream attribute*), 2334
concat_matrix (*matplotlib.backends.backend_pdf.Op attribute*), 2324
ConciseDateConverter (*class in matplotlib.dates*), 2774
ConciseDateFormatter (*class in matplotlib.dates*), 2774
configure_subplots() (*matplotlib.backend_bases.NavigationToolbar2 method*), 2271
ConfigureSubplotsBase (*class in matplotlib.backend_tools*), 2288
connect() (*in module matplotlib.pyplot*), 3516
connect() (*matplotlib.cbook.CallbackRegistry method*), 2372
connect() (*matplotlib.patches.ConnectionStyle.Angle method*), 3133
connect() (*matplotlib.patches.ConnectionStyle.Angle3 method*), 3133
connect() (*matplotlib.patches.ConnectionStyle.Arc method*), 3134
connect() (*matplotlib.patches.ConnectionStyle.Arc3 method*), 3134
connect() (*matplotlib.patches.ConnectionStyle.Bar method*), 3135
connect() (*matplotlib.widgets.MultiCursor method*), 3783
connect_bbox() (*mpl_toolkits.axes_grid1.inset_locator.BboxConnector static method*), 3962
connect_default_events() (*matplotlib.widgets.SpanSelector method*), 3798
connect_event() (*matplotlib.widgets.AxesWidget method*), 3773
connected (*matplotlib.backends.backend_nbagg.FigureManagerNbAgg property*), 2321
connection_info() (*in module matplotlib.backends.backend_nbagg*), 2322
ConnectionPatch (*class in matplotlib.patches*), 3127
ConnectionStyle (*class in matplotlib.patches*), 3132
ConnectionStyle.Angle (*class in matplotlib.patches*), 3132
ConnectionStyle.Angle3 (*class in matplotlib.patches*), 3133
ConnectionStyle.Arc (*class in matplotlib.patches*), 3133
ConnectionStyle.Arc3 (*class in matplotlib.patches*), 3134
ConnectionStyle.Bar (*class in matplotlib.patches*), 3134
ConstrainedLayoutEngine (*class in matplotlib.layout_engine*), 2985
Container (*class in matplotlib.container*), 2753
contains() (*matplotlib.artist.Artist method*), 1872
contains() (*matplotlib.axes.Axes method*), 2202
contains() (*matplotlib.axis.Axis method*), 2239
contains() (*matplotlib.collections.AsteriskPolygonCollection method*), 2390
contains() (*matplotlib.collections.CircleCollection method*), 2412
contains() (*matplotlib.collections.Collection method*), 2435
contains() (*matplotlib.collections.EllipseCollection method*), 2457
contains() (*matplotlib.collections.EventCollection method*), 2479
contains() (*matplotlib.collections.LineCollection method*), 2503
contains() (*matplotlib.collections.PatchCollection method*), 2525
contains() (*matplotlib.collections.PathCollection method*), 2546
contains() (*matplotlib.collections.PolyCollection method*), 2569
contains() (*matplotlib.collections.PolyQuadMesh method*), 2592
contains() (*matplotlib.collections.QuadMesh method*), 2616
contains() (*matplotlib.collections.RegularPolyCollection method*), 2638
contains() (*matplotlib.collections.StarPolygonCollection method*), 2660
contains() (*matplotlib.collections.TriMesh method*), 2683
contains() (*matplotlib.figure.FigureBase method*), 2918
contains() (*matplotlib.image.BboxImage method*), 2971
contains() (*matplotlib.legend.Legend method*), 2995
contains() (*matplotlib.lines.Line2D method*), 3017
contains() (*matplotlib.offsetbox.AnnotationBbox method*), 3071
contains() (*matplotlib.offsetbox.OffsetBox method*), 3081
contains() (*matplotlib.patches.Patch method*), 3159
contains() (*matplotlib.quiver.QuiverKey method*), 3597
contains() (*matplotlib.table.Table method*), 3652
contains() (*matplotlib.text.Annotation method*), 3681
contains() (*matplotlib.text.Text method*), 3664
contains() (*matplotlib.transforms.BboxBase method*), 3732
contains() (*matplotlib.transforms.TransformedBbox method*), 3752
contains_branch() (*matplotlib.transforms.BlendedGenericTransform*

*method*), 3737
contains_branch() (*matplotlib.transforms.Transform method*), 3746
contains_branch_seperately() (*matplotlib.transforms.Transform method*), 3746
contains_path() (*matplotlib.path.Path method*), 3197
contains_point() (*matplotlib.axes.Axes method*), 2203
contains_point() (*matplotlib.patches.Patch method*), 3159
contains_point() (*matplotlib.path.Path method*), 3197
contains_points() (*matplotlib.patches.Patch method*), 3161
contains_points() (*matplotlib.path.Path method*), 3198
containsx() (*matplotlib.transforms.BboxBase method*), 3732
containsy() (*matplotlib.transforms.BboxBase method*), 3732
context() (*in module matplotlib.style*), 3643
contiguous_regions() (*in module matplotlib.cbook*), 2376
contour() (*in module matplotlib.pyplot*), 3364
contour() (*matplotlib.axes.Axes method*), 2030
contour() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3826
contourf() (*in module matplotlib.pyplot*), 3370
contourf() (*matplotlib.axes.Axes method*), 2035
contourf() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3828
ContourLabeler (*class in matplotlib.contour*), 2756
ContourSet (*class in matplotlib.contour*), 2759
control_points (*matplotlib.bezier.BezierSegment property*), 2364
ConversionError, 3770
ConversionInterface (*class in matplotlib.units*), 3770
convert() (*matplotlib.category.StrCategoryConverter static method*), 2368
convert() (*matplotlib.dates.DateConverter static method*), 2776
convert() (*matplotlib.units.ConversionInterface static method*), 3770
convert() (*matplotlib.units.DecimalConverter static method*), 3770
convert_mesh_to_paths() (*matplotlib.collections.TriMesh static method*), 2683
convert_to_pct() (*matplotlib.ticker.PercentFormatter method*), 3708
convert_units() (*matplotlib.axis.Axis method*), 2240
convert_xunits() (*matplotlib.artist.Artist method*), 1893
convert_xunits() (*matplotlib.axes.Axes method*), 2184
convert_xunits() (*matplotlib.collections.AsteriskPolygonCollection method*), 2390
convert_xunits() (*matplotlib.collections.CircleCollection method*), 2412
convert_xunits() (*matplotlib.collections.Collection method*), 2435
convert_xunits() (*matplotlib.collections.EllipseCollection method*), 2457
convert_xunits() (*matplotlib.collections.EventCollection method*), 2479
convert_xunits() (*matplotlib.collections.LineCollection method*), 2503
convert_xunits() (*matplotlib.collections.PatchCollection method*), 2525
convert_xunits() (*matplotlib.collections.PathCollection method*), 2546
convert_xunits() (*matplotlib.collections.PolyCollection method*), 2569
convert_xunits() (*matplotlib.collections.PolyQuadMesh method*), 2592
convert_xunits() (*matplotlib.collections.QuadMesh method*), 2616
convert_xunits() (*matplotlib.collections.RegularPolyCollection method*), 2638
convert_xunits() (*matplotlib.collections.StarPolygonCollection method*), 2660
convert_xunits() (*matplotlib.collections.TriMesh method*), 2683
convert_yunits() (*matplotlib.artist.Artist method*), 1893
convert_yunits() (*matplotlib.axes.Axes method*), 2184
convert_yunits() (*matplotlib.collections.AsteriskPolygonCollection method*), 2390
convert_yunits() (*matplotlib.collections.CircleCollection method*), 2412
convert_yunits() (*matplotlib.collections.Collection method*), 2435
convert_yunits() (*matplotlib.collections.EllipseCollection method*), 2457
convert_yunits() (*matplotlib.collections.EventCollection method*), 2479
convert_yunits() (*matplotlib.collections.LineCollection method*), 2503
convert_yunits() (*matplotlib.collections.PatchCollection method*), 2525
convert_yunits() (*matplotlib.collections.PathCollection method*), 2546
convert_yunits() (*matplotlib.collections.PolyCollection method*), 2569
convert_yunits() (*matplotlib.collections.PolyQuadMesh method*), 2592
convert_yunits() (*matplotlib.collections.QuadMesh method*), 2616
convert_yunits() (*matplotlib.collections.RegularPolyCollection method*), 2638
convert_yunits() (*matplotlib.collections.StarPolygonCollection method*), 2660
convert_yunits() (*matplotlib.collections.TriMesh method*), 2683
convert_zunits() (*mpl_toolkits.mplot3d.axes3d.Axes3D*

*method*), 3863
copy() (*in module matplotlib._docstring*), 2787
copy() (*matplotlib.colors.Colormap method*), 2733
copy() (*matplotlib.font_manager.FontProperties method*), 2945
copy() (*matplotlib.path.Path method*), 3198
copy() (*matplotlib.RcParams method*), 1823
copy_from_bbox() (*matplotlib.backends.backend_agg.FigureCanvasAgg method*), 2309
copy_from_bbox() (*matplotlib.backends.backend_cairo.FigureCanvasCairo method*), 2315
copy_properties() (*matplotlib.backend_bases.GraphicsContextBase method*), 2264
copy_properties() (*matplotlib.backends.backend_pdf.GraphicsContextPdf method*), 2323
copy_with_path_effect() (*matplotlib.patheffects.PathEffectRenderer method*), 3204
corners (*matplotlib.widgets.RectangleSelector property*), 3793
corners() (*matplotlib.transforms.BboxBase method*), 3732
count_contains() (*matplotlib.transforms.BboxBase method*), 3732
count_overlaps() (*matplotlib.transforms.BboxBase method*), 3732
covariance_factor() (*matplotlib.mlab.GaussianKDE method*), 3048
create_artists() (*matplotlib.legend_handler.HandlerBase method*), 3003
create_artists() (*matplotlib.legend_handler.HandlerErrorbar method*), 3005
create_artists() (*matplotlib.legend_handler.HandlerLine2D method*), 3006
create_artists() (*matplotlib.legend_handler.HandlerLine2DCompound method*), 3007
create_artists() (*matplotlib.legend_handler.HandlerLineCollection method*), 3008
create_artists() (*matplotlib.legend_handler.HandlerPatch method*), 3010
create_artists() (*matplotlib.legend_handler.HandlerPolyCollection method*), 3011
create_artists() (*matplotlib.legend_handler.HandlerRegularPolyCollection method*), 3012
create_artists() (*matplotlib.legend_handler.HandlerStem method*), 3013
create_artists() (*matplotlib.legend_handler.HandlerStepPatch method*), 3014
create_artists() (*matplotlib.legend_handler.HandlerTuple method*), 3015
create_collection() (*matplotlib.legend_handler.HandlerCircleCollection method*), 3005
create_collection() (*matplotlib.legend_handler.HandlerPathCollection method*), 3011
create_collection() (*matplotlib.legend_handler.HandlerRegularPolyCollection method*), 3012
create_dummy_axis() (*matplotlib.ticker.TickHelper method*), 3714
create_hatch() (*matplotlib.backends.backend_ps.RendererPS method*), 2342
create_with_canvas() (*matplotlib.backend_bases.FigureManagerBase class method*), 2263
create_with_canvas() (*matplotlib.backends.backend_nbagg.FigureManagerNbAgg class method*), 2321
createType1Descriptor() (*matplotlib.backends.backend_pdf.PdfFile method*), 2327
csd() (*in module matplotlib.mlab*), 3053
csd() (*in module matplotlib.pyplot*), 3322
csd() (*matplotlib.axes.Axes method*), 1983
CSS4_COLORS (*in module matplotlib.colors*), 2753
CubicTriInterpolator (*class in matplotlib.tri*), 3760
Cursor (*class in matplotlib.widgets*), 3777
cursor (*matplotlib.backend_tools.ToolPan attribute*), 2295
cursor (*matplotlib.backend_tools.ToolToggleBase attribute*), 2297
cursor (*matplotlib.backend_tools.ToolZoom attribute*), 2299
Cursors (*class in matplotlib.backend_tools*), 2289
cursors (*in module matplotlib.backend_tools*), 2303
CURVE3 (*matplotlib.path.Path attribute*), 3195
CURVE4 (*matplotlib.path.Path attribute*), 3195
curveto (*matplotlib.backends.backend_pdf.Op attribute*), 2324
CustomCell (*in module matplotlib.table*), 3650
cycler() (*in module matplotlib.rcsetup*), 3606

## D

d_interval (*mpl_toolkits.mplot3d.axis3d.Axis property*), 3875
dash_cmd() (*matplotlib.backends.backend_pdf.GraphicsContextPdf method*), 2323
data() (*matplotlib.backends.backend_svg.XMLWriter method*), 2356
datalim_to_dt() (*matplotlib.dates.DateLocator method*), 2776
date2num() (*in module matplotlib.dates*), 2781

DateConverter (*class in matplotlib.dates*), 2775
DateFormatter (*class in matplotlib.dates*), 2776
DateLocator (*class in matplotlib.dates*), 2776
datestr2num() (*in module matplotlib.dates*), 2782
DayLocator (*class in matplotlib.dates*), 2777
DecimalConverter (*class in matplotlib.units*), 3770
decrypted (*matplotlib._type1font.Type1Font attribute*), 3767
deepcopy() (*matplotlib.path.Path method*), 3199
default_keymap (*matplotlib.backend_tools.SaveFigureBase property*), 2290
default_keymap (*matplotlib.backend_tools.ToolBack property*), 2290
default_keymap (*matplotlib.backend_tools.ToolBase attribute*), 2291
default_keymap (*matplotlib.backend_tools.ToolCopyToClipboardBase property*), 2292
default_keymap (*matplotlib.backend_tools.ToolForward property*), 2292
default_keymap (*matplotlib.backend_tools.ToolFullScreen property*), 2293
default_keymap (*matplotlib.backend_tools.ToolGrid property*), 2293
default_keymap (*matplotlib.backend_tools.ToolHelpBase property*), 2294
default_keymap (*matplotlib.backend_tools.ToolHome property*), 2294
default_keymap (*matplotlib.backend_tools.ToolMinorGrid property*), 2295
default_keymap (*matplotlib.backend_tools.ToolPan property*), 2295
default_keymap (*matplotlib.backend_tools.ToolQuit property*), 2296
default_keymap (*matplotlib.backend_tools.ToolQuitAll property*), 2296
default_keymap (*matplotlib.backend_tools.ToolXScale property*), 2299
default_keymap (*matplotlib.backend_tools.ToolYScale property*), 2299
default_keymap (*matplotlib.backend_tools.ToolZoom property*), 2299
default_params (*matplotlib.ticker.MaxNLocator attribute*), 3706
default_toggled (*matplotlib.backend_tools.ToolToggleBase attribute*), 2297
default_units() (*matplotlib.category.StrCategoryConverter static method*), 2369
default_units() (*matplotlib.dates.DateConverter static method*), 2776
default_units() (*matplotlib.units.ConversionInterface static method*), 3770
defaultFont (*matplotlib.font_manager.FontManager property*), 2942
define_aliases() (*in module matplotlib._api*), 3804
deg_mark (*mpl_toolkits.axisartist.angle_helper.FormatterDMS attribute*), 3992
deg_mark (*mpl_toolkits.axisartist.angle_helper.FormatterHMS attribute*), 3993
degree (*matplotlib.bezier.BezierSegment property*), 2365
delaxes() (*in module matplotlib.pyplot*), 3216
delaxes() (*matplotlib.figure.Figure method*), 2813
delaxes() (*matplotlib.figure.FigureBase method*), 2918
delaxes() (*matplotlib.figure.SubFigure method*), 2879
delete_masked_points() (*in module matplotlib.cbook*), 2376
delete_parameter() (*in module matplotlib._api.deprecation*), 3806
delta() (*matplotlib.backends.backend_pdf.GraphicsContextPdf method*), 2323
demo() (*matplotlib._enums.CapStyle static method*), 3811
demo() (*matplotlib._enums.JoinStyle static method*), 3810
deprecate_method_override() (*in module matplotlib._api.deprecation*), 3807
deprecate_privatize_attribute (*class in matplotlib._api.deprecation*), 3807
deprecated() (*in module matplotlib._api.deprecation*), 3807
depth (*matplotlib.dviread.Tfm attribute*), 2791
depth (*matplotlib.mathtext.RasterParse attribute*), 3044
depth (*matplotlib.mathtext.VectorParse attribute*), 3045
depth (*matplotlib.transforms.BlendedGenericTransform property*), 3737
depth (*matplotlib.transforms.CompositeAffine2D property*), 3739
depth (*matplotlib.transforms.CompositeGenericTransform property*), 3740
depth (*matplotlib.transforms.Transform property*), 3746
descender (*matplotlib.ft2font.FT2Font attribute*), 2953
description (*matplotlib.backend_tools.ConfigureSubplotsBase attribute*), 2289
description (*matplotlib.backend_tools.SaveFigureBase attribute*), 2290
description (*matplotlib.backend_tools.ToolBack attribute*), 2290
description (*matplotlib.backend_tools.ToolBase attribute*), 2291
description (*matplotlib.backend_tools.ToolCopyToClipboardBase attribute*), 2292
description (*matplotlib.backend_tools.ToolForward attribute*), 2292
description (*matplotlib.backend_tools.ToolFullScreen attribute*), 2293
description (*matplotlib.backend_tools.ToolGrid attribute*), 2293
description (*matplotlib.backend_tools.ToolHelpBase attribute*), 2294
description (*matplotlib.backend_tools.ToolHome attribute*), 2294
description (*matplotlib.backend_tools.ToolMinorGrid attribute*), 2295

description (*matplotlib.backend_tools.ToolPan attribute*), 2295
description (*matplotlib.backend_tools.ToolQuit attribute*), 2296
description (*matplotlib.backend_tools.ToolQuitAll attribute*), 2296
description (*matplotlib.backend_tools.ToolXScale attribute*), 2299
description (*matplotlib.backend_tools.ToolYScale attribute*), 2299
description (*matplotlib.backend_tools.ToolZoom attribute*), 2299
design_size (*matplotlib.dviread.Tfm attribute*), 2791
destroy() (*matplotlib.backend_bases.FigureManagerBase method*), 2263
destroy() (*matplotlib.backends.backend_nbagg.FigureManagerNbAgg method*), 2321
detrend() (*in module matplotlib.mlab*), 3054
detrend_linear() (*in module matplotlib.mlab*), 3055
detrend_mean() (*in module matplotlib.mlab*), 3055
detrend_none() (*in module matplotlib.mlab*), 3056
device_pixel_ratio (*matplotlib.backend_bases.FigureCanvasBase property*), 2255
DictFormatter (*class in mpl_toolkits.axisartist.grid_finder*), 4045
dimension (*matplotlib.bezier.BezierSegment property*), 2365
direction (*matplotlib.colors.LightSource property*), 2743
direction (*matplotlib.widgets.SpanSelector property*), 3798
direction (*matplotlib.widgets.ToolLineHandles property*), 3801
disable() (*matplotlib.backend_tools.AxisScaleBase method*), 2288
disable() (*matplotlib.backend_tools.ToolToggleBase method*), 2297
disable() (*matplotlib.backend_tools.ZoomPanBase method*), 2300
disable_mouse_rotation() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3865
disconnect() (*in module matplotlib.pyplot*), 3518
disconnect() (*matplotlib.cbook.CallbackRegistry method*), 2372
disconnect() (*matplotlib.offsetbox.DraggableBase method*), 3076
disconnect() (*matplotlib.widgets.Button method*), 3774
disconnect() (*matplotlib.widgets.CheckButtons method*), 3776
disconnect() (*matplotlib.widgets.MultiCursor method*), 3783
disconnect() (*matplotlib.widgets.RadioButtons method*), 3786
disconnect() (*matplotlib.widgets.SliderBase method*), 3795
disconnect() (*matplotlib.widgets.TextBox method*), 3799
disconnect_events() (*matplotlib.widgets.AxesWidget method*), 3773
display_js() (*matplotlib.backends.backend_nbagg.FigureManagerNbAgg method*), 2321
Divider (*class in mpl_toolkits.axes_grid1.axes_divider*), 3932
do_3d_projection() (*mpl_toolkits.mplot3d.art3d.Line3DCollection method*), 3884
do_3d_projection() (*mpl_toolkits.mplot3d.art3d.Patch3D method*), 3886
do_3d_projection() (*mpl_toolkits.mplot3d.art3d.Patch3DCollection method*), 3888
do_3d_projection() (*mpl_toolkits.mplot3d.art3d.Path3DCollection method*), 3890
do_3d_projection() (*mpl_toolkits.mplot3d.art3d.PathPatch3D method*), 3893
do_3d_projection() (*mpl_toolkits.mplot3d.art3d.Poly3DCollection method*), 3896
dpi (*matplotlib.figure.Figure property*), 2844
DPI (*matplotlib.text.TextToPath attribute*), 3687
drag_pan() (*matplotlib.axes.Axes method*), 2200
drag_pan() (*matplotlib.backend_bases.NavigationToolbar2 method*), 2271
drag_pan() (*matplotlib.colorbar.Colorbar method*), 2707
drag_pan() (*matplotlib.projections.geo.GeoAxes method*), 3565
drag_pan() (*matplotlib.projections.polar.PolarAxes method*), 3535
drag_pan() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3865
drag_zoom() (*matplotlib.backend_bases.NavigationToolbar2 method*), 2271
DraggableAnnotation (*class in matplotlib.offsetbox*), 3075
DraggableBase (*class in matplotlib.offsetbox*), 3076
DraggableLegend (*class in matplotlib.legend*), 2990
DraggableOffsetBox (*class in matplotlib.offsetbox*), 3076
drange() (*in module matplotlib.dates*), 2782
draw() (*in module matplotlib.pyplot*), 3490
draw() (*matplotlib.artist.Artist method*), 1882
draw() (*matplotlib.axes.Axes method*), 2205
draw() (*matplotlib.backend_bases.FigureCanvasBase method*), 2255
draw() (*matplotlib.backends.backend_agg.FigureCanvasAgg method*), 2309
draw() (*matplotlib.backends.backend_pdf.FigureCanvasPdf method*), 2322
draw() (*matplotlib.backends.backend_pgf.FigureCanvasPgf method*), 2335
draw() (*matplotlib.backends.backend_ps.FigureCanvasPS method*), 2341
draw() (*matplotlib.backends.backend_svg.FigureCanvasSVG method*), 2351
draw() (*mat-*

*plotlib.backends.backend_template.FigureCanvasTemplate method*), 2305
draw() (*matplotlib.backends.backend_tkagg.FigureCanvasTkAgg method*), 2357
draw() (*matplotlib.backends.backend_tkcairo.FigureCanvasTkCairo method*), 2357
draw() (*matplotlib.backends.backend_webagg_core.FigureCanvasWebAggCore method*), 2358
draw() (*matplotlib.collections.AsteriskPolygonCollection method*), 2390
draw() (*matplotlib.collections.CircleCollection method*), 2412
draw() (*matplotlib.collections.Collection method*), 2435
draw() (*matplotlib.collections.EllipseCollection method*), 2457
draw() (*matplotlib.collections.EventCollection method*), 2479
draw() (*matplotlib.collections.LineCollection method*), 2503
draw() (*matplotlib.collections.PatchCollection method*), 2525
draw() (*matplotlib.collections.PathCollection method*), 2546
draw() (*matplotlib.collections.PolyCollection method*), 2569
draw() (*matplotlib.collections.PolyQuadMesh method*), 2592
draw() (*matplotlib.collections.QuadMesh method*), 2616
draw() (*matplotlib.collections.RegularPolyCollection method*), 2638
draw() (*matplotlib.collections.StarPolygonCollection method*), 2660
draw() (*matplotlib.collections.TriMesh method*), 2683
draw() (*matplotlib.contour.ContourSet method*), 2761
draw() (*matplotlib.figure.Figure method*), 2863
draw() (*matplotlib.figure.FigureBase method*), 2918
draw() (*matplotlib.legend.Legend method*), 2996
draw() (*matplotlib.lines.AxLine method*), 3034
draw() (*matplotlib.lines.Line2D method*), 3018
draw() (*matplotlib.offsetbox.AnchoredOffsetbox method*), 3066
draw() (*matplotlib.offsetbox.AnnotationBbox method*), 3071
draw() (*matplotlib.offsetbox.AuxTransformBox method*), 3074
draw() (*matplotlib.offsetbox.DrawingArea method*), 3077
draw() (*matplotlib.offsetbox.OffsetBox method*), 3081
draw() (*matplotlib.offsetbox.OffsetImage method*), 3084
draw() (*matplotlib.offsetbox.PaddedBox method*), 3088
draw() (*matplotlib.offsetbox.TextArea method*), 3090
draw() (*matplotlib.patches.Arc method*), 3103
draw() (*matplotlib.patches.ConnectionPatch method*), 3129
draw() (*matplotlib.patches.FancyArrowPatch method*), 3147
draw() (*matplotlib.patches.Patch method*), 3161
draw() (*matplotlib.patches.Shadow method*), 3188
draw() (*matplotlib.projections.polar.PolarAxes method*), 3536
draw() (*matplotlib.quiver.Quiver method*), 3594
draw() (*matplotlib.quiver.QuiverKey method*), 3598
draw() (*matplotlib.spines.Spine method*), 3638
draw() (*matplotlib.table.Cell method*), 3646
draw() (*matplotlib.table.Table method*), 3652
draw() (*matplotlib.text.Annotation method*), 3681
draw() (*matplotlib.text.Text method*), 3664
draw() (*mpl_toolkits.axes_grid1.inset_locator.AnchoredLocatorBase method*), 3954
draw() (*mpl_toolkits.axes_grid1.mpl_axes.SimpleAxisArtist method*), 3983
draw() (*mpl_toolkits.axes_grid1.parasite_axes.HostAxesBase method*), 3986
draw() (*mpl_toolkits.axisartist.axis_artist.AxisArtist method*), 4004
draw() (*mpl_toolkits.axisartist.axis_artist.AxisLabel method*), 4009
draw() (*mpl_toolkits.axisartist.axis_artist.GridlinesCollection method*), 4013
draw() (*mpl_toolkits.axisartist.axis_artist.LabelBase method*), 4017
draw() (*mpl_toolkits.axisartist.axis_artist.TickLabels method*), 4019
draw() (*mpl_toolkits.axisartist.axis_artist.Ticks method*), 4022
draw() (*mpl_toolkits.mplot3d.art3d.Line3D method*), 3880
draw() (*mpl_toolkits.mplot3d.art3d.Path3DCollection method*), 3890
draw() (*mpl_toolkits.mplot3d.art3d.Text3D method*), 3901
draw() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3868
draw() (*mpl_toolkits.mplot3d.axis3d.Axis method*), 3875
draw_artist() (*matplotlib.axes.Axes method*), 2206
draw_artist() (*matplotlib.figure.Figure method*), 2864
draw_bbox() (*in module matplotlib.patches*), 3193
draw_frame() (*matplotlib.legend.Legend method*), 2996
draw_frame() (*matplotlib.offsetbox.PaddedBox method*), 3088
draw_glyph_to_bitmap() (*matplotlib.ft2font.FT2Font method*), 2953
draw_glyphs_to_bitmap() (*matplotlib.ft2font.FT2Font method*), 2953
draw_gouraud_triangles() (*matplotlib.backend_bases.RendererBase method*), 2274
draw_gouraud_triangles() (*matplotlib.backends.backend_pdf.RendererPdf method*), 2330
draw_gouraud_triangles() (*matplotlib.backends.backend_ps.RendererPS method*), 2342
draw_gouraud_triangles() (*matplotlib.backends.backend_svg.RendererSVG method*), 2352
draw_grid() (*mpl_toolkits.mplot3d.axis3d.Axis method*), 3875
draw_idle() (*matplotlib.backend_bases.FigureCanvasBase method*), 2255
draw_idle() (*matplotlib.backends.backend_webagg_core.FigureCanvasWebAggCore method*), 2358
draw_if_interactive() (*in module matplotlib.pyplot*), 3491
draw_image() (*matplotlib.backend_bases.RendererBase method*), 2274

draw_image() (*matplotlib.backends.backend_cairo.RendererCairo method*), 2317
draw_image() (*matplotlib.backends.backend_pdf.RendererPdf method*), 2330
draw_image() (*matplotlib.backends.backend_pgf.RendererPgf method*), 2338
draw_image() (*matplotlib.backends.backend_ps.RendererPS method*), 2342
draw_image() (*matplotlib.backends.backend_svg.RendererSVG method*), 2352
draw_image() (*matplotlib.backends.backend_template.RendererTemplate method*), 2306
draw_markers() (*matplotlib.backend_bases.RendererBase method*), 2275
draw_markers() (*matplotlib.backends.backend_cairo.RendererCairo method*), 2318
draw_markers() (*matplotlib.backends.backend_pdf.RendererPdf method*), 2331
draw_markers() (*matplotlib.backends.backend_pgf.RendererPgf method*), 2338
draw_markers() (*matplotlib.backends.backend_ps.RendererPS method*), 2343
draw_markers() (*matplotlib.backends.backend_svg.RendererSVG method*), 2353
draw_markers() (*matplotlib.patheffects.PathEffectRenderer method*), 3205
draw_mathtext() (*matplotlib.backends.backend_agg.RendererAgg method*), 2312
draw_mathtext() (*matplotlib.backends.backend_pdf.RendererPdf method*), 2332
draw_mathtext() (*matplotlib.backends.backend_ps.RendererPS method*), 2343
draw_pane() (*mpl_toolkits.mplot3d.axis3d.Axis method*), 3875
draw_path() (*matplotlib.backend_bases.RendererBase method*), 2275
draw_path() (*matplotlib.backends.backend_agg.RendererAgg method*), 2312
draw_path() (*matplotlib.backends.backend_cairo.RendererCairo method*), 2319
draw_path() (*matplotlib.backends.backend_pdf.RendererPdf method*), 2332
draw_path() (*matplotlib.backends.backend_pgf.RendererPgf method*), 2339
draw_path() (*matplotlib.backends.backend_ps.RendererPS method*), 2344
draw_path() (*matplotlib.backends.backend_svg.RendererSVG method*), 2353
draw_path() (*matplotlib.backends.backend_template.RendererTemplate method*), 2307
draw_path() (*matplotlib.patheffects.AbstractPathEffect method*), 3204
draw_path() (*matplotlib.patheffects.PathEffectRenderer method*), 3205
draw_path() (*matplotlib.patheffects.PathPatchEffect method*), 3206
draw_path() (*matplotlib.patheffects.SimpleLineShadow method*), 3207
draw_path() (*matplotlib.patheffects.SimplePatchShadow method*), 3207
draw_path() (*matplotlib.patheffects.Stroke method*), 3207
draw_path() (*matplotlib.patheffects.TickedStroke method*), 3208
draw_path() (*matplotlib.patheffects.withSimplePatchShadow method*), 3209
draw_path() (*matplotlib.patheffects.withStroke method*), 3209
draw_path() (*matplotlib.patheffects.withTickedStroke method*), 3211
draw_path_collection() (*matplotlib.backend_bases.RendererBase method*), 2275
draw_path_collection() (*matplotlib.backends.backend_pdf.RendererPdf method*), 2332
draw_path_collection() (*matplotlib.backends.backend_ps.RendererPS method*), 2344
draw_path_collection() (*matplotlib.backends.backend_svg.RendererSVG method*), 2353
draw_path_collection() (*matplotlib.patheffects.PathEffectRenderer method*), 3205
draw_quad_mesh() (*matplotlib.backend_bases.RendererBase method*), 2276
draw_rect() (*matplotlib.ft2font.FT2Image method*), 2956
draw_rect_filled() (*matplotlib.ft2font.FT2Image method*), 2956
draw_rubberband() (*matplotlib.backend_bases.NavigationToolbar2 method*), 2271
draw_rubberband() (*matplotlib.backend_tools.RubberbandBase method*), 2289

draw_rubberband() (*matplotlib.backends.backend_webagg_core.NavigationToolbar2WebAgg method*), 2360
draw_tex() (*matplotlib.backend_bases.RendererBase method*), 2276
draw_tex() (*matplotlib.backends.backend_agg.RendererAgg method*), 2312
draw_tex() (*matplotlib.backends.backend_pdf.RendererPdf method*), 2332
draw_tex() (*matplotlib.backends.backend_pgf.RendererPgf method*), 2339
draw_tex() (*matplotlib.backends.backend_ps.RendererPS method*), 2344
draw_text() (*matplotlib.backend_bases.RendererBase method*), 2276
draw_text() (*matplotlib.backends.backend_agg.RendererAgg method*), 2313
draw_text() (*matplotlib.backends.backend_cairo.RendererCairo method*), 2319
draw_text() (*matplotlib.backends.backend_pdf.RendererPdf method*), 2333
draw_text() (*matplotlib.backends.backend_pgf.RendererPgf method*), 2339
draw_text() (*matplotlib.backends.backend_ps.RendererPS method*), 2345
draw_text() (*matplotlib.backends.backend_svg.RendererSVG method*), 2354
draw_text() (*matplotlib.backends.backend_template.RendererTemplate method*), 2307
draw_without_rendering() (*matplotlib.figure.Figure method*), 2863
DrawEvent (*class in matplotlib.backend_bases*), 2253
DrawingArea (*class in matplotlib.offsetbox*), 3077
drawon (*matplotlib.widgets.Widget attribute*), 3802
drawStyleKeys (*matplotlib.lines.Line2D attribute*), 3018
drawStyles (*matplotlib.lines.Line2D attribute*), 3018
DrawStyleType (*in module matplotlib.typing*), 3768
Dvi (*class in matplotlib.dviread*), 2788
DviFont (*class in matplotlib.dviread*), 2788
dviFontName() (*matplotlib.backends.backend_pdf.PdfFile method*), 2327
dx (*matplotlib._afm.CompositePart attribute*), 1831
dy (*matplotlib._afm.CompositePart attribute*), 1832

## E

ecdf() (*in module matplotlib.pyplot*), 3340
ecdf() (*matplotlib.axes.Axes method*), 2001
edge_centers (*matplotlib.widgets.RectangleSelector property*), 3793
edges (*matplotlib.table.Table property*), 3653
edges (*matplotlib.tri.Triangulation property*), 3756
effects (*matplotlib.dviread.PsFont attribute*), 2789
embed() (*matplotlib.backends.backend_svg.XMLWriter method*), 2356
Ellipse (*class in matplotlib.patches*), 3135
EllipseCollection (*class in matplotlib.collections*), 2455
EllipseSelector (*class in matplotlib.widgets*), 3778
embedTTF() (*matplotlib.backends.backend_pdf.PdfFile method*), 2327
empty() (*matplotlib.cbook.Stack method*), 2374
enable() (*matplotlib.backend_tools.AxisScaleBase method*), 2288
enable() (*matplotlib.backend_tools.ToolToggleBase method*), 2298
enable() (*matplotlib.backend_tools.ZoomPanBase method*), 2300
encode_string() (*matplotlib.backends.backend_pdf.RendererPdf method*), 2334
encoding (*matplotlib.dviread.PsFont attribute*), 2789
end() (*matplotlib.backends.backend_pdf.Stream method*), 2334
end() (*matplotlib.backends.backend_svg.XMLWriter method*), 2356
end_pan() (*matplotlib.axes.Axes method*), 2201
end_pan() (*matplotlib.projections.geo.GeoAxes method*), 3565
end_pan() (*matplotlib.projections.polar.PolarAxes method*), 3536
end_text (*matplotlib.backends.backend_pdf.Op attribute*), 2324
endpath (*matplotlib.backends.backend_pdf.Op attribute*), 2324
endStream() (*matplotlib.backends.backend_pdf.PdfFile method*), 2327
ENG_PREFIXES (*matplotlib.ticker.EngFormatter attribute*), 3694
EngFormatter (*class in matplotlib.ticker*), 3693
environment variable
    HOME, 9, 20
    MESONPY_EDITABLE_VERBOSE, 4650
    MPLBACKEND, 9, 59, 60, 1811, 4415, 4477
    MPLCONFIGDIR, 9, 21, 292, 1811
    MPLSETUPCFG, 9, 4209
    PATH, 9, 433, 436, 437, 439, 440
    PYTHONPATH, 9, 10
    QT_API, 9, 62, 1697, 2350, 4185, 4326
errorbar() (*in module matplotlib.pyplot*), 3257
errorbar() (*matplotlib.axes.Axes method*), 1919
errorbar() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3831
ErrorbarContainer (*class in matplotlib.container*), 2754
evaluate() (*matplotlib.mlab.GaussianKDE method*), 3048
Event (*class in matplotlib.backend_bases*), 2254
EventCollection (*class in matplotlib.collections*), 2477
eventplot() (*in module matplotlib.pyplot*), 3289
eventplot() (*matplotlib.axes.Axes method*), 1951
events (*matplotlib.backend_bases.FigureCanvasBase attribute*), 2255
eventson (*matplotlib.widgets.Widget attribute*), 3802

execute() (*matplotlib.layout_engine.ConstrainedLayoutEngine method*), 2986
execute() (*matplotlib.layout_engine.LayoutEngine method*), 2987
execute() (*matplotlib.layout_engine.PlaceHolderLayoutEngine method*), 2988
execute() (*matplotlib.layout_engine.TightLayoutEngine method*), 2988
expanded() (*matplotlib.transforms.BboxBase method*), 3732
extend_positions() (*matplotlib.collections.EventCollection method*), 2480
extents (*matplotlib.transforms.BboxBase property*), 3732
extents (*matplotlib.widgets.RectangleSelector property*), 3793
extents (*matplotlib.widgets.SpanSelector property*), 3798
extra (*matplotlib.backends.backend_pdf.Stream attribute*), 2335
ExtremeFinderCycle (*class in mpl_toolkits.axisartist.angle_helper*), 3991
ExtremeFinderFixed (*class in mpl_toolkits.axisartist.floating_axes*), 4041
ExtremeFinderSimple (*class in mpl_toolkits.axisartist.grid_finder*), 4046

## F

face_flags (*matplotlib.ft2font.FT2Font attribute*), 2953
family_name (*matplotlib._afm.AFM property*), 1830
family_name (*matplotlib.ft2font.FT2Font attribute*), 2953
FancyArrow (*class in matplotlib.patches*), 3140
FancyArrowPatch (*class in matplotlib.patches*), 3143
FancyBboxPatch (*class in matplotlib.patches*), 3152
FFMpegBase (*class in matplotlib.animation*), 1866
FFMpegFileWriter (*class in matplotlib.animation*), 1853
FFMpegWriter (*class in matplotlib.animation*), 1849
fget (*matplotlib._api.classproperty property*), 3804
figaspect() (*in module matplotlib.figure*), 2940
figimage() (*in module matplotlib.pyplot*), 3394
figimage() (*matplotlib.figure.Figure method*), 2858
figlegend() (*in module matplotlib.pyplot*), 3421
figmplnode (*class in matplotlib.sphinxext.figmpl_directive*), 3637
fignum_exists() (*in module matplotlib.pyplot*), 3216
figtext() (*in module matplotlib.pyplot*), 3414
Figure (*class in matplotlib.figure*), 2793
figure (*matplotlib.backend_managers.ToolManager property*), 2285
figure (*matplotlib.backend_tools.ToolBase property*), 2291
figure() (*in module matplotlib.pyplot*), 3217
FigureBase (*class in matplotlib.figure*), 2904
FigureCanvas (*in module matplotlib.backends.backend_agg*), 2309
FigureCanvas (*in module matplotlib.backends.backend_cairo*), 2315
FigureCanvas (*in module matplotlib.backends.backend_nbagg*), 2321
FigureCanvas (*in module matplotlib.backends.backend_pdf*), 2322
FigureCanvas (*in module matplotlib.backends.backend_pgf*), 2335
FigureCanvas (*in module matplotlib.backends.backend_ps*), 2341
FigureCanvas (*in module matplotlib.backends.backend_svg*), 2351
FigureCanvas (*in module matplotlib.backends.backend_template*), 2304
FigureCanvas (*in module matplotlib.backends.backend_tkagg*), 2357
FigureCanvas (*in module matplotlib.backends.backend_tkcairo*), 2357
FigureCanvas (*in module matplotlib.backends.backend_webagg*), 2362
FigureCanvas (*in module matplotlib.backends.backend_webagg_core*), 2358
FigureCanvasAgg (*class in matplotlib.backends.backend_agg*), 2309
FigureCanvasBase (*class in matplotlib.backend_bases*), 2254
FigureCanvasCairo (*class in matplotlib.backends.backend_cairo*), 2315
FigureCanvasNbAgg (*class in matplotlib.backends.backend_nbagg*), 2321
FigureCanvasPdf (*class in matplotlib.backends.backend_pdf*), 2322
FigureCanvasPgf (*class in matplotlib.backends.backend_pgf*), 2335
FigureCanvasPS (*class in matplotlib.backends.backend_ps*), 2341
FigureCanvasSVG (*class in matplotlib.backends.backend_svg*), 2351
FigureCanvasTemplate (*class in matplotlib.backends.backend_template*), 2304
FigureCanvasTkAgg (*class in matplotlib.backends.backend_tkagg*), 2357
FigureCanvasTkCairo (*class in matplotlib.backends.backend_tkcairo*), 2357
FigureCanvasWebAgg (*class in matplotlib.backends.backend_webagg*), 2362
FigureCanvasWebAggCore (*class in matplotlib.backends.backend_webagg_core*), 2358
FigureImage (*class in matplotlib.image*), 2973
FigureManager (*in module matplotlib.backends.backend_nbagg*), 2321
FigureManager (*in module matplotlib.backends.backend_template*), 2305
FigureManager (*in module matplotlib.backends.backend_webagg*), 2362
FigureManager (*in module matplotlib.backends.backend_webagg_core*), 2360
FigureManagerBase (*class in matplotlib.backend_bases*), 2261
FigureManagerNbAgg (*class in matplotlib.backends.backend_nbagg*), 2321
FigureManagerTemplate (*class in matplotlib.backends.backend_template*), 2305

FigureManagerWebAgg (*class in matplotlib.backends.backend_webagg*), 2362
FigureManagerWebAgg (*class in matplotlib.backends.backend_webagg_core*), 2360
FigureMpl (*class in matplotlib.sphinxext.figmpl_directive*), 3636
file (*matplotlib.backends.backend_pdf.Stream attribute*), 2335
file_requires_unicode() (*in module matplotlib.cbook*), 2377
FileMovieWriter (*class in matplotlib.animation*), 1863
filename (*matplotlib.dviread.PsFont attribute*), 2789
filetypes (*matplotlib.backend_bases.FigureCanvasBase attribute*), 2255
filetypes (*matplotlib.backends.backend_pdf.FigureCanvasPdf attribute*), 2322
filetypes (*matplotlib.backends.backend_pgf.FigureCanvasPgf attribute*), 2335
filetypes (*matplotlib.backends.backend_ps.FigureCanvasPS attribute*), 2341
filetypes (*matplotlib.backends.backend_svg.FigureCanvasSVG attribute*), 2351
filetypes (*matplotlib.backends.backend_template.FigureCanvasTemplate attribute*), 2305
fill (*matplotlib.backends.backend_pdf.Op attribute*), 2324
fill (*matplotlib.patches.Patch property*), 3162
fill() (*in module matplotlib.pyplot*), 3300
fill() (*matplotlib.axes.Axes method*), 1962
fill() (*matplotlib.backends.backend_pdf.GraphicsContextPdf method*), 2323
fill_between() (*in module matplotlib.pyplot*), 3272
fill_between() (*matplotlib.axes.Axes method*), 1934
fill_betweenx() (*in module matplotlib.pyplot*), 3275
fill_betweenx() (*matplotlib.axes.Axes method*), 1937
fill_stroke (*matplotlib.backends.backend_pdf.Op attribute*), 2325
fillcolor_cmd() (*matplotlib.backends.backend_pdf.GraphicsContextPdf method*), 2323
filled (*matplotlib.hatch.Shapes attribute*), 2966
filled (*matplotlib.hatch.SmallFilledCircles attribute*), 2966
filled (*matplotlib.hatch.Stars attribute*), 2967
filled_markers (*matplotlib.lines.Line2D attribute*), 3018
filled_markers (*matplotlib.markers.MarkerStyle attribute*), 3041
fillStyles (*matplotlib.lines.Line2D attribute*), 3018
fillstyles (*matplotlib.markers.MarkerStyle attribute*), 3041
FillStyleType (*in module matplotlib.typing*), 3768
final_argument_whitespace (*matplotlib.sphinxext.figmpl_directive.FigureMpl attribute*), 3636
final_argument_whitespace (*matplotlib.sphinxext.mathmpl.MathDirective attribute*), 3631
final_argument_whitespace (*matplotlib.sphinxext.plot_directive.PlotDirective attribute*), 3634
finalize() (*matplotlib.backends.backend_pdf.GraphicsContextPdf method*), 2323
finalize() (*matplotlib.backends.backend_pdf.PdfFile method*), 2327
finalize() (*matplotlib.backends.backend_pdf.RendererPdf method*), 2334
finalize() (*matplotlib.backends.backend_svg.RendererSVG method*), 2355
finalize_offset() (*matplotlib.legend.DraggableLegend method*), 2990
finalize_offset() (*matplotlib.offsetbox.DraggableBase method*), 3076
find_all() (*matplotlib.RcParams method*), 1823
find_bezier_t_intersecting_with_closedpath() (*in module matplotlib.bezier*), 2366
find_control_points() (*in module matplotlib.bezier*), 2366
find_nearest_contour() (*matplotlib.contour.ContourSet method*), 2762
find_tex_file() (*in module matplotlib.dviread*), 2792
findfont() (*in module matplotlib.font_manager*), 2948
findfont() (*matplotlib.font_manager.FontManager method*), 2942
findobj() (*in module matplotlib.pyplot*), 3518
findobj() (*matplotlib.artist.Artist method*), 1891
findobj() (*matplotlib.axes.Axes method*), 2204
findobj() (*matplotlib.collections.AsteriskPolygonCollection method*), 2391
findobj() (*matplotlib.collections.CircleCollection method*), 2412
findobj() (*matplotlib.collections.Collection method*), 2436
findobj() (*matplotlib.collections.EllipseCollection method*), 2457
findobj() (*matplotlib.collections.EventCollection method*), 2480
findobj() (*matplotlib.collections.LineCollection method*), 2503
findobj() (*matplotlib.collections.PatchCollection method*), 2525
findobj() (*matplotlib.collections.PathCollection method*), 2546
findobj() (*matplotlib.collections.PolyCollection method*), 2569
findobj() (*matplotlib.collections.PolyQuadMesh method*), 2592
findobj() (*matplotlib.collections.QuadMesh method*), 2617
findobj() (*matplotlib.collections.RegularPolyCollection method*), 2639
findobj() (*matplotlib.collections.StarPolygonCollection method*), 2661
findobj() (*matplotlib.collections.TriMesh method*), 2684

findSystemFonts() (*in module matplotlib.font_manager*), 2948
finish() (*matplotlib.animation.AbstractMovieWriter method*), 1859
finish() (*matplotlib.animation.FileMovieWriter method*), 1865
finish() (*matplotlib.animation.HTMLWriter method*), 1847
finish() (*matplotlib.animation.MovieWriter method*), 1862
finish() (*matplotlib.animation.PillowWriter method*), 1845
finish() (*matplotlib.sankey.Sankey method*), 3613
fix_minus() (*matplotlib.ticker.Formatter static method*), 3696
Fixed (*class in mpl_toolkits.axes_grid1.axes_size*), 3950
Fixed (*mpl_toolkits.axisartist.axislines.AxisArtistHelper attribute*), 4038
Fixed (*mpl_toolkits.axisartist.axislines.AxisArtistHelperRectilinear attribute*), 4038
fixed_dpi (*matplotlib.backend_bases.FigureCanvasBase attribute*), 2255
fixed_dpi (*matplotlib.backends.backend_pdf.FigureCanvasPdf attribute*), 2322
fixed_dpi (*matplotlib.backends.backend_ps.FigureCanvasPS attribute*), 2342
fixed_dpi (*matplotlib.backends.backend_svg.FigureCanvasSVG attribute*), 2351
FixedAxisArtistHelper (*class in mpl_toolkits.axisartist.floating_axes*), 4042
FixedAxisArtistHelper (*class in mpl_toolkits.axisartist.grid_helper_curvelinear*), 4049
FixedAxisArtistHelperRectilinear (*class in mpl_toolkits.axisartist.axislines*), 4039
FixedFormatter (*class in matplotlib.ticker*), 3695
FixedLocator (*class in matplotlib.ticker*), 3695
FixedLocator (*class in mpl_toolkits.axisartist.grid_finder*), 4046
flatten() (*in module matplotlib.cbook*), 2377
flipy() (*matplotlib.backend_bases.RendererBase method*), 2277
flipy() (*matplotlib.backends.backend_pgf.RendererPgf method*), 2340
flipy() (*matplotlib.backends.backend_svg.RendererSVG method*), 2355
flipy() (*matplotlib.backends.backend_template.RendererTemplate method*), 2307
Floating (*mpl_toolkits.axisartist.axislines.AxisArtistHelper attribute*), 4038
Floating (*mpl_toolkits.axisartist.axislines.AxisArtistHelperRectilinear attribute*), 4038
FloatingAxes (*in module mpl_toolkits.axisartist.floating_axes*), 4042
floatingaxes_class_factory() (*in module mpl_toolkits.axisartist.floating_axes*), 4045
FloatingAxesBase (*class in mpl_toolkits.axisartist.floating_axes*), 4042
FloatingAxisArtistHelper (*class in mpl_toolkits.axisartist.floating_axes*), 4043
FloatingAxisArtistHelper (*class in mpl_toolkits.axisartist.grid_helper_curvelinear*), 4050
FloatingAxisArtistHelperRectilinear (*class in mpl_toolkits.axisartist.axislines*), 4039
FloatingSubplot (*in module mpl_toolkits.axisartist.floating_axes*), 4043
flush() (*matplotlib.backends.backend_svg.XMLWriter method*), 2356
flush_events() (*matplotlib.backend_bases.FigureCanvasBase method*), 2256
fmt_d (*mpl_toolkits.axisartist.angle_helper.FormatterDMS attribute*), 3992
fmt_d (*mpl_toolkits.axisartist.angle_helper.FormatterHMS attribute*), 3993
fmt_d_m (*mpl_toolkits.axisartist.angle_helper.FormatterDMS attribute*), 3993
fmt_d_m (*mpl_toolkits.axisartist.angle_helper.FormatterHMS attribute*), 3993
fmt_d_m_partial (*mpl_toolkits.axisartist.angle_helper.FormatterDMS attribute*), 3993
fmt_d_m_partial (*mpl_toolkits.axisartist.angle_helper.FormatterHMS attribute*), 3993
fmt_d_ms (*mpl_toolkits.axisartist.angle_helper.FormatterDMS attribute*), 3993
fmt_d_ms (*mpl_toolkits.axisartist.angle_helper.FormatterHMS attribute*), 3993
fmt_ds (*mpl_toolkits.axisartist.angle_helper.FormatterDMS attribute*), 3993
fmt_ds (*mpl_toolkits.axisartist.angle_helper.FormatterHMS attribute*), 3993
fmt_s_partial (*mpl_toolkits.axisartist.angle_helper.FormatterDMS attribute*), 3993
fmt_s_partial (*mpl_toolkits.axisartist.angle_helper.FormatterHMS attribute*), 3993
fmt_ss_partial (*mpl_toolkits.axisartist.angle_helper.FormatterDMS attribute*), 3993
fmt_ss_partial (*mpl_toolkits.axisartist.angle_helper.FormatterHMS attribute*), 3993
fname (*matplotlib.ft2font.FT2Font attribute*), 2953
FONT_SCALE (*matplotlib.text.TextToPath attribute*), 3687
FontEntry (*class in matplotlib.font_manager*), 2951
FontManager (*class in matplotlib.font_manager*), 2941
fontManager (*in module matplotlib.font_manager*), 2951
fontName() (*matplotlib.backends.backend_pdf.PdfFile method*), 2327

FontProperties (*class in matplotlib.font_manager*), 2944
FONTSIZE (*matplotlib.table.Table attribute*), 3651
format_coord() (*matplotlib.axes.Axes method*), 2201
format_coord() (*matplotlib.projections.geo.GeoAxes method*), 3565
format_coord() (*matplotlib.projections.polar.PolarAxes method*), 3536
format_coord() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3866
format_cursor_data() (*matplotlib.artist.Artist method*), 1871
format_cursor_data() (*matplotlib.axes.Axes method*), 2201
format_cursor_data() (*matplotlib.collections.AsteriskPolygonCollection method*), 2391
format_cursor_data() (*matplotlib.collections.CircleCollection method*), 2413
format_cursor_data() (*matplotlib.collections.Collection method*), 2436
format_cursor_data() (*matplotlib.collections.EllipseCollection method*), 2458
format_cursor_data() (*matplotlib.collections.EventCollection method*), 2480
format_cursor_data() (*matplotlib.collections.LineCollection method*), 2504
format_cursor_data() (*matplotlib.collections.PatchCollection method*), 2526
format_cursor_data() (*matplotlib.collections.PathCollection method*), 2547
format_cursor_data() (*matplotlib.collections.PolyCollection method*), 2570
format_cursor_data() (*matplotlib.collections.PolyQuadMesh method*), 2593
format_cursor_data() (*matplotlib.collections.QuadMesh method*), 2617
format_cursor_data() (*matplotlib.collections.RegularPolyCollection method*), 2639
format_cursor_data() (*matplotlib.collections.StarPolygonCollection method*), 2661
format_cursor_data() (*matplotlib.collections.TriMesh method*), 2684
format_data() (*matplotlib.ticker.Formatter method*), 3696
format_data() (*matplotlib.ticker.LogFormatter method*), 3700
format_data() (*matplotlib.ticker.ScalarFormatter method*), 3710
format_data_short() (*matplotlib.dates.ConciseDateFormatter method*), 2775
format_data_short() (*matplotlib.ticker.Formatter method*), 3696
format_data_short() (*matplotlib.ticker.LogFormatter method*), 3700
format_data_short() (*matplotlib.ticker.LogitFormatter method*), 3703
format_data_short() (*matplotlib.ticker.ScalarFormatter method*), 3710
format_eng() (*matplotlib.ticker.EngFormatter method*), 3694
format_pct() (*matplotlib.ticker.PercentFormatter method*), 3708
format_shortcut() (*matplotlib.backend_tools.ToolHelpBase static method*), 2294
format_ticks() (*matplotlib.category.StrCategoryFormatter method*), 2369
format_ticks() (*matplotlib.dates.ConciseDateFormatter method*), 2775
format_ticks() (*matplotlib.ticker.Formatter method*), 3696
format_xdata() (*matplotlib.axes.Axes method*), 2202
format_ydata() (*matplotlib.axes.Axes method*), 2202
format_zdata() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3866
FormatStrFormatter (*class in matplotlib.ticker*), 3696
Formatter (*class in matplotlib.ticker*), 3696
formatter (*matplotlib.colorbar.Colorbar property*), 2707
FormatterDMS (*class in mpl_toolkits.axisartist.angle_helper*), 3992
FormatterHMS (*class in mpl_toolkits.axisartist.angle_helper*), 3993
FormatterPrettyPrint (*class in mpl_toolkits.axisartist.grid_finder*), 4047
FORWARD (*matplotlib.backend_bases.MouseButton attribute*), 2269
forward() (*matplotlib.backend_bases.NavigationToolbar2 method*), 2271
forward() (*matplotlib.backend_tools.ToolViewsPositions method*), 2298
forward() (*matplotlib.cbook.Stack method*), 2374
Fraction (*class in mpl_toolkits.axes_grid1.axes_size*), 3951
frame_format (*matplotlib.animation.FileMovieWriter property*), 1865
frame_size (*matplotlib.animation.AbstractMovieWriter property*), 1859
frameon (*matplotlib.figure.FigureBase property*), 2919
from_any() (*in module mpl_toolkits.axes_grid1.axes_size*), 3952
from_bounds() (*matplotlib.transforms.Bbox static method*), 3727
from_dict() (*matplotlib.spines.Spines class method*), 3642
from_extents() (*matplotlib.transforms.Bbox static method*), 3727
from_levels_and_colors() (*in module matplotlib.colors*), 2748

from_list() (*matplotlib.colors.LinearSegmentedColormap static method*), 2735
from_values() (*matplotlib.transforms.Affine2D static method*), 3720
frozen() (*matplotlib.transforms.Affine2DBase method*), 3722
frozen() (*matplotlib.transforms.Bbox method*), 3727
frozen() (*matplotlib.transforms.BboxBase method*), 3733
frozen() (*matplotlib.transforms.BlendedGenericTransform method*), 3738
frozen() (*matplotlib.transforms.CompositeGenericTransform method*), 3740
frozen() (*matplotlib.transforms.IdentityTransform method*), 3742
frozen() (*matplotlib.transforms.TransformNode method*), 3750
frozen() (*matplotlib.transforms.TransformWrapper method*), 3751
FT2Font (*class in matplotlib.ft2font*), 2951
FT2Image (*class in matplotlib.ft2font*), 2956
full_screen_toggle() (*matplotlib.backend_bases.FigureManagerBase method*), 2263
fully_contains() (*matplotlib.transforms.BboxBase method*), 3733
fully_contains() (*matplotlib.transforms.TransformedBbox method*), 3752
fully_containsx() (*matplotlib.transforms.BboxBase method*), 3733
fully_containsy() (*matplotlib.transforms.BboxBase method*), 3733
fully_overlaps() (*matplotlib.transforms.BboxBase method*), 3733
FuncAnimation (*class in matplotlib.animation*), 1837
FuncFormatter (*class in matplotlib.ticker*), 3696
FuncNorm (*class in matplotlib.colors*), 2722
FuncScale (*class in matplotlib.scale*), 3616
FuncScaleLog (*class in matplotlib.scale*), 3617
FuncTransform (*class in matplotlib.scale*), 3617

## G

GaussianKDE (*class in matplotlib.mlab*), 3047
gca() (*in module matplotlib.pyplot*), 3224
gca() (*matplotlib.figure.Figure method*), 2861
gca() (*matplotlib.figure.FigureBase method*), 2919
gcf() (*in module matplotlib.pyplot*), 3225
gci() (*in module matplotlib.pyplot*), 3481
GeoAxes (*class in matplotlib.projections.geo*), 3562
GeoAxes.ThetaFormatter (*class in matplotlib.projections.geo*), 3564
geometry (*matplotlib.widgets.RectangleSelector property*), 3793
get() (*in module matplotlib.artist*), 1898
get() (*in module matplotlib.pyplot*), 3519
get() (*matplotlib.backends.backend_webagg.WebAggApplication.AllFiguresPage method*), 2363
get() (*matplotlib.backends.backend_webagg.WebAggApplication.Download method*), 2363
get() (*matplotlib.backends.backend_webagg.WebAggApplication.FavIcon method*), 2363
get() (*matplotlib.backends.backend_webagg.WebAggApplication.MplJs method*), 2363
get() (*matplotlib.backends.backend_webagg.WebAggApplication.SingleFigurePage method*), 2363
get() (*matplotlib.layout_engine.ConstrainedLayoutEngine method*), 2986
get() (*matplotlib.layout_engine.LayoutEngine method*), 2987
get() (*matplotlib.layout_engine.PlaceHolderLayoutEngine method*), 2988
get() (*matplotlib.layout_engine.TightLayoutEngine method*), 2989
get_aa() (*matplotlib.collections.AsteriskPolygonCollection method*), 2391
get_aa() (*matplotlib.collections.CircleCollection method*), 2413
get_aa() (*matplotlib.collections.Collection method*), 2436
get_aa() (*matplotlib.collections.EllipseCollection method*), 2458
get_aa() (*matplotlib.collections.EventCollection method*), 2481
get_aa() (*matplotlib.collections.LineCollection method*), 2504
get_aa() (*matplotlib.collections.PatchCollection method*), 2526
get_aa() (*matplotlib.collections.PathCollection method*), 2547
get_aa() (*matplotlib.collections.PolyCollection method*), 2570
get_aa() (*matplotlib.collections.PolyQuadMesh method*), 2593
get_aa() (*matplotlib.collections.QuadMesh method*), 2617
get_aa() (*matplotlib.collections.RegularPolyCollection method*), 2639
get_aa() (*matplotlib.collections.StarPolygonCollection method*), 2661
get_aa() (*matplotlib.collections.TriMesh method*), 2684
get_aa() (*matplotlib.lines.Line2D method*), 3018
get_aa() (*matplotlib.patches.Patch method*), 3162
get_active() (*matplotlib.widgets.Widget method*), 3802
get_adjustable() (*matplotlib.axes.Axes method*), 2167
get_affine() (*matplotlib.transforms.AffineBase method*), 3723
get_affine() (*matplotlib.transforms.BlendedGenericTransform method*), 3738
get_affine() (*matplotlib.transforms.CompositeGenericTransform method*), 3740
get_affine() (*matplotlib.transforms.IdentityTransform method*), 3742

get_affine() (*matplotlib.transforms.Transform method*), 3747
get_affine() (*matplotlib.transforms.TransformedPath method*), 3752
get_agg_filter() (*matplotlib.artist.Artist method*), 1888
get_agg_filter() (*matplotlib.collections.AsteriskPolygonCollection method*), 2391
get_agg_filter() (*matplotlib.collections.CircleCollection method*), 2413
get_agg_filter() (*matplotlib.collections.Collection method*), 2436
get_agg_filter() (*matplotlib.collections.EllipseCollection method*), 2458
get_agg_filter() (*matplotlib.collections.EventCollection method*), 2481
get_agg_filter() (*matplotlib.collections.LineCollection method*), 2504
get_agg_filter() (*matplotlib.collections.PatchCollection method*), 2526
get_agg_filter() (*matplotlib.collections.PathCollection method*), 2547
get_agg_filter() (*matplotlib.collections.PolyCollection method*), 2570
get_agg_filter() (*matplotlib.collections.PolyQuadMesh method*), 2593
get_agg_filter() (*matplotlib.collections.QuadMesh method*), 2617
get_agg_filter() (*matplotlib.collections.RegularPolyCollection method*), 2639
get_agg_filter() (*matplotlib.collections.StarPolygonCollection method*), 2661
get_agg_filter() (*matplotlib.collections.TriMesh method*), 2685
get_aliases() (*matplotlib.artist.ArtistInspector method*), 1901
get_alignment() (*matplotlib.legend.Legend method*), 2996
get_alpha() (*matplotlib.artist.Artist method*), 1883
get_alpha() (*matplotlib.backend_bases.GraphicsContextBase method*), 2264
get_alpha() (*matplotlib.cm.ScalarMappable method*), 2386
get_alpha() (*matplotlib.collections.AsteriskPolygonCollection method*), 2391
get_alpha() (*matplotlib.collections.CircleCollection method*), 2413
get_alpha() (*matplotlib.collections.Collection method*), 2436
get_alpha() (*matplotlib.collections.EllipseCollection method*), 2458
get_alpha() (*matplotlib.collections.EventCollection method*), 2481
get_alpha() (*matplotlib.collections.LineCollection method*), 2504
get_alpha() (*matplotlib.collections.PatchCollection method*), 2526
get_alpha() (*matplotlib.collections.PathCollection method*), 2547
get_alpha() (*matplotlib.collections.PolyCollection method*), 2570
get_alpha() (*matplotlib.collections.PolyQuadMesh method*), 2593
get_alpha() (*matplotlib.collections.QuadMesh method*), 2617
get_alpha() (*matplotlib.collections.RegularPolyCollection method*), 2639
get_alpha() (*matplotlib.collections.StarPolygonCollection method*), 2661
get_alpha() (*matplotlib.collections.TriMesh method*), 2685
get_alt_path() (*matplotlib.markers.MarkerStyle method*), 3041
get_alt_transform() (*matplotlib.markers.MarkerStyle method*), 3041
get_anchor() (*matplotlib.axes.Axes method*), 2192
get_anchor() (*mpl_toolkits.axes_grid1.axes_divider.AxesDivider method*), 3931
get_anchor() (*mpl_toolkits.axes_grid1.axes_divider.Divider method*), 3933
get_angle() (*matplotlib._afm.AFM method*), 1830
get_angle() (*matplotlib.patches.Annulus method*), 3098
get_angle() (*matplotlib.patches.Ellipse method*), 3136
get_angle() (*matplotlib.patches.Rectangle method*), 3181
get_angles() (*matplotlib.collections.EllipseCollection method*), 2458
get_animated() (*matplotlib.artist.Artist method*), 1882
get_animated() (*matplotlib.collections.AsteriskPolygonCollection method*), 2392
get_animated() (*matplotlib.collections.CircleCollection method*), 2413
get_animated() (*matplotlib.collections.Collection method*), 2437
get_animated() (*matplotlib.collections.EllipseCollection method*), 2458
get_animated() (*matplotlib.collections.EventCollection method*), 2481
get_animated() (*matplotlib.collections.LineCollection method*), 2504
get_animated() (*matplotlib.collections.PatchCollection method*), 2526
get_animated() (*matplotlib.collections.PathCollection method*), 2547
get_animated() (*matplotlib.collections.PolyCollection method*), 2570
get_animated() (*matplotlib.collections.PolyQuadMesh method*), 2593
get_animated() (*matplotlib.collections.QuadMesh method*), 2618
get_animated()

(*matplotlib.collections.RegularPolyCollection method*), 2640
get_animated() (*matplotlib.collections.StarPolygonCollection method*), 2662
get_animated() (*matplotlib.collections.TriMesh method*), 2685
get_anncoords() (*matplotlib.text.Annotation method*), 3682
get_annotation_clip() (*matplotlib.patches.ConnectionPatch method*), 3130
get_antialiased() (*matplotlib.backend_bases.GraphicsContextBase method*), 2264
get_antialiased() (*matplotlib.backends.backend_cairo.GraphicsContextCairo method*), 2316
get_antialiased() (*matplotlib.collections.AsteriskPolygonCollection method*), 2392
get_antialiased() (*matplotlib.collections.CircleCollection method*), 2413
get_antialiased() (*matplotlib.collections.Collection method*), 2437
get_antialiased() (*matplotlib.collections.EllipseCollection method*), 2458
get_antialiased() (*matplotlib.collections.EventCollection method*), 2481
get_antialiased() (*matplotlib.collections.LineCollection method*), 2504
get_antialiased() (*matplotlib.collections.PatchCollection method*), 2526
get_antialiased() (*matplotlib.collections.PathCollection method*), 2547
get_antialiased() (*matplotlib.collections.PolyCollection method*), 2570
get_antialiased() (*matplotlib.collections.PolyQuadMesh method*), 2593
get_antialiased() (*matplotlib.collections.QuadMesh method*), 2618
get_antialiased() (*matplotlib.collections.RegularPolyCollection method*), 2640
get_antialiased() (*matplotlib.collections.StarPolygonCollection method*), 2662
get_antialiased() (*matplotlib.collections.TriMesh method*), 2685
get_antialiased() (*matplotlib.lines.Line2D method*), 3018
get_antialiased() (*matplotlib.patches.Patch method*), 3162
get_antialiased() (*matplotlib.text.Text method*), 3664
get_antialiaseds() (*matplotlib.collections.AsteriskPolygonCollection method*), 2392
get_antialiaseds() (*matplotlib.collections.CircleCollection method*), 2413
get_antialiaseds() (*matplotlib.collections.Collection method*), 2437
get_antialiaseds() (*matplotlib.collections.EllipseCollection method*), 2459
get_antialiaseds() (*matplotlib.collections.EventCollection method*), 2481
get_antialiaseds() (*matplotlib.collections.LineCollection method*), 2504
get_antialiaseds() (*matplotlib.collections.PatchCollection method*), 2527
get_antialiaseds() (*matplotlib.collections.PathCollection method*), 2548
get_antialiaseds() (*matplotlib.collections.PolyCollection method*), 2571
get_antialiaseds() (*matplotlib.collections.PolyQuadMesh method*), 2593
get_antialiaseds() (*matplotlib.collections.QuadMesh method*), 2618
get_antialiaseds() (*matplotlib.collections.RegularPolyCollection method*), 2640
get_antialiaseds() (*matplotlib.collections.StarPolygonCollection method*), 2662
get_antialiaseds() (*matplotlib.collections.TriMesh method*), 2685
get_array() (*matplotlib.cm.ScalarMappable method*), 2386
get_array() (*matplotlib.collections.AsteriskPolygonCollection method*), 2392
get_array() (*matplotlib.collections.CircleCollection method*), 2413
get_array() (*matplotlib.collections.Collection method*), 2437
get_array() (*matplotlib.collections.EllipseCollection method*), 2459
get_array() (*matplotlib.collections.EventCollection method*), 2481
get_array() (*matplotlib.collections.LineCollection method*), 2504
get_array() (*matplotlib.collections.PatchCollection method*), 2527
get_array() (*matplotlib.collections.PathCollection*

*method*), 2548
get_array() (*matplotlib.collections.PolyCollection method*), 2571
get_array() (*matplotlib.collections.PolyQuadMesh method*), 2593
get_array() (*matplotlib.collections.QuadMesh method*), 2618
get_array() (*matplotlib.collections.RegularPolyCollection method*), 2640
get_array() (*matplotlib.collections.StarPolygonCollection method*), 2662
get_array() (*matplotlib.collections.TriMesh method*), 2685
get_arrowstyle() (*matplotlib.patches.FancyArrowPatch method*), 3147
get_aspect() (*matplotlib.axes.Axes method*), 2165
get_aspect() (*mpl_toolkits.axes_grid1.axes_divider.AxesDivider method*), 3931
get_aspect() (*mpl_toolkits.axes_grid1.axes_divider.Divider method*), 3933
get_aspect() (*mpl_toolkits.axes_grid1.axes_grid.Grid method*), 3942
get_attribute_from_ref_artist() (*mpl_toolkits.axisartist.axis_artist.AttributeCopier method*), 4004
get_autoscale_on() (*matplotlib.axes.Axes method*), 2161
get_autoscalex_on() (*matplotlib.axes.Axes method*), 2162
get_autoscaley_on() (*matplotlib.axes.Axes method*), 2162
get_autoscalez_on() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3855
get_aux_axes() (*mpl_toolkits.axes_grid1.parasite_axes.HostAxesBase method*), 3986
get_axes() (*matplotlib.figure.Figure method*), 2812
get_axes_locator() (*matplotlib.axes.Axes method*), 2193
get_axes_locator() (*mpl_toolkits.axes_grid1.axes_grid.Grid method*), 3942
get_axes_pad() (*mpl_toolkits.axes_grid1.axes_grid.Grid method*), 3943
get_axis_position() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3872
get_axisbelow() (*matplotlib.axes.Axes method*), 2113
get_axislabel_pos_angle() (*mpl_toolkits.axisartist.axislines.FloatingAxisArtistHelperRectilinear method*), 4039
get_axislabel_pos_angle() (*mpl_toolkits.axisartist.grid_helper_curvelinear.FloatingAxisArtistHelper method*), 4050
get_axislabel_transform() (*mpl_toolkits.axisartist.axislines.FloatingAxisArtistHelperRectilinear method*), 4039
get_axislabel_transform() (*mpl_toolkits.axisartist.grid_helper_curvelinear.FloatingAxisArtistHelper method*), 4050
get_axisline_style() (*mpl_toolkits.axisartist.axis_artist.AxisArtist method*), 4005
get_backend() (*in module matplotlib*), 1812
get_bad() (*matplotlib.colors.Colormap method*), 2733
get_basefile() (*matplotlib.texmanager.TexManager class method*), 3688
get_bbox() (*matplotlib.lines.Line2D method*), 3018
get_bbox() (*matplotlib.offsetbox.AnchoredOffsetbox method*), 3066
get_bbox() (*matplotlib.offsetbox.AuxTransformBox method*), 3074
get_bbox() (*matplotlib.offsetbox.DrawingArea method*), 3077
get_bbox() (*matplotlib.offsetbox.OffsetBox method*), 3081
get_bbox() (*matplotlib.offsetbox.OffsetImage method*), 3084
get_bbox() (*matplotlib.offsetbox.TextArea method*), 3090
get_bbox() (*matplotlib.patches.FancyBboxPatch method*), 3154
get_bbox() (*matplotlib.patches.Rectangle method*), 3181
get_bbox() (*mpl_toolkits.axes_grid1.inset_locator.AnchoredSizeLocator method*), 3956
get_bbox() (*mpl_toolkits.axes_grid1.inset_locator.AnchoredZoomLocator method*), 3959
get_bbox_char() (*matplotlib._afm.AFM method*), 1830
get_bbox_edge_pos() (*mpl_toolkits.axes_grid1.inset_locator.BboxConnector static method*), 3962
get_bbox_header() (*in module matplotlib.backends.backend_ps*), 2346
get_bbox_patch() (*matplotlib.text.Text method*), 3664
get_bbox_to_anchor() (*matplotlib.legend.Legend method*), 2996
get_bbox_to_anchor() (*matplotlib.offsetbox.AnchoredOffsetbox method*), 3066
get_bitmap_offset() (*matplotlib.ft2font.FT2Font method*), 2953
get_bounds() (*matplotlib.spines.Spine method*), 3639
get_box_aspect() (*matplotlib.axes.Axes method*), 2166
get_boxstyle() (*matplotlib.patches.FancyBboxPatch method*), 3154
get_c() (*matplotlib.lines.Line2D method*), 3018
get_c() (*matplotlib.text.Text method*), 3664
get_cachedir() (*in module matplotlib*), 1829
get_canvas_width_height() (*matplotlib.backend_bases.RendererBase method*), 2277
get_canvas_width_height() (*matplotlib.backends.backend_agg.RendererAgg method*), 2313
get_canvas_width_height() (*matplotlib.backends.backend_cairo.RendererCairo method*), 2319

get_canvas_width_height() (*matplotlib.backends.backend_pgf.RendererPgf method*), 2340
get_canvas_width_height() (*matplotlib.backends.backend_svg.RendererSVG method*), 2355
get_canvas_width_height() (*matplotlib.backends.backend_template.RendererTemplate method*), 2307
get_capheight() (*matplotlib._afm.AFM method*), 1830
get_capstyle() (*matplotlib.backend_bases.GraphicsContextBase method*), 2264
get_capstyle() (*matplotlib.collections.AsteriskPolygonCollection method*), 2392
get_capstyle() (*matplotlib.collections.CircleCollection method*), 2413
get_capstyle() (*matplotlib.collections.Collection method*), 2437
get_capstyle() (*matplotlib.collections.EllipseCollection method*), 2459
get_capstyle() (*matplotlib.collections.EventCollection method*), 2481
get_capstyle() (*matplotlib.collections.LineCollection method*), 2504
get_capstyle() (*matplotlib.collections.PatchCollection method*), 2527
get_capstyle() (*matplotlib.collections.PathCollection method*), 2548
get_capstyle() (*matplotlib.collections.PolyCollection method*), 2571
get_capstyle() (*matplotlib.collections.PolyQuadMesh method*), 2593
get_capstyle() (*matplotlib.collections.QuadMesh method*), 2618
get_capstyle() (*matplotlib.collections.RegularPolyCollection method*), 2640
get_capstyle() (*matplotlib.collections.StarPolygonCollection method*), 2662
get_capstyle() (*matplotlib.collections.TriMesh method*), 2685
get_capstyle() (*matplotlib.markers.MarkerStyle method*), 3041
get_capstyle() (*matplotlib.patches.Patch method*), 3162
get_celld() (*matplotlib.table.Table method*), 3653
get_center() (*matplotlib.patches.Annulus method*), 3098
get_center() (*matplotlib.patches.Ellipse method*), 3136
get_center() (*matplotlib.patches.Rectangle method*), 3181
get_char_index() (*matplotlib.ft2font.FT2Font method*), 2953
get_charmap() (*matplotlib.ft2font.FT2Font method*), 2953
get_checked_labels() (*matplotlib.widgets.CheckButtons method*), 3776
get_child() (*matplotlib.offsetbox.AnchoredOffsetbox method*), 3066
get_children() (*matplotlib.artist.Artist method*), 1891
get_children() (*matplotlib.axes.Axes method*), 2204
get_children() (*matplotlib.collections.AsteriskPolygonCollection method*), 2392
get_children() (*matplotlib.collections.CircleCollection method*), 2414
get_children() (*matplotlib.collections.Collection method*), 2437
get_children() (*matplotlib.collections.EllipseCollection method*), 2459
get_children() (*matplotlib.collections.EventCollection method*), 2481
get_children() (*matplotlib.collections.LineCollection method*), 2505
get_children() (*matplotlib.collections.PatchCollection method*), 2527
get_children() (*matplotlib.collections.PathCollection method*), 2548
get_children() (*matplotlib.collections.PolyCollection method*), 2571
get_children() (*matplotlib.collections.PolyQuadMesh method*), 2594
get_children() (*matplotlib.collections.QuadMesh method*), 2618
get_children() (*matplotlib.collections.RegularPolyCollection method*), 2640
get_children() (*matplotlib.collections.StarPolygonCollection method*), 2662
get_children() (*matplotlib.collections.TriMesh method*), 2685
get_children() (*matplotlib.container.Container method*), 2754
get_children() (*matplotlib.figure.Figure method*), 2858
get_children() (*matplotlib.figure.FigureBase method*), 2919
get_children() (*matplotlib.figure.SubFigure method*), 2901
get_children() (*matplotlib.legend.Legend method*), 2996
get_children() (*matplotlib.offsetbox.AnchoredOffsetbox method*), 3066
get_children() (*matplotlib.offsetbox.AnnotationBbox method*), 3072
get_children() (*matplotlib.offsetbox.OffsetBox method*), 3081
get_children() (*matplotlib.offsetbox.OffsetImage method*), 3084
get_children() (*matplotlib.table.Table method*), 3653
get_children() (*mpl_toolkits.axisartist.axislines.Axes method*), 4032
get_clim() (*matplotlib.cm.ScalarMappable method*), 2386
get_clim() (*matplotlib.collections.AsteriskPolygonCollection method*), 2392
get_clim() (*matplotlib.collections.CircleCollection method*), 2414
get_clim() (*matplotlib.collections.Collection method*), 2437

get_clim() (*matplotlib.collections.EllipseCollection method*), 2459
get_clim() (*matplotlib.collections.EventCollection method*), 2481
get_clim() (*matplotlib.collections.LineCollection method*), 2505
get_clim() (*matplotlib.collections.PatchCollection method*), 2527
get_clim() (*matplotlib.collections.PathCollection method*), 2548
get_clim() (*matplotlib.collections.PolyCollection method*), 2571
get_clim() (*matplotlib.collections.PolyQuadMesh method*), 2594
get_clim() (*matplotlib.collections.QuadMesh method*), 2618
get_clim() (*matplotlib.collections.RegularPolyCollection method*), 2640
get_clim() (*matplotlib.collections.StarPolygonCollection method*), 2662
get_clim() (*matplotlib.collections.TriMesh method*), 2685
get_clip_box() (*matplotlib.artist.Artist method*), 1875
get_clip_box() (*matplotlib.collections.AsteriskPolygonCollection method*), 2392
get_clip_box() (*matplotlib.collections.CircleCollection method*), 2414
get_clip_box() (*matplotlib.collections.Collection method*), 2437
get_clip_box() (*matplotlib.collections.EllipseCollection method*), 2459
get_clip_box() (*matplotlib.collections.EventCollection method*), 2481
get_clip_box() (*matplotlib.collections.LineCollection method*), 2505
get_clip_box() (*matplotlib.collections.PatchCollection method*), 2527
get_clip_box() (*matplotlib.collections.PathCollection method*), 2548
get_clip_box() (*matplotlib.collections.PolyCollection method*), 2571
get_clip_box() (*matplotlib.collections.PolyQuadMesh method*), 2594
get_clip_box() (*matplotlib.collections.QuadMesh method*), 2618
get_clip_box() (*matplotlib.collections.RegularPolyCollection method*), 2640
get_clip_box() (*matplotlib.collections.StarPolygonCollection method*), 2662
get_clip_box() (*matplotlib.collections.TriMesh method*), 2685
get_clip_on() (*matplotlib.artist.Artist method*), 1875
get_clip_on() (*matplotlib.collections.AsteriskPolygonCollection method*), 2392
get_clip_on() (*matplotlib.collections.CircleCollection method*), 2414
get_clip_on() (*matplotlib.collections.Collection method*), 2437
get_clip_on() (*matplotlib.collections.EllipseCollection method*), 2459
get_clip_on() (*matplotlib.collections.EventCollection method*), 2481
get_clip_on() (*matplotlib.collections.LineCollection method*), 2505
get_clip_on() (*matplotlib.collections.PatchCollection method*), 2527
get_clip_on() (*matplotlib.collections.PathCollection method*), 2548
get_clip_on() (*matplotlib.collections.PolyCollection method*), 2571
get_clip_on() (*matplotlib.collections.PolyQuadMesh method*), 2594
get_clip_on() (*matplotlib.collections.QuadMesh method*), 2618
get_clip_on() (*matplotlib.collections.RegularPolyCollection method*), 2640
get_clip_on() (*matplotlib.collections.StarPolygonCollection method*), 2662
get_clip_on() (*matplotlib.collections.TriMesh method*), 2685
get_clip_path() (*matplotlib.artist.Artist method*), 1876
get_clip_path() (*matplotlib.backend_bases.GraphicsContextBase method*), 2264
get_clip_path() (*matplotlib.collections.AsteriskPolygonCollection method*), 2392
get_clip_path() (*matplotlib.collections.CircleCollection method*), 2414
get_clip_path() (*matplotlib.collections.Collection method*), 2437
get_clip_path() (*matplotlib.collections.EllipseCollection method*), 2459
get_clip_path() (*matplotlib.collections.EventCollection method*), 2481
get_clip_path() (*matplotlib.collections.LineCollection method*), 2505
get_clip_path() (*matplotlib.collections.PatchCollection method*), 2527
get_clip_path() (*matplotlib.collections.PathCollection method*), 2548
get_clip_path() (*matplotlib.collections.PolyCollection method*), 2571
get_clip_path() (*matplotlib.collections.PolyQuadMesh method*), 2594
get_clip_path() (*matplotlib.collections.QuadMesh method*), 2618
get_clip_path() (*matplotlib.collections.RegularPolyCollection method*), 2640
get_clip_path() (*matplotlib.collections.StarPolygonCollection method*), 2662

get_clip_path() (*matplotlib.collections.TriMesh method*), 2685
get_clip_rectangle() (*matplotlib.backend_bases.GraphicsContextBase method*), 2264
get_closed() (*matplotlib.patches.Polygon method*), 3177
get_cmap() (*in module matplotlib.pyplot*), 3482
get_cmap() (*matplotlib.cm.ColormapRegistry method*), 2384
get_cmap() (*matplotlib.cm.ScalarMappable method*), 2387
get_cmap() (*matplotlib.collections.AsteriskPolygonCollection method*), 2392
get_cmap() (*matplotlib.collections.CircleCollection method*), 2414
get_cmap() (*matplotlib.collections.Collection method*), 2437
get_cmap() (*matplotlib.collections.EllipseCollection method*), 2459
get_cmap() (*matplotlib.collections.EventCollection method*), 2481
get_cmap() (*matplotlib.collections.LineCollection method*), 2505
get_cmap() (*matplotlib.collections.PatchCollection method*), 2527
get_cmap() (*matplotlib.collections.PathCollection method*), 2548
get_cmap() (*matplotlib.collections.PolyCollection method*), 2571
get_cmap() (*matplotlib.collections.PolyQuadMesh method*), 2594
get_cmap() (*matplotlib.collections.QuadMesh method*), 2618
get_cmap() (*matplotlib.collections.RegularPolyCollection method*), 2640
get_cmap() (*matplotlib.collections.StarPolygonCollection method*), 2662
get_cmap() (*matplotlib.collections.TriMesh method*), 2685
get_co_vertices() (*matplotlib.patches.Ellipse method*), 3136
get_color() (*matplotlib.collections.EventCollection method*), 2482
get_color() (*matplotlib.collections.LineCollection method*), 2505
get_color() (*matplotlib.lines.Line2D method*), 3018
get_color() (*matplotlib.text.Text method*), 3664
get_color() (*mpl_toolkits.axisartist.axis_artist.AxisLabel method*), 4009
get_color() (*mpl_toolkits.axisartist.axis_artist.Ticks method*), 4023
get_colors() (*matplotlib.collections.EventCollection method*), 2482
get_colors() (*matplotlib.collections.LineCollection method*), 2505
get_configdir() (*in module matplotlib*), 1826
get_connectionstyle() (*matplotlib.patches.FancyArrowPatch method*), 3147
get_constrained_layout() (*matplotlib.figure.Figure method*), 2851
get_constrained_layout_pads() (*matplotlib.figure.Figure method*), 2852
get_converter() (*matplotlib.units.Registry method*), 3771
get_coordinates() (*matplotlib.collections.PolyQuadMesh method*), 2594
get_coordinates() (*matplotlib.collections.QuadMesh method*), 2618
get_corners() (*matplotlib.patches.Ellipse method*), 3136
get_corners() (*matplotlib.patches.Rectangle method*), 3181
get_cos_sin() (*in module matplotlib.bezier*), 2367
get_cpp_triangulation() (*matplotlib.tri.Triangulation method*), 3756
get_current_fig_manager() (*in module matplotlib.pyplot*), 3520
get_cursor_data() (*matplotlib.artist.Artist method*), 1870
get_cursor_data() (*matplotlib.axes.Axes method*), 2203
get_cursor_data() (*matplotlib.collections.AsteriskPolygonCollection method*), 2392
get_cursor_data() (*matplotlib.collections.CircleCollection method*), 2414
get_cursor_data() (*matplotlib.collections.Collection method*), 2437
get_cursor_data() (*matplotlib.collections.EllipseCollection method*), 2459
get_cursor_data() (*matplotlib.collections.EventCollection method*), 2482
get_cursor_data() (*matplotlib.collections.LineCollection method*), 2505
get_cursor_data() (*matplotlib.collections.PatchCollection method*), 2527
get_cursor_data() (*matplotlib.collections.PathCollection method*), 2548
get_cursor_data() (*matplotlib.collections.PolyCollection method*), 2571
get_cursor_data() (*matplotlib.collections.PolyQuadMesh method*), 2594
get_cursor_data() (*matplotlib.collections.QuadMesh method*), 2618
get_cursor_data() (*matplotlib.collections.RegularPolyCollection method*), 2640
get_cursor_data() (*matplotlib.collections.StarPolygonCollection method*), 2662
get_cursor_data() (*matplotlib.collections.TriMesh*

*method*), 2686
get_cursor_data() (*matplotlib.image.AxesImage method*), 2968
get_cursor_data() (*matplotlib.image.NonUniformImage method*), 2975
get_cursor_data() (*matplotlib.image.PcolorImage method*), 2979
get_custom_preamble() (*matplotlib.texmanager.TexManager class method*), 3688
get_dash_capstyle() (*matplotlib.lines.Line2D method*), 3019
get_dash_joinstyle() (*matplotlib.lines.Line2D method*), 3019
get_dashes() (*matplotlib.backend_bases.GraphicsContextBase method*), 2264
get_dashes() (*matplotlib.collections.AsteriskPolygonCollection method*), 2393
get_dashes() (*matplotlib.collections.CircleCollection method*), 2414
get_dashes() (*matplotlib.collections.Collection method*), 2438
get_dashes() (*matplotlib.collections.EllipseCollection method*), 2460
get_dashes() (*matplotlib.collections.EventCollection method*), 2482
get_dashes() (*matplotlib.collections.LineCollection method*), 2505
get_dashes() (*matplotlib.collections.PatchCollection method*), 2528
get_dashes() (*matplotlib.collections.PathCollection method*), 2549
get_dashes() (*matplotlib.collections.PolyCollection method*), 2572
get_dashes() (*matplotlib.collections.PolyQuadMesh method*), 2595
get_dashes() (*matplotlib.collections.QuadMesh method*), 2619
get_dashes() (*matplotlib.collections.RegularPolyCollection method*), 2641
get_dashes() (*matplotlib.collections.StarPolygonCollection method*), 2663
get_dashes() (*matplotlib.collections.TriMesh method*), 2686
get_data() (*matplotlib.lines.Line2D method*), 3019
get_data() (*matplotlib.offsetbox.OffsetImage method*), 3084
get_data() (*matplotlib.patches.StepPatch method*), 3174
get_data_3d() (*mpl_toolkits.mplot3d.art3d.Line3D method*), 3880
get_data_boundary() (*mpl_toolkits.axisartist.floating_axes.GridHelperCurveLinear method*), 4044
get_data_interval() (*matplotlib.axis.Axis method*), 2237
get_data_path() (*in module matplotlib*), 1827
get_data_ratio() (*matplotlib.axes.Axes method*), 2210
get_data_ratio() (*matplotlib.projections.geo.GeoAxes method*), 3565
get_data_ratio() (*matplotlib.projections.polar.PolarAxes method*), 3536
get_data_transform() (*matplotlib.patches.Patch method*), 3162
get_datalim() (*matplotlib.collections.AsteriskPolygonCollection method*), 2393
get_datalim() (*matplotlib.collections.CircleCollection method*), 2414
get_datalim() (*matplotlib.collections.Collection method*), 2438
get_datalim() (*matplotlib.collections.EllipseCollection method*), 2460
get_datalim() (*matplotlib.collections.EventCollection method*), 2482
get_datalim() (*matplotlib.collections.LineCollection method*), 2506
get_datalim() (*matplotlib.collections.PatchCollection method*), 2528
get_datalim() (*matplotlib.collections.PathCollection method*), 2549
get_datalim() (*matplotlib.collections.PolyCollection method*), 2572
get_datalim() (*matplotlib.collections.PolyQuadMesh method*), 2595
get_datalim() (*matplotlib.collections.QuadMesh method*), 2619
get_datalim() (*matplotlib.collections.RegularPolyCollection method*), 2641
get_datalim() (*matplotlib.collections.StarPolygonCollection method*), 2663
get_datalim() (*matplotlib.collections.TriMesh method*), 2686
get_datalim() (*matplotlib.quiver.Quiver method*), 3594
get_default_bbox_extra_artists() (*matplotlib.axes.Axes method*), 2213
get_default_bbox_extra_artists() (*matplotlib.figure.FigureBase method*), 2919
get_default_filename() (*matplotlib.backend_bases.FigureCanvasBase method*), 2256
get_default_filetype() (*matplotlib.backend_bases.FigureCanvasBase class method*), 2256
get_default_filetype() (*matplotlib.backends.backend_pdf.FigureCanvasPdf method*), 2323
get_default_filetype() (*matplotlib.backends.backend_pgf.FigureCanvasPgf method*), 2335
get_default_filetype() (*matplotlib.backends.backend_ps.FigureCanvasPS method*), 2342

get_default_filetype() (*matplotlib.backends.backend_svg.FigureCanvasSVG method*), 2351
get_default_filetype() (*matplotlib.backends.backend_template.FigureCanvasTemplate method*), 2305
get_default_handler_map() (*matplotlib.legend.Legend class method*), 2997
get_default_size() (*matplotlib.font_manager.FontManager static method*), 2943
get_default_weight() (*matplotlib.font_manager.FontManager method*), 2943
get_depthshade() (*mpl_toolkits.mplot3d.art3d.Patch3DCollection method*), 3888
get_depthshade() (*mpl_toolkits.mplot3d.art3d.Path3DCollection method*), 3891
get_descent() (*matplotlib.ft2font.FT2Font method*), 2953
get_diff_image() (*matplotlib.backends.backend_webagg_core.FigureCanvasWebAggCore method*), 2358
get_dir_vector() (*in module mpl_toolkits.mplot3d.art3d*), 3904
get_divider() (*mpl_toolkits.axes_grid1.axes_grid.Grid method*), 3943
get_dpi() (*matplotlib.figure.SubFigure method*), 2903
get_draggable() (*matplotlib.legend.Legend method*), 2997
get_drawstyle() (*matplotlib.lines.Line2D method*), 3019
get_ds() (*matplotlib.lines.Line2D method*), 3019
get_ec() (*matplotlib.collections.AsteriskPolygonCollection method*), 2393
get_ec() (*matplotlib.collections.CircleCollection method*), 2414
get_ec() (*matplotlib.collections.Collection method*), 2438
get_ec() (*matplotlib.collections.EllipseCollection method*), 2460
get_ec() (*matplotlib.collections.EventCollection method*), 2482
get_ec() (*matplotlib.collections.LineCollection method*), 2506
get_ec() (*matplotlib.collections.PatchCollection method*), 2528
get_ec() (*matplotlib.collections.PathCollection method*), 2549
get_ec() (*matplotlib.collections.PolyCollection method*), 2572
get_ec() (*matplotlib.collections.PolyQuadMesh method*), 2595
get_ec() (*matplotlib.collections.QuadMesh method*), 2619
get_ec() (*matplotlib.collections.RegularPolyCollection method*), 2641
get_ec() (*matplotlib.collections.StarPolygonCollection method*), 2663
get_ec() (*matplotlib.collections.TriMesh method*), 2686
get_ec() (*matplotlib.patches.Patch method*), 3162
get_edgecolor() (*matplotlib.collections.AsteriskPolygonCollection method*), 2393
get_edgecolor() (*matplotlib.collections.CircleCollection method*), 2415
get_edgecolor() (*matplotlib.collections.Collection method*), 2438
get_edgecolor() (*matplotlib.collections.EllipseCollection method*), 2460
get_edgecolor() (*matplotlib.collections.EventCollection method*), 2482
get_edgecolor() (*matplotlib.collections.LineCollection method*), 2506
get_edgecolor() (*matplotlib.collections.PatchCollection method*), 2528
get_edgecolor() (*matplotlib.collections.PathCollection method*), 2549
get_edgecolor() (*matplotlib.collections.PolyCollection method*), 2572
get_edgecolor() (*matplotlib.collections.PolyQuadMesh method*), 2595
get_edgecolor() (*matplotlib.collections.QuadMesh method*), 2619
get_edgecolor() (*matplotlib.collections.RegularPolyCollection method*), 2641
get_edgecolor() (*matplotlib.collections.StarPolygonCollection method*), 2663
get_edgecolor() (*matplotlib.collections.TriMesh method*), 2686
get_edgecolor() (*matplotlib.figure.Figure method*), 2857
get_edgecolor() (*matplotlib.figure.FigureBase method*), 2919
get_edgecolor() (*matplotlib.figure.SubFigure method*), 2903
get_edgecolor() (*matplotlib.patches.Patch method*), 3162
get_edgecolor() (*mpl_toolkits.mplot3d.art3d.Patch3DCollection method*), 3888
get_edgecolor() (*mpl_toolkits.mplot3d.art3d.Path3DCollection method*), 3891
get_edgecolor() (*mpl_toolkits.mplot3d.art3d.Poly3DCollection method*), 3896
get_edgecolors() (*matplotlib.collections.AsteriskPolygonCollection method*), 2393
get_edgecolors() (*matplotlib.collections.CircleCollection method*), 2415
get_edgecolors() (*matplotlib.collections.Collection method*), 2438
get_edgecolors() (*matplotlib.collections.EllipseCollection method*), 2460
get_edgecolors() (*matplotlib.collections.EventCollection method*), 2482

get_edgecolors() (*matplotlib.collections.LineCollection method*), 2506
get_edgecolors() (*matplotlib.collections.PatchCollection method*), 2528
get_edgecolors() (*matplotlib.collections.PathCollection method*), 2549
get_edgecolors() (*matplotlib.collections.PolyCollection method*), 2572
get_edgecolors() (*matplotlib.collections.PolyQuadMesh method*), 2595
get_edgecolors() (*matplotlib.collections.QuadMesh method*), 2619
get_edgecolors() (*matplotlib.collections.RegularPolyCollection method*), 2641
get_edgecolors() (*matplotlib.collections.StarPolygonCollection method*), 2663
get_edgecolors() (*matplotlib.collections.TriMesh method*), 2686
get_epoch() (*in module matplotlib.dates*), 2783
get_err_size() (*matplotlib.legend_handler.HandlerErrorbar method*), 3006
get_extent() (*matplotlib.image.AxesImage method*), 2969
get_extent() (*matplotlib.image.FigureImage method*), 2973
get_extent() (*matplotlib.image.NonUniformImage method*), 2975
get_extents() (*matplotlib.patches.Patch method*), 3162
get_extents() (*matplotlib.path.Path method*), 3199
get_facecolor() (*matplotlib.axes.Axes method*), 2116
get_facecolor() (*matplotlib.collections.AsteriskPolygonCollection method*), 2393
get_facecolor() (*matplotlib.collections.CircleCollection method*), 2415
get_facecolor() (*matplotlib.collections.Collection method*), 2438
get_facecolor() (*matplotlib.collections.EllipseCollection method*), 2460
get_facecolor() (*matplotlib.collections.EventCollection method*), 2482
get_facecolor() (*matplotlib.collections.LineCollection method*), 2506
get_facecolor() (*matplotlib.collections.PatchCollection method*), 2528
get_facecolor() (*matplotlib.collections.PathCollection method*), 2549
get_facecolor() (*matplotlib.collections.PolyCollection method*), 2572
get_facecolor() (*matplotlib.collections.PolyQuadMesh method*), 2595
get_facecolor() (*matplotlib.collections.QuadMesh method*), 2619
get_facecolor() (*matplotlib.collections.RegularPolyCollection method*), 2641
get_facecolor() (*matplotlib.collections.StarPolygonCollection method*), 2663
get_facecolor() (*matplotlib.collections.TriMesh method*), 2686
get_facecolor() (*matplotlib.figure.Figure method*), 2856
get_facecolor() (*matplotlib.figure.FigureBase method*), 2919
get_facecolor() (*matplotlib.figure.SubFigure method*), 2902
get_facecolor() (*matplotlib.patches.Patch method*), 3162
get_facecolor() (*mpl_toolkits.mplot3d.art3d.Patch3DCollection method*), 3888
get_facecolor() (*mpl_toolkits.mplot3d.art3d.Path3DCollection method*), 3891
get_facecolor() (*mpl_toolkits.mplot3d.art3d.Poly3DCollection method*), 3896
get_facecolors() (*matplotlib.collections.AsteriskPolygonCollection method*), 2393
get_facecolors() (*matplotlib.collections.CircleCollection method*), 2415
get_facecolors() (*matplotlib.collections.Collection method*), 2438
get_facecolors() (*matplotlib.collections.EllipseCollection method*), 2460
get_facecolors() (*matplotlib.collections.EventCollection method*), 2482
get_facecolors() (*matplotlib.collections.LineCollection method*), 2506
get_facecolors() (*matplotlib.collections.PatchCollection method*), 2528
get_facecolors() (*matplotlib.collections.PathCollection method*), 2549
get_facecolors() (*matplotlib.collections.PolyCollection method*), 2572
get_facecolors() (*matplotlib.collections.PolyQuadMesh method*), 2595
get_facecolors() (*matplotlib.collections.QuadMesh method*), 2619
get_facecolors() (*matplotlib.collections.RegularPolyCollection method*), 2641
get_facecolors() (*matplotlib.collections.StarPolygonCollection method*), 2663
get_facecolors() (*matplotlib.collections.TriMesh method*), 2686
get_family() (*matplotlib.font_manager.FontProperties method*), 2945
get_family() (*matplotlib.text.Text method*), 3664
get_familyname() (*matplotlib._afm.AFM method*), 1830
get_fc() (*matplotlib.collections.AsteriskPolygonCollection method*), 2393
get_fc() (*matplotlib.collections.CircleCollection method*),

2415
get_fc() (*matplotlib.collections.Collection method*), 2438
get_fc() (*matplotlib.collections.EllipseCollection method*), 2460
get_fc() (*matplotlib.collections.EventCollection method*), 2482
get_fc() (*matplotlib.collections.LineCollection method*), 2506
get_fc() (*matplotlib.collections.PatchCollection method*), 2528
get_fc() (*matplotlib.collections.PathCollection method*), 2549
get_fc() (*matplotlib.collections.PolyCollection method*), 2572
get_fc() (*matplotlib.collections.PolyQuadMesh method*), 2595
get_fc() (*matplotlib.collections.QuadMesh method*), 2619
get_fc() (*matplotlib.collections.RegularPolyCollection method*), 2641
get_fc() (*matplotlib.collections.StarPolygonCollection method*), 2663
get_fc() (*matplotlib.collections.TriMesh method*), 2686
get_fc() (*matplotlib.patches.Patch method*), 3162
get_figheight() (*matplotlib.figure.Figure method*), 2843
get_figlabels() (*in module matplotlib.pyplot*), 3225
get_fignums() (*in module matplotlib.pyplot*), 3225
get_figure() (*matplotlib.artist.Artist method*), 1890
get_figure() (*matplotlib.collections.AsteriskPolygonCollection method*), 2393
get_figure() (*matplotlib.collections.CircleCollection method*), 2415
get_figure() (*matplotlib.collections.Collection method*), 2438
get_figure() (*matplotlib.collections.EllipseCollection method*), 2460
get_figure() (*matplotlib.collections.EventCollection method*), 2483
get_figure() (*matplotlib.collections.LineCollection method*), 2506
get_figure() (*matplotlib.collections.PatchCollection method*), 2528
get_figure() (*matplotlib.collections.PathCollection method*), 2549
get_figure() (*matplotlib.collections.PolyCollection method*), 2572
get_figure() (*matplotlib.collections.PolyQuadMesh method*), 2595
get_figure() (*matplotlib.collections.QuadMesh method*), 2619
get_figure() (*matplotlib.collections.RegularPolyCollection method*), 2641
get_figure() (*matplotlib.collections.StarPolygonCollection method*), 2663
get_figure() (*matplotlib.collections.TriMesh method*), 2686
get_figwidth() (*matplotlib.figure.Figure method*), 2844
get_file() (*matplotlib.font_manager.FontProperties method*), 2945
get_fill() (*matplotlib.collections.AsteriskPolygonCollection method*), 2393
get_fill() (*matplotlib.collections.CircleCollection method*), 2415
get_fill() (*matplotlib.collections.Collection method*), 2438
get_fill() (*matplotlib.collections.EllipseCollection method*), 2460
get_fill() (*matplotlib.collections.EventCollection method*), 2483
get_fill() (*matplotlib.collections.LineCollection method*), 2506
get_fill() (*matplotlib.collections.PatchCollection method*), 2528
get_fill() (*matplotlib.collections.PathCollection method*), 2549
get_fill() (*matplotlib.collections.PolyCollection method*), 2572
get_fill() (*matplotlib.collections.PolyQuadMesh method*), 2595
get_fill() (*matplotlib.collections.QuadMesh method*), 2619
get_fill() (*matplotlib.collections.RegularPolyCollection method*), 2641
get_fill() (*matplotlib.collections.StarPolygonCollection method*), 2663
get_fill() (*matplotlib.collections.TriMesh method*), 2687
get_fill() (*matplotlib.patches.Patch method*), 3162
get_fillstyle() (*matplotlib.lines.Line2D method*), 3019
get_fillstyle() (*matplotlib.markers.MarkerStyle method*), 3041
get_flat_tri_mask() (*matplotlib.tri.TriAnalyzer method*), 3765
get_font() (*in module matplotlib.font_manager*), 2949
get_font() (*matplotlib.text.Text method*), 3664
get_font_names() (*in module matplotlib.font_manager*), 2950
get_font_names() (*matplotlib.font_manager.FontManager method*), 2943
get_font_preamble() (*matplotlib.texmanager.TexManager class method*), 3688
get_font_properties() (*matplotlib.text.Text method*), 3664
get_fontconfig_pattern() (*matplotlib.font_manager.FontProperties method*), 2945
get_fontext_synonyms() (*in module matplotlib.font_manager*), 2950
get_fontfamily() (*matplotlib.text.Text method*), 3665
get_fontname() (*matplotlib._afm.AFM method*), 1830
get_fontname() (*matplotlib.text.Text method*), 3665
get_fontproperties() (*matplotlib.text.Text method*), 3665

get_fontsize() (*matplotlib.offsetbox.AnnotationBbox method*), 3072
get_fontsize() (*matplotlib.table.Cell method*), 3647
get_fontsize() (*matplotlib.text.Text method*), 3665
get_fontstyle() (*matplotlib.text.Text method*), 3665
get_fontvariant() (*matplotlib.text.Text method*), 3665
get_fontweight() (*matplotlib.text.Text method*), 3665
get_forced_alpha() (*matplotlib.backend_bases.GraphicsContextBase method*), 2264
get_forward_navigation_events() (*matplotlib.axes.Axes method*), 2199
get_frame() (*matplotlib.legend.Legend method*), 2997
get_frame_on() (*matplotlib.axes.Axes method*), 2112
get_frame_on() (*matplotlib.legend.Legend method*), 2997
get_frameon() (*matplotlib.figure.Figure method*), 2855
get_frameon() (*matplotlib.figure.FigureBase method*), 2919
get_frameon() (*matplotlib.figure.SubFigure method*), 2901
get_from_args_and_kwargs() (*matplotlib.tri.Triangulation static method*), 3756
get_fullname() (*matplotlib._afm.AFM method*), 1830
get_fully_transformed_path() (*matplotlib.transforms.TransformedPath method*), 3752
get_gapcolor() (*matplotlib.collections.EventCollection method*), 2483
get_gapcolor() (*matplotlib.collections.LineCollection method*), 2506
get_gapcolor() (*matplotlib.lines.Line2D method*), 3019
get_geometry() (*matplotlib.gridspec.GridSpecBase method*), 2962
get_geometry() (*matplotlib.gridspec.SubplotSpec method*), 2960
get_geometry() (*mpl_toolkits.axes_grid1.axes_grid.Grid method*), 3943
get_gid() (*matplotlib.artist.Artist method*), 1894
get_gid() (*matplotlib.backend_bases.GraphicsContextBase method*), 2264
get_gid() (*matplotlib.collections.AsteriskPolygonCollection method*), 2393
get_gid() (*matplotlib.collections.CircleCollection method*), 2415
get_gid() (*matplotlib.collections.Collection method*), 2438
get_gid() (*matplotlib.collections.EllipseCollection method*), 2460
get_gid() (*matplotlib.collections.EventCollection method*), 2483
get_gid() (*matplotlib.collections.LineCollection method*), 2506
get_gid() (*matplotlib.collections.PatchCollection method*), 2528
get_gid() (*matplotlib.collections.PathCollection method*), 2549
get_gid() (*matplotlib.collections.PolyCollection method*), 2572
get_gid() (*matplotlib.collections.PolyQuadMesh method*), 2595
get_gid() (*matplotlib.collections.QuadMesh method*), 2619
get_gid() (*matplotlib.collections.RegularPolyCollection method*), 2641
get_gid() (*matplotlib.collections.StarPolygonCollection method*), 2663
get_gid() (*matplotlib.collections.TriMesh method*), 2687
get_glyph_name() (*matplotlib.ft2font.FT2Font method*), 2953
get_glyphs_mathtext() (*matplotlib.text.TextToPath method*), 3687
get_glyphs_tex() (*matplotlib.text.TextToPath method*), 3687
get_glyphs_with_font() (*matplotlib.text.TextToPath method*), 3687
get_grey() (*matplotlib.texmanager.TexManager class method*), 3689
get_grid_helper() (*mpl_toolkits.axisartist.axislines.Axes method*), 4032
get_grid_info() (*mpl_toolkits.axisartist.grid_finder.GridFinder method*), 4047
get_grid_positions() (*matplotlib.gridspec.GridSpecBase method*), 2962
get_gridlines() (*matplotlib.axis.Axis method*), 2235
get_gridlines() (*mpl_toolkits.axisartist.axislines.GridHelperBase method*), 4040
get_gridlines() (*mpl_toolkits.axisartist.axislines.GridHelperRectlinear method*), 4040
get_gridlines() (*mpl_toolkits.axisartist.floating_axes.GridHelperCurveLinear method*), 4044
get_gridlines() (*mpl_toolkits.axisartist.grid_helper_curvelinear.GridHelperCurveLinear method*), 4051
get_gridspec() (*matplotlib.gridspec.SubplotSpec method*), 2960
get_ha() (*matplotlib.text.Text method*), 3665
get_hatch() (*matplotlib.backend_bases.GraphicsContextBase method*), 2264
get_hatch() (*matplotlib.collections.AsteriskPolygonCollection method*), 2394
get_hatch() (*matplotlib.collections.CircleCollection method*), 2415
get_hatch() (*matplotlib.collections.Collection method*), 2439
get_hatch() (*matplotlib.collections.EllipseCollection method*), 2460
get_hatch() (*matplotlib.collections.EventCollection method*), 2483
get_hatch() (*matplotlib.collections.LineCollection method*), 2506
get_hatch() (*matplotlib.collections.PatchCollection method*), 2528
get_hatch() (*matplotlib.collections.PathCollection method*), 2549

get_hatch() (*matplotlib.collections.PolyCollection method*), 2572
get_hatch() (*matplotlib.collections.PolyQuadMesh method*), 2595
get_hatch() (*matplotlib.collections.QuadMesh method*), 2620
get_hatch() (*matplotlib.collections.RegularPolyCollection method*), 2642
get_hatch() (*matplotlib.collections.StarPolygonCollection method*), 2664
get_hatch() (*matplotlib.collections.TriMesh method*), 2687
get_hatch() (*matplotlib.patches.Patch method*), 3162
get_hatch_color() (*matplotlib.backend_bases.GraphicsContextBase method*), 2264
get_hatch_linewidth() (*matplotlib.backend_bases.GraphicsContextBase method*), 2264
get_hatch_path() (*matplotlib.backend_bases.GraphicsContextBase method*), 2265
get_height() (*matplotlib.patches.Ellipse method*), 3136
get_height() (*matplotlib.patches.FancyBboxPatch method*), 3154
get_height() (*matplotlib.patches.Rectangle method*), 3181
get_height_char() (*matplotlib._afm.AFM method*), 1830
get_height_ratios() (*matplotlib.gridspec.GridSpecBase method*), 2963
get_heights() (*matplotlib.collections.EllipseCollection method*), 2460
get_helper() (*mpl_toolkits.axisartist.axis_artist.AxisArtist method*), 4005
get_hinting_flag() (*in module matplotlib.backends.backend_agg*), 2315
get_horizontal() (*mpl_toolkits.axes_grid1.axes_divider.Divider method*), 3933
get_horizontal_sizes() (*mpl_toolkits.axes_grid1.axes_divider.Divider method*), 3933
get_horizontal_stem_width() (*matplotlib._afm.AFM method*), 1830
get_horizontalalignment() (*matplotlib.text.Text method*), 3666
get_image() (*matplotlib.ft2font.FT2Font method*), 2954
get_image_magnification() (*matplotlib.backend_bases.RendererBase method*), 2277
get_image_magnification() (*matplotlib.backends.backend_pdf.RendererPdf method*), 2334
get_image_magnification() (*matplotlib.backends.backend_ps.RendererPS method*), 2345
get_image_magnification() (*matplotlib.backends.backend_svg.RendererSVG method*), 2355
get_images() (*matplotlib.axes.Axes method*), 2204
get_in_layout() (*matplotlib.artist.Artist method*), 1897
get_in_layout() (*matplotlib.collections.AsteriskPolygonCollection method*), 2394
get_in_layout() (*matplotlib.collections.CircleCollection method*), 2415
get_in_layout() (*matplotlib.collections.Collection method*), 2439
get_in_layout() (*matplotlib.collections.EllipseCollection method*), 2460
get_in_layout() (*matplotlib.collections.EventCollection method*), 2483
get_in_layout() (*matplotlib.collections.LineCollection method*), 2506
get_in_layout() (*matplotlib.collections.PatchCollection method*), 2528
get_in_layout() (*matplotlib.collections.PathCollection method*), 2549
get_in_layout() (*matplotlib.collections.PolyCollection method*), 2572
get_in_layout() (*matplotlib.collections.PolyQuadMesh method*), 2595
get_in_layout() (*matplotlib.collections.QuadMesh method*), 2620
get_in_layout() (*matplotlib.collections.RegularPolyCollection method*), 2642
get_in_layout() (*matplotlib.collections.StarPolygonCollection method*), 2664
get_in_layout() (*matplotlib.collections.TriMesh method*), 2687
get_intersection() (*in module matplotlib.bezier*), 2367
get_inverted() (*matplotlib.axis.Axis method*), 2237
get_javascript() (*matplotlib.backends.backend_nbagg.FigureManagerNbAgg class method*), 2322
get_javascript() (*matplotlib.backends.backend_webagg_core.FigureManagerWebAgg class method*), 2360
get_joinstyle() (*matplotlib.backend_bases.GraphicsContextBase method*), 2265
get_joinstyle() (*matplotlib.collections.AsteriskPolygonCollection method*), 2394
get_joinstyle() (*matplotlib.collections.CircleCollection method*), 2415
get_joinstyle() (*matplotlib.collections.Collection method*), 2439
get_joinstyle() (*matplotlib.collections.EllipseCollection method*), 2460
get_joinstyle() (*matplotlib.collections.EventCollection method*), 2483
get_joinstyle() (*matplotlib.collections.LineCollection method*), 2506
get_joinstyle() (*matplotlib.collections.PatchCollection method*), 2528
get_joinstyle() (*matplotlib.collections.PathCollection method*), 2549

get_joinstyle() (*matplotlib.collections.PolyCollection method*), 2572
get_joinstyle() (*matplotlib.collections.PolyQuadMesh method*), 2595
get_joinstyle() (*matplotlib.collections.QuadMesh method*), 2620
get_joinstyle() (*matplotlib.collections.RegularPolyCollection method*), 2642
get_joinstyle() (*matplotlib.collections.StarPolygonCollection method*), 2664
get_joinstyle() (*matplotlib.collections.TriMesh method*), 2687
get_joinstyle() (*matplotlib.markers.MarkerStyle method*), 3041
get_joinstyle() (*matplotlib.patches.Patch method*), 3162
get_kern_dist() (*matplotlib._afm.AFM method*), 1830
get_kern_dist_from_name() (*matplotlib._afm.AFM method*), 1830
get_kerning() (*matplotlib.ft2font.FT2Font method*), 2954
get_label() (*matplotlib.artist.Artist method*), 1895
get_label() (*matplotlib.axis.Axis method*), 2229
get_label() (*matplotlib.collections.AsteriskPolygonCollection method*), 2394
get_label() (*matplotlib.collections.CircleCollection method*), 2415
get_label() (*matplotlib.collections.Collection method*), 2439
get_label() (*matplotlib.collections.EllipseCollection method*), 2461
get_label() (*matplotlib.collections.EventCollection method*), 2483
get_label() (*matplotlib.collections.LineCollection method*), 2506
get_label() (*matplotlib.collections.PatchCollection method*), 2529
get_label() (*matplotlib.collections.PathCollection method*), 2550
get_label() (*matplotlib.collections.PolyCollection method*), 2573
get_label() (*matplotlib.collections.PolyQuadMesh method*), 2595
get_label() (*matplotlib.collections.QuadMesh method*), 2620
get_label() (*matplotlib.collections.RegularPolyCollection method*), 2642
get_label() (*matplotlib.collections.StarPolygonCollection method*), 2664
get_label() (*matplotlib.collections.TriMesh method*), 2687
get_label() (*matplotlib.container.Container method*), 2754
get_label_position() (*matplotlib.axis.Axis method*), 2229
get_label_position() (*mpl_toolkits.mplot3d.axis3d.Axis method*), 3875
get_label_text() (*matplotlib.axis.Axis method*), 2229
get_layout_engine() (*matplotlib.figure.Figure method*), 2848
get_legend() (*matplotlib.axes.Axes method*), 2152
get_legend_handler() (*matplotlib.legend.Legend static method*), 2997
get_legend_handler_map() (*matplotlib.legend.Legend method*), 2997
get_legend_handles_labels() (*matplotlib.axes.Axes method*), 2152
get_line() (*mpl_toolkits.axisartist.axislines.FloatingAxisArtistHelperRectilinear method*), 4039
get_line() (*mpl_toolkits.axisartist.floating_axes.FixedAxisArtistHelper method*), 4042
get_line() (*mpl_toolkits.axisartist.grid_helper_curvelinear.FloatingAxisArtistHelpe method*), 4050
get_line_transform() (*mpl_toolkits.axisartist.axislines.FloatingAxisArtistHelperRectilinear method*), 4039
get_line_transform() (*mpl_toolkits.axisartist.grid_helper_curvelinear.FloatingAxisArtistHelpe method*), 4050
get_linelength() (*matplotlib.collections.EventCollection method*), 2483
get_lineoffset() (*matplotlib.collections.EventCollection method*), 2483
get_lines() (*matplotlib.axes.Axes method*), 2204
get_lines() (*matplotlib.legend.Legend method*), 2997
get_linestyle() (*matplotlib.collections.AsteriskPolygonCollection method*), 2394
get_linestyle() (*matplotlib.collections.CircleCollection method*), 2415
get_linestyle() (*matplotlib.collections.Collection method*), 2439
get_linestyle() (*matplotlib.collections.EllipseCollection method*), 2461
get_linestyle() (*matplotlib.collections.EventCollection method*), 2483
get_linestyle() (*matplotlib.collections.LineCollection method*), 2506
get_linestyle() (*matplotlib.collections.PatchCollection method*), 2529
get_linestyle() (*matplotlib.collections.PathCollection method*), 2550
get_linestyle() (*matplotlib.collections.PolyCollection method*), 2573
get_linestyle() (*matplotlib.collections.PolyQuadMesh method*), 2595
get_linestyle() (*matplotlib.collections.QuadMesh method*), 2620
get_linestyle() (*matplotlib.collections.RegularPolyCollection method*), 2642
get_linestyle() (*matplotlib.collections.StarPolygonCollection method*), 2664

get_linestyle() (*matplotlib.collections.TriMesh method*), 2687
get_linestyle() (*matplotlib.lines.Line2D method*), 3019
get_linestyle() (*matplotlib.patches.Patch method*), 3162
get_linestyles() (*matplotlib.collections.AsteriskPolygonCollection method*), 2394
get_linestyles() (*matplotlib.collections.CircleCollection method*), 2415
get_linestyles() (*matplotlib.collections.Collection method*), 2439
get_linestyles() (*matplotlib.collections.EllipseCollection method*), 2461
get_linestyles() (*matplotlib.collections.EventCollection method*), 2483
get_linestyles() (*matplotlib.collections.LineCollection method*), 2506
get_linestyles() (*matplotlib.collections.PatchCollection method*), 2529
get_linestyles() (*matplotlib.collections.PathCollection method*), 2550
get_linestyles() (*matplotlib.collections.PolyCollection method*), 2573
get_linestyles() (*matplotlib.collections.PolyQuadMesh method*), 2596
get_linestyles() (*matplotlib.collections.QuadMesh method*), 2620
get_linestyles() (*matplotlib.collections.RegularPolyCollection method*), 2642
get_linestyles() (*matplotlib.collections.StarPolygonCollection method*), 2664
get_linestyles() (*matplotlib.collections.TriMesh method*), 2687
get_linewidth() (*matplotlib.backend_bases.GraphicsContextBase method*), 2265
get_linewidth() (*matplotlib.collections.AsteriskPolygonCollection method*), 2394
get_linewidth() (*matplotlib.collections.CircleCollection method*), 2415
get_linewidth() (*matplotlib.collections.Collection method*), 2439
get_linewidth() (*matplotlib.collections.EllipseCollection method*), 2461
get_linewidth() (*matplotlib.collections.EventCollection method*), 2483
get_linewidth() (*matplotlib.collections.LineCollection method*), 2507
get_linewidth() (*matplotlib.collections.PatchCollection method*), 2529
get_linewidth() (*matplotlib.collections.PathCollection method*), 2550
get_linewidth() (*matplotlib.collections.PolyCollection method*), 2573
get_linewidth() (*matplotlib.collections.PolyQuadMesh method*), 2596
get_linewidth() (*matplotlib.collections.QuadMesh method*), 2620
get_linewidth() (*matplotlib.collections.RegularPolyCollection method*), 2642
get_linewidth() (*matplotlib.collections.StarPolygonCollection method*), 2664
get_linewidth() (*matplotlib.collections.TriMesh method*), 2687
get_linewidth() (*matplotlib.figure.Figure method*), 2856
get_linewidth() (*matplotlib.figure.FigureBase method*), 2919
get_linewidth() (*matplotlib.figure.SubFigure method*), 2902
get_linewidth() (*matplotlib.lines.Line2D method*), 3019
get_linewidth() (*matplotlib.patches.Patch method*), 3162
get_linewidths() (*matplotlib.collections.AsteriskPolygonCollection method*), 2394
get_linewidths() (*matplotlib.collections.CircleCollection method*), 2415
get_linewidths() (*matplotlib.collections.Collection method*), 2439
get_linewidths() (*matplotlib.collections.EllipseCollection method*), 2461
get_linewidths() (*matplotlib.collections.EventCollection method*), 2483
get_linewidths() (*matplotlib.collections.LineCollection method*), 2507
get_linewidths() (*matplotlib.collections.PatchCollection method*), 2529
get_linewidths() (*matplotlib.collections.PathCollection method*), 2550
get_linewidths() (*matplotlib.collections.PolyCollection method*), 2573
get_linewidths() (*matplotlib.collections.PolyQuadMesh method*), 2596
get_linewidths() (*matplotlib.collections.QuadMesh method*), 2620
get_linewidths() (*matplotlib.collections.RegularPolyCollection method*), 2642
get_linewidths() (*matplotlib.collections.StarPolygonCollection method*), 2664
get_linewidths() (*matplotlib.collections.TriMesh method*), 2687
get_loc() (*matplotlib.axis.Tick method*), 2251
get_loc_in_canvas() (*matplotlib.offsetbox.DraggableOffsetBox method*), 3076
get_locator() (*matplotlib.dates.AutoDateLocator method*), 2773
get_locator()

(*mpl_toolkits.axes_grid1.axes_divider.Divider method*), 3933
get_ls() (*matplotlib.collections.AsteriskPolygonCollection method*), 2394
get_ls() (*matplotlib.collections.CircleCollection method*), 2416
get_ls() (*matplotlib.collections.Collection method*), 2439
get_ls() (*matplotlib.collections.EllipseCollection method*), 2461
get_ls() (*matplotlib.collections.EventCollection method*), 2483
get_ls() (*matplotlib.collections.LineCollection method*), 2507
get_ls() (*matplotlib.collections.PatchCollection method*), 2529
get_ls() (*matplotlib.collections.PathCollection method*), 2550
get_ls() (*matplotlib.collections.PolyCollection method*), 2573
get_ls() (*matplotlib.collections.PolyQuadMesh method*), 2596
get_ls() (*matplotlib.collections.QuadMesh method*), 2620
get_ls() (*matplotlib.collections.RegularPolyCollection method*), 2642
get_ls() (*matplotlib.collections.StarPolygonCollection method*), 2664
get_ls() (*matplotlib.collections.TriMesh method*), 2687
get_ls() (*matplotlib.lines.Line2D method*), 3019
get_ls() (*matplotlib.patches.Patch method*), 3162
get_lw() (*matplotlib.collections.AsteriskPolygonCollection method*), 2394
get_lw() (*matplotlib.collections.CircleCollection method*), 2416
get_lw() (*matplotlib.collections.Collection method*), 2439
get_lw() (*matplotlib.collections.EllipseCollection method*), 2461
get_lw() (*matplotlib.collections.EventCollection method*), 2483
get_lw() (*matplotlib.collections.LineCollection method*), 2507
get_lw() (*matplotlib.collections.PatchCollection method*), 2529
get_lw() (*matplotlib.collections.PathCollection method*), 2550
get_lw() (*matplotlib.collections.PolyCollection method*), 2573
get_lw() (*matplotlib.collections.PolyQuadMesh method*), 2596
get_lw() (*matplotlib.collections.QuadMesh method*), 2620
get_lw() (*matplotlib.collections.RegularPolyCollection method*), 2642
get_lw() (*matplotlib.collections.StarPolygonCollection method*), 2664
get_lw() (*matplotlib.collections.TriMesh method*), 2687
get_lw() (*matplotlib.lines.Line2D method*), 3019
get_lw() (*matplotlib.patches.Patch method*), 3162
get_major_formatter() (*matplotlib.axis.Axis method*), 2222
get_major_locator() (*matplotlib.axis.Axis method*), 2223
get_major_ticks() (*matplotlib.axis.Axis method*), 2230
get_major_ticks() (*mpl_toolkits.mplot3d.axis3d.Axis method*), 3875
get_majorticklabels() (*matplotlib.axis.Axis method*), 2231
get_majorticklines() (*matplotlib.axis.Axis method*), 2231
get_majorticklocs() (*matplotlib.axis.Axis method*), 2231
get_marker() (*matplotlib.lines.Line2D method*), 3019
get_marker() (*matplotlib.markers.MarkerStyle method*), 3041
get_markeredgecolor() (*matplotlib.lines.Line2D method*), 3020
get_markeredgecolor() (*mpl_toolkits.axisartist.axis_artist.Ticks method*), 4023
get_markeredgewidth() (*matplotlib.lines.Line2D method*), 3020
get_markeredgewidth() (*mpl_toolkits.axisartist.axis_artist.Ticks method*), 4023
get_markerfacecolor() (*matplotlib.lines.Line2D method*), 3020
get_markerfacecoloralt() (*matplotlib.lines.Line2D method*), 3020
get_markersize() (*matplotlib.lines.Line2D method*), 3020
get_markevery() (*matplotlib.lines.Line2D method*), 3020
get_masked_triangles() (*matplotlib.tri.Triangulation method*), 3756
get_math_fontfamily() (*matplotlib.font_manager.FontProperties method*), 2945
get_math_fontfamily() (*matplotlib.text.Text method*), 3666
get_matrix() (*matplotlib.projections.polar.PolarAffine method*), 3529
get_matrix() (*matplotlib.projections.polar.PolarAxes.PolarAffine method*), 3533
get_matrix() (*matplotlib.transforms.Affine2D method*), 3720
get_matrix() (*matplotlib.transforms.AffineDeltaTransform method*), 3725
get_matrix() (*matplotlib.transforms.BboxTransform method*), 3735
get_matrix() (*matplotlib.transforms.BboxTransformFrom method*), 3736
get_matrix() (*matplotlib.transforms.BboxTransformTo method*), 3736
get_matrix() (*matplotlib.transforms.BboxTransformToMaxOnly method*), 3737
get_matrix() (*matplotlib.transforms.BlendedAffine2D method*), 3737
get_matrix() (*matplotlib.transforms.CompositeAffine2D*

*method*), 3739
get_matrix() (*matplotlib.transforms.IdentityTransform method*), 3742
get_matrix() (*matplotlib.transforms.ScaledTranslation method*), 3745
get_matrix() (*matplotlib.transforms.Transform method*), 3747
get_mec() (*matplotlib.lines.Line2D method*), 3020
get_mew() (*matplotlib.lines.Line2D method*), 3020
get_mfc() (*matplotlib.lines.Line2D method*), 3020
get_mfcalt() (*matplotlib.lines.Line2D method*), 3020
get_minor_formatter() (*matplotlib.axis.Axis method*), 2223
get_minor_locator() (*matplotlib.axis.Axis method*), 2223
get_minor_ticks() (*matplotlib.axis.Axis method*), 2231
get_minor_ticks() (*mpl_toolkits.mplot3d.axis3d.Axis method*), 3876
get_minorticklabels() (*matplotlib.axis.Axis method*), 2231
get_minorticklines() (*matplotlib.axis.Axis method*), 2231
get_minorticklocs() (*matplotlib.axis.Axis method*), 2232
get_minpos() (*matplotlib.axis.Axis method*), 2238
get_mouseover() (*matplotlib.artist.Artist method*), 1872
get_mouseover() (*matplotlib.collections.AsteriskPolygonCollection method*), 2394
get_mouseover() (*matplotlib.collections.CircleCollection method*), 2416
get_mouseover() (*matplotlib.collections.Collection method*), 2439
get_mouseover() (*matplotlib.collections.EllipseCollection method*), 2461
get_mouseover() (*matplotlib.collections.EventCollection method*), 2484
get_mouseover() (*matplotlib.collections.LineCollection method*), 2507
get_mouseover() (*matplotlib.collections.PatchCollection method*), 2529
get_mouseover() (*matplotlib.collections.PathCollection method*), 2550
get_mouseover() (*matplotlib.collections.PolyCollection method*), 2573
get_mouseover() (*matplotlib.collections.PolyQuadMesh method*), 2596
get_mouseover() (*matplotlib.collections.QuadMesh method*), 2620
get_mouseover() (*matplotlib.collections.RegularPolyCollection method*), 2642
get_mouseover() (*matplotlib.collections.StarPolygonCollection method*), 2664
get_mouseover() (*matplotlib.collections.TriMesh method*), 2687
get_ms() (*matplotlib.lines.Line2D method*), 3020
get_multilinebaseline() (*matplotlib.offsetbox.TextArea method*), 3090
get_mutation_aspect() (*matplotlib.patches.FancyArrowPatch method*), 3147
get_mutation_aspect() (*matplotlib.patches.FancyBboxPatch method*), 3154
get_mutation_scale() (*matplotlib.patches.FancyArrowPatch method*), 3147
get_mutation_scale() (*matplotlib.patches.FancyBboxPatch method*), 3154
get_name() (*matplotlib.font_manager.FontProperties method*), 2945
get_name() (*matplotlib.text.Text method*), 3666
get_name_char() (*matplotlib._afm.AFM method*), 1830
get_name_index() (*matplotlib.ft2font.FT2Font method*), 2954
get_named_colors_mapping() (*in module matplotlib.colors*), 2752
get_navigate() (*matplotlib.axes.Axes method*), 2198
get_navigate_mode() (*matplotlib.axes.Axes method*), 2198
get_normal_points() (*in module matplotlib.bezier*), 2367
get_num_glyphs() (*matplotlib.ft2font.FT2Font method*), 2954
get_numpoints() (*matplotlib.legend_handler.HandlerLineCollection method*), 3008
get_numpoints() (*matplotlib.legend_handler.HandlerNpoints method*), 3009
get_numpoints() (*matplotlib.legend_handler.HandlerRegularPolyCollection method*), 3012
get_numsides() (*matplotlib.collections.AsteriskPolygonCollection method*), 2394
get_numsides() (*matplotlib.collections.RegularPolyCollection method*), 2642
get_numsides() (*matplotlib.collections.StarPolygonCollection method*), 2664
get_offset() (*matplotlib.dates.ConciseDateFormatter method*), 2775
get_offset() (*matplotlib.offsetbox.AnchoredOffsetbox method*), 3066
get_offset() (*matplotlib.offsetbox.AuxTransformBox method*), 3074
get_offset() (*matplotlib.offsetbox.DrawingArea method*), 3077
get_offset() (*matplotlib.offsetbox.OffsetBox method*), 3081
get_offset() (*matplotlib.offsetbox.OffsetImage method*), 3084
get_offset() (*matplotlib.offsetbox.TextArea method*), 3090
get_offset() (*matplotlib.ticker.FixedFormatter method*),

3695
get_offset() (*matplotlib.ticker.Formatter method*), 3696
get_offset() (*matplotlib.ticker.FuncFormatter method*), 3697
get_offset() (*matplotlib.ticker.ScalarFormatter method*), 3710
get_offset_text() (*matplotlib.axis.Axis method*), 2232
get_offset_transform() (*matplotlib.collections.AsteriskPolygonCollection method*), 2394
get_offset_transform() (*matplotlib.collections.CircleCollection method*), 2416
get_offset_transform() (*matplotlib.collections.Collection method*), 2439
get_offset_transform() (*matplotlib.collections.EllipseCollection method*), 2461
get_offset_transform() (*matplotlib.collections.EventCollection method*), 2484
get_offset_transform() (*matplotlib.collections.LineCollection method*), 2507
get_offset_transform() (*matplotlib.collections.PatchCollection method*), 2529
get_offset_transform() (*matplotlib.collections.PathCollection method*), 2550
get_offset_transform() (*matplotlib.collections.PolyCollection method*), 2573
get_offset_transform() (*matplotlib.collections.PolyQuadMesh method*), 2596
get_offset_transform() (*matplotlib.collections.QuadMesh method*), 2620
get_offset_transform() (*matplotlib.collections.RegularPolyCollection method*), 2642
get_offset_transform() (*matplotlib.collections.StarPolygonCollection method*), 2664
get_offset_transform() (*matplotlib.collections.TriMesh method*), 2687
get_offsets() (*matplotlib.collections.AsteriskPolygonCollection method*), 2394
get_offsets() (*matplotlib.collections.CircleCollection method*), 2416
get_offsets() (*matplotlib.collections.Collection method*), 2439
get_offsets() (*matplotlib.collections.EllipseCollection method*), 2461
get_offsets() (*matplotlib.collections.EventCollection method*), 2484
get_offsets() (*matplotlib.collections.LineCollection method*), 2507
get_offsets() (*matplotlib.collections.PatchCollection method*), 2529
get_offsets() (*matplotlib.collections.PathCollection method*), 2550
get_offsets() (*matplotlib.collections.PolyCollection method*), 2573
get_offsets() (*matplotlib.collections.PolyQuadMesh method*), 2596
get_offsets() (*matplotlib.collections.QuadMesh method*), 2620
get_offsets() (*matplotlib.collections.RegularPolyCollection method*), 2642
get_offsets() (*matplotlib.collections.StarPolygonCollection method*), 2664
get_offsets() (*matplotlib.collections.TriMesh method*), 2687
get_orientation() (*matplotlib.collections.EventCollection method*), 2484
get_over() (*matplotlib.colors.Colormap method*), 2733
get_pad() (*matplotlib.axis.Tick method*), 2251
get_pad() (*mpl_toolkits.axisartist.axis_artist.AxisLabel method*), 4009
get_pagecount() (*matplotlib.backends.backend_pdf.PdfPages method*), 2329
get_pagecount() (*matplotlib.backends.backend_pgf.PdfPages method*), 2337
get_parallels() (*in module matplotlib.bezier*), 2367
get_parse_math() (*matplotlib.text.Text method*), 3666
get_patch_transform() (*matplotlib.patches.Arrow method*), 3106
get_patch_transform() (*matplotlib.patches.Ellipse method*), 3136
get_patch_transform() (*matplotlib.patches.Patch method*), 3163
get_patch_transform() (*matplotlib.patches.Rectangle method*), 3181
get_patch_transform() (*matplotlib.patches.RegularPolygon method*), 3186
get_patch_transform() (*matplotlib.patches.Shadow method*), 3189
get_patch_transform() (*matplotlib.spines.Spine method*), 3639
get_patches() (*matplotlib.legend.Legend method*), 2997
get_path() (*in module matplotlib.hatch*), 2967
get_path() (*matplotlib.ft2font.FT2Font method*), 2954
get_path() (*matplotlib.lines.Line2D method*), 3020
get_path() (*matplotlib.markers.MarkerStyle method*), 3041
get_path() (*matplotlib.patches.Annulus method*), 3098
get_path() (*matplotlib.patches.Arrow method*), 3106
get_path() (*matplotlib.patches.Ellipse method*), 3137
get_path() (*matplotlib.patches.FancyArrowPatch method*), 3148
get_path() (*matplotlib.patches.FancyBboxPatch method*), 3154

get_path() (*matplotlib.patches.Patch method*), 3163
get_path() (*matplotlib.patches.PathPatch method*), 3171
get_path() (*matplotlib.patches.Polygon method*), 3177
get_path() (*matplotlib.patches.Rectangle method*), 3181
get_path() (*matplotlib.patches.RegularPolygon method*), 3186
get_path() (*matplotlib.patches.Shadow method*), 3189
get_path() (*matplotlib.patches.Wedge method*), 3191
get_path() (*matplotlib.spines.Spine method*), 3639
get_path() (*matplotlib.table.Cell method*), 3647
get_path() (*mpl_toolkits.axes_grid1.inset_locator.BboxConnector method*), 3962
get_path() (*mpl_toolkits.axes_grid1.inset_locator.BboxConnectorPatch method*), 3964
get_path() (*mpl_toolkits.axes_grid1.inset_locator.BboxPatch method*), 3967
get_path() (*mpl_toolkits.mplot3d.art3d.Patch3D method*), 3886
get_path_collection_extents() (*in module matplotlib.path*), 3202
get_path_effects() (*matplotlib.artist.Artist method*), 1888
get_path_effects() (*matplotlib.collections.AsteriskPolygonCollection method*), 2394
get_path_effects() (*matplotlib.collections.CircleCollection method*), 2416
get_path_effects() (*matplotlib.collections.Collection method*), 2439
get_path_effects() (*matplotlib.collections.EllipseCollection method*), 2461
get_path_effects() (*matplotlib.collections.EventCollection method*), 2484
get_path_effects() (*matplotlib.collections.LineCollection method*), 2507
get_path_effects() (*matplotlib.collections.PatchCollection method*), 2529
get_path_effects() (*matplotlib.collections.PathCollection method*), 2550
get_path_effects() (*matplotlib.collections.PolyCollection method*), 2573
get_path_effects() (*matplotlib.collections.PolyQuadMesh method*), 2596
get_path_effects() (*matplotlib.collections.QuadMesh method*), 2620
get_path_effects() (*matplotlib.collections.RegularPolyCollection method*), 2642
get_path_effects() (*matplotlib.collections.StarPolygonCollection method*), 2664
get_path_effects() (*matplotlib.collections.TriMesh method*), 2688
get_paths() (*matplotlib.collections.AsteriskPolygonCollection method*), 2394
get_paths() (*matplotlib.collections.CircleCollection method*), 2416
get_paths() (*matplotlib.collections.Collection method*), 2439
get_paths() (*matplotlib.collections.EllipseCollection method*), 2461
get_paths() (*matplotlib.collections.EventCollection method*), 2484
get_paths() (*matplotlib.collections.LineCollection method*), 2507
get_paths() (*matplotlib.collections.PatchCollection method*), 2529
get_paths() (*matplotlib.collections.PathCollection method*), 2550
get_paths() (*matplotlib.collections.PolyCollection method*), 2573
get_paths() (*matplotlib.collections.PolyQuadMesh method*), 2596
get_paths() (*matplotlib.collections.QuadMesh method*), 2620
get_paths() (*matplotlib.collections.RegularPolyCollection method*), 2642
get_paths() (*matplotlib.collections.StarPolygonCollection method*), 2664
get_paths() (*matplotlib.collections.TriMesh method*), 2688
get_picker() (*matplotlib.artist.Artist method*), 1874
get_picker() (*matplotlib.collections.AsteriskPolygonCollection method*), 2395
get_picker() (*matplotlib.collections.CircleCollection method*), 2416
get_picker() (*matplotlib.collections.Collection method*), 2439
get_picker() (*matplotlib.collections.EllipseCollection method*), 2461
get_picker() (*matplotlib.collections.EventCollection method*), 2484
get_picker() (*matplotlib.collections.LineCollection method*), 2507
get_picker() (*matplotlib.collections.PatchCollection method*), 2529
get_picker() (*matplotlib.collections.PathCollection method*), 2550
get_picker() (*matplotlib.collections.PolyCollection method*), 2573
get_picker() (*matplotlib.collections.PolyQuadMesh method*), 2596
get_picker() (*matplotlib.collections.QuadMesh method*), 2620
get_picker() (*matplotlib.collections.RegularPolyCollection*

*method*), 2643
get_picker() (*matplotlib.collections.StarPolygonCollection method*), 2665
get_picker() (*matplotlib.collections.TriMesh method*), 2688
get_pickradius() (*matplotlib.axis.Axis method*), 2240
get_pickradius() (*matplotlib.collections.AsteriskPolygonCollection method*), 2395
get_pickradius() (*matplotlib.collections.CircleCollection method*), 2416
get_pickradius() (*matplotlib.collections.Collection method*), 2440
get_pickradius() (*matplotlib.collections.EllipseCollection method*), 2461
get_pickradius() (*matplotlib.collections.EventCollection method*), 2484
get_pickradius() (*matplotlib.collections.LineCollection method*), 2507
get_pickradius() (*matplotlib.collections.PatchCollection method*), 2529
get_pickradius() (*matplotlib.collections.PathCollection method*), 2550
get_pickradius() (*matplotlib.collections.PolyCollection method*), 2573
get_pickradius() (*matplotlib.collections.PolyQuadMesh method*), 2596
get_pickradius() (*matplotlib.collections.QuadMesh method*), 2621
get_pickradius() (*matplotlib.collections.RegularPolyCollection method*), 2643
get_pickradius() (*matplotlib.collections.StarPolygonCollection method*), 2665
get_pickradius() (*matplotlib.collections.TriMesh method*), 2688
get_pickradius() (*matplotlib.lines.Line2D method*), 3020
get_points() (*matplotlib.transforms.Bbox method*), 3728
get_points() (*matplotlib.transforms.BboxBase method*), 3733
get_points() (*matplotlib.transforms.LockableBbox method*), 3744
get_points() (*matplotlib.transforms.TransformedBbox method*), 3752
get_position() (*matplotlib.axes.Axes method*), 2194
get_position() (*matplotlib.gridspec.SubplotSpec method*), 2960
get_position() (*matplotlib.spines.Spine method*), 3639
get_position() (*matplotlib.text.Text method*), 3666
get_position() (*mpl_toolkits.axes_grid1.axes_divider.AxesDivider method*), 3931
get_position() (*mpl_toolkits.axes_grid1.axes_divider.Divider method*), 3933
get_position() (*mpl_toolkits.axes_grid1.axes_divider.SubplotDivider method*), 3937
get_position_3d() (*mpl_toolkits.mplot3d.art3d.Text3D method*), 3901
get_position_runtime() (*mpl_toolkits.axes_grid1.axes_divider.Divider method*), 3933
get_positions() (*matplotlib.collections.EventCollection method*), 2484
get_proj() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3868
get_projection_class() (*in module matplotlib.projections*), 3527
get_projection_class() (*matplotlib.projections.ProjectionRegistry method*), 3527
get_projection_names() (*in module matplotlib.projections*), 3527
get_projection_names() (*matplotlib.projections.ProjectionRegistry method*), 3527
get_ps_font_info() (*matplotlib.ft2font.FT2Font method*), 2954
get_radii() (*matplotlib.patches.Annulus method*), 3098
get_radius() (*matplotlib.patches.Circle method*), 3123
get_rasterization_zorder() (*matplotlib.axes.Axes method*), 2206
get_rasterized() (*matplotlib.artist.Artist method*), 1887
get_rasterized() (*matplotlib.collections.AsteriskPolygonCollection method*), 2395
get_rasterized() (*matplotlib.collections.CircleCollection method*), 2416
get_rasterized() (*matplotlib.collections.Collection method*), 2440
get_rasterized() (*matplotlib.collections.EllipseCollection method*), 2461
get_rasterized() (*matplotlib.collections.EventCollection method*), 2484
get_rasterized() (*matplotlib.collections.LineCollection method*), 2507
get_rasterized() (*matplotlib.collections.PatchCollection method*), 2529
get_rasterized() (*matplotlib.collections.PathCollection method*), 2550
get_rasterized() (*matplotlib.collections.PolyCollection method*), 2573
get_rasterized() (*matplotlib.collections.PolyQuadMesh method*), 2596
get_rasterized() (*matplotlib.collections.QuadMesh method*), 2621
get_rasterized() (*matplotlib.collections.RegularPolyCollection method*), 2643
get_rasterized()

(*matplotlib.collections.StarPolygonCollection method*), 2665
get_rasterized() (*matplotlib.collections.TriMesh method*), 2688
get_ref_artist() (*mpl_toolkits.axisartist.axis_artist.AttributeCopier method*), 4004
get_ref_artist() (*mpl_toolkits.axisartist.axis_artist.AxisLabel method*), 4009
get_ref_artist() (*mpl_toolkits.axisartist.axis_artist.TickLabels method*), 4020
get_ref_artist() (*mpl_toolkits.axisartist.axis_artist.Ticks method*), 4023
get_registered_canvas_class() (*in module matplotlib.backend_bases*), 2283
get_remove_overlapping_locs() (*matplotlib.axis.Axis method*), 2227
get_renderer() (*matplotlib.backends.backend_agg.FigureCanvasAgg method*), 2309
get_renderer() (*matplotlib.backends.backend_cairo.FigureCanvasCairo method*), 2315
get_renderer() (*matplotlib.backends.backend_pgf.FigureCanvasPgf method*), 2335
get_required_width() (*matplotlib.table.Cell method*), 3647
get_rgb() (*matplotlib.backend_bases.GraphicsContextBase method*), 2265
get_rgb() (*matplotlib.backends.backend_cairo.GraphicsContextCairo method*), 2316
get_rgba() (*matplotlib.texmanager.TexManager class method*), 3689
get_rlabel_position() (*matplotlib.projections.polar.PolarAxes method*), 3536
get_rmax() (*matplotlib.projections.polar.PolarAxes method*), 3536
get_rmin() (*matplotlib.projections.polar.PolarAxes method*), 3537
get_rorigin() (*matplotlib.projections.polar.PolarAxes method*), 3537
get_rotate_label() (*mpl_toolkits.mplot3d.axis3d.Axis method*), 3876
get_rotation() (*matplotlib.collections.AsteriskPolygonCollection method*), 2395
get_rotation() (*matplotlib.collections.RegularPolyCollection method*), 2643
get_rotation() (*matplotlib.collections.StarPolygonCollection method*), 2665
get_rotation() (*matplotlib.text.Text method*), 3666
get_rotation_mode() (*matplotlib.text.Text method*), 3666
get_rsign() (*matplotlib.projections.polar.PolarAxes method*), 3537
get_sample_data() (*in module matplotlib.cbook*), 2377
get_scale() (*matplotlib.axis.Axis method*), 2222
get_scale_names() (*in module matplotlib.scale*), 3629
get_segments() (*matplotlib.collections.EventCollection method*), 2484
get_segments() (*matplotlib.collections.LineCollection method*), 2507
get_setters() (*matplotlib.artist.ArtistInspector method*), 1901
get_sfnt() (*matplotlib.ft2font.FT2Font method*), 2954
get_sfnt_table() (*matplotlib.ft2font.FT2Font method*), 2954
get_shared_x_axes() (*matplotlib.axes.Axes method*), 2191
get_shared_y_axes() (*matplotlib.axes.Axes method*), 2191
get_siblings() (*matplotlib.cbook.Grouper method*), 2373
get_siblings() (*matplotlib.cbook.GrouperView method*), 2373
get_size() (*matplotlib.font_manager.FontProperties method*), 2945
get_size() (*matplotlib.text.Text method*), 3666
get_size() (*matplotlib.text.TextPath method*), 3686
get_size() (*mpl_toolkits.axes_grid1.axes_size.Add method*), 3949
get_size() (*mpl_toolkits.axes_grid1.axes_size.AxesX method*), 3949
get_size() (*mpl_toolkits.axes_grid1.axes_size.AxesY method*), 3950
get_size() (*mpl_toolkits.axes_grid1.axes_size.Fixed method*), 3950
get_size() (*mpl_toolkits.axes_grid1.axes_size.Fraction method*), 3951
get_size() (*mpl_toolkits.axes_grid1.axes_size.MaxExtent method*), 3951
get_size() (*mpl_toolkits.axes_grid1.axes_size.Scaled method*), 3952
get_size_in_points() (*matplotlib.font_manager.FontProperties method*), 2946
get_size_inches() (*matplotlib.figure.Figure method*), 2842
get_sizes() (*matplotlib.collections.AsteriskPolygonCollection method*), 2395
get_sizes() (*matplotlib.collections.CircleCollection method*), 2416
get_sizes() (*matplotlib.collections.PathCollection method*), 2550
get_sizes() (*matplotlib.collections.PolyCollection method*), 2573
get_sizes() (*matplotlib.collections.PolyQuadMesh method*), 2596
get_sizes() (*matplotlib.collections.RegularPolyCollection method*), 2643

get_sizes() (*matplotlib.collections.StarPolygonCollection method*), 2665
get_sizes() (*matplotlib.legend_handler.HandlerRegularPolyCollection method*), 3012
get_sketch_params() (*matplotlib.artist.Artist method*), 1887
get_sketch_params() (*matplotlib.backend_bases.GraphicsContextBase method*), 2265
get_sketch_params() (*matplotlib.collections.AsteriskPolygonCollection method*), 2395
get_sketch_params() (*matplotlib.collections.CircleCollection method*), 2416
get_sketch_params() (*matplotlib.collections.Collection method*), 2440
get_sketch_params() (*matplotlib.collections.EllipseCollection method*), 2462
get_sketch_params() (*matplotlib.collections.EventCollection method*), 2484
get_sketch_params() (*matplotlib.collections.LineCollection method*), 2507
get_sketch_params() (*matplotlib.collections.PatchCollection method*), 2529
get_sketch_params() (*matplotlib.collections.PathCollection method*), 2551
get_sketch_params() (*matplotlib.collections.PolyCollection method*), 2574
get_sketch_params() (*matplotlib.collections.PolyQuadMesh method*), 2596
get_sketch_params() (*matplotlib.collections.QuadMesh method*), 2621
get_sketch_params() (*matplotlib.collections.RegularPolyCollection method*), 2643
get_sketch_params() (*matplotlib.collections.StarPolygonCollection method*), 2665
get_sketch_params() (*matplotlib.collections.TriMesh method*), 2688
get_slant() (*matplotlib.font_manager.FontProperties method*), 2946
get_slope() (*matplotlib.lines.AxLine method*), 3035
get_snap() (*matplotlib.artist.Artist method*), 1884
get_snap() (*matplotlib.backend_bases.GraphicsContextBase method*), 2265
get_snap() (*matplotlib.collections.AsteriskPolygonCollection method*), 2395
get_snap() (*matplotlib.collections.CircleCollection method*), 2417
get_snap() (*matplotlib.collections.Collection method*), 2440
get_snap() (*matplotlib.collections.EllipseCollection method*), 2462
get_snap() (*matplotlib.collections.EventCollection method*), 2485
get_snap() (*matplotlib.collections.LineCollection method*), 2508
get_snap() (*matplotlib.collections.PatchCollection method*), 2530
get_snap() (*matplotlib.collections.PathCollection method*), 2551
get_snap() (*matplotlib.collections.PolyCollection method*), 2574
get_snap() (*matplotlib.collections.PolyQuadMesh method*), 2597
get_snap() (*matplotlib.collections.QuadMesh method*), 2621
get_snap() (*matplotlib.collections.RegularPolyCollection method*), 2643
get_snap() (*matplotlib.collections.StarPolygonCollection method*), 2665
get_snap() (*matplotlib.collections.TriMesh method*), 2688
get_snap_threshold() (*matplotlib.markers.MarkerStyle method*), 3041
get_solid_capstyle() (*matplotlib.lines.Line2D method*), 3021
get_solid_joinstyle() (*matplotlib.lines.Line2D method*), 3021
get_spine_transform() (*matplotlib.spines.Spine method*), 3639
get_static_file_path() (*matplotlib.backends.backend_webagg_core.FigureManagerWebAgg class method*), 2360
get_status() (*matplotlib.widgets.CheckButtons method*), 3776
get_str_bbox() (*matplotlib._afm.AFM method*), 1830
get_str_bbox_and_descent() (*matplotlib._afm.AFM method*), 1830
get_stretch() (*matplotlib.font_manager.FontProperties method*), 2946
get_stretch() (*matplotlib.text.Text method*), 3666
get_style() (*matplotlib.font_manager.FontProperties method*), 2946
get_style() (*matplotlib.text.Text method*), 3666
get_subplot_params() (*matplotlib.gridspec.GridSpec method*), 2958
get_subplot_params() (*matplotlib.gridspec.GridSpecBase method*), 2963
get_subplot_params() (*matplotlib.gridspec.GridSpecFromSubplotSpec method*), 2964
get_subplotspec() (*matplotlib.axes.Axes method*), 2194
get_subplotspec() (*mpl_toolkits.axes_grid1.axes_divider.AxesDivider method*), 3931
get_subplotspec()

(*mpl_toolkits.axes_grid1.axes_divider.AxesLocator method*), 3932
get_subplotspec() (*mpl_toolkits.axes_grid1.axes_divider.Divider method*), 3933
get_subplotspec() (*mpl_toolkits.axes_grid1.axes_divider.SubplotDivider method*), 3937
get_subplotspec_list() (*in module matplotlib._tight_layout*), 3716
get_supported_filetypes() (*matplotlib.backend_bases.FigureCanvasBase class method*), 2256
get_supported_filetypes_grouped() (*matplotlib.backend_bases.FigureCanvasBase class method*), 2256
get_suptitle() (*matplotlib.figure.Figure method*), 2834
get_suptitle() (*matplotlib.figure.FigureBase method*), 2919
get_suptitle() (*matplotlib.figure.SubFigure method*), 2894
get_supxlabel() (*matplotlib.figure.Figure method*), 2836
get_supxlabel() (*matplotlib.figure.FigureBase method*), 2919
get_supxlabel() (*matplotlib.figure.SubFigure method*), 2895
get_supylabel() (*matplotlib.figure.Figure method*), 2837
get_supylabel() (*matplotlib.figure.FigureBase method*), 2919
get_supylabel() (*matplotlib.figure.SubFigure method*), 2896
get_test_data() (*in module mpl_toolkits.mplot3d.axes3d*), 3873
get_texmanager() (*matplotlib.backend_bases.RendererBase method*), 2277
get_text() (*matplotlib.contour.ContourLabeler method*), 2758
get_text() (*matplotlib.offsetbox.TextArea method*), 3090
get_text() (*matplotlib.table.Cell method*), 3647
get_text() (*matplotlib.text.Text method*), 3666
get_text() (*mpl_toolkits.axisartist.axis_artist.AxisLabel method*), 4009
get_text_bounds() (*matplotlib.table.Cell method*), 3647
get_text_path() (*matplotlib.text.TextToPath method*), 3687
get_text_width_height_descent() (*matplotlib.backend_bases.RendererBase method*), 2277
get_text_width_height_descent() (*matplotlib.backends.backend_agg.RendererAgg method*), 2313
get_text_width_height_descent() (*matplotlib.backends.backend_cairo.RendererCairo method*), 2320
get_text_width_height_descent() (*matplotlib.backends.backend_pgf.RendererPgf method*), 2340
get_text_width_height_descent() (*matplotlib.backends.backend_svg.RendererSVG method*), 2355
get_text_width_height_descent() (*matplotlib.backends.backend_template.RendererTemplate method*), 2308
get_text_width_height_descent() (*matplotlib.texmanager.TexManager class method*), 3689
get_text_width_height_descent() (*matplotlib.text.TextToPath method*), 3688
get_texts() (*matplotlib.legend.Legend method*), 2997
get_texts_widths_heights_descents() (*mpl_toolkits.axisartist.axis_artist.TickLabels method*), 4020
get_theta_direction() (*matplotlib.projections.polar.PolarAxes method*), 3537
get_theta_offset() (*matplotlib.projections.polar.PolarAxes method*), 3537
get_thetamax() (*matplotlib.projections.polar.PolarAxes method*), 3537
get_thetamin() (*matplotlib.projections.polar.PolarAxes method*), 3537
get_tick_iterator() (*mpl_toolkits.axisartist.grid_helper_curvelinear.GridHelperCurveLinear method*), 4051
get_tick_iterators() (*mpl_toolkits.axisartist.axislines.FixedAxisArtistHelperRectilinear method*), 4039
get_tick_iterators() (*mpl_toolkits.axisartist.axislines.FloatingAxisArtistHelperRectilinear method*), 4039
get_tick_iterators() (*mpl_toolkits.axisartist.floating_axes.FixedAxisArtistHelper method*), 4042
get_tick_iterators() (*mpl_toolkits.axisartist.grid_helper_curvelinear.FixedAxisArtistHelper method*), 4049
get_tick_iterators() (*mpl_toolkits.axisartist.grid_helper_curvelinear.FloatingAxisArtistHelpe method*), 4050
get_tick_out() (*mpl_toolkits.axisartist.axis_artist.Ticks method*), 4023
get_tick_padding() (*matplotlib.axis.Axis method*), 2232
get_tick_padding() (*matplotlib.axis.Tick method*), 2251
get_tick_params() (*matplotlib.axis.Axis method*), 2232
get_tick_space() (*matplotlib.axis.Axis method*), 2239
get_tick_transform() (*mpl_toolkits.axisartist.axislines.FloatingAxisArtistHelperRectilinear method*), 4039
get_tick_transform() (*mpl_toolkits.axisartist.grid_helper_curvelinear.FixedAxisArtistHelper method*), 4049
get_tick_transform() (*mpl_toolkits.axisartist.grid_helper_curvelinear.FloatingAxisArtistHelpe method*), 4050
get_tickdir() (*matplotlib.axis.Tick method*), 2251
get_ticklabels() (*matplotlib.axis.Axis method*), 2233

get_ticklines() (*matplotlib.axis.Axis method*), 2234
get_ticklocs() (*matplotlib.axis.Axis method*), 2234
get_ticks() (*matplotlib.colorbar.Colorbar method*), 2707
get_ticks_position() (*matplotlib.axis.XAxis method*), 2242
get_ticks_position() (*matplotlib.axis.YAxis method*), 2244
get_ticks_position() (*mpl_toolkits.mplot3d.axis3d.Axis method*), 3876
get_ticksize() (*mpl_toolkits.axisartist.axis_artist.Ticks method*), 4023
get_tight_layout() (*matplotlib.figure.Figure method*), 2851
get_tight_layout_figure() (*in module matplotlib._tight_layout*), 3716
get_tightbbox() (*matplotlib.artist.Artist method*), 1889
get_tightbbox() (*matplotlib.axes.Axes method*), 2207
get_tightbbox() (*matplotlib.axis.Axis method*), 2239
get_tightbbox() (*matplotlib.collections.AsteriskPolygonCollection method*), 2395
get_tightbbox() (*matplotlib.collections.CircleCollection method*), 2417
get_tightbbox() (*matplotlib.collections.Collection method*), 2440
get_tightbbox() (*matplotlib.collections.EllipseCollection method*), 2462
get_tightbbox() (*matplotlib.collections.EventCollection method*), 2485
get_tightbbox() (*matplotlib.collections.LineCollection method*), 2508
get_tightbbox() (*matplotlib.collections.PatchCollection method*), 2530
get_tightbbox() (*matplotlib.collections.PathCollection method*), 2551
get_tightbbox() (*matplotlib.collections.PolyCollection method*), 2574
get_tightbbox() (*matplotlib.collections.PolyQuadMesh method*), 2597
get_tightbbox() (*matplotlib.collections.QuadMesh method*), 2621
get_tightbbox() (*matplotlib.collections.RegularPolyCollection method*), 2643
get_tightbbox() (*matplotlib.collections.StarPolygonCollection method*), 2665
get_tightbbox() (*matplotlib.collections.TriMesh method*), 2688
get_tightbbox() (*matplotlib.figure.Figure method*), 2861
get_tightbbox() (*matplotlib.figure.FigureBase method*), 2919
get_tightbbox() (*matplotlib.legend.Legend method*), 2997
get_tightbbox() (*matplotlib.offsetbox.AnnotationBbox method*), 3072
get_tightbbox() (*matplotlib.text.Annotation method*), 3682
get_tightbbox() (*mpl_toolkits.axes_grid1.parasite_axes.HostAxesBase method*), 3986
get_tightbbox() (*mpl_toolkits.axisartist.axis_artist.AxisArtist method*), 4005
get_tightbbox() (*mpl_toolkits.mplot3d.art3d.Text3D method*), 3901
get_tightbbox() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3869
get_tightbbox() (*mpl_toolkits.mplot3d.axis3d.Axis method*), 3876
get_title() (*matplotlib.axes.Axes method*), 2143
get_title() (*matplotlib.legend.Legend method*), 2998
get_tool() (*matplotlib.backend_managers.ToolManager method*), 2285
get_tool_keymap() (*matplotlib.backend_managers.ToolManager method*), 2285
get_topmost_subplotspec() (*matplotlib.gridspec.GridSpecFromSubplotSpec method*), 2964
get_topmost_subplotspec() (*matplotlib.gridspec.SubplotSpec method*), 2960
get_transform() (*matplotlib.artist.Artist method*), 1892
get_transform() (*matplotlib.collections.AsteriskPolygonCollection method*), 2396
get_transform() (*matplotlib.collections.CircleCollection method*), 2417
get_transform() (*matplotlib.collections.Collection method*), 2441
get_transform() (*matplotlib.collections.EllipseCollection method*), 2462
get_transform() (*matplotlib.collections.EventCollection method*), 2485
get_transform() (*matplotlib.collections.LineCollection method*), 2508
get_transform() (*matplotlib.collections.PatchCollection method*), 2530
get_transform() (*matplotlib.collections.PathCollection method*), 2551
get_transform() (*matplotlib.collections.PolyCollection method*), 2574
get_transform() (*matplotlib.collections.PolyQuadMesh method*), 2597
get_transform() (*matplotlib.collections.QuadMesh method*), 2622
get_transform() (*matplotlib.collections.RegularPolyCollection method*), 2644
get_transform() (*matplotlib.collections.StarPolygonCollection method*), 2666
get_transform() (*matplotlib.collections.TriMesh method*), 2689
get_transform() (*matplotlib.contour.ContourSet method*), 2762
get_transform() (*matplotlib.lines.AxLine method*), 3035
get_transform() (*matplotlib.markers.MarkerStyle*

*method*), 3041
get_transform() (*matplotlib.offsetbox.AuxTransformBox method*), 3074
get_transform() (*matplotlib.offsetbox.DrawingArea method*), 3077
get_transform() (*matplotlib.patches.Patch method*), 3163
get_transform() (*matplotlib.scale.AsinhScale method*), 3615
get_transform() (*matplotlib.scale.FuncScale method*), 3617
get_transform() (*matplotlib.scale.FuncScaleLog method*), 3617
get_transform() (*matplotlib.scale.LinearScale method*), 3622
get_transform() (*matplotlib.scale.LogitScale method*), 3625
get_transform() (*matplotlib.scale.LogScale method*), 3622
get_transform() (*matplotlib.scale.ScaleBase method*), 3627
get_transform() (*matplotlib.scale.SymmetricalLogScale method*), 3628
get_transform() (*mpl_toolkits.axisartist.axis_artist.AxisArtist method*), 4005
get_transform() (*mpl_toolkits.axisartist.grid_finder.GridFinder method*), 4047
get_transform_rotates_text() (*matplotlib.text.Text method*), 3666
get_transformed_clip_path_and_affine() (*matplotlib.artist.Artist method*), 1889
get_transformed_clip_path_and_affine() (*matplotlib.axes.Axes method*), 2213
get_transformed_clip_path_and_affine() (*matplotlib.collections.AsteriskPolygonCollection method*), 2396
get_transformed_clip_path_and_affine() (*matplotlib.collections.CircleCollection method*), 2417
get_transformed_clip_path_and_affine() (*matplotlib.collections.Collection method*), 2441
get_transformed_clip_path_and_affine() (*matplotlib.collections.EllipseCollection method*), 2462
get_transformed_clip_path_and_affine() (*matplotlib.collections.EventCollection method*), 2485
get_transformed_clip_path_and_affine() (*matplotlib.collections.LineCollection method*), 2508
get_transformed_clip_path_and_affine() (*matplotlib.collections.PatchCollection method*), 2530
get_transformed_clip_path_and_affine() (*matplotlib.collections.PathCollection method*), 2551
get_transformed_clip_path_and_affine() (*matplotlib.collections.PolyCollection method*), 2574
get_transformed_clip_path_and_affine() (*matplotlib.collections.PolyQuadMesh method*), 2597
get_transformed_clip_path_and_affine() (*matplotlib.collections.QuadMesh method*), 2622
get_transformed_clip_path_and_affine() (*matplotlib.collections.RegularPolyCollection method*), 2644
get_transformed_clip_path_and_affine() (*matplotlib.collections.StarPolygonCollection method*), 2666
get_transformed_clip_path_and_affine() (*matplotlib.collections.TriMesh method*), 2689
get_transformed_path_and_affine() (*matplotlib.transforms.TransformedPath method*), 3752
get_transformed_points_and_affine() (*matplotlib.transforms.TransformedPath method*), 3752
get_transforms() (*matplotlib.collections.AsteriskPolygonCollection method*), 2396
get_transforms() (*matplotlib.collections.CircleCollection method*), 2417
get_transforms() (*matplotlib.collections.Collection method*), 2441
get_transforms() (*matplotlib.collections.EllipseCollection method*), 2462
get_transforms() (*matplotlib.collections.EventCollection method*), 2485
get_transforms() (*matplotlib.collections.LineCollection method*), 2508
get_transforms() (*matplotlib.collections.PatchCollection method*), 2530
get_transforms() (*matplotlib.collections.PathCollection method*), 2551
get_transforms() (*matplotlib.collections.PolyCollection method*), 2574
get_transforms() (*matplotlib.collections.PolyQuadMesh method*), 2597
get_transforms() (*matplotlib.collections.QuadMesh method*), 2622
get_transforms() (*matplotlib.collections.RegularPolyCollection method*), 2644
get_transforms() (*matplotlib.collections.StarPolygonCollection method*), 2666
get_transforms() (*matplotlib.collections.TriMesh method*), 2689
get_transOffset() (*matplotlib.collections.AsteriskPolygonCollection method*), 2396
get_transOffset() (*matplotlib.collections.CircleCollection method*), 2417

get_transOffset() (*matplotlib.collections.Collection method*), 2440
get_transOffset() (*matplotlib.collections.EllipseCollection method*), 2462
get_transOffset() (*matplotlib.collections.EventCollection method*), 2485
get_transOffset() (*matplotlib.collections.LineCollection method*), 2508
get_transOffset() (*matplotlib.collections.PatchCollection method*), 2530
get_transOffset() (*matplotlib.collections.PathCollection method*), 2551
get_transOffset() (*matplotlib.collections.PolyCollection method*), 2574
get_transOffset() (*matplotlib.collections.PolyQuadMesh method*), 2597
get_transOffset() (*matplotlib.collections.QuadMesh method*), 2621
get_transOffset() (*matplotlib.collections.RegularPolyCollection method*), 2644
get_transOffset() (*matplotlib.collections.StarPolygonCollection method*), 2666
get_transOffset() (*matplotlib.collections.TriMesh method*), 2689
get_trifinder() (*matplotlib.tri.Triangulation method*), 3756
get_under() (*matplotlib.colors.Colormap method*), 2733
get_underline_thickness() (*matplotlib._afm.AFM method*), 1830
get_unicode_index() (*in module matplotlib.mathtext*), 3045
get_unit() (*matplotlib.text.OffsetFrom method*), 3685
get_unit_generic() (*matplotlib.dates.RRuleLocator static method*), 2780
get_unitless_position() (*matplotlib.text.Text method*), 3666
get_units() (*matplotlib.axis.Axis method*), 2241
get_url() (*matplotlib.artist.Artist method*), 1895
get_url() (*matplotlib.backend_bases.GraphicsContextBase method*), 2265
get_url() (*matplotlib.collections.AsteriskPolygonCollection method*), 2396
get_url() (*matplotlib.collections.CircleCollection method*), 2417
get_url() (*matplotlib.collections.Collection method*), 2441
get_url() (*matplotlib.collections.EllipseCollection method*), 2462
get_url() (*matplotlib.collections.EventCollection method*), 2485
get_url() (*matplotlib.collections.LineCollection method*), 2508
get_url() (*matplotlib.collections.PatchCollection method*), 2530
get_url() (*matplotlib.collections.PathCollection method*), 2552
get_url() (*matplotlib.collections.PolyCollection method*), 2575
get_url() (*matplotlib.collections.PolyQuadMesh method*), 2597
get_url() (*matplotlib.collections.QuadMesh method*), 2622
get_url() (*matplotlib.collections.RegularPolyCollection method*), 2644
get_url() (*matplotlib.collections.StarPolygonCollection method*), 2666
get_url() (*matplotlib.collections.TriMesh method*), 2689
get_urls() (*matplotlib.collections.AsteriskPolygonCollection method*), 2396
get_urls() (*matplotlib.collections.CircleCollection method*), 2417
get_urls() (*matplotlib.collections.Collection method*), 2441
get_urls() (*matplotlib.collections.EllipseCollection method*), 2463
get_urls() (*matplotlib.collections.EventCollection method*), 2485
get_urls() (*matplotlib.collections.LineCollection method*), 2508
get_urls() (*matplotlib.collections.PatchCollection method*), 2530
get_urls() (*matplotlib.collections.PathCollection method*), 2552
get_urls() (*matplotlib.collections.PolyCollection method*), 2575
get_urls() (*matplotlib.collections.PolyQuadMesh method*), 2597
get_urls() (*matplotlib.collections.QuadMesh method*), 2622
get_urls() (*matplotlib.collections.RegularPolyCollection method*), 2644
get_urls() (*matplotlib.collections.StarPolygonCollection method*), 2666
get_urls() (*matplotlib.collections.TriMesh method*), 2689
get_useLocale() (*matplotlib.ticker.ScalarFormatter method*), 3710
get_useMathText() (*matplotlib.ticker.EngFormatter method*), 3695
get_useMathText() (*matplotlib.ticker.ScalarFormatter method*), 3710
get_useOffset() (*matplotlib.ticker.ScalarFormatter method*), 3710
get_user_transform() (*matplotlib.markers.MarkerStyle method*), 3041
get_usetex() (*matplotlib.text.Text method*), 3666
get_usetex() (*matplotlib.ticker.EngFormatter method*), 3695
get_va() (*matplotlib.text.Text method*), 3667
get_valid_values() (*matplotlib.artist.ArtistInspector method*), 1901

get_variant() (*matplotlib.font_manager.FontProperties method*), 2946
get_variant() (*matplotlib.text.Text method*), 3667
get_vector() (*mpl_toolkits.mplot3d.art3d.Poly3DCollection method*), 3896
get_vertical() (*mpl_toolkits.axes_grid1.axes_divider.Divider method*), 3933
get_vertical_sizes() (*mpl_toolkits.axes_grid1.axes_divider.Divider method*), 3934
get_vertical_stem_width() (*matplotlib._afm.AFM method*), 1831
get_verticalalignment() (*matplotlib.text.Text method*), 3667
get_vertices() (*matplotlib.patches.Ellipse method*), 3137
get_verts() (*matplotlib.patches.Patch method*), 3163
get_view_interval() (*matplotlib.axis.Axis method*), 2237
get_view_interval() (*matplotlib.axis.Tick method*), 2252
get_viewlim_mode() (*mpl_toolkits.axes_grid1.parasite_axes.ParasiteAxesBase method*), 3987
get_visible() (*matplotlib.artist.Artist method*), 1885
get_visible() (*matplotlib.collections.AsteriskPolygonCollection method*), 2396
get_visible() (*matplotlib.collections.CircleCollection method*), 2417
get_visible() (*matplotlib.collections.Collection method*), 2441
get_visible() (*matplotlib.collections.EllipseCollection method*), 2463
get_visible() (*matplotlib.collections.EventCollection method*), 2485
get_visible() (*matplotlib.collections.LineCollection method*), 2509
get_visible() (*matplotlib.collections.PatchCollection method*), 2530
get_visible() (*matplotlib.collections.PathCollection method*), 2552
get_visible() (*matplotlib.collections.PolyCollection method*), 2575
get_visible() (*matplotlib.collections.PolyQuadMesh method*), 2598
get_visible() (*matplotlib.collections.QuadMesh method*), 2622
get_visible() (*matplotlib.collections.RegularPolyCollection method*), 2644
get_visible() (*matplotlib.collections.StarPolygonCollection method*), 2666
get_visible() (*matplotlib.collections.TriMesh method*), 2689
get_visible_children() (*matplotlib.offsetbox.OffsetBox method*), 3082
get_w_lims() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3847
get_weight() (*matplotlib._afm.AFM method*), 1831
get_weight() (*matplotlib.font_manager.FontProperties method*), 2946
get_weight() (*matplotlib.text.Text method*), 3667
get_width() (*matplotlib.patches.Annulus method*), 3098
get_width() (*matplotlib.patches.Ellipse method*), 3137
get_width() (*matplotlib.patches.FancyBboxPatch method*), 3154
get_width() (*matplotlib.patches.Rectangle method*), 3181
get_width_char() (*matplotlib._afm.AFM method*), 1831
get_width_from_char_name() (*matplotlib._afm.AFM method*), 1831
get_width_height() (*matplotlib.backend_bases.FigureCanvasBase method*), 2256
get_width_height() (*matplotlib.ft2font.FT2Font method*), 2954
get_width_height_descent() (*matplotlib.backends.backend_pgf.LatexManager method*), 2336
get_width_ratios() (*matplotlib.gridspec.GridSpecBase method*), 2963
get_widths() (*matplotlib.collections.EllipseCollection method*), 2463
get_window_extent() (*matplotlib.artist.Artist method*), 1888
get_window_extent() (*matplotlib.axes.Axes method*), 2207
get_window_extent() (*matplotlib.collections.AsteriskPolygonCollection method*), 2396
get_window_extent() (*matplotlib.collections.CircleCollection method*), 2418
get_window_extent() (*matplotlib.collections.Collection method*), 2441
get_window_extent() (*matplotlib.collections.EllipseCollection method*), 2463
get_window_extent() (*matplotlib.collections.EventCollection method*), 2486
get_window_extent() (*matplotlib.collections.LineCollection method*), 2509
get_window_extent() (*matplotlib.collections.PatchCollection method*), 2531
get_window_extent() (*matplotlib.collections.PathCollection method*), 2552
get_window_extent() (*matplotlib.collections.PolyCollection method*), 2575
get_window_extent() (*matplotlib.collections.PolyQuadMesh method*), 2598

get_window_extent() (*matplotlib.collections.QuadMesh method*), 2622
get_window_extent() (*matplotlib.collections.RegularPolyCollection method*), 2644
get_window_extent() (*matplotlib.collections.StarPolygonCollection method*), 2666
get_window_extent() (*matplotlib.collections.TriMesh method*), 2689
get_window_extent() (*matplotlib.figure.Figure method*), 2862
get_window_extent() (*matplotlib.figure.FigureBase method*), 2920
get_window_extent() (*matplotlib.image.AxesImage method*), 2969
get_window_extent() (*matplotlib.image.BboxImage method*), 2971
get_window_extent() (*matplotlib.legend.Legend method*), 2998
get_window_extent() (*matplotlib.lines.Line2D method*), 3021
get_window_extent() (*matplotlib.offsetbox.AnnotationBbox method*), 3072
get_window_extent() (*matplotlib.offsetbox.OffsetBox method*), 3082
get_window_extent() (*matplotlib.patches.Patch method*), 3163
get_window_extent() (*matplotlib.spines.Spine method*), 3639
get_window_extent() (*matplotlib.table.Table method*), 3653
get_window_extent() (*matplotlib.text.Annotation method*), 3682
get_window_extent() (*matplotlib.text.Text method*), 3667
get_window_extent() (*mpl_toolkits.axisartist.axis_artist.AxisLabel method*), 4009
get_window_extent() (*mpl_toolkits.axisartist.axis_artist.LabelBase method*), 4017
get_window_extents() (*mpl_toolkits.axisartist.axis_artist.TickLabels method*), 4020
get_window_title() (*matplotlib.backend_bases.FigureManagerBase method*), 2263
get_window_title() (*matplotlib.backends.backend_webagg_core.FigureManagerWebAgg method*), 2360
get_wrap() (*matplotlib.text.Text method*), 3667
get_x() (*matplotlib.patches.FancyBboxPatch method*), 3154
get_x() (*matplotlib.patches.Rectangle method*), 3181
get_xaxis() (*matplotlib.axes.Axes method*), 2118
get_xaxis_text1_transform() (*matplotlib.axes.Axes method*), 2210
get_xaxis_text1_transform() (*matplotlib.projections.geo.GeoAxes method*), 3565
get_xaxis_text1_transform() (*matplotlib.projections.polar.PolarAxes method*), 3537
get_xaxis_text2_transform() (*matplotlib.axes.Axes method*), 2211
get_xaxis_text2_transform() (*matplotlib.projections.geo.GeoAxes method*), 3566
get_xaxis_text2_transform() (*matplotlib.projections.polar.PolarAxes method*), 3538
get_xaxis_transform() (*matplotlib.axes.Axes method*), 2209
get_xaxis_transform() (*matplotlib.projections.geo.GeoAxes method*), 3566
get_xaxis_transform() (*matplotlib.projections.polar.PolarAxes method*), 3538
get_xbound() (*matplotlib.axes.Axes method*), 2129
get_xbound() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3848
get_xdata() (*matplotlib.legend_handler.HandlerNpoints method*), 3009
get_xdata() (*matplotlib.lines.Line2D method*), 3021
get_xgridlines() (*matplotlib.axes.Axes method*), 2173
get_xheight() (*matplotlib._afm.AFM method*), 1831
get_xlabel() (*matplotlib.axes.Axes method*), 2134
get_xlim() (*matplotlib.axes.Axes method*), 2124
get_xlim() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3841
get_xmajorticklabels() (*matplotlib.axes.Axes method*), 2173
get_xmargin() (*matplotlib.axes.Axes method*), 2157
get_xminorticklabels() (*matplotlib.axes.Axes method*), 2173
get_xscale() (*matplotlib.axes.Axes method*), 2154
get_xticklabels() (*matplotlib.axes.Axes method*), 2172
get_xticklines() (*matplotlib.axes.Axes method*), 2173
get_xticks() (*matplotlib.axes.Axes method*), 2170
get_xy() (*matplotlib.patches.Polygon method*), 3177
get_xy() (*matplotlib.patches.Rectangle method*), 3181
get_xy1() (*matplotlib.lines.AxLine method*), 3035
get_xy2() (*matplotlib.lines.AxLine method*), 3035
get_xydata() (*matplotlib.lines.Line2D method*), 3021
get_xys() (*matplotlib.ft2font.FT2Font method*), 2954
get_y() (*matplotlib.patches.FancyBboxPatch method*), 3154
get_y() (*matplotlib.patches.Rectangle method*), 3181
get_yaxis() (*matplotlib.axes.Axes method*), 2119
get_yaxis_text1_transform() (*matplotlib.axes.Axes method*), 2211
get_yaxis_text1_transform() (*matplotlib.projections.geo.GeoAxes method*), 3566
get_yaxis_text1_transform() (*matplotlib.projections.polar.PolarAxes method*), 3538
get_yaxis_text2_transform() (*matplotlib.axes.Axes method*), 2212
get_yaxis_text2_transform() (*matplotlib.projections.geo.GeoAxes method*), 3567
get_yaxis_text2_transform()

(*matplotlib.projections.polar.PolarAxes method*), 3539
get_yaxis_transform() (*matplotlib.axes.Axes method*), 2209
get_yaxis_transform() (*matplotlib.projections.geo.GeoAxes method*), 3567
get_yaxis_transform() (*matplotlib.projections.polar.PolarAxes method*), 3539
get_ybound() (*matplotlib.axes.Axes method*), 2130
get_ybound() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3849
get_ydata() (*matplotlib.legend_handler.HandlerNpointsYoffsets method*), 3009
get_ydata() (*matplotlib.legend_handler.HandlerStem method*), 3013
get_ydata() (*matplotlib.lines.Line2D method*), 3021
get_ygridlines() (*matplotlib.axes.Axes method*), 2178
get_ylabel() (*matplotlib.axes.Axes method*), 2137
get_ylim() (*matplotlib.axes.Axes method*), 2128
get_ylim() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3843
get_ymajorticklabels() (*matplotlib.axes.Axes method*), 2178
get_ymargin() (*matplotlib.axes.Axes method*), 2158
get_yminorticklabels() (*matplotlib.axes.Axes method*), 2178
get_yscale() (*matplotlib.axes.Axes method*), 2155
get_yticklabels() (*matplotlib.axes.Axes method*), 2177
get_yticklines() (*matplotlib.axes.Axes method*), 2178
get_yticks() (*matplotlib.axes.Axes method*), 2175
get_zaxis() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3841
get_zbound() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3850
get_zgridlines() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3862
get_zlabel() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3851
get_zlim() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3845
get_zmajorticklabels() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3862
get_zmargin() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3854
get_zminorticklabels() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3862
get_zoom() (*matplotlib.offsetbox.OffsetImage method*), 3084
get_zorder() (*matplotlib.artist.Artist method*), 1885
get_zorder() (*matplotlib.collections.AsteriskPolygonCollection method*), 2396
get_zorder() (*matplotlib.collections.CircleCollection method*), 2418
get_zorder() (*matplotlib.collections.Collection method*), 2441
get_zorder() (*matplotlib.collections.EllipseCollection method*), 2463
get_zorder() (*matplotlib.collections.EventCollection method*), 2486
get_zorder() (*matplotlib.collections.LineCollection method*), 2509
get_zorder() (*matplotlib.collections.PatchCollection method*), 2531
get_zorder() (*matplotlib.collections.PathCollection method*), 2552
get_zorder() (*matplotlib.collections.PolyCollection method*), 2575
get_zorder() (*matplotlib.collections.PolyQuadMesh method*), 2598
get_zorder() (*matplotlib.collections.QuadMesh method*), 2622
get_zorder() (*matplotlib.collections.RegularPolyCollection method*), 2644
get_zorder() (*matplotlib.collections.StarPolygonCollection method*), 2666
get_zorder() (*matplotlib.collections.TriMesh method*), 2689
get_zscale() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3853
get_zticklines() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3862
get_zticks() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3860
getp() (*in module matplotlib.artist*), 1898
getp() (*in module matplotlib.pyplot*), 3520
ginput() (*in module matplotlib.pyplot*), 3521
ginput() (*matplotlib.figure.Figure method*), 2853
glyphs (*matplotlib.mathtext.VectorParse attribute*), 3045
grab_frame() (*matplotlib.animation.AbstractMovieWriter method*), 1859
grab_frame() (*matplotlib.animation.FileMovieWriter method*), 1865
grab_frame() (*matplotlib.animation.HTMLWriter method*), 1848
grab_frame() (*matplotlib.animation.MovieWriter method*), 1862
grab_frame() (*matplotlib.animation.PillowWriter method*), 1845
grab_mouse() (*matplotlib.backend_bases.FigureCanvasBase method*), 2256
gradient() (*matplotlib.tri.CubicTriInterpolator method*), 3762
gradient() (*matplotlib.tri.LinearTriInterpolator method*), 3760
GraphicsContextBase (*class in matplotlib.backend_bases*), 2264
GraphicsContextCairo (*class in matplotlib.backends.backend_cairo*), 2316
GraphicsContextPdf (*class in matplotlib.backends.backend_pdf*), 2323

GraphicsContextTemplate (*class in matplotlib.backends.backend_template*), 2305
grestore (*matplotlib.backends.backend_pdf.Op attribute*), 2325
Grid (*class in mpl_toolkits.axes_grid1.axes_grid*), 3941
Grid (*class in mpl_toolkits.axisartist.axes_grid*), 3998
grid() (*in module matplotlib.pyplot*), 3451
grid() (*matplotlib.axes.Axes method*), 2113
grid() (*matplotlib.axis.Axis method*), 2235
grid() (*mpl_toolkits.axisartist.axislines.Axes method*), 4032
grid() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3839
GridFinder (*class in mpl_toolkits.axisartist.grid_finder*), 4047
GridHelperBase (*class in mpl_toolkits.axisartist.axislines*), 4040
GridHelperCurveLinear (*class in mpl_toolkits.axisartist.floating_axes*), 4043
GridHelperCurveLinear (*class in mpl_toolkits.axisartist.grid_helper_curvelinear*), 4050
GridHelperRectlinear (*class in mpl_toolkits.axisartist.axislines*), 4040
GridlinesCollection (*class in mpl_toolkits.axisartist.axis_artist*), 4013
GridSpec (*class in matplotlib.gridspec*), 2957
GridSpecBase (*class in matplotlib.gridspec*), 2962
GridSpecFromSubplotSpec (*class in matplotlib.gridspec*), 2964
Grouper (*class in matplotlib.cbook*), 2372
GrouperView (*class in matplotlib.cbook*), 2373
gs_distill() (*in module matplotlib.backends.backend_ps*), 2346
gsave (*matplotlib.backends.backend_pdf.Op attribute*), 2325
guiEvent (*matplotlib.backend_bases.Event property*), 2254

## H

halfrange (*matplotlib.colors.CenteredNorm property*), 2722
halign (*matplotlib.quiver.QuiverKey attribute*), 3598
HammerAxes (*class in matplotlib.projections.geo*), 3571
HammerAxes.HammerTransform (*class in matplotlib.projections.geo*), 3573
HammerAxes.InvertedHammerTransform (*class in matplotlib.projections.geo*), 3574
HAND (*matplotlib.backend_tools.Cursors attribute*), 2289
handle_ack() (*matplotlib.backends.backend_webagg_core.FigureCanvasWebAggCore method*), 2358
handle_button_press() (*matplotlib.backends.backend_webagg_core.FigureCanvasWebAggCore method*), 2358
handle_button_release() (*matplotlib.backends.backend_webagg_core.FigureCanvasWebAggCore method*), 2358
handle_dblclick() (*matplotlib.backends.backend_webagg_core.FigureCanvasWebAggCore method*), 2358
handle_draw() (*matplotlib.backends.backend_webagg_core.FigureCanvasWebAggCore method*), 2358
handle_event() (*matplotlib.backends.backend_webagg_core.FigureCanvasWebAggCore method*), 2358
handle_figure_enter() (*matplotlib.backends.backend_webagg_core.FigureCanvasWebAggCore method*), 2358
handle_figure_leave() (*matplotlib.backends.backend_webagg_core.FigureCanvasWebAggCore method*), 2359
handle_json() (*matplotlib.backends.backend_webagg_core.FigureManagerWebAgg method*), 2360
handle_key_press() (*matplotlib.backends.backend_webagg_core.FigureCanvasWebAggCore method*), 2359
handle_key_release() (*matplotlib.backends.backend_webagg_core.FigureCanvasWebAggCore method*), 2359
handle_motion_notify() (*matplotlib.backends.backend_webagg_core.FigureCanvasWebAggCore method*), 2359
handle_refresh() (*matplotlib.backends.backend_webagg_core.FigureCanvasWebAggCore method*), 2359
handle_resize() (*matplotlib.backends.backend_webagg_core.FigureCanvasWebAggCore method*), 2359
handle_scroll() (*matplotlib.backends.backend_webagg_core.FigureCanvasWebAggCore method*), 2359
handle_send_image_mode() (*matplotlib.backends.backend_webagg_core.FigureCanvasWebAggCore method*), 2359
handle_set_device_pixel_ratio() (*matplotlib.backends.backend_webagg_core.FigureCanvasWebAggCore method*), 2359
handle_set_dpi_ratio() (*matplotlib.backends.backend_webagg_core.FigureCanvasWebAggCore method*), 2359
handle_toolbar_button() (*matplotlib.backends.backend_webagg_core.FigureCanvasWebAggCore method*), 2359
handle_unknown_event() (*matplotlib.backends.backend_webagg_core.FigureCanvasWebAggCore method*), 2359
HandlerBase (*class in matplotlib.legend_handler*), 3003
HandlerCircleCollection (*class in matplotlib.legend_handler*), 3004
HandlerErrorbar (*class in matplotlib.legend_handler*), 3005
HandlerLine2D (*class in matplotlib.legend_handler*), 3006
HandlerLine2DCompound (*class in matplotlib.legend_handler*), 3007
HandlerLineCollection (*class in matplotlib.legend_handler*), 3008
HandlerNpoints (*class in matplotlib.legend_handler*), 3008
HandlerNpointsYoffsets (*class in matplotlib.legend_handler*), 3009

HandlerPatch (*class in matplotlib.legend_handler*), 3009
HandlerPathCollection (*class in matplotlib.legend_handler*), 3010
HandlerPolyCollection (*class in matplotlib.legend_handler*), 3011
HandlerRegularPolyCollection (*class in matplotlib.legend_handler*), 3011
HandlerStem (*class in matplotlib.legend_handler*), 3012
HandlerStepPatch (*class in matplotlib.legend_handler*), 3014
HandlerTuple (*class in matplotlib.legend_handler*), 3014
has_content (*matplotlib.sphinxext.figmpl_directive.FigureMpl attribute*), 3636
has_content (*matplotlib.sphinxext.mathmpl.MathDirective attribute*), 3631
has_content (*matplotlib.sphinxext.plot_directive.PlotDirective attribute*), 3634
has_data() (*matplotlib.axes.Axes method*), 2213
has_inverse (*matplotlib.projections.geo.AitoffAxes.AitoffTransform attribute*), 3559
has_inverse (*matplotlib.projections.geo.AitoffAxes.InvertedAitoffTransform attribute*), 3559
has_inverse (*matplotlib.projections.geo.HammerAxes.HammerTransform attribute*), 3574
has_inverse (*matplotlib.projections.geo.HammerAxes.InvertedHammerTransform attribute*), 3574
has_inverse (*matplotlib.projections.geo.LambertAxes.InvertedLambertTransform attribute*), 3579
has_inverse (*matplotlib.projections.geo.LambertAxes.LambertTransform attribute*), 3580
has_inverse (*matplotlib.projections.geo.MollweideAxes.InvertedMollweideTransform attribute*), 3585
has_inverse (*matplotlib.projections.geo.MollweideAxes.MollweideTransform attribute*), 3585
has_inverse (*matplotlib.projections.polar.InvertedPolarTransform attribute*), 3528
has_inverse (*matplotlib.projections.polar.PolarAxes.InvertedPolarTransform attribute*), 3532
has_inverse (*matplotlib.projections.polar.PolarAxes.PolarTransform attribute*), 3533
has_inverse (*matplotlib.projections.polar.PolarTransform attribute*), 3547
has_inverse (*matplotlib.scale.AsinhTransform attribute*), 3615
has_inverse (*matplotlib.scale.FuncTransform attribute*), 3618
has_inverse (*matplotlib.scale.InvertedAsinhTransform attribute*), 3619
has_inverse (*matplotlib.scale.InvertedLogTransform attribute*), 3620
has_inverse (*matplotlib.scale.InvertedSymmetricalLogTransform attribute*), 3621
has_inverse (*matplotlib.scale.LogisticTransform attribute*), 3624
has_inverse (*matplotlib.scale.LogitTransform attribute*), 3625
has_inverse (*matplotlib.scale.LogTransform attribute*), 3623
has_inverse (*matplotlib.scale.SymmetricalLogTransform attribute*), 3628
has_inverse (*matplotlib.transforms.Affine2DBase attribute*), 3722
has_inverse (*matplotlib.transforms.BlendedGenericTransform property*), 3738
has_inverse (*matplotlib.transforms.CompositeGenericTransform property*), 3740
has_inverse (*matplotlib.transforms.Transform attribute*), 3747
has_inverse (*matplotlib.transforms.TransformWrapper property*), 3751
HashableList (*in module matplotlib.typing*), 3769
hatch() (*matplotlib.path.Path static method*), 3199
hatch_cmd() (*matplotlib.backends.backend_pdf.GraphicsContextPdf method*), 2323
hatchPattern() (*matplotlib.backends.backend_pdf.PdfFile method*), 2327
HatchPatternBase (*class in matplotlib.hatch*), 2966
have_units() (*matplotlib.artist.Artist method*), 1893
have_units() (*matplotlib.axes.Axes method*), 2184
have_units() (*matplotlib.collections.AsteriskPolygonCollection method*), 2396
have_units() (*matplotlib.collections.CircleCollection method*), 2418
have_units() (*matplotlib.collections.Collection method*), 2441
have_units() (*matplotlib.collections.EllipseCollection method*), 2463
have_units() (*matplotlib.collections.EventCollection method*), 2486
have_units() (*matplotlib.collections.LineCollection method*), 2509
have_units() (*matplotlib.collections.PatchCollection method*), 2531
have_units() (*matplotlib.collections.PathCollection method*), 2552
have_units() (*matplotlib.collections.PolyCollection method*), 2575
have_units() (*matplotlib.collections.PolyQuadMesh method*), 2598

have_units() (*matplotlib.collections.QuadMesh method*), 2622
have_units() (*matplotlib.collections.RegularPolyCollection method*), 2644
have_units() (*matplotlib.collections.StarPolygonCollection method*), 2666
have_units() (*matplotlib.collections.TriMesh method*), 2689
HBoxDivider (*class in mpl_toolkits.axes_grid1.axes_divider*), 3935
height (*matplotlib.dviread.Tfm attribute*), 2791
height (*matplotlib.ft2font.FT2Font attribute*), 2954
height (*matplotlib.mathtext.RasterParse attribute*), 3044
height (*matplotlib.mathtext.VectorParse attribute*), 3045
height (*matplotlib.patches.Ellipse property*), 3137
height (*matplotlib.transforms.BboxBase property*), 3733
hexbin() (*in module matplotlib.pyplot*), 3350
hexbin() (*matplotlib.axes.Axes method*), 2016
hillshade() (*matplotlib.colors.LightSource method*), 2743
hist() (*in module matplotlib.pyplot*), 3355
hist() (*matplotlib.axes.Axes method*), 2020
hist2d() (*in module matplotlib.pyplot*), 3359
hist2d() (*matplotlib.axes.Axes method*), 2025
hlines() (*in module matplotlib.pyplot*), 3299
hlines() (*matplotlib.axes.Axes method*), 1961
hms0d (*matplotlib.dates.DateLocator attribute*), 2777
HOME, 9, 20
home() (*matplotlib.backend_bases.NavigationToolbar2 method*), 2271
home() (*matplotlib.backend_tools.ToolViewsPositions method*), 2298
home() (*matplotlib.cbook.Stack method*), 2374
HorizontalHatch (*class in matplotlib.hatch*), 2966
host_axes() (*in module mpl_toolkits.axes_grid1.parasite_axes*), 3988
host_axes_class_factory() (*in module mpl_toolkits.axes_grid1.parasite_axes*), 3988
host_subplot() (*in module mpl_toolkits.axes_grid1.parasite_axes*), 3989
host_subplot_class_factory() (*in module mpl_toolkits.axes_grid1.parasite_axes*), 3989
HostAxes (*in module mpl_toolkits.axes_grid1.parasite_axes*), 3985
HostAxesBase (*class in mpl_toolkits.axes_grid1.parasite_axes*), 3986
HourLocator (*class in matplotlib.dates*), 2777
HPacker (*class in matplotlib.offsetbox*), 3079
hsv_to_rgb() (*in module matplotlib.colors*), 2748
HTMLWriter (*class in matplotlib.animation*), 1846

## I

id (*matplotlib.backends.backend_pdf.Stream attribute*), 2335
IdentityTransform (*class in matplotlib.transforms*), 3741
ignore() (*matplotlib.transforms.Bbox method*), 3728
ignore() (*matplotlib.widgets.Widget method*), 3802
image (*matplotlib.backend_tools.ConfigureSubplotsBase attribute*), 2289
image (*matplotlib.backend_tools.SaveFigureBase attribute*), 2290
image (*matplotlib.backend_tools.ToolBack attribute*), 2290
image (*matplotlib.backend_tools.ToolBase attribute*), 2291
image (*matplotlib.backend_tools.ToolForward attribute*), 2292
image (*matplotlib.backend_tools.ToolHelpBase attribute*), 2294
image (*matplotlib.backend_tools.ToolHome attribute*), 2294
image (*matplotlib.backend_tools.ToolPan attribute*), 2295
image (*matplotlib.backend_tools.ToolZoom attribute*), 2299
image (*matplotlib.mathtext.RasterParse attribute*), 3044
image_comparison() (*in module matplotlib.testing.decorators*), 3661
ImageComparisonFailure, 3662
ImageGrid (*class in mpl_toolkits.axes_grid1.axes_grid*), 3944
ImageGrid (*class in mpl_toolkits.axisartist.axes_grid*), 3999
ImageMagickBase (*class in matplotlib.animation*), 1867
ImageMagickFileWriter (*class in matplotlib.animation*), 1854
ImageMagickWriter (*class in matplotlib.animation*), 1851
imageObject() (*matplotlib.backends.backend_pdf.PdfFile method*), 2327
imread() (*in module matplotlib.image*), 2981
imread() (*in module matplotlib.pyplot*), 3483
imsave() (*in module matplotlib.image*), 2982
imsave() (*in module matplotlib.pyplot*), 3484
imshow() (*in module matplotlib.pyplot*), 3376
imshow() (*matplotlib.axes.Axes method*), 2041
imshow_rgb() (*mpl_toolkits.axes_grid1.axes_rgb.RGBAxes method*), 3946
in_axes() (*matplotlib.axes.Axes method*), 2202
inaxes() (*matplotlib.backend_bases.FigureCanvasBase method*), 2256
index_of() (*in module matplotlib.cbook*), 2377
IndexLocator (*class in matplotlib.ticker*), 3697
indicate_inset() (*matplotlib.axes.Axes method*), 2088
indicate_inset_zoom() (*matplotlib.axes.Axes method*), 2090
infodict() (*matplotlib.backends.backend_pdf.PdfPages method*), 2329
init3d() (*mpl_toolkits.mplot3d.axis3d.Axis method*), 3876
initialize() (*matplotlib.backends.backend_webagg.WebAggApplication class method*), 2364
initialized (*matplotlib.backends.backend_webagg.WebAggApplication attribute*), 2364
input_dims (*matplotlib.projections.polar.InvertedPolarTransform attribute*), 3528
input_dims (*matplotlib.projections.polar.PolarAxes.InvertedPolarTransform attribute*), 3532
input_dims (*matplotlib.projections.polar.PolarAxes.PolarTransform attribute*), 3533

input_dims (*matplotlib.projections.polar.PolarTransform attribute*), 3547
input_dims (*matplotlib.scale.AsinhTransform attribute*), 3615
input_dims (*matplotlib.scale.FuncTransform attribute*), 3618
input_dims (*matplotlib.scale.InvertedAsinhTransform attribute*), 3619
input_dims (*matplotlib.scale.InvertedLogTransform attribute*), 3620
input_dims (*matplotlib.scale.InvertedSymmetricalLogTransform attribute*), 3621
input_dims (*matplotlib.scale.LogisticTransform attribute*), 3624
input_dims (*matplotlib.scale.LogitTransform attribute*), 3625
input_dims (*matplotlib.scale.LogTransform attribute*), 3623
input_dims (*matplotlib.scale.SymmetricalLogTransform attribute*), 3628
input_dims (*matplotlib.transforms.Affine2DBase attribute*), 3722
input_dims (*matplotlib.transforms.BlendedGenericTransform attribute*), 3738
input_dims (*matplotlib.transforms.Transform attribute*), 3747
input_dims (*matplotlib.transforms.TransformWrapper property*), 3751
input_names (*matplotlib.animation.ImageMagickFileWriter property*), 1855
input_names (*matplotlib.animation.ImageMagickWriter attribute*), 1852
inset_axes() (*in module mpl_toolkits.axes_grid1.inset_locator*), 3969
inset_axes() (*matplotlib.axes.Axes method*), 2087
InsetPosition (*class in mpl_toolkits.axes_grid1.inset_locator*), 3968
inside_circle() (*in module matplotlib.bezier*), 2367
install_repl_displayhook() (*in module matplotlib.pyplot*), 3493
INTERACTIVE (*matplotlib.backends.registry.BackendFilter attribute*), 2346
interactive() (*in module matplotlib*), 1812
interpolated() (*matplotlib.path.Path method*), 3199
intersection() (*matplotlib.transforms.BboxBase static method*), 3733
intersects_bbox() (*matplotlib.path.Path method*), 3199
intersects_path() (*matplotlib.path.Path method*), 3199
interval (*matplotlib.backend_bases.TimerBase property*), 2280
interval_contains() (*in module matplotlib.transforms*), 3753
interval_contains_open() (*in module matplotlib.transforms*), 3753
intervalx (*matplotlib.transforms.Bbox property*), 3728
intervalx (*matplotlib.transforms.BboxBase property*), 3733
intervaly (*matplotlib.transforms.Bbox property*), 3728
intervaly (*matplotlib.transforms.BboxBase property*), 3733
inv_transform() (*in module mpl_toolkits.mplot3d.proj3d*), 3908
inv_transform_xy() (*mpl_toolkits.axisartist.grid_finder.GridFinder method*), 4047
INVALID (*matplotlib.transforms.TransformNode attribute*), 3750
INVALID_AFFINE (*matplotlib.transforms.TransformNode attribute*), 3750
INVALID_NON_AFFINE (*matplotlib.transforms.TransformNode attribute*), 3750
invalidate() (*matplotlib.transforms.TransformNode method*), 3750
inverse() (*matplotlib.colors.AsinhNorm method*), 2719
inverse() (*matplotlib.colors.BoundaryNorm method*), 2720
inverse() (*matplotlib.colors.FuncNorm method*), 2724
inverse() (*matplotlib.colors.LogNorm method*), 2725
inverse() (*matplotlib.colors.NoNorm method*), 2718
inverse() (*matplotlib.colors.Normalize method*), 2715
inverse() (*matplotlib.colors.PowerNorm method*), 2728
inverse() (*matplotlib.colors.SymLogNorm method*), 2730
inverse() (*matplotlib.colors.TwoSlopeNorm method*), 2731
invert_axis_direction() (*mpl_toolkits.axisartist.axis_artist.TickLabels method*), 4020
invert_ticklabel_direction() (*mpl_toolkits.axisartist.axis_artist.AxisArtist method*), 4005
invert_xaxis() (*matplotlib.axes.Axes method*), 2119
invert_xaxis() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3847
invert_yaxis() (*matplotlib.axes.Axes method*), 2120
invert_yaxis() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3847
invert_zaxis() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3848
inverted() (*matplotlib.projections.geo.AitoffAxes.AitoffTransform method*), 3559
inverted() (*matplotlib.projections.geo.AitoffAxes.InvertedAitoffTransform method*), 3559
inverted() (*matplotlib.projections.geo.HammerAxes.HammerTransform method*), 3574
inverted() (*matplotlib.projections.geo.HammerAxes.InvertedHammerTransform method*), 3574
inverted() (*matplotlib.projections.geo.LambertAxes.InvertedLambertTransform method*), 3579
inverted() (*matplotlib.projections.geo.LambertAxes.LambertTransform method*), 3580
inverted() (*matplotlib.projections.geo.MollweideAxes.InvertedMollweideTransform method*), 3585
inverted() (*mat-*

*plotlib.projections.geo.MollweideAxes.MollweideTransform method*), 3585
inverted() (*matplotlib.projections.polar.InvertedPolarTransform method*), 3528
inverted() (*matplotlib.projections.polar.PolarAxes.InvertedPolarTransform method*), 3532
inverted() (*matplotlib.projections.polar.PolarAxes.PolarTransform method*), 3534
inverted() (*matplotlib.projections.polar.PolarTransform method*), 3547
inverted() (*matplotlib.scale.AsinhTransform method*), 3615
inverted() (*matplotlib.scale.FuncTransform method*), 3618
inverted() (*matplotlib.scale.InvertedAsinhTransform method*), 3619
inverted() (*matplotlib.scale.InvertedLogTransform method*), 3620
inverted() (*matplotlib.scale.InvertedSymmetricalLogTransform method*), 3621
inverted() (*matplotlib.scale.LogisticTransform method*), 3624
inverted() (*matplotlib.scale.LogitTransform method*), 3626
inverted() (*matplotlib.scale.LogTransform method*), 3623
inverted() (*matplotlib.scale.SymmetricalLogTransform method*), 3628
inverted() (*matplotlib.transforms.Affine2DBase method*), 3722
inverted() (*matplotlib.transforms.BlendedGenericTransform method*), 3738
inverted() (*matplotlib.transforms.CompositeGenericTransform method*), 3740
inverted() (*matplotlib.transforms.IdentityTransform method*), 3742
inverted() (*matplotlib.transforms.Transform method*), 3747
InvertedAsinhTransform (*class in matplotlib.scale*), 3618
InvertedLogTransform (*class in matplotlib.scale*), 3620
InvertedPolarTransform (*class in matplotlib.projections.polar*), 3527
InvertedSymmetricalLogTransform (*class in matplotlib.scale*), 3621
ioff() (*in module matplotlib.pyplot*), 3491
ion() (*in module matplotlib.pyplot*), 3492
ipython_in_subprocess() (*in module matplotlib.testing*), 3657
ipython_inline_display() (*in module matplotlib.backends.backend_webagg*), 2364
is_affine (*matplotlib.transforms.AffineBase attribute*), 3723
is_affine (*matplotlib.transforms.BboxBase attribute*), 3733
is_affine (*matplotlib.transforms.BlendedGenericTransform property*), 3738
is_affine (*matplotlib.transforms.CompositeGenericTransform property*), 3740
is_affine (*matplotlib.transforms.TransformNode attribute*), 3750
is_affine (*matplotlib.transforms.TransformWrapper property*), 3751
is_alias() (*matplotlib.artist.ArtistInspector static method*), 1901
is_available() (*matplotlib.animation.MovieWriterRegistry method*), 1858
is_bbox (*matplotlib.transforms.BboxBase attribute*), 3733
is_bbox (*matplotlib.transforms.TransformNode attribute*), 3750
is_color_like() (*in module matplotlib.colors*), 2752
is_dashed() (*matplotlib.lines.Line2D method*), 3021
is_filled() (*matplotlib.markers.MarkerStyle method*), 3041
is_first_col() (*matplotlib.gridspec.SubplotSpec method*), 2960
is_first_row() (*matplotlib.gridspec.SubplotSpec method*), 2960
is_gray() (*matplotlib.colors.Colormap method*), 2733
is_horizontal() (*matplotlib.collections.EventCollection method*), 2486
is_interactive() (*in module matplotlib*), 1812
is_last_col() (*matplotlib.gridspec.SubplotSpec method*), 2960
is_last_row() (*matplotlib.gridspec.SubplotSpec method*), 2961
is_math_text() (*in module matplotlib.cbook*), 2377
is_open() (*matplotlib.backends.backend_nbagg.CommSocket method*), 2321
is_opentype_cff_font() (*in module matplotlib.font_manager*), 2950
is_saving() (*matplotlib.backend_bases.FigureCanvasBase method*), 2257
is_scalar_or_string() (*in module matplotlib.cbook*), 2378
is_separable (*matplotlib.scale.AsinhTransform attribute*), 3616
is_separable (*matplotlib.scale.FuncTransform attribute*), 3618
is_separable (*matplotlib.scale.InvertedAsinhTransform attribute*), 3619
is_separable (*matplotlib.scale.InvertedLogTransform attribute*), 3620
is_separable (*matplotlib.scale.InvertedSymmetricalLogTransform attribute*), 3621
is_separable (*matplotlib.scale.LogisticTransform attribute*), 3624
is_separable (*matplotlib.scale.LogitTransform attribute*), 3626
is_separable (*matplotlib.scale.LogTransform attribute*), 3623

is_separable (*matplotlib.scale.SymmetricalLogTransform attribute*), 3628
is_separable (*matplotlib.transforms.Affine2DBase property*), 3722
is_separable (*matplotlib.transforms.BboxTransform attribute*), 3736
is_separable (*matplotlib.transforms.BboxTransformFrom attribute*), 3736
is_separable (*matplotlib.transforms.BboxTransformTo attribute*), 3736
is_separable (*matplotlib.transforms.BlendedAffine2D attribute*), 3737
is_separable (*matplotlib.transforms.BlendedGenericTransform attribute*), 3738
is_separable (*matplotlib.transforms.CompositeGenericTransform property*), 3740
is_separable (*matplotlib.transforms.Transform attribute*), 3747
is_separable (*matplotlib.transforms.TransformWrapper property*), 3751
is_transform_set() (*matplotlib.artist.Artist method*), 1893
is_transform_set() (*matplotlib.collections.AsteriskPolygonCollection method*), 2397
is_transform_set() (*matplotlib.collections.CircleCollection method*), 2418
is_transform_set() (*matplotlib.collections.Collection method*), 2441
is_transform_set() (*matplotlib.collections.EllipseCollection method*), 2463
is_transform_set() (*matplotlib.collections.EventCollection method*), 2486
is_transform_set() (*matplotlib.collections.LineCollection method*), 2509
is_transform_set() (*matplotlib.collections.PatchCollection method*), 2531
is_transform_set() (*matplotlib.collections.PathCollection method*), 2552
is_transform_set() (*matplotlib.collections.PolyCollection method*), 2575
is_transform_set() (*matplotlib.collections.PolyQuadMesh method*), 2598
is_transform_set() (*matplotlib.collections.QuadMesh method*), 2622
is_transform_set() (*matplotlib.collections.RegularPolyCollection method*), 2645
is_transform_set() (*matplotlib.collections.StarPolygonCollection method*), 2667
is_transform_set() (*matplotlib.collections.TriMesh method*), 2689
is_valid_backend() (*matplotlib.backends.registry.BackendRegistry method*), 2347
is_writable_file_like() (*in module matplotlib.cbook*), 2378
isAvailable() (*matplotlib.animation.HTMLWriter class method*), 1848
isAvailable() (*matplotlib.animation.ImageMagickBase class method*), 1867
isAvailable() (*matplotlib.animation.MovieWriter class method*), 1863
isAvailable() (*matplotlib.animation.PillowWriter class method*), 1845
isinteractive() (*in module matplotlib.pyplot*), 3493
isowner() (*matplotlib.widgets.LockDraw method*), 3782
iter_bezier() (*matplotlib.path.Path method*), 3199
iter_segments() (*matplotlib.path.Path method*), 3200

## J

join() (*matplotlib.cbook.Grouper method*), 2373
joined() (*matplotlib.cbook.Grouper method*), 2373
joined() (*matplotlib.cbook.GrouperView method*), 2373
JoinStyle (*class in matplotlib._enums*), 3810
joinstyle_cmd() (*matplotlib.backends.backend_pdf.GraphicsContextPdf method*), 2324
joinstyles (*matplotlib.backends.backend_pdf.GraphicsContextPdf attribute*), 2324
JoinStyleType (*in module matplotlib.typing*), 3768
json_dump() (*in module matplotlib.font_manager*), 2950
json_load() (*in module matplotlib.font_manager*), 2950
juggle_axes() (*in module mpl_toolkits.mplot3d.art3d*), 3905

## K

keep_empty (*matplotlib.backends.backend_pdf.PdfPages property*), 2329
keep_empty (*matplotlib.backends.backend_pgf.PdfPages property*), 2337
key_press_handler() (*in module matplotlib.backend_bases*), 2283
KeyEvent (*class in matplotlib.backend_bases*), 2267
kwarg_doc() (*in module matplotlib._docstring*), 2787
kwarg_error() (*in module matplotlib._api*), 3805
kwdoc() (*in module matplotlib.artist*), 1900

## L

label (*mpl_toolkits.axes_grid1.mpl_axes.SimpleAxisArtist property*), 3983
label_outer() (*matplotlib.axes.Axes method*), 2137
LabelBase (*class in mpl_toolkits.axisartist.axis_artist*), 4015
LABELPAD (*mpl_toolkits.axisartist.axis_artist.AxisArtist property*), 4004

labels() (*matplotlib.contour.ContourLabeler method*), 2758
labelsep (*matplotlib.quiver.QuiverKey property*), 3598
LambertAxes (*class in matplotlib.projections.geo*), 3577
LambertAxes.InvertedLambertTransform (*class in matplotlib.projections.geo*), 3579
LambertAxes.LambertTransform (*class in matplotlib.projections.geo*), 3580
LargeCircles (*class in matplotlib.hatch*), 2966
Lasso (*class in matplotlib.widgets*), 3780
LassoSelector (*class in matplotlib.widgets*), 3781
lastevent (*matplotlib.backend_bases.LocationEvent attribute*), 2269
LatexError, 2336
LatexManager (*class in matplotlib.backends.backend_pgf*), 2336
LayoutEngine (*class in matplotlib.layout_engine*), 2986
LEFT (*matplotlib.backend_bases.MouseButton attribute*), 2269
Legend (*class in matplotlib.legend*), 2990
legend entry, **163**
legend handle, **164**
legend key, **163**
legend label, **164**
legend() (*in module matplotlib.pyplot*), 3427
legend() (*matplotlib.axes.Axes method*), 2144
legend() (*matplotlib.figure.Figure method*), 2823
legend() (*matplotlib.figure.FigureBase method*), 2920
legend() (*matplotlib.figure.SubFigure method*), 2885
legend_artist() (*matplotlib.legend_handler.HandlerBase method*), 3004
legend_elements() (*matplotlib.collections.PathCollection method*), 2552
legend_elements() (*matplotlib.contour.ContourSet method*), 2762
len (*matplotlib.backends.backend_pdf.Stream attribute*), 2335
library (*in module matplotlib.style*), 3645
LightSource (*class in matplotlib.colors*), 2740
limit_range_for_scale() (*matplotlib.axis.Axis method*), 2246
limit_range_for_scale() (*matplotlib.scale.LogitScale method*), 3625
limit_range_for_scale() (*matplotlib.scale.LogScale method*), 3623
limit_range_for_scale() (*matplotlib.scale.ScaleBase method*), 3627
Line2D (*class in matplotlib.lines*), 3016
Line3D (*class in mpl_toolkits.mplot3d.art3d*), 3880
Line3DCollection (*class in mpl_toolkits.mplot3d.art3d*), 3883
line_2d_to_3d() (*in module mpl_toolkits.mplot3d.art3d*), 3905
line_collection_2d_to_3d() (*in module mpl_toolkits.mplot3d.art3d*), 3905
linear_spine() (*matplotlib.spines.Spine class method*), 3639
linear_width (*matplotlib.scale.AsinhScale property*), 3615
LinearLocator (*class in matplotlib.ticker*), 3697
LinearScale (*class in matplotlib.scale*), 3622
LinearSegmentedColormap (*class in matplotlib.colors*), 2734
LinearTriInterpolator (*class in matplotlib.tri*), 3759
LineCollection (*class in matplotlib.collections*), 2501
linestyles (*matplotlib.contour.ContourSet property*), 2763
lineStyles (*matplotlib.lines.Line2D attribute*), 3021
LineStyleType (*in module matplotlib.typing*), 3768
lineto (*matplotlib.backends.backend_pdf.Op attribute*), 2325
LINETO (*matplotlib.path.Path attribute*), 3195
linewidth_cmd() (*matplotlib.backends.backend_pdf.GraphicsContextPdf method*), 2324
linscale (*matplotlib.scale.SymmetricalLogScale property*), 3628
linthresh (*matplotlib.scale.SymmetricalLogScale property*), 3628
list() (*matplotlib.animation.MovieWriterRegistry method*), 1858
list_all() (*matplotlib.backends.registry.BackendRegistry method*), 2348
list_builtin() (*matplotlib.backends.registry.BackendRegistry method*), 2348
list_fonts() (*in module matplotlib.font_manager*), 2950
list_gui_frameworks() (*matplotlib.backends.registry.BackendRegistry method*), 2348
ListedColormap (*class in matplotlib.colors*), 2737
load_backend_module() (*matplotlib.backends.registry.BackendRegistry method*), 2348
load_char() (*matplotlib.ft2font.FT2Font method*), 2954
load_glyph() (*matplotlib.ft2font.FT2Font method*), 2955
locally_modified_subplot_params() (*matplotlib.gridspec.GridSpec method*), 2958
locate() (*mpl_toolkits.axes_grid1.axes_divider.Divider method*), 3934
locate_label() (*matplotlib.contour.ContourLabeler method*), 2758
LocationEvent (*class in matplotlib.backend_bases*), 2268
Locator (*class in matplotlib.ticker*), 3698
locator (*matplotlib.colorbar.Colorbar property*), 2707
locator_params() (*in module matplotlib.pyplot*), 3453
locator_params() (*matplotlib.axes.Axes method*), 2183
LocatorBase (*class in mpl_toolkits.axisartist.angle_helper*), 3994
LocatorD (*class in mpl_toolkits.axisartist.angle_helper*), 3994
LocatorDM (*class in mpl_toolkits.axisartist.angle_helper*), 3995
LocatorDMS (*class in mpl_toolkits.axisartist.angle_helper*), 3995
LocatorH (*class in mpl_toolkits.axisartist.angle_helper*), 3995
LocatorHM (*class in mpl_toolkits.axisartist.angle_helper*), 3995
LocatorHMS (*class in mpl_toolkits.axisartist.angle_helper*), 3996
LockableBbox (*class in matplotlib.transforms*), 3744

LockDraw (*class in matplotlib.widgets*), 3782
locked() (*matplotlib.widgets.LockDraw method*), 3782
locked_x0 (*matplotlib.transforms.LockableBbox property*), 3744
locked_x1 (*matplotlib.transforms.LockableBbox property*), 3744
locked_y0 (*matplotlib.transforms.LockableBbox property*), 3744
locked_y1 (*matplotlib.transforms.LockableBbox property*), 3744
locs (*matplotlib.ticker.Formatter attribute*), 3696
LogFormatter (*class in matplotlib.ticker*), 3699
LogFormatterExponent (*class in matplotlib.ticker*), 3700
LogFormatterMathtext (*class in matplotlib.ticker*), 3701
LogFormatterSciNotation (*class in matplotlib.ticker*), 3701
LogisticTransform (*class in matplotlib.scale*), 3624
LogitFormatter (*class in matplotlib.ticker*), 3702
LogitLocator (*class in matplotlib.ticker*), 3704
LogitScale (*class in matplotlib.scale*), 3624
LogitTransform (*class in matplotlib.scale*), 3625
LogLocator (*class in matplotlib.ticker*), 3701
loglog() (*in module matplotlib.pyplot*), 3269
loglog() (*matplotlib.axes.Axes method*), 1931
LogNorm (*class in matplotlib.colors*), 2724
LogScale (*class in matplotlib.scale*), 3622
LogTransform (*class in matplotlib.scale*), 3623
ls_mapper (*in module matplotlib.cbook*), 2378
ls_mapper_r (*in module matplotlib.cbook*), 2378

## M

magnitude_spectrum() (*in module matplotlib.mlab*), 3056
magnitude_spectrum() (*in module matplotlib.pyplot*), 3325
magnitude_spectrum() (*matplotlib.axes.Axes method*), 1986
major_ticklabels (*mpl_toolkits.axes_grid1.mpl_axes.SimpleAxisArtist property*), 3983
major_ticks (*mpl_toolkits.axes_grid1.mpl_axes.SimpleAxisArtist property*), 3983
make_axes() (*in module matplotlib.colorbar*), 2710
make_axes_area_auto_adjustable() (*in module mpl_toolkits.axes_grid1.axes_divider*), 3939
make_axes_gridspec() (*in module matplotlib.colorbar*), 2711
make_axes_locatable() (*in module mpl_toolkits.axes_grid1.axes_divider*), 3940
make_compound_path() (*matplotlib.path.Path class method*), 3201
make_compound_path_from_polys() (*matplotlib.path.Path class method*), 3201
make_dvi() (*matplotlib.texmanager.TexManager class method*), 3689
make_image() (*matplotlib.image.AxesImage method*), 2969
make_image() (*matplotlib.image.BboxImage method*), 2971
make_image() (*matplotlib.image.FigureImage method*), 2973
make_image() (*matplotlib.image.NonUniformImage method*), 2975
make_image() (*matplotlib.image.PcolorImage method*), 2979
make_keyword_only() (*in module matplotlib._api.deprecation*), 3808
make_norm_from_scale() (*in module matplotlib.colors*), 2752
make_pdf_to_png_converter() (*in module matplotlib.backends.backend_pgf*), 2341
make_png() (*matplotlib.texmanager.TexManager class method*), 3689
make_rgb_axes() (*in module mpl_toolkits.axes_grid1.axes_rgb*), 3948
make_tex() (*matplotlib.texmanager.TexManager class method*), 3689
make_wedged_bezier2() (*in module matplotlib.bezier*), 2367
manager_class (*matplotlib.backend_bases.FigureCanvasBase attribute*), 2257
manager_class (*matplotlib.backends.backend_nbagg.FigureCanvasNbAgg attribute*), 2321
manager_class (*matplotlib.backends.backend_template.FigureCanvasTemplate attribute*), 2305
manager_class (*matplotlib.backends.backend_webagg_core.FigureCanvasWebAggCore attribute*), 2359
manager_class (*matplotlib.backends.backend_webagg.FigureCanvasWebAgg attribute*), 2362
margins() (*in module matplotlib.pyplot*), 3473
margins() (*matplotlib.axes.Axes method*), 2156
margins() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3855
mark_inset() (*in module mpl_toolkits.axes_grid1.inset_locator*), 3973
mark_plot_labels() (*in module matplotlib.sphinxext.plot_directive*), 3635
markerObject() (*matplotlib.backends.backend_pdf.PdfFile method*), 2327
markers (*matplotlib.lines.Line2D attribute*), 3021
markers (*matplotlib.markers.MarkerStyle attribute*), 3041
MarkerStyle (*class in matplotlib.markers*), 3040
MarkEveryType (*in module matplotlib.typing*), 3768
math_to_image() (*in module matplotlib.mathtext*), 3045
MathDirective (*class in matplotlib.sphinxext.mathmpl*), 3631
MathTextParser (*class in matplotlib.mathtext*), 3043
matplotlib
    module, 1
matplotlib._afm
    module, 1829
matplotlib._api

module, 3802
matplotlib._api.deprecation
module, 3806
matplotlib._docstring
module, 2787
matplotlib._enums
module, 3810
matplotlib_fname() (*in module matplotlib*), 1826
matplotlib._tight_bbox
module, 3716
matplotlib._tight_layout
module, 3716
matplotlib._type1font
module, 3766
matplotlib.animation
module, 1832
matplotlib.artist
module, 1867
matplotlib.axes
module, 1903
matplotlib.axis
module, 2218
matplotlib.backend_bases
module, 2253
matplotlib.backend_managers
module, 2284
matplotlib.backend_tools
module, 2288
matplotlib.backends
module, 2303
matplotlib.backends.backend_agg
module, 2308
matplotlib.backends.backend_cairo
module, 2315
matplotlib.backends.backend_gtk3
module, 2320
matplotlib.backends.backend_gtk3agg
module, 2320
matplotlib.backends.backend_gtk3cairo
module, 2320
matplotlib.backends.backend_gtk4
module, 2320
matplotlib.backends.backend_gtk4agg
module, 2320
matplotlib.backends.backend_gtk4cairo
module, 2320
matplotlib.backends.backend_mixed
module, 2303
matplotlib.backends.backend_nbagg
module, 2321
matplotlib.backends.backend_pdf
module, 2322
matplotlib.backends.backend_pgf
module, 2335
matplotlib.backends.backend_ps
module, 2341
matplotlib.backends.backend_qt
module, 2350
matplotlib.backends.backend_qt5agg
module, 2350
matplotlib.backends.backend_qt5cairo
module, 2350
matplotlib.backends.backend_qtagg
module, 2350
matplotlib.backends.backend_qtcairo
module, 2350
matplotlib.backends.backend_svg
module, 2351
matplotlib.backends.backend_template
module, 2304
matplotlib.backends.backend_tkagg
module, 2357
matplotlib.backends.backend_tkcairo
module, 2357
matplotlib.backends.backend_webagg
module, 2362
matplotlib.backends.backend_webagg_core
module, 2358
matplotlib.backends.backend_wx
module, 2364
matplotlib.backends.backend_wxagg
module, 2364
matplotlib.backends.backend_wxcairo
module, 2364
matplotlib.backends.qt_compat
module, 2350
matplotlib.backends.registry
module, 2346
matplotlib.bezier
module, 2364
matplotlib.category
module, 2368
matplotlib.cbook
module, 2371
matplotlib.cm
module, 2384
matplotlib.collections
module, 2388
matplotlib.colorbar
module, 2704
matplotlib.colors
module, 2712
matplotlib.container
module, 2753
matplotlib.contour
module, 2756
matplotlib.dates
module, 2768
MatplotlibDeprecationWarning, 3806
MatplotlibDeprecationWarning (*class in matplotlib*), 1829
matplotlib.dviread
module, 2788
matplotlib.figure
module, 2792
matplotlib.font_manager
module, 2941
matplotlib.ft2font

module, 2951
matplotlib.gridspec
module, 2956
matplotlib.hatch
module, 2966
matplotlib.image
module, 2967
matplotlib.layout_engine
module, 2985
matplotlib.legend
module, 2989
matplotlib.legend_handler
module, 3003
matplotlib.lines
module, 3015
matplotlib.markers
module, 3038
matplotlib.mathtext
module, 3043
matplotlib.mlab
module, 3046
matplotlib.offsetbox
module, 3064
matplotlib.patches
module, 3095
matplotlib.path
module, 3194
matplotlib.patheffects
module, 3203
matplotlib.projections
module, 3526
matplotlib.projections.geo
module, 3556
matplotlib.projections.polar
module, 3527
matplotlib.pyplot
module, 3211
matplotlib.quiver
module, 3588
matplotlib.rcsetup
module, 3605
matplotlib.sankey
module, 3609
matplotlib.scale
module, 3614
matplotlib.sphinxext.figmpl_directive
module, 3635
matplotlib.sphinxext.mathmpl
module, 3630
matplotlib.sphinxext.plot_directive
module, 3631
matplotlib.spines
module, 3637
matplotlib.style
module, 3643
matplotlib.table
module, 3645
matplotlib.testing
module, 3657
matplotlib.testing.compare
module, 3659
matplotlib.testing.decorators
module, 3660
matplotlib.testing.exceptions
module, 3662
matplotlib.texmanager
module, 3688
matplotlib.text
module, 3662
matplotlib.ticker
module, 3689
matplotlib.transforms
module, 3718
matplotlib.tri
module, 3755
matplotlib.units
module, 3769
matplotlib.widgets
module, 3772
matshow() (*in module matplotlib.pyplot*), 3381
matshow() (*matplotlib.axes.Axes method*), 2047
max (*matplotlib.transforms.BboxBase property*), 3733
max_advance_height (*matplotlib.ft2font.FT2Font attribute*), 2955
max_advance_width (*matplotlib.ft2font.FT2Font attribute*), 2955
MaxExtent (*class in mpl_toolkits.axes_grid1.axes_size*), 3951
MaxHeight (*class in mpl_toolkits.axes_grid1.axes_size*), 3951
MaxNLocator (*class in matplotlib.ticker*), 3705
MaxNLocator (*class in mpl_toolkits.axisartist.grid_finder*), 4048
MAXTICKS (*matplotlib.ticker.Locator attribute*), 3698
MaxWidth (*class in mpl_toolkits.axes_grid1.axes_size*), 3951
MESONPY_EDITABLE_VERBOSE, 4650
message_event() (*matplotlib.backend_managers.ToolManager method*), 2286
MicrosecondLocator (*class in matplotlib.dates*), 2778
MIDDLE (*matplotlib.backend_bases.MouseButton attribute*), 2269
min (*matplotlib.transforms.BboxBase property*), 3733
min_mark (*mpl_toolkits.axisartist.angle_helper.FormatterDMS attribute*), 3993
min_mark (*mpl_toolkits.axisartist.angle_helper.FormatterHMS attribute*), 3994
minor (*matplotlib.ticker.LogitLocator property*), 3704
minorformatter (*matplotlib.colorbar.Colorbar property*), 2707
minorlocator (*matplotlib.colorbar.Colorbar property*), 2707
minorticks_off() (*in module matplotlib.pyplot*), 3454
minorticks_off() (*matplotlib.axes.Axes method*), 2179
minorticks_off() (*matplotlib.axis.Axis method*), 2236

minorticks_off() (*matplotlib.colorbar.Colorbar method*), 2707
minorticks_on() (*in module matplotlib.pyplot*), 3455
minorticks_on() (*matplotlib.axes.Axes method*), 2179
minorticks_on() (*matplotlib.axis.Axis method*), 2236
minorticks_on() (*matplotlib.colorbar.Colorbar method*), 2708
minpos (*matplotlib.transforms.Bbox property*), 3728
minposx (*matplotlib.transforms.Bbox property*), 3728
minposy (*matplotlib.transforms.Bbox property*), 3728
MinuteLocator (*class in matplotlib.dates*), 2779
MixedModeRenderer (*class in matplotlib.backends.backend_mixed*), 2303
module
matplotlib, 1
matplotlib._afm, 1829
matplotlib._api, 3802
matplotlib._api.deprecation, 3806
matplotlib._docstring, 2787
matplotlib._enums, 3810
matplotlib._tight_bbox, 3716
matplotlib._tight_layout, 3716
matplotlib._type1font, 3766
matplotlib.animation, 1832
matplotlib.artist, 1867
matplotlib.axes, 1903
matplotlib.axis, 2218
matplotlib.backend_bases, 2253
matplotlib.backend_managers, 2284
matplotlib.backend_tools, 2288
matplotlib.backends, 2303
matplotlib.backends.backend_agg, 2308
matplotlib.backends.backend_cairo, 2315
matplotlib.backends.backend_gtk3, 2320
matplotlib.backends.backend_gtk3agg, 2320
matplotlib.backends.backend_gtk3cairo, 2320
matplotlib.backends.backend_gtk4, 2320
matplotlib.backends.backend_gtk4agg, 2320
matplotlib.backends.backend_gtk4cairo, 2320
matplotlib.backends.backend_mixed, 2303
matplotlib.backends.backend_nbagg, 2321
matplotlib.backends.backend_pdf, 2322
matplotlib.backends.backend_pgf, 2335
matplotlib.backends.backend_ps, 2341
matplotlib.backends.backend_qt, 2350
matplotlib.backends.backend_qt5agg, 2350
matplotlib.backends.backend_qt5cairo, 2350
matplotlib.backends.backend_qtagg, 2350
matplotlib.backends.backend_qtcairo, 2350
matplotlib.backends.backend_svg, 2351
matplotlib.backends.backend_template, 2304
matplotlib.backends.backend_tkagg, 2357
matplotlib.backends.backend_tkcairo, 2357
matplotlib.backends.backend_webagg, 2362
matplotlib.backends.backend_webagg_core, 2358
matplotlib.backends.backend_wx, 2364
matplotlib.backends.backend_wxagg, 2364
matplotlib.backends.backend_wxcairo, 2364
matplotlib.backends.qt_compat, 2350
matplotlib.backends.registry, 2346
matplotlib.bezier, 2364
matplotlib.category, 2368
matplotlib.cbook, 2371
matplotlib.cm, 2384
matplotlib.collections, 2388
matplotlib.colorbar, 2704
matplotlib.colors, 2712
matplotlib.container, 2753
matplotlib.contour, 2756
matplotlib.dates, 2768
matplotlib.dviread, 2788
matplotlib.figure, 2792
matplotlib.font_manager, 2941
matplotlib.ft2font, 2951
matplotlib.gridspec, 2956
matplotlib.hatch, 2966
matplotlib.image, 2967
matplotlib.layout_engine, 2985
matplotlib.legend, 2989
matplotlib.legend_handler, 3003
matplotlib.lines, 3015
matplotlib.markers, 3038
matplotlib.mathtext, 3043
matplotlib.mlab, 3046
matplotlib.offsetbox, 3064
matplotlib.patches, 3095
matplotlib.path, 3194
matplotlib.patheffects, 3203
matplotlib.projections, 3526
matplotlib.projections.geo, 3556
matplotlib.projections.polar, 3527
matplotlib.pyplot, 3211
matplotlib.quiver, 3588
matplotlib.rcsetup, 3605
matplotlib.sankey, 3609
matplotlib.scale, 3614
matplotlib.sphinxext.figmpl_directive, 3635
matplotlib.sphinxext.mathmpl, 3630
matplotlib.sphinxext.plot_directive, 3631
matplotlib.spines, 3637
matplotlib.style, 3643
matplotlib.table, 3645
matplotlib.testing, 3657
matplotlib.testing.compare, 3659
matplotlib.testing.decorators, 3660

matplotlib.testing.exceptions, 3662
matplotlib.texmanager, 3688
matplotlib.text, 3662
matplotlib.ticker, 3689
matplotlib.transforms, 3718
matplotlib.tri, 3755
matplotlib.units, 3769
matplotlib.widgets, 3772
mpl_toolkits.axes_grid1, 3910
mpl_toolkits.axes_grid1.anchored_artists, 3911
mpl_toolkits.axes_grid1.axes_divider, 3930
mpl_toolkits.axes_grid1.axes_grid, 3940
mpl_toolkits.axes_grid1.axes_rgb, 3945
mpl_toolkits.axes_grid1.axes_size, 3948
mpl_toolkits.axes_grid1.inset_locator, 3953
mpl_toolkits.axes_grid1.mpl_axes, 3977
mpl_toolkits.axes_grid1.parasite_axes, 3985
mpl_toolkits.axisartist, 3989
mpl_toolkits.axisartist.angle_helper, 3990
mpl_toolkits.axisartist.axes_divider, 3997
mpl_toolkits.axisartist.axes_grid, 3997
mpl_toolkits.axisartist.axes_rgb, 4001
mpl_toolkits.axisartist.axis_artist, 4002
mpl_toolkits.axisartist.axisline_style, 4025
mpl_toolkits.axisartist.axislines, 4027
mpl_toolkits.axisartist.floating_axes, 4041
mpl_toolkits.axisartist.grid_finder, 4045
mpl_toolkits.axisartist.grid_helper_curvelinear, 4049
mpl_toolkits.axisartist.parasite_axes, 4051
mpl_toolkits.mplot3d, 3873
mpl_toolkits.mplot3d.art3d, 3879
mpl_toolkits.mplot3d.axes3d, 3873
mpl_toolkits.mplot3d.axis3d, 3874
mpl_toolkits.mplot3d.proj3d, 3907
pylab, 4051
MollweideAxes (*class in matplotlib.projections.geo*), 3582
MollweideAxes.InvertedMollweideTransform (*class in matplotlib.projections.geo*), 3585
MollweideAxes.MollweideTransform (*class in matplotlib.projections.geo*), 3585
MonthLocator (*class in matplotlib.dates*), 2779
mouse_init() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3865
mouse_move() (*matplotlib.backend_bases.NavigationToolbar2 method*), 2271
MouseButton (*class in matplotlib.backend_bases*), 2269
MouseEvent (*class in matplotlib.backend_bases*), 2269
mouseover (*matplotlib.artist.Artist property*), 1872
mouseover (*matplotlib.axes.Axes property*), 2202
mouseover (*matplotlib.collections.AsteriskPolygonCollection property*), 2397
mouseover (*matplotlib.collections.CircleCollection property*), 2418
mouseover (*matplotlib.collections.Collection property*), 2441
mouseover (*matplotlib.collections.EllipseCollection property*), 2463
mouseover (*matplotlib.collections.EventCollection property*), 2486
mouseover (*matplotlib.collections.LineCollection property*), 2509
mouseover (*matplotlib.collections.PatchCollection property*), 2531
mouseover (*matplotlib.collections.PathCollection property*), 2553
mouseover (*matplotlib.collections.PolyCollection property*), 2575
mouseover (*matplotlib.collections.PolyQuadMesh property*), 2598
mouseover (*matplotlib.collections.QuadMesh property*), 2622
mouseover (*matplotlib.collections.RegularPolyCollection property*), 2645
mouseover (*matplotlib.collections.StarPolygonCollection property*), 2667
mouseover (*matplotlib.collections.TriMesh property*), 2689
MOVE (*matplotlib.backend_tools.Cursors attribute*), 2289
moveto (*matplotlib.backends.backend_pdf.Op attribute*), 2325
MOVETO (*matplotlib.path.Path attribute*), 3195
MovieWriter (*class in matplotlib.animation*), 1860
MovieWriterRegistry (*class in matplotlib.animation*), 1857
mpl_connect() (*matplotlib.backend_bases.FigureCanvasBase method*), 2257
mpl_disconnect() (*matplotlib.backend_bases.FigureCanvasBase method*), 2258
mpl_toolkits.axes_grid1 module, 3910
mpl_toolkits.axes_grid1.anchored_artists module, 3911
mpl_toolkits.axes_grid1.axes_divider module, 3930
mpl_toolkits.axes_grid1.axes_grid module, 3940
mpl_toolkits.axes_grid1.axes_rgb module, 3945
mpl_toolkits.axes_grid1.axes_size module, 3948
mpl_toolkits.axes_grid1.inset_locator module, 3953
mpl_toolkits.axes_grid1.mpl_axes module, 3977
mpl_toolkits.axes_grid1.parasite_axes module, 3985

mpl_toolkits.axisartist
module, 3989
mpl_toolkits.axisartist.angle_helper
module, 3990
mpl_toolkits.axisartist.axes_divider
module, 3997
mpl_toolkits.axisartist.axes_grid
module, 3997
mpl_toolkits.axisartist.axes_rgb
module, 4001
mpl_toolkits.axisartist.axis_artist
module, 4002
mpl_toolkits.axisartist.axisline_style
module, 4025
mpl_toolkits.axisartist.axislines
module, 4027
mpl_toolkits.axisartist.floating_axes
module, 4041
mpl_toolkits.axisartist.grid_finder
module, 4045
mpl_toolkits.axisartist.grid_helper_curvelinear
module, 4049
mpl_toolkits.axisartist.parasite_axes
module, 4051
mpl_toolkits.mplot3d
module, 3873
mpl_toolkits.mplot3d.art3d
module, 3879
mpl_toolkits.mplot3d.axes3d
module, 3873
mpl_toolkits.mplot3d.axis3d
module, 3874
mpl_toolkits.mplot3d.proj3d
module, 3907
MPLBACKEND, 59, 60, 1811, 4415, 4477
MPLCONFIGDIR, 9, 21, 292, 1811
MPLSETUPCFG, 4209
MultiCursor (*class in matplotlib.widgets*), 3782
MultipleLocator (*class in matplotlib.ticker*), 3706
mutated() (*matplotlib.transforms.Bbox method*), 3728
mutatedx() (*matplotlib.transforms.Bbox method*), 3728
mutatedy() (*matplotlib.transforms.Bbox method*), 3729

## N

n_rasterize (*matplotlib.colorbar.Colorbar attribute*), 2708
Name (*class in matplotlib.backends.backend_pdf*), 2324
name (*matplotlib._afm.CharMetrics attribute*), 1831
name (*matplotlib._afm.CompositePart attribute*), 1832
name (*matplotlib.axes.Axes attribute*), 2209
name (*matplotlib.backend_tools.ToolBase property*), 2291
name (*matplotlib.backends.backend_pdf.Name attribute*), 2324
name (*matplotlib.projections.geo.AitoffAxes attribute*), 3560
name (*matplotlib.projections.geo.HammerAxes attribute*), 3575
name (*matplotlib.projections.geo.LambertAxes attribute*), 3581
name (*matplotlib.projections.geo.MollweideAxes attribute*), 3586
name (*matplotlib.projections.polar.PolarAxes attribute*), 3540
name (*matplotlib.scale.AsinhScale attribute*), 3615
name (*matplotlib.scale.FuncScale attribute*), 3617
name (*matplotlib.scale.FuncScaleLog attribute*), 3617
name (*matplotlib.scale.LinearScale attribute*), 3622
name (*matplotlib.scale.LogitScale attribute*), 3625
name (*matplotlib.scale.LogScale attribute*), 3623
name (*matplotlib.scale.SymmetricalLogScale attribute*), 3628
nargs_error() (*in module matplotlib._api*), 3805
NavigationIPy (*class in matplotlib.backends.backend_nbagg*), 2322
NavigationToolbar2 (*class in matplotlib.backend_bases*), 2270
NavigationToolbar2WebAgg (*class in matplotlib.backends.backend_webagg_core*), 2360
ncols (*matplotlib.gridspec.GridSpecBase property*), 2963
neighbors (*matplotlib.tri.Triangulation property*), 3756
new_axes() (*matplotlib.widgets.SpanSelector method*), 3798
new_figure_manager() (*in module matplotlib.pyplot*), 3522
new_fixed_axis() (*mpl_toolkits.axisartist.axislines.Axes method*), 4032
new_fixed_axis() (*mpl_toolkits.axisartist.axislines.GridHelperRectlinear method*), 4040
new_fixed_axis() (*mpl_toolkits.axisartist.floating_axes.GridHelperCurveLinear method*), 4044
new_fixed_axis() (*mpl_toolkits.axisartist.grid_helper_curvelinear.GridHelperCurveLinear method*), 4051
new_floating_axis() (*mpl_toolkits.axisartist.axislines.Axes method*), 4032
new_floating_axis() (*mpl_toolkits.axisartist.axislines.GridHelperRectlinear method*), 4040
new_floating_axis() (*mpl_toolkits.axisartist.grid_helper_curvelinear.GridHelperCurveLinear method*), 4051
new_frame_seq() (*matplotlib.animation.Animation method*), 1834
new_frame_seq() (*matplotlib.animation.FuncAnimation method*), 1839
new_gc() (*matplotlib.backend_bases.RendererBase method*), 2277
new_gc() (*matplotlib.backends.backend_cairo.RendererCairo method*), 2320
new_gc() (*matplotlib.backends.backend_pdf.RendererPdf method*), 2334
new_gc() (*matplotlib.backends.backend_template.RendererTemplate method*), 2308
new_line() (*mpl_toolkits.axisartist.axisline_style.AxislineStyle.FilledArrow method*), 4026
new_line() (*mpl_toolkits.axisartist.axisline_style.AxislineStyle.SimpleArrow method*), 4026

new_locator() (*mpl_toolkits.axes_grid1.axes_divider.Divider method*), 3934
new_locator() (*mpl_toolkits.axes_grid1.axes_divider.HBoxDivider method*), 3936
new_locator() (*mpl_toolkits.axes_grid1.axes_divider.VBoxDivider method*), 3938
new_manager() (*matplotlib.backend_bases.FigureCanvasBase class method*), 2258
new_saved_frame_seq() (*matplotlib.animation.Animation method*), 1834
new_saved_frame_seq() (*matplotlib.animation.FuncAnimation method*), 1839
new_subplotspec() (*matplotlib.gridspec.GridSpecBase method*), 2963
new_timer() (*matplotlib.backend_bases.FigureCanvasBase method*), 2259
newPage() (*matplotlib.backends.backend_pdf.PdfFile method*), 2327
newTextnote() (*matplotlib.backends.backend_pdf.PdfFile method*), 2327
NON_INTERACTIVE (*matplotlib.backends.registry.BackendFilter attribute*), 2346
NonGuiException, 2272
NonIntersectingPathException, 2365
NoNorm (*class in matplotlib.colors*), 2717
nonsingular() (*in module matplotlib.transforms*), 3754
nonsingular() (*matplotlib.dates.AutoDateLocator method*), 2774
nonsingular() (*matplotlib.dates.DateLocator method*), 2777
nonsingular() (*matplotlib.projections.polar.PolarAxes.RadialLocator method*), 3534
nonsingular() (*matplotlib.projections.polar.RadialLocator method*), 3550
nonsingular() (*matplotlib.ticker.Locator method*), 3698
nonsingular() (*matplotlib.ticker.LogitLocator method*), 3704
nonsingular() (*matplotlib.ticker.LogLocator method*), 3701
NonUniformImage (*class in matplotlib.image*), 2975
norm (*matplotlib.cm.ScalarMappable property*), 2387
norm (*matplotlib.collections.AsteriskPolygonCollection property*), 2397
norm (*matplotlib.collections.CircleCollection property*), 2418
norm (*matplotlib.collections.Collection property*), 2441
norm (*matplotlib.collections.EllipseCollection property*), 2463
norm (*matplotlib.collections.EventCollection property*), 2486
norm (*matplotlib.collections.LineCollection property*), 2509
norm (*matplotlib.collections.PatchCollection property*), 2531
norm (*matplotlib.collections.PathCollection property*), 2554
norm (*matplotlib.collections.PolyCollection property*), 2575
norm (*matplotlib.collections.PolyQuadMesh property*), 2598
norm (*matplotlib.collections.QuadMesh property*), 2622
norm (*matplotlib.collections.RegularPolyCollection property*), 2645
norm (*matplotlib.collections.StarPolygonCollection property*), 2667
norm (*matplotlib.collections.TriMesh property*), 2690
Normal (*class in matplotlib.patheffects*), 3204
Normalize (*class in matplotlib.colors*), 2714
normalize_kwargs() (*in module matplotlib.cbook*), 2378
normalized() (*matplotlib.dates.relativedelta method*), 2785
NorthEastHatch (*class in matplotlib.hatch*), 2966
nrows (*matplotlib.gridspec.GridSpecBase property*), 2963
null() (*matplotlib.transforms.Bbox static method*), 3729
NullFormatter (*class in matplotlib.ticker*), 3707
NullLocator (*class in matplotlib.ticker*), 3707
num2 (*matplotlib.gridspec.SubplotSpec property*), 2961
num2date() (*in module matplotlib.dates*), 2783
num2timedelta() (*in module matplotlib.dates*), 2783
num_charmaps (*matplotlib.ft2font.FT2Font attribute*), 2955
num_faces (*matplotlib.ft2font.FT2Font attribute*), 2955
num_fixed_sizes (*matplotlib.ft2font.FT2Font attribute*), 2955
num_glyphs (*matplotlib.ft2font.FT2Font attribute*), 2955
NUM_VERTICES_FOR_CODE (*matplotlib.path.Path attribute*), 3195
number_of_parameters() (*matplotlib.artist.ArtistInspector static method*), 1901
numdecs (*matplotlib.ticker.LogLocator property*), 3702
numticks (*matplotlib.ticker.LinearLocator property*), 3697

## O

offset_copy() (*in module matplotlib.transforms*), 3754
OffsetBox (*class in matplotlib.offsetbox*), 3080
OffsetFrom (*class in matplotlib.text*), 3685
OffsetImage (*class in matplotlib.offsetbox*), 3084
OFFSETTEXTPAD (*matplotlib.axis.Axis attribute*), 2246
on_changed() (*matplotlib.widgets.RangeSlider method*), 3789
on_changed() (*matplotlib.widgets.Slider method*), 3795
on_clicked() (*matplotlib.widgets.Button method*), 3774
on_clicked() (*matplotlib.widgets.CheckButtons method*), 3776
on_clicked() (*matplotlib.widgets.RadioButtons method*), 3786
on_close() (*matplotlib.backends.backend_nbagg.CommSocket method*), 2321
on_close() (*matplotlib.backends.backend_webagg.WebAggApplication.WebSocket method*), 2363
on_message() (*matplotlib.backends.backend_nbagg.CommSocket method*), 2321
on_message() (*matplotlib.backends.backend_webagg.WebAggApplication.WebSocket method*), 2363

on_motion() (*matplotlib.offsetbox.DraggableBase method*), 3076
on_pick() (*matplotlib.offsetbox.DraggableBase method*), 3076
on_release() (*matplotlib.offsetbox.DraggableBase method*), 3076
on_submit() (*matplotlib.widgets.TextBox method*), 3799
on_text_change() (*matplotlib.widgets.TextBox method*), 3799
onmove() (*matplotlib.widgets.Cursor method*), 3778
onmove() (*matplotlib.widgets.Lasso method*), 3781
onmove() (*matplotlib.widgets.MultiCursor method*), 3783
onmove() (*matplotlib.widgets.PolygonSelector method*), 3785
onpick() (*matplotlib.lines.VertexSelector method*), 3034
onrelease() (*matplotlib.widgets.Lasso method*), 3781
Op (*class in matplotlib.backends.backend_pdf*), 2324
open() (*matplotlib.backends.backend_webagg.WebAggApplication.WebSocket method*), 2363
open_file_cm() (*in module matplotlib.cbook*), 2378
open_group() (*matplotlib.backend_bases.RendererBase method*), 2278
open_group() (*matplotlib.backends.backend_svg.RendererSVG method*), 2355
open_group() (*matplotlib.patheffects.PathEffectRenderer method*), 3205
option_image_nocomposite() (*matplotlib.backend_bases.RendererBase method*), 2278
option_image_nocomposite() (*matplotlib.backends.backend_agg.RendererAgg method*), 2314
option_image_nocomposite() (*matplotlib.backends.backend_pgf.RendererPgf method*), 2340
option_image_nocomposite() (*matplotlib.backends.backend_svg.RendererSVG method*), 2355
option_scale_image() (*matplotlib.backend_bases.RendererBase method*), 2278
option_scale_image() (*matplotlib.backends.backend_agg.RendererAgg method*), 2314
option_scale_image() (*matplotlib.backends.backend_pgf.RendererPgf method*), 2341
option_scale_image() (*matplotlib.backends.backend_svg.RendererSVG method*), 2355
option_spec (*matplotlib.sphinxext.figmpl_directive.FigureMpl attribute*), 3636
option_spec (*matplotlib.sphinxext.mathmpl.MathDirective attribute*), 3631
option_spec (*matplotlib.sphinxext.plot_directive.PlotDirective attribute*), 3634
optional_arguments (*matplotlib.sphinxext.figmpl_directive.FigureMpl attribute*), 3637
optional_arguments (*matplotlib.sphinxext.mathmpl.MathDirective attribute*), 3631
optional_arguments (*matplotlib.sphinxext.plot_directive.PlotDirective attribute*), 3635
out_of_date() (*in module matplotlib.sphinxext.plot_directive*), 3635
output() (*matplotlib.backends.backend_pdf.PdfFile method*), 2327
output_args (*matplotlib.animation.FFMpegBase property*), 1867
output_dims (*matplotlib.projections.polar.InvertedPolarTransform attribute*), 3528
output_dims (*matplotlib.projections.polar.PolarAxes.InvertedPolarTransform attribute*), 3532
output_dims (*matplotlib.projections.polar.PolarAxes.PolarTransform attribute*), 3534
output_dims (*matplotlib.projections.polar.PolarTransform attribute*), 3547
output_dims (*matplotlib.scale.AsinhTransform attribute*), 3616
output_dims (*matplotlib.scale.FuncTransform attribute*), 3618
output_dims (*matplotlib.scale.InvertedAsinhTransform attribute*), 3619
output_dims (*matplotlib.scale.InvertedLogTransform attribute*), 3620
output_dims (*matplotlib.scale.InvertedSymmetricalLogTransform attribute*), 3621
output_dims (*matplotlib.scale.LogisticTransform attribute*), 3624
output_dims (*matplotlib.scale.LogitTransform attribute*), 3626
output_dims (*matplotlib.scale.LogTransform attribute*), 3623
output_dims (*matplotlib.scale.SymmetricalLogTransform attribute*), 3628
output_dims (*matplotlib.transforms.Affine2DBase attribute*), 3722
output_dims (*matplotlib.transforms.BlendedGenericTransform attribute*), 3738
output_dims (*matplotlib.transforms.Transform attribute*), 3747
output_dims (*matplotlib.transforms.TransformWrapper property*), 3751
outputStream() (*matplotlib.backends.backend_pdf.PdfFile method*), 2327
overlaps() (*matplotlib.transforms.BboxBase method*), 3733
ox (*matplotlib.mathtext.RasterParse attribute*), 3044

oy (*matplotlib.mathtext.RasterParse attribute*), 3044

## P

p0 (*matplotlib.transforms.Bbox property*), 3729
p0 (*matplotlib.transforms.BboxBase property*), 3734
p1 (*matplotlib.transforms.Bbox property*), 3729
p1 (*matplotlib.transforms.BboxBase property*), 3734
PackerBase (*class in matplotlib.offsetbox*), 3086
PAD (*matplotlib.table.Cell attribute*), 3646
padded() (*matplotlib.transforms.BboxBase method*), 3734
PaddedBox (*class in matplotlib.offsetbox*), 3087
paint() (*matplotlib.backends.backend_pdf.GraphicsContextPdf method*), 2324
paint_path() (*matplotlib.backends.backend_pdf.Op class method*), 2325
pan() (*matplotlib.backend_bases.NavigationToolbar2 method*), 2271
pan() (*matplotlib.backends.backend_webagg_core.NavigationToolbar2WebAgg method*), 2361
parasite_axes_class_factory() (*in module mpl_toolkits.axes_grid1.parasite_axes*), 3989
ParasiteAxes (*in module mpl_toolkits.axes_grid1.parasite_axes*), 3987
ParasiteAxesBase (*class in mpl_toolkits.axes_grid1.parasite_axes*), 3987
parse() (*matplotlib.mathtext.MathTextParser method*), 3043
parts (*matplotlib._type1font.Type1Font attribute*), 3767
pass_through (*matplotlib.transforms.BlendedGenericTransform attribute*), 3738
pass_through (*matplotlib.transforms.CompositeGenericTransform attribute*), 3740
pass_through (*matplotlib.transforms.TransformNode attribute*), 3750
pass_through (*matplotlib.transforms.TransformWrapper attribute*), 3751
Patch (*class in matplotlib.patches*), 3158
Patch3D (*class in mpl_toolkits.mplot3d.art3d*), 3886
Patch3DCollection (*class in mpl_toolkits.mplot3d.art3d*), 3888
patch_2d_to_3d() (*in module mpl_toolkits.mplot3d.art3d*), 3905
patch_collection_2d_to_3d() (*in module mpl_toolkits.mplot3d.art3d*), 3905
PatchCollection (*class in matplotlib.collections*), 2523
PATH, 433, 436, 437, 439, 440
Path (*class in matplotlib.path*), 3194
Path3DCollection (*class in mpl_toolkits.mplot3d.art3d*), 3890
PathCollection (*class in matplotlib.collections*), 2545
pathCollectionObject() (*matplotlib.backends.backend_pdf.PdfFile method*), 2327
PathEffectRenderer (*class in matplotlib.patheffects*), 3204
pathOperations() (*matplotlib.backends.backend_pdf.PdfFile static method*), 2327
PathPatch (*class in matplotlib.patches*), 3170
PathPatch3D (*class in mpl_toolkits.mplot3d.art3d*), 3893
pathpatch_2d_to_3d() (*in module mpl_toolkits.mplot3d.art3d*), 3906
PathPatchEffect (*class in matplotlib.patheffects*), 3206
pause() (*in module matplotlib.pyplot*), 3494
pause() (*matplotlib.animation.Animation method*), 1834
pchanged() (*matplotlib.artist.Artist method*), 1870
pchanged() (*matplotlib.axes.Axes method*), 2196
pchanged() (*matplotlib.collections.AsteriskPolygonCollection method*), 2397
pchanged() (*matplotlib.collections.CircleCollection method*), 2418
pchanged() (*matplotlib.collections.Collection method*), 2441
pchanged() (*matplotlib.collections.EllipseCollection method*), 2463
pchanged() (*matplotlib.collections.EventCollection method*), 2486
pchanged() (*matplotlib.collections.LineCollection method*), 2509
pchanged() (*matplotlib.collections.PatchCollection method*), 2531
pchanged() (*matplotlib.collections.PathCollection method*), 2554
pchanged() (*matplotlib.collections.PolyCollection method*), 2575
pchanged() (*matplotlib.collections.PolyQuadMesh method*), 2598
pchanged() (*matplotlib.collections.QuadMesh method*), 2622
pchanged() (*matplotlib.collections.RegularPolyCollection method*), 2645
pchanged() (*matplotlib.collections.StarPolygonCollection method*), 2667
pchanged() (*matplotlib.collections.TriMesh method*), 2690
pchanged() (*matplotlib.container.Container method*), 2754
pcolor() (*in module matplotlib.pyplot*), 3382
pcolor() (*matplotlib.axes.Axes method*), 2048
pcolorfast() (*matplotlib.axes.Axes method*), 2052
PcolorImage (*class in matplotlib.image*), 2978
pcolormesh() (*in module matplotlib.pyplot*), 3386
pcolormesh() (*matplotlib.axes.Axes method*), 2055
PdfFile (*class in matplotlib.backends.backend_pdf*), 2326
pdfFile (*matplotlib.backends.backend_pdf.Stream attribute*), 2335
PdfPages (*class in matplotlib.backends.backend_pdf*), 2328
PdfPages (*class in matplotlib.backends.backend_pgf*), 2336
pdfRepr() (*in module matplotlib.backends.backend_pdf*), 2335
pdfRepr() (*matplotlib.backends.backend_pdf.Name method*), 2324
pdfRepr() (*matplotlib.backends.backend_pdf.Op method*), 2325

pdfRepr() (*matplotlib.backends.backend_pdf.Reference method*), 2330
pdfRepr() (*matplotlib.backends.backend_pdf.Verbatim method*), 2335
PercentFormatter (*class in matplotlib.ticker*), 3708
persp_transformation() (*in module mpl_toolkits.mplot3d.proj3d*), 3908
phase_spectrum() (*in module matplotlib.mlab*), 3058
phase_spectrum() (*in module matplotlib.pyplot*), 3328
phase_spectrum() (*matplotlib.axes.Axes method*), 1989
pick() (*matplotlib.artist.Artist method*), 1873
pick() (*matplotlib.collections.AsteriskPolygonCollection method*), 2397
pick() (*matplotlib.collections.CircleCollection method*), 2418
pick() (*matplotlib.collections.Collection method*), 2442
pick() (*matplotlib.collections.EllipseCollection method*), 2464
pick() (*matplotlib.collections.EventCollection method*), 2486
pick() (*matplotlib.collections.LineCollection method*), 2509
pick() (*matplotlib.collections.PatchCollection method*), 2531
pick() (*matplotlib.collections.PathCollection method*), 2554
pick() (*matplotlib.collections.PolyCollection method*), 2575
pick() (*matplotlib.collections.PolyQuadMesh method*), 2598
pick() (*matplotlib.collections.QuadMesh method*), 2623
pick() (*matplotlib.collections.RegularPolyCollection method*), 2645
pick() (*matplotlib.collections.StarPolygonCollection method*), 2667
pick() (*matplotlib.collections.TriMesh method*), 2690
pick() (*matplotlib.figure.Figure method*), 2855
pick() (*mpl_toolkits.axes_grid1.parasite_axes.HostAxesBase method*), 3986
pick() (*mpl_toolkits.axes_grid1.parasite_axes.ParasiteAxesBase method*), 3987
pickable() (*matplotlib.artist.Artist method*), 1873
pickable() (*matplotlib.collections.AsteriskPolygonCollection method*), 2397
pickable() (*matplotlib.collections.CircleCollection method*), 2418
pickable() (*matplotlib.collections.Collection method*), 2442
pickable() (*matplotlib.collections.EllipseCollection method*), 2464
pickable() (*matplotlib.collections.EventCollection method*), 2487
pickable() (*matplotlib.collections.LineCollection method*), 2510
pickable() (*matplotlib.collections.PatchCollection method*), 2531
pickable() (*matplotlib.collections.PathCollection method*), 2554
pickable() (*matplotlib.collections.PolyCollection method*), 2576
pickable() (*matplotlib.collections.PolyQuadMesh method*), 2599
pickable() (*matplotlib.collections.QuadMesh method*), 2623
pickable() (*matplotlib.collections.RegularPolyCollection method*), 2645
pickable() (*matplotlib.collections.StarPolygonCollection method*), 2667
pickable() (*matplotlib.collections.TriMesh method*), 2690
PickEvent (*class in matplotlib.backend_bases*), 2272
pickradius (*matplotlib.axis.Axis property*), 2240
pickradius (*matplotlib.lines.Line2D property*), 3022
pie() (*in module matplotlib.pyplot*), 3292
pie() (*matplotlib.axes.Axes method*), 1953
pil_to_array() (*in module matplotlib.image*), 2983
PillowWriter (*class in matplotlib.animation*), 1844
pivot (*matplotlib.quiver.QuiverKey attribute*), 3598
PlaceHolderLayoutEngine (*class in matplotlib.layout_engine*), 2987
plot() (*in module matplotlib.pyplot*), 3250
plot() (*matplotlib.axes.Axes method*), 1907
plot() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3818
plot_date() (*in module matplotlib.pyplot*), 3265
plot_date() (*matplotlib.axes.Axes method*), 1927
plot_surface() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3822
plot_trisurf() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3825
plot_wireframe() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3824
PlotDirective (*class in matplotlib.sphinxext.plot_directive*), 3634
PlotError, 3635
point_at_t() (*matplotlib.bezier.BezierSegment method*), 2365
POINTER (*matplotlib.backend_tools.Cursors attribute*), 2289
points_to_pixels() (*matplotlib.backend_bases.RendererBase method*), 2278
points_to_pixels() (*matplotlib.backends.backend_agg.RendererAgg method*), 2314
points_to_pixels() (*matplotlib.backends.backend_cairo.RendererCairo method*), 2320
points_to_pixels() (*matplotlib.backends.backend_pgf.RendererPgf method*), 2341
points_to_pixels() (*matplotlib.backends.backend_template.RendererTemplate method*), 2308
polar() (*in module matplotlib.pyplot*), 3302
PolarAffine (*class in matplotlib.projections.polar*), 3528
PolarAxes (*class in matplotlib.projections.polar*), 3529
PolarAxes.InvertedPolarTransform (*class in matplotlib.projections.polar*), 3531
PolarAxes.PolarAffine (*class in matplotlib.projections.polar*), 3532
PolarAxes.PolarTransform (*class in matplotlib.projections.polar*), 3533

PolarAxes.RadialLocator (*class in matplotlib.projections.polar*), 3534
PolarAxes.ThetaFormatter (*class in matplotlib.projections.polar*), 3535
PolarAxes.ThetaLocator (*class in matplotlib.projections.polar*), 3535
PolarTransform (*class in matplotlib.projections.polar*), 3547
Poly3DCollection (*class in mpl_toolkits.mplot3d.art3d*), 3895
poly_collection_2d_to_3d() (*in module mpl_toolkits.mplot3d.art3d*), 3906
PolyCollection (*class in matplotlib.collections*), 2567
Polygon (*class in matplotlib.patches*), 3176
PolygonSelector (*class in matplotlib.widgets*), 3783
polynomial_coefficients (*matplotlib.bezier.BezierSegment property*), 2365
PolyQuadMesh (*class in matplotlib.collections*), 2590
pop() (*matplotlib.backends.backend_pdf.GraphicsContextPdf method*), 2324
pop_label() (*matplotlib.contour.ContourLabeler method*), 2759
pos (*matplotlib.backends.backend_pdf.Stream attribute*), 2335
positions (*matplotlib.widgets.ToolLineHandles property*), 3801
postscript_name (*matplotlib._afm.AFM property*), 1831
postscript_name (*matplotlib.ft2font.FT2Font attribute*), 2955
PowerNorm (*class in matplotlib.colors*), 2726
pprint_getters() (*matplotlib.artist.ArtistInspector method*), 1901
pprint_setters() (*matplotlib.artist.ArtistInspector method*), 1902
pprint_setters_rest() (*matplotlib.artist.ArtistInspector method*), 1902
press_pan() (*matplotlib.backend_bases.NavigationToolbar2 method*), 2271
press_zoom() (*matplotlib.backend_bases.NavigationToolbar2 method*), 2271
print_cycles() (*in module matplotlib.cbook*), 2378
print_eps() (*matplotlib.backends.backend_ps.FigureCanvasPS method*), 2342
print_figure() (*matplotlib.backend_bases.FigureCanvasBase method*), 2259
print_foo() (*matplotlib.backends.backend_template.FigureCanvasTemplate method*), 2305
print_jpeg() (*matplotlib.backends.backend_agg.FigureCanvasAgg method*), 2309
print_jpg() (*matplotlib.backends.backend_agg.FigureCanvasAgg method*), 2309
print_label() (*matplotlib.contour.ContourLabeler method*), 2759
print_pdf() (*matplotlib.backends.backend_cairo.FigureCanvasCairo method*), 2315
print_pdf() (*matplotlib.backends.backend_pdf.FigureCanvasPdf method*), 2323
print_pdf() (*matplotlib.backends.backend_pgf.FigureCanvasPgf method*), 2336
print_pgf() (*matplotlib.backends.backend_pgf.FigureCanvasPgf method*), 2336
print_png() (*matplotlib.backends.backend_agg.FigureCanvasAgg method*), 2310
print_png() (*matplotlib.backends.backend_cairo.FigureCanvasCairo method*), 2315
print_png() (*matplotlib.backends.backend_pgf.FigureCanvasPgf method*), 2336
print_ps() (*matplotlib.backends.backend_cairo.FigureCanvasCairo method*), 2316
print_ps() (*matplotlib.backends.backend_ps.FigureCanvasPS method*), 2342
print_raw() (*matplotlib.backends.backend_agg.FigureCanvasAgg method*), 2310
print_raw() (*matplotlib.backends.backend_cairo.FigureCanvasCairo method*), 2316
print_rgba() (*matplotlib.backends.backend_agg.FigureCanvasAgg method*), 2310
print_rgba() (*matplotlib.backends.backend_cairo.FigureCanvasCairo method*), 2316
print_svg() (*matplotlib.backends.backend_cairo.FigureCanvasCairo method*), 2316
print_svg() (*matplotlib.backends.backend_svg.FigureCanvasSVG method*), 2351
print_svgz() (*matplotlib.backends.backend_cairo.FigureCanvasCairo method*), 2316
print_svgz() (*matplotlib.backends.backend_svg.FigureCanvasSVG method*), 2352
print_tif() (*matplotlib.backends.backend_agg.FigureCanvasAgg method*), 2310
print_tiff() (*matplotlib.backends.backend_agg.FigureCanvasAgg method*), 2311
print_to_buffer() (*matplotlib.backends.backend_agg.FigureCanvasAgg*

method), 2311
print_webp() (matplotlib.backends.backend_agg.FigureCanvasAgg method), 2311
process() (matplotlib.cbook.CallbackRegistry method), 2372
process_figure_for_rasterizing() (in module matplotlib._tight_bbox), 3716
process_selected() (matplotlib.lines.VertexSelector method), 3034
process_value() (matplotlib.colors.Normalize static method), 2715
proj_points() (in module mpl_toolkits.mplot3d.proj3d), 3908
proj_trans_points() (in module mpl_toolkits.mplot3d.proj3d), 3908
proj_transform() (in module mpl_toolkits.mplot3d.proj3d), 3908
proj_transform_clip() (in module mpl_toolkits.mplot3d.proj3d), 3909
ProjectionRegistry (class in matplotlib.projections), 3527
prop (matplotlib._type1font.Type1Font attribute), 3767
properties() (matplotlib.artist.Artist method), 1877
properties() (matplotlib.artist.ArtistInspector method), 1902
properties() (matplotlib.collections.AsteriskPolygonCollection method), 2397
properties() (matplotlib.collections.CircleCollection method), 2419
properties() (matplotlib.collections.Collection method), 2442
properties() (matplotlib.collections.EllipseCollection method), 2464
properties() (matplotlib.collections.EventCollection method), 2487
properties() (matplotlib.collections.LineCollection method), 2510
properties() (matplotlib.collections.PatchCollection method), 2532
properties() (matplotlib.collections.PathCollection method), 2554
properties() (matplotlib.collections.PolyCollection method), 2576
properties() (matplotlib.collections.PolyQuadMesh method), 2599
properties() (matplotlib.collections.QuadMesh method), 2623
properties() (matplotlib.collections.RegularPolyCollection method), 2645
properties() (matplotlib.collections.StarPolygonCollection method), 2667
properties() (matplotlib.collections.TriMesh method), 2690
psd() (in module matplotlib.mlab), 3059
psd() (in module matplotlib.pyplot), 3331
psd() (matplotlib.axes.Axes method), 1992
PsFont (class in matplotlib.dviread), 2789
PsfontsMap (class in matplotlib.dviread), 2790
psname (matplotlib.dviread.PsFont attribute), 2789
pstoeps() (in module matplotlib.backends.backend_ps), 2346
pts_to_midstep() (in module matplotlib.cbook), 2379
pts_to_poststep() (in module matplotlib.cbook), 2379
pts_to_prestep() (in module matplotlib.cbook), 2380
push() (matplotlib.backends.backend_pdf.GraphicsContextPdf method), 2324
push() (matplotlib.cbook.Stack method), 2374
push_current() (matplotlib.backend_bases.NavigationToolbar2 method), 2271
push_current() (matplotlib.backend_tools.ToolViewsPositions method), 2298
pylab
    module, 4051
pyplot_show() (matplotlib.backend_bases.FigureManagerBase class method), 2263
pyplot_show() (matplotlib.backends.backend_webagg.FigureManagerWebAgg class method), 2362
Python Enhancement Proposals
    PEP 440, 4720
    PEP 3102, 4348
PYTHONPATH, 10

## Q

QT_API, 62, 1697, 2350, 4185, 4326
QuadContourSet (class in matplotlib.contour), 2765
QuadMesh (class in matplotlib.collections), 2613
Quiver (class in matplotlib.quiver), 3588
quiver() (in module matplotlib.pyplot), 3439
quiver() (matplotlib.axes.Axes method), 2098
quiver() (mpl_toolkits.mplot3d.axes3d.Axes3D method), 3829
QuiverKey (class in matplotlib.quiver), 3596
quiverkey() (in module matplotlib.pyplot), 3444
quiverkey() (matplotlib.axes.Axes method), 2103

## R

RadialAxis (class in matplotlib.projections.polar), 3548
RadialLocator (class in matplotlib.projections.polar), 3550
RadialTick (class in matplotlib.projections.polar), 3551
radii (matplotlib.patches.Annulus property), 3098
radio_group (matplotlib.backend_tools.ToolPan attribute), 2296
radio_group (matplotlib.backend_tools.ToolToggleBase attribute), 2298
radio_group (matplotlib.backend_tools.ToolZoom attribute), 2300
RadioButtons (class in matplotlib.widgets), 3785

radius (*matplotlib.patches.Circle property*), 3123
raise_if_exceeds() (*matplotlib.ticker.Locator method*), 3698
RangeSlider (*class in matplotlib.widgets*), 3787
RasterParse (*class in matplotlib.mathtext*), 3044
rc() (*in module matplotlib*), 1824
rc() (*in module matplotlib.pyplot*), 3487
rc_context() (*in module matplotlib*), 1823
rc_context() (*in module matplotlib.pyplot*), 3488
rc_file() (*in module matplotlib*), 1825
rc_file_defaults() (*in module matplotlib*), 1825
rc_params() (*in module matplotlib*), 1826
rc_params_from_file() (*in module matplotlib*), 1826
rcdefaults() (*in module matplotlib*), 1825
rcdefaults() (*in module matplotlib.pyplot*), 3489
RcParams (*class in matplotlib*), 1813
rcParams (*in module matplotlib*), 1813
RcStyleType (*in module matplotlib.typing*), 3769
readonly (*matplotlib.path.Path property*), 3201
recache() (*matplotlib.lines.Line2D method*), 3022
recache_always() (*matplotlib.lines.Line2D method*), 3022
recordXref() (*matplotlib.backends.backend_pdf.PdfFile method*), 2327
Rectangle (*class in matplotlib.patches*), 3179
rectangle (*matplotlib.backends.backend_pdf.Op attribute*), 2325
RectangleSelector (*class in matplotlib.widgets*), 3790
rects (*matplotlib.mathtext.VectorParse attribute*), 3045
recursive_subclasses() (*in module matplotlib._api*), 3805
redraw_in_frame() (*matplotlib.axes.Axes method*), 2206
Reference (*class in matplotlib.backends.backend_pdf*), 2330
refine_field() (*matplotlib.tri.UniformTriRefiner method*), 3763
refine_triangulation() (*matplotlib.tri.UniformTriRefiner method*), 3763
refresh_all() (*matplotlib.backends.backend_webagg_core.FigureManagerWebAgg method*), 2360
register() (*matplotlib.animation.MovieWriterRegistry method*), 1858
register() (*matplotlib.cm.ColormapRegistry method*), 2385
register() (*matplotlib.colors.ColorSequenceRegistry method*), 2740
register() (*matplotlib.projections.ProjectionRegistry method*), 3527
register_axis() (*matplotlib.spines.Spine method*), 3639
register_backend() (*in module matplotlib.backend_bases*), 2283
register_projection() (*in module matplotlib.projections*), 3527
register_scale() (*in module matplotlib.scale*), 3629
Registry (*class in matplotlib.units*), 3771
RegularPolyCollection (*class in matplotlib.collections*), 2636
RegularPolygon (*class in matplotlib.patches*), 3184
relativedelta (*class in matplotlib.dates*), 2784
release() (*matplotlib.widgets.LockDraw method*), 3782
release_mouse() (*matplotlib.backend_bases.FigureCanvasBase method*), 2260
release_pan() (*matplotlib.backend_bases.NavigationToolbar2 method*), 2271
release_zoom() (*matplotlib.backend_bases.NavigationToolbar2 method*), 2272
relim() (*matplotlib.axes.Axes method*), 2159
reload_library() (*in module matplotlib.style*), 3643
remove() (*matplotlib.artist.Artist method*), 1890
remove() (*matplotlib.cbook.Grouper method*), 2373
remove() (*matplotlib.cbook.Stack method*), 2374
remove() (*matplotlib.collections.AsteriskPolygonCollection method*), 2397
remove() (*matplotlib.collections.CircleCollection method*), 2419
remove() (*matplotlib.collections.Collection method*), 2442
remove() (*matplotlib.collections.EllipseCollection method*), 2464
remove() (*matplotlib.collections.EventCollection method*), 2487
remove() (*matplotlib.collections.LineCollection method*), 2510
remove() (*matplotlib.collections.PatchCollection method*), 2532
remove() (*matplotlib.collections.PathCollection method*), 2554
remove() (*matplotlib.collections.PolyCollection method*), 2576
remove() (*matplotlib.collections.PolyQuadMesh method*), 2599
remove() (*matplotlib.collections.QuadMesh method*), 2623
remove() (*matplotlib.collections.RegularPolyCollection method*), 2645
remove() (*matplotlib.collections.StarPolygonCollection method*), 2667
remove() (*matplotlib.collections.TriMesh method*), 2690
remove() (*matplotlib.colorbar.Colorbar method*), 2708
remove() (*matplotlib.container.Container method*), 2754
remove() (*matplotlib.contour.ContourLabeler method*), 2759
remove() (*matplotlib.widgets.ToolLineHandles method*), 3801
remove_callback() (*matplotlib.artist.Artist method*), 1870
remove_callback() (*matplotlib.axes.Axes method*), 2197
remove_callback() (*matplotlib.backend_bases.TimerBase method*), 2280
remove_callback() (*matplotlib.collections.AsteriskPolygonCollection method*), 2398
remove_callback() (*matplotlib.collections.CircleCollection method*), 2419
remove_callback() (*matplotlib.collections.Collection method*), 2442

remove_callback() (*matplotlib.collections.EllipseCollection method*), 2464
remove_callback() (*matplotlib.collections.EventCollection method*), 2487
remove_callback() (*matplotlib.collections.LineCollection method*), 2510
remove_callback() (*matplotlib.collections.PatchCollection method*), 2532
remove_callback() (*matplotlib.collections.PathCollection method*), 2554
remove_callback() (*matplotlib.collections.PolyCollection method*), 2576
remove_callback() (*matplotlib.collections.PolyQuadMesh method*), 2599
remove_callback() (*matplotlib.collections.QuadMesh method*), 2623
remove_callback() (*matplotlib.collections.RegularPolyCollection method*), 2646
remove_callback() (*matplotlib.collections.StarPolygonCollection method*), 2668
remove_callback() (*matplotlib.collections.TriMesh method*), 2690
remove_callback() (*matplotlib.container.Container method*), 2754
remove_comm() (*matplotlib.backends.backend_nbagg.FigureManagerNbAgg method*), 2322
remove_overlapping_locs (*matplotlib.axis.Axis property*), 2227
remove_rubberband() (*matplotlib.backend_bases.NavigationToolbar2 method*), 2272
remove_rubberband() (*matplotlib.backend_tools.RubberbandBase method*), 2289
remove_rubberband() (*matplotlib.backends.backend_webagg_core.NavigationToolbar2WebAgg method*), 2361
remove_ticks_and_titles() (*in module matplotlib.testing.decorators*), 3662
remove_tool() (*matplotlib.backend_managers.ToolManager method*), 2286
remove_toolitem() (*matplotlib.backend_bases.ToolContainerBase method*), 2282
remove_web_socket() (*matplotlib.backends.backend_webagg_core.FigureManagerWebAgg method*), 2360
rename_parameter() (*in module matplotlib._api.deprecation*), 3808
render_figures() (*in module matplotlib.sphinxext.plot_directive*), 3635
RendererAgg (*class in matplotlib.backends.backend_agg*), 2312
RendererBase (*class in matplotlib.backend_bases*), 2273
RendererCairo (*class in matplotlib.backends.backend_cairo*), 2317
RendererPdf (*class in matplotlib.backends.backend_pdf*), 2330
RendererPgf (*class in matplotlib.backends.backend_pgf*), 2337
RendererPS (*class in matplotlib.backends.backend_ps*), 2342
RendererSVG (*class in matplotlib.backends.backend_svg*), 2352
RendererTemplate (*class in matplotlib.backends.backend_template*), 2306
required_arguments (*matplotlib.sphinxext.figmpl_directive.FigureMpl attribute*), 3637
required_arguments (*matplotlib.sphinxext.mathmpl.MathDirective attribute*), 3631
required_arguments (*matplotlib.sphinxext.plot_directive.PlotDirective attribute*), 3635
required_interactive_framework (*matplotlib.backend_bases.FigureCanvasBase attribute*), 2260
resampled() (*matplotlib.colors.Colormap method*), 2733
resampled() (*matplotlib.colors.LinearSegmentedColormap method*), 2735
resampled() (*matplotlib.colors.ListedColormap method*), 2738
reserveObject() (*matplotlib.backends.backend_pdf.PdfFile method*), 2327
reset() (*matplotlib.widgets.SliderBase method*), 3796
reset_position() (*matplotlib.axes.Axes method*), 2194
reset_ticks() (*matplotlib.axis.Axis method*), 2247
reshow() (*matplotlib.backends.backend_nbagg.FigureManagerNbAgg method*), 2322
resize() (*matplotlib.backend_bases.FigureManagerBase method*), 2263
resize() (*matplotlib.backends.backend_webagg_core.FigureManagerWebAgg method*), 2360
RESIZE_HORIZONTAL (*matplotlib.backend_tools.Cursors attribute*), 2289
RESIZE_VERTICAL (*matplotlib.backend_tools.Cursors attribute*), 2289
ResizeEvent (*class in matplotlib.backend_bases*), 2279
RESOLUTION (*matplotlib.projections.geo.GeoAxes attribute*), 3564
resolve_backend() (*matplotlib.backends.registry.BackendRegistry method*), 2349

resolve_gui_or_backend() (*matplotlib.backends.registry.BackendRegistry method*), 2349
restore() (*matplotlib.backend_bases.GraphicsContextBase method*), 2265
restore() (*matplotlib.backends.backend_cairo.GraphicsContextCairo method*), 2316
restore_region() (*matplotlib.backends.backend_agg.FigureCanvasAgg method*), 2311
restore_region() (*matplotlib.backends.backend_agg.RendererAgg method*), 2314
restore_region() (*matplotlib.backends.backend_cairo.FigureCanvasCairo method*), 2316
resume() (*matplotlib.animation.Animation method*), 1834
reversed() (*matplotlib.colors.Colormap method*), 2733
reversed() (*matplotlib.colors.LinearSegmentedColormap method*), 2735
reversed() (*matplotlib.colors.ListedColormap method*), 2738
rgb_cmd() (*matplotlib.backends.backend_pdf.GraphicsContextPdf method*), 2324
rgb_to_hsv() (*in module matplotlib.colors*), 2749
RGBAColorType (*in module matplotlib.typing*), 3768
RGBAColourType (*in module matplotlib.typing*), 3768
RGBAxes (*class in mpl_toolkits.axes_grid1.axes_rgb*), 3946
RGBAxes (*class in mpl_toolkits.axisartist.axes_rgb*), 4001
RGBColorType (*in module matplotlib.typing*), 3768
RGBColourType (*in module matplotlib.typing*), 3768
rgrids() (*in module matplotlib.pyplot*), 3455
RIGHT (*matplotlib.backend_bases.MouseButton attribute*), 2269
rot_x() (*in module mpl_toolkits.mplot3d.proj3d*), 3909
rotate() (*matplotlib.transforms.Affine2D method*), 3720
rotate_around() (*matplotlib.transforms.Affine2D method*), 3720
rotate_axes() (*in module mpl_toolkits.mplot3d.art3d*), 3907
rotate_deg() (*matplotlib.transforms.Affine2D method*), 3720
rotate_deg_around() (*matplotlib.transforms.Affine2D method*), 3720
rotated() (*matplotlib.markers.MarkerStyle method*), 3042
rotated() (*matplotlib.transforms.BboxBase method*), 3734
rotation (*matplotlib.widgets.RectangleSelector property*), 3793
rotation_point (*matplotlib.patches.Rectangle property*), 3181
rowspan (*matplotlib.gridspec.SubplotSpec property*), 2961
RRuleLocator (*class in matplotlib.dates*), 2779
rrulewrapper (*class in matplotlib.dates*), 2785
RubberbandBase (*class in matplotlib.backend_tools*), 2289
run() (*matplotlib.sphinxext.plot_directive.PlotDirective method*), 3635

## S

safe_first_element() (*in module matplotlib.cbook*), 2381
safe_masked_invalid() (*in module matplotlib.cbook*), 2381
same_color() (*in module matplotlib.colors*), 2752
sanitize_sequence() (*in module matplotlib.cbook*), 2381
Sankey (*class in matplotlib.sankey*), 3609
save() (*matplotlib.animation.Animation method*), 1834
save_figure() (*matplotlib.backend_bases.NavigationToolbar2 method*), 2272
save_figure() (*matplotlib.backends.backend_webagg_core.NavigationToolbar2WebAgg method*), 2361
save_offset() (*matplotlib.offsetbox.DraggableAnnotation method*), 3075
save_offset() (*matplotlib.offsetbox.DraggableBase method*), 3076
save_offset() (*matplotlib.offsetbox.DraggableOffsetBox method*), 3077
savefig() (*in module matplotlib.pyplot*), 3495
savefig() (*matplotlib.backends.backend_pdf.PdfPages method*), 2330
savefig() (*matplotlib.backends.backend_pgf.PdfPages method*), 2337
savefig() (*matplotlib.figure.Figure method*), 2815
SaveFigureBase (*class in matplotlib.backend_tools*), 2290
saving() (*matplotlib.animation.AbstractMovieWriter method*), 1860
sca() (*in module matplotlib.pyplot*), 3225
sca() (*matplotlib.figure.Figure method*), 2861
sca() (*matplotlib.figure.FigureBase method*), 2927
Scalable (*in module mpl_toolkits.axes_grid1.axes_size*), 3952
scalable (*matplotlib.ft2font.FT2Font attribute*), 2955
ScalarFormatter (*class in matplotlib.ticker*), 3709
ScalarMappable (*class in matplotlib.cm*), 2385
scale() (*matplotlib.table.Table method*), 3653
scale() (*matplotlib.transforms.Affine2D method*), 3721
scale_factors (*matplotlib.tri.TriAnalyzer property*), 3766
scale_factory() (*in module matplotlib.scale*), 3629
ScaleBase (*class in matplotlib.scale*), 3626
Scaled (*class in mpl_toolkits.axes_grid1.axes_size*), 3952
scaled() (*matplotlib.colors.Normalize method*), 2716
scaled() (*matplotlib.markers.MarkerStyle method*), 3042
ScaledTranslation (*class in matplotlib.transforms*), 3744
scatter() (*in module matplotlib.pyplot*), 3262
scatter() (*matplotlib.axes.Axes method*), 1923
scatter() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3819
sci() (*in module matplotlib.pyplot*), 3482
score_family() (*matplotlib.font_manager.FontManager method*), 2943
score_size() (*matplotlib.font_manager.FontManager method*), 2943
score_stretch() (*matplotlib.font_manager.FontManager*

*method*), 2943
score_style() (*matplotlib.font_manager.FontManager method*), 2944
score_variant() (*matplotlib.font_manager.FontManager method*), 2944
score_weight() (*matplotlib.font_manager.FontManager method*), 2944
scotts_factor() (*matplotlib.mlab.GaussianKDE method*), 3048
scroll_pick_id (*matplotlib.backend_bases.FigureCanvasBase property*), 2260
scroll_zoom() (*matplotlib.backend_tools.ZoomPanBase method*), 2300
sec_mark (*mpl_toolkits.axisartist.angle_helper.FormatterDMS attribute*), 3993
sec_mark (*mpl_toolkits.axisartist.angle_helper.FormatterHMS attribute*), 3994
secondary_xaxis() (*matplotlib.axes.Axes method*), 2091
secondary_yaxis() (*matplotlib.axes.Axes method*), 2092
SecondLocator (*class in matplotlib.dates*), 2780
segment_hits() (*in module matplotlib.lines*), 3037
select_charmap() (*matplotlib.ft2font.FT2Font method*), 2955
select_matching_signature() (*in module matplotlib._api*), 3805
SELECT_REGION (*matplotlib.backend_tools.Cursors attribute*), 2289
select_step() (*in module mpl_toolkits.axisartist.angle_helper*), 3996
select_step24() (*in module mpl_toolkits.axisartist.angle_helper*), 3997
select_step360() (*in module mpl_toolkits.axisartist.angle_helper*), 3997
select_step_degree() (*in module mpl_toolkits.axisartist.angle_helper*), 3997
select_step_hour() (*in module mpl_toolkits.axisartist.angle_helper*), 3997
select_step_sub() (*in module mpl_toolkits.axisartist.angle_helper*), 3997
selectfont (*matplotlib.backends.backend_pdf.Op attribute*), 2325
semilogx() (*in module matplotlib.pyplot*), 3270
semilogx() (*matplotlib.axes.Axes method*), 1932
semilogy() (*in module matplotlib.pyplot*), 3271
semilogy() (*matplotlib.axes.Axes method*), 1933
send_binary() (*matplotlib.backends.backend_nbagg.CommSocket method*), 2321
send_binary() (*matplotlib.backends.backend_webagg.WebAggApplication.WebSocket method*), 2363
send_event() (*matplotlib.backends.backend_webagg_core.FigureCanvasWebAggCore method*), 2359
send_json() (*matplotlib.backends.backend_nbagg.CommSocket method*), 2321
send_json() (*matplotlib.backends.backend_webagg.WebAggApplication.WebSocket method*), 2364
send_message() (*matplotlib.backend_tools.ToolCursorPosition method*), 2292
set() (*matplotlib.artist.Artist method*), 1877
set() (*matplotlib.axes.Axes method*), 2213
set() (*matplotlib.collections.AsteriskPolygonCollection method*), 2398
set() (*matplotlib.collections.CircleCollection method*), 2419
set() (*matplotlib.collections.Collection method*), 2442
set() (*matplotlib.collections.EllipseCollection method*), 2464
set() (*matplotlib.collections.EventCollection method*), 2487
set() (*matplotlib.collections.LineCollection method*), 2510
set() (*matplotlib.collections.PatchCollection method*), 2532
set() (*matplotlib.collections.PathCollection method*), 2555
set() (*matplotlib.collections.PolyCollection method*), 2576
set() (*matplotlib.collections.PolyQuadMesh method*), 2599
set() (*matplotlib.collections.QuadMesh method*), 2623
set() (*matplotlib.collections.RegularPolyCollection method*), 2646
set() (*matplotlib.collections.StarPolygonCollection method*), 2668
set() (*matplotlib.collections.TriMesh method*), 2691
set() (*matplotlib.contour.ContourSet method*), 2763
set() (*matplotlib.contour.QuadContourSet method*), 2766
set() (*matplotlib.dates.rrulewrapper method*), 2786
set() (*matplotlib.figure.FigureBase method*), 2927
set() (*matplotlib.image.AxesImage method*), 2969
set() (*matplotlib.image.BboxImage method*), 2972
set() (*matplotlib.image.FigureImage method*), 2973
set() (*matplotlib.image.NonUniformImage method*), 2975
set() (*matplotlib.image.PcolorImage method*), 2979
set() (*matplotlib.layout_engine.ConstrainedLayoutEngine method*), 2986
set() (*matplotlib.layout_engine.LayoutEngine method*), 2987
set() (*matplotlib.layout_engine.PlaceHolderLayoutEngine method*), 2988
set() (*matplotlib.layout_engine.TightLayoutEngine method*), 2989
set() (*matplotlib.legend.Legend method*), 2998
set() (*matplotlib.lines.AxLine method*), 3035
set() (*matplotlib.lines.Line2D method*), 3022
set() (*matplotlib.offsetbox.AnchoredOffsetbox method*), 3066
set() (*matplotlib.offsetbox.AnchoredText method*), 3068
set() (*matplotlib.offsetbox.AnnotationBbox method*), 3072
set() (*matplotlib.offsetbox.AuxTransformBox method*), 3074
set() (*matplotlib.offsetbox.DrawingArea method*), 3077
set() (*matplotlib.offsetbox.HPacker method*), 3079
set() (*matplotlib.offsetbox.OffsetBox method*), 3082
set() (*matplotlib.offsetbox.OffsetImage method*), 3084
set() (*matplotlib.offsetbox.PackerBase method*), 3086
set() (*matplotlib.offsetbox.PaddedBox method*), 3088
set() (*matplotlib.offsetbox.TextArea method*), 3090
set() (*matplotlib.offsetbox.VPacker method*), 3093
set() (*matplotlib.patches.Annulus method*), 3098
set() (*matplotlib.patches.Arc method*), 3103

set() (*matplotlib.patches.Arrow method*), 3106
set() (*matplotlib.patches.Circle method*), 3123
set() (*matplotlib.patches.CirclePolygon method*), 3126
set() (*matplotlib.patches.ConnectionPatch method*), 3130
set() (*matplotlib.patches.Ellipse method*), 3137
set() (*matplotlib.patches.FancyArrow method*), 3141
set() (*matplotlib.patches.FancyArrowPatch method*), 3148
set() (*matplotlib.patches.FancyBboxPatch method*), 3154
set() (*matplotlib.patches.Patch method*), 3163
set() (*matplotlib.patches.PathPatch method*), 3171
set() (*matplotlib.patches.Polygon method*), 3177
set() (*matplotlib.patches.Rectangle method*), 3182
set() (*matplotlib.patches.RegularPolygon method*), 3186
set() (*matplotlib.patches.Shadow method*), 3189
set() (*matplotlib.patches.StepPatch method*), 3174
set() (*matplotlib.patches.Wedge method*), 3191
set() (*matplotlib.projections.geo.AitoffAxes method*), 3560
set() (*matplotlib.projections.geo.GeoAxes method*), 3568
set() (*matplotlib.projections.geo.HammerAxes method*), 3575
set() (*matplotlib.projections.geo.LambertAxes method*), 3581
set() (*matplotlib.projections.geo.MollweideAxes method*), 3586
set() (*matplotlib.projections.polar.PolarAxes method*), 3540
set() (*matplotlib.projections.polar.RadialAxis method*), 3549
set() (*matplotlib.projections.polar.RadialTick method*), 3551
set() (*matplotlib.projections.polar.ThetaAxis method*), 3553
set() (*matplotlib.projections.polar.ThetaTick method*), 3555
set() (*matplotlib.quiver.Barbs method*), 3604
set() (*matplotlib.quiver.Quiver method*), 3594
set() (*matplotlib.quiver.QuiverKey method*), 3598
set() (*matplotlib.spines.Spine method*), 3639
set() (*matplotlib.table.Cell method*), 3647
set() (*matplotlib.table.Table method*), 3653
set() (*matplotlib.text.Annotation method*), 3683
set() (*matplotlib.text.Text method*), 3667
set() (*matplotlib.transforms.Affine2D method*), 3721
set() (*matplotlib.transforms.Bbox method*), 3729
set() (*matplotlib.transforms.TransformWrapper method*), 3751
set() (*matplotlib.tri.TriContourSet method*), 3757
set() (*mpl_toolkits.axes_grid1.anchored_artists.AnchoredAuxTransformBox method*), 3913
set() (*mpl_toolkits.axes_grid1.anchored_artists.AnchoredDirectionArrows method*), 3918
set() (*mpl_toolkits.axes_grid1.anchored_artists.AnchoredDrawingArea method*), 3921
set() (*mpl_toolkits.axes_grid1.anchored_artists.AnchoredEllipse method*), 3924
set() (*mpl_toolkits.axes_grid1.anchored_artists.AnchoredSizeBar method*), 3928
set() (*mpl_toolkits.axes_grid1.inset_locator.AnchoredLocatorBase method*), 3954
set() (*mpl_toolkits.axes_grid1.inset_locator.AnchoredSizeLocator method*), 3956
set() (*mpl_toolkits.axes_grid1.inset_locator.AnchoredZoomLocator method*), 3959
set() (*mpl_toolkits.axes_grid1.inset_locator.BboxConnector method*), 3962
set() (*mpl_toolkits.axes_grid1.inset_locator.BboxConnectorPatch method*), 3964
set() (*mpl_toolkits.axes_grid1.inset_locator.BboxPatch method*), 3967
set() (*mpl_toolkits.axes_grid1.mpl_axes.Axes method*), 3981
set() (*mpl_toolkits.axes_grid1.mpl_axes.SimpleAxisArtist method*), 3983
set() (*mpl_toolkits.axisartist.axis_artist.AxisArtist method*), 4005
set() (*mpl_toolkits.axisartist.axis_artist.AxisLabel method*), 4010
set() (*mpl_toolkits.axisartist.axis_artist.GridlinesCollection method*), 4013
set() (*mpl_toolkits.axisartist.axis_artist.LabelBase method*), 4017
set() (*mpl_toolkits.axisartist.axis_artist.TickLabels method*), 4020
set() (*mpl_toolkits.axisartist.axis_artist.Ticks method*), 4023
set() (*mpl_toolkits.axisartist.axislines.Axes method*), 4032
set() (*mpl_toolkits.axisartist.axislines.AxesZero method*), 4036
set() (*mpl_toolkits.mplot3d.art3d.Line3D method*), 3881
set() (*mpl_toolkits.mplot3d.art3d.Line3DCollection method*), 3884
set() (*mpl_toolkits.mplot3d.art3d.Patch3D method*), 3886
set() (*mpl_toolkits.mplot3d.art3d.Patch3DCollection method*), 3888
set() (*mpl_toolkits.mplot3d.art3d.Path3DCollection method*), 3891
set() (*mpl_toolkits.mplot3d.art3d.PathPatch3D method*), 3893
set() (*mpl_toolkits.mplot3d.art3d.Poly3DCollection method*), 3896
set() (*mpl_toolkits.mplot3d.art3d.Text3D method*), 3902
set() (*mpl_toolkits.mplot3d.axis3d.Axis method*), 3877
set_3d_properties() (*mpl_toolkits.mplot3d.art3d.Line3D method*), 3882
set_3d_properties() (*mpl_toolkits.mplot3d.art3d.Patch3D method*), 3887
set_3d_properties() (*mpl_toolkits.mplot3d.art3d.Patch3DCollection method*), 3889
set_3d_properties() (*mpl_toolkits.mplot3d.art3d.Path3DCollection method*), 3892
set_3d_properties() (*mpl_toolkits.mplot3d.art3d.PathPatch3D method*), 3894
set_3d_properties()

(*mpl_toolkits.mplot3d.art3d.Poly3DCollection method*), 3897
set_3d_properties() (*mpl_toolkits.mplot3d.art3d.Text3D method*), 3903
set_aa() (*matplotlib.collections.AsteriskPolygonCollection method*), 2399
set_aa() (*matplotlib.collections.CircleCollection method*), 2420
set_aa() (*matplotlib.collections.Collection method*), 2444
set_aa() (*matplotlib.collections.EllipseCollection method*), 2465
set_aa() (*matplotlib.collections.EventCollection method*), 2489
set_aa() (*matplotlib.collections.LineCollection method*), 2511
set_aa() (*matplotlib.collections.PatchCollection method*), 2533
set_aa() (*matplotlib.collections.PathCollection method*), 2556
set_aa() (*matplotlib.collections.PolyCollection method*), 2577
set_aa() (*matplotlib.collections.PolyQuadMesh method*), 2600
set_aa() (*matplotlib.collections.QuadMesh method*), 2625
set_aa() (*matplotlib.collections.RegularPolyCollection method*), 2647
set_aa() (*matplotlib.collections.StarPolygonCollection method*), 2669
set_aa() (*matplotlib.collections.TriMesh method*), 2692
set_aa() (*matplotlib.lines.Line2D method*), 3023
set_aa() (*matplotlib.patches.Patch method*), 3164
set_active() (*matplotlib.widgets.CheckButtons method*), 3776
set_active() (*matplotlib.widgets.RadioButtons method*), 3787
set_active() (*matplotlib.widgets.Widget method*), 3802
set_adjustable() (*matplotlib.axes.Axes method*), 2167
set_agg_filter() (*matplotlib.artist.Artist method*), 1886
set_agg_filter() (*matplotlib.collections.AsteriskPolygonCollection method*), 2399
set_agg_filter() (*matplotlib.collections.CircleCollection method*), 2420
set_agg_filter() (*matplotlib.collections.Collection method*), 2444
set_agg_filter() (*matplotlib.collections.EllipseCollection method*), 2465
set_agg_filter() (*matplotlib.collections.EventCollection method*), 2489
set_agg_filter() (*matplotlib.collections.LineCollection method*), 2511
set_agg_filter() (*matplotlib.collections.PatchCollection method*), 2533
set_agg_filter() (*matplotlib.collections.PathCollection method*), 2556
set_agg_filter() (*matplotlib.collections.PolyCollection method*), 2578
set_agg_filter() (*matplotlib.collections.PolyQuadMesh method*), 2600
set_agg_filter() (*matplotlib.collections.QuadMesh method*), 2625
set_agg_filter() (*matplotlib.collections.RegularPolyCollection method*), 2647
set_agg_filter() (*matplotlib.collections.StarPolygonCollection method*), 2669
set_agg_filter() (*matplotlib.collections.TriMesh method*), 2692
set_alignment() (*matplotlib.legend.Legend method*), 2999
set_alpha() (*matplotlib.artist.Artist method*), 1883
set_alpha() (*matplotlib.backend_bases.GraphicsContextBase method*), 2265
set_alpha() (*matplotlib.backends.backend_cairo.GraphicsContextCairo method*), 2316
set_alpha() (*matplotlib.collections.AsteriskPolygonCollection method*), 2399
set_alpha() (*matplotlib.collections.CircleCollection method*), 2421
set_alpha() (*matplotlib.collections.Collection method*), 2444
set_alpha() (*matplotlib.collections.EllipseCollection method*), 2466
set_alpha() (*matplotlib.collections.EventCollection method*), 2489
set_alpha() (*matplotlib.collections.LineCollection method*), 2512
set_alpha() (*matplotlib.collections.PatchCollection method*), 2534
set_alpha() (*matplotlib.collections.PathCollection method*), 2556
set_alpha() (*matplotlib.collections.PolyCollection method*), 2578
set_alpha() (*matplotlib.collections.PolyQuadMesh method*), 2601
set_alpha() (*matplotlib.collections.QuadMesh method*), 2625
set_alpha() (*matplotlib.collections.RegularPolyCollection method*), 2647
set_alpha() (*matplotlib.collections.StarPolygonCollection method*), 2669
set_alpha() (*matplotlib.collections.TriMesh method*), 2692
set_alpha() (*matplotlib.colorbar.Colorbar method*), 2708
set_alpha() (*matplotlib.patches.Patch method*), 3164
set_alpha() (*mpl_toolkits.mplot3d.art3d.Poly3DCollection method*), 3898
set_anchor() (*matplotlib.axes.Axes method*), 2192
set_anchor() (*mpl_toolkits.axes_grid1.axes_divider.Divider method*), 3934
set_angle() (*matplotlib.patches.Annulus method*), 3100
set_angle() (*matplotlib.patches.Ellipse method*), 3138

set_angle() (*matplotlib.patches.Rectangle method*), 3183
set_angles() (*matplotlib.collections.EllipseCollection method*), 2466
set_animated() (*matplotlib.artist.Artist method*), 1882
set_animated() (*matplotlib.collections.AsteriskPolygonCollection method*), 2399
set_animated() (*matplotlib.collections.CircleCollection method*), 2421
set_animated() (*matplotlib.collections.Collection method*), 2444
set_animated() (*matplotlib.collections.EllipseCollection method*), 2466
set_animated() (*matplotlib.collections.EventCollection method*), 2489
set_animated() (*matplotlib.collections.LineCollection method*), 2512
set_animated() (*matplotlib.collections.PatchCollection method*), 2534
set_animated() (*matplotlib.collections.PathCollection method*), 2556
set_animated() (*matplotlib.collections.PolyCollection method*), 2578
set_animated() (*matplotlib.collections.PolyQuadMesh method*), 2601
set_animated() (*matplotlib.collections.QuadMesh method*), 2625
set_animated() (*matplotlib.collections.RegularPolyCollection method*), 2647
set_animated() (*matplotlib.collections.StarPolygonCollection method*), 2669
set_animated() (*matplotlib.collections.TriMesh method*), 2692
set_animated() (*matplotlib.widgets.ToolHandles method*), 3800
set_animated() (*matplotlib.widgets.ToolLineHandles method*), 3802
set_anncoords() (*matplotlib.text.Annotation method*), 3684
set_annotation_clip() (*matplotlib.patches.ConnectionPatch method*), 3131
set_antialiased() (*matplotlib.backend_bases.GraphicsContextBase method*), 2265
set_antialiased() (*matplotlib.backends.backend_cairo.GraphicsContextCairo method*), 2316
set_antialiased() (*matplotlib.collections.AsteriskPolygonCollection method*), 2400
set_antialiased() (*matplotlib.collections.CircleCollection method*), 2421
set_antialiased() (*matplotlib.collections.Collection method*), 2444
set_antialiased() (*matplotlib.collections.EllipseCollection method*), 2466
set_antialiased() (*matplotlib.collections.EventCollection method*), 2489
set_antialiased() (*matplotlib.collections.LineCollection method*), 2512
set_antialiased() (*matplotlib.collections.PatchCollection method*), 2534
set_antialiased() (*matplotlib.collections.PathCollection method*), 2557
set_antialiased() (*matplotlib.collections.PolyCollection method*), 2578
set_antialiased() (*matplotlib.collections.PolyQuadMesh method*), 2601
set_antialiased() (*matplotlib.collections.QuadMesh method*), 2625
set_antialiased() (*matplotlib.collections.RegularPolyCollection method*), 2648
set_antialiased() (*matplotlib.collections.StarPolygonCollection method*), 2670
set_antialiased() (*matplotlib.collections.TriMesh method*), 2692
set_antialiased() (*matplotlib.lines.Line2D method*), 3023
set_antialiased() (*matplotlib.patches.Patch method*), 3164
set_antialiased() (*matplotlib.text.Text method*), 3669
set_antialiaseds() (*matplotlib.collections.AsteriskPolygonCollection method*), 2400
set_antialiaseds() (*matplotlib.collections.CircleCollection method*), 2421
set_antialiaseds() (*matplotlib.collections.Collection method*), 2445
set_antialiaseds() (*matplotlib.collections.EllipseCollection method*), 2466
set_antialiaseds() (*matplotlib.collections.EventCollection method*), 2490
set_antialiaseds() (*matplotlib.collections.LineCollection method*), 2512
set_antialiaseds() (*matplotlib.collections.PatchCollection method*), 2534
set_antialiaseds() (*matplotlib.collections.PathCollection method*), 2557
set_antialiaseds() (*matplotlib.collections.PolyCollection method*),

2578
set_antialiaseds() (*matplotlib.collections.PolyQuadMesh method*), 2601
set_antialiaseds() (*matplotlib.collections.QuadMesh method*), 2626
set_antialiaseds() (*matplotlib.collections.RegularPolyCollection method*), 2648
set_antialiaseds() (*matplotlib.collections.StarPolygonCollection method*), 2670
set_antialiaseds() (*matplotlib.collections.TriMesh method*), 2693
set_array() (*matplotlib.cm.ScalarMappable method*), 2387
set_array() (*matplotlib.collections.AsteriskPolygonCollection method*), 2400
set_array() (*matplotlib.collections.CircleCollection method*), 2421
set_array() (*matplotlib.collections.Collection method*), 2445
set_array() (*matplotlib.collections.EllipseCollection method*), 2466
set_array() (*matplotlib.collections.EventCollection method*), 2490
set_array() (*matplotlib.collections.LineCollection method*), 2512
set_array() (*matplotlib.collections.PatchCollection method*), 2534
set_array() (*matplotlib.collections.PathCollection method*), 2557
set_array() (*matplotlib.collections.PolyCollection method*), 2578
set_array() (*matplotlib.collections.PolyQuadMesh method*), 2601
set_array() (*matplotlib.collections.QuadMesh method*), 2626
set_array() (*matplotlib.collections.RegularPolyCollection method*), 2648
set_array() (*matplotlib.collections.StarPolygonCollection method*), 2670
set_array() (*matplotlib.collections.TriMesh method*), 2693
set_array() (*matplotlib.image.NonUniformImage method*), 2976
set_array() (*matplotlib.image.PcolorImage method*), 2980
set_arrowstyle() (*matplotlib.patches.FancyArrowPatch method*), 3149
set_aspect() (*matplotlib.axes.Axes method*), 2163
set_aspect() (*mpl_toolkits.axes_grid1.axes_divider.Divider method*), 3935
set_aspect() (*mpl_toolkits.axes_grid1.axes_grid.Grid method*), 3943
set_aspect() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3856
set_autoscale_on() (*matplotlib.axes.Axes method*), 2161
set_autoscalex_on() (*matplotlib.axes.Axes method*), 2161
set_autoscaley_on() (*matplotlib.axes.Axes method*), 2162
set_autoscalez_on() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3855
set_axes_locator() (*matplotlib.axes.Axes method*), 2193
set_axes_locator() (*mpl_toolkits.axes_grid1.axes_grid.Grid method*), 3943
set_axes_pad() (*mpl_toolkits.axes_grid1.axes_grid.Grid method*), 3943
set_axis() (*matplotlib.dates.MicrosecondLocator method*), 2778
set_axis() (*matplotlib.projections.polar.PolarAxes.RadialLocator method*), 3535
set_axis() (*matplotlib.projections.polar.PolarAxes.ThetaLocator method*), 3535
set_axis() (*matplotlib.projections.polar.RadialLocator method*), 3550
set_axis() (*matplotlib.projections.polar.ThetaLocator method*), 3554
set_axis() (*matplotlib.ticker.TickHelper method*), 3714
set_axis() (*mpl_toolkits.axisartist.axis_artist.GridlinesCollection method*), 4015
set_axis_direction() (*mpl_toolkits.axisartist.axis_artist.AxisArtist method*), 4006
set_axis_direction() (*mpl_toolkits.axisartist.axis_artist.AxisLabel method*), 4011
set_axis_direction() (*mpl_toolkits.axisartist.axis_artist.TickLabels method*), 4021
set_axis_off() (*matplotlib.axes.Axes method*), 2111
set_axis_off() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3839
set_axis_on() (*matplotlib.axes.Axes method*), 2111
set_axis_on() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3839
set_axisbelow() (*matplotlib.axes.Axes method*), 2112
set_axislabel_direction() (*mpl_toolkits.axisartist.axis_artist.AxisArtist method*), 4007
set_axisline_style() (*mpl_toolkits.axisartist.axis_artist.AxisArtist method*), 4007
set_backgroundcolor() (*matplotlib.text.Text method*), 3669
set_bad() (*matplotlib.colors.Colormap method*), 2733
set_base() (*matplotlib.ticker.LogFormatter method*), 3700
set_bbox() (*matplotlib.text.Text method*), 3670
set_bbox_to_anchor() (*matplotlib.legend.Legend method*), 3000

set_bbox_to_anchor() (*matplotlib.offsetbox.AnchoredOffsetbox method*), 3067
set_bounds() (*matplotlib.patches.FancyBboxPatch method*), 3155
set_bounds() (*matplotlib.patches.Rectangle method*), 3183
set_bounds() (*matplotlib.spines.Spine method*), 3641
set_box_aspect() (*matplotlib.axes.Axes method*), 2165
set_box_aspect() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3857
set_boxstyle() (*matplotlib.patches.FancyBboxPatch method*), 3156
set_c() (*matplotlib.lines.Line2D method*), 3024
set_c() (*matplotlib.text.Text method*), 3670
set_canvas() (*matplotlib.figure.Figure method*), 2863
set_capstyle() (*matplotlib.backend_bases.GraphicsContextBase method*), 2266
set_capstyle() (*matplotlib.backends.backend_cairo.GraphicsContextCairo method*), 2316
set_capstyle() (*matplotlib.collections.AsteriskPolygonCollection method*), 2400
set_capstyle() (*matplotlib.collections.CircleCollection method*), 2421
set_capstyle() (*matplotlib.collections.Collection method*), 2445
set_capstyle() (*matplotlib.collections.EllipseCollection method*), 2467
set_capstyle() (*matplotlib.collections.EventCollection method*), 2490
set_capstyle() (*matplotlib.collections.LineCollection method*), 2513
set_capstyle() (*matplotlib.collections.PatchCollection method*), 2534
set_capstyle() (*matplotlib.collections.PathCollection method*), 2557
set_capstyle() (*matplotlib.collections.PolyCollection method*), 2579
set_capstyle() (*matplotlib.collections.PolyQuadMesh method*), 2602
set_capstyle() (*matplotlib.collections.QuadMesh method*), 2626
set_capstyle() (*matplotlib.collections.RegularPolyCollection method*), 2648
set_capstyle() (*matplotlib.collections.StarPolygonCollection method*), 2670
set_capstyle() (*matplotlib.collections.TriMesh method*), 2693
set_capstyle() (*matplotlib.patches.Patch method*), 3165
set_center() (*matplotlib.patches.Annulus method*), 3100
set_center() (*matplotlib.patches.Ellipse method*), 3138
set_center() (*matplotlib.patches.Wedge method*), 3192
set_charmap() (*matplotlib.ft2font.FT2Font method*), 2956
set_check_props() (*matplotlib.widgets.CheckButtons method*), 3777
set_child() (*matplotlib.offsetbox.AnchoredOffsetbox method*), 3067
set_children() (*matplotlib.transforms.TransformNode method*), 3750
set_clim() (*matplotlib.cm.ScalarMappable method*), 2387
set_clim() (*matplotlib.collections.AsteriskPolygonCollection method*), 2400
set_clim() (*matplotlib.collections.CircleCollection method*), 2422
set_clim() (*matplotlib.collections.Collection method*), 2445
set_clim() (*matplotlib.collections.EllipseCollection method*), 2467
set_clim() (*matplotlib.collections.EventCollection method*), 2490
set_clim() (*matplotlib.collections.LineCollection method*), 2513
set_clim() (*matplotlib.collections.PatchCollection method*), 2535
set_clim() (*matplotlib.collections.PathCollection method*), 2557
set_clim() (*matplotlib.collections.PolyCollection method*), 2579
set_clim() (*matplotlib.collections.PolyQuadMesh method*), 2602
set_clim() (*matplotlib.collections.QuadMesh method*), 2626
set_clim() (*matplotlib.collections.RegularPolyCollection method*), 2648
set_clim() (*matplotlib.collections.StarPolygonCollection method*), 2670
set_clim() (*matplotlib.collections.TriMesh method*), 2693
set_clip_box() (*matplotlib.artist.Artist method*), 1875
set_clip_box() (*matplotlib.collections.AsteriskPolygonCollection method*), 2400
set_clip_box() (*matplotlib.collections.CircleCollection method*), 2422
set_clip_box() (*matplotlib.collections.Collection method*), 2445
set_clip_box() (*matplotlib.collections.EllipseCollection method*), 2467
set_clip_box() (*matplotlib.collections.EventCollection method*), 2490
set_clip_box() (*matplotlib.collections.LineCollection method*), 2513
set_clip_box() (*matplotlib.collections.PatchCollection method*), 2535
set_clip_box() (*matplotlib.collections.PathCollection method*), 2557
set_clip_box() (*matplotlib.collections.PolyCollection method*), 2579
set_clip_box() (*matplotlib.collections.PolyQuadMesh method*), 2602
set_clip_box() (*matplotlib.collections.QuadMesh method*), 2627
set_clip_box() (*matplotlib.collections.RegularPolyCollection*

*method*), 2648
set_clip_box() (*matplotlib.collections.StarPolygonCollection method*), 2670
set_clip_box() (*matplotlib.collections.TriMesh method*), 2693
set_clip_box() (*matplotlib.text.Text method*), 3670
set_clip_on() (*matplotlib.artist.Artist method*), 1874
set_clip_on() (*matplotlib.collections.AsteriskPolygonCollection method*), 2401
set_clip_on() (*matplotlib.collections.CircleCollection method*), 2422
set_clip_on() (*matplotlib.collections.Collection method*), 2446
set_clip_on() (*matplotlib.collections.EllipseCollection method*), 2467
set_clip_on() (*matplotlib.collections.EventCollection method*), 2490
set_clip_on() (*matplotlib.collections.LineCollection method*), 2513
set_clip_on() (*matplotlib.collections.PatchCollection method*), 2535
set_clip_on() (*matplotlib.collections.PathCollection method*), 2558
set_clip_on() (*matplotlib.collections.PolyCollection method*), 2579
set_clip_on() (*matplotlib.collections.PolyQuadMesh method*), 2602
set_clip_on() (*matplotlib.collections.QuadMesh method*), 2627
set_clip_on() (*matplotlib.collections.RegularPolyCollection method*), 2649
set_clip_on() (*matplotlib.collections.StarPolygonCollection method*), 2671
set_clip_on() (*matplotlib.collections.TriMesh method*), 2694
set_clip_on() (*matplotlib.text.Text method*), 3670
set_clip_path() (*matplotlib.artist.Artist method*), 1875
set_clip_path() (*matplotlib.backend_bases.GraphicsContextBase method*), 2266
set_clip_path() (*matplotlib.backends.backend_cairo.GraphicsContextCairo method*), 2316
set_clip_path() (*matplotlib.collections.AsteriskPolygonCollection method*), 2401
set_clip_path() (*matplotlib.collections.CircleCollection method*), 2422
set_clip_path() (*matplotlib.collections.Collection method*), 2446
set_clip_path() (*matplotlib.collections.EllipseCollection method*), 2468
set_clip_path() (*matplotlib.collections.EventCollection method*), 2491
set_clip_path() (*matplotlib.collections.LineCollection method*), 2513
set_clip_path() (*matplotlib.collections.PatchCollection method*), 2535
set_clip_path() (*matplotlib.collections.PathCollection method*), 2558
set_clip_path() (*matplotlib.collections.PolyCollection method*), 2580
set_clip_path() (*matplotlib.collections.PolyQuadMesh method*), 2603
set_clip_path() (*matplotlib.collections.QuadMesh method*), 2627
set_clip_path() (*matplotlib.collections.RegularPolyCollection method*), 2649
set_clip_path() (*matplotlib.collections.StarPolygonCollection method*), 2671
set_clip_path() (*matplotlib.collections.TriMesh method*), 2694
set_clip_path() (*matplotlib.text.Text method*), 3670
set_clip_rectangle() (*matplotlib.backend_bases.GraphicsContextBase method*), 2266
set_clip_rectangle() (*matplotlib.backends.backend_cairo.GraphicsContextCairo method*), 2316
set_closed() (*matplotlib.patches.Polygon method*), 3178
set_cmap() (*in module matplotlib.pyplot*), 3483
set_cmap() (*matplotlib.cm.ScalarMappable method*), 2387
set_cmap() (*matplotlib.collections.AsteriskPolygonCollection method*), 2401
set_cmap() (*matplotlib.collections.CircleCollection method*), 2423
set_cmap() (*matplotlib.collections.Collection method*), 2446
set_cmap() (*matplotlib.collections.EllipseCollection method*), 2468
set_cmap() (*matplotlib.collections.EventCollection method*), 2491
set_cmap() (*matplotlib.collections.LineCollection method*), 2514
set_cmap() (*matplotlib.collections.PatchCollection method*), 2536
set_cmap() (*matplotlib.collections.PathCollection method*), 2558
set_cmap() (*matplotlib.collections.PolyCollection method*), 2580
set_cmap() (*matplotlib.collections.PolyQuadMesh method*), 2603
set_cmap() (*matplotlib.collections.QuadMesh method*), 2627
set_cmap() (*matplotlib.collections.RegularPolyCollection method*), 2649
set_cmap() (*matplotlib.collections.StarPolygonCollection method*), 2671
set_cmap() (*matplotlib.collections.TriMesh method*), 2694
set_cmap() (*matplotlib.image.NonUniformImage method*), 2977

set_color() (*matplotlib.backends.backend_ps.RendererPS method*), 2345
set_color() (*matplotlib.collections.AsteriskPolygonCollection method*), 2402
set_color() (*matplotlib.collections.CircleCollection method*), 2423
set_color() (*matplotlib.collections.Collection method*), 2446
set_color() (*matplotlib.collections.EllipseCollection method*), 2468
set_color() (*matplotlib.collections.EventCollection method*), 2491
set_color() (*matplotlib.collections.LineCollection method*), 2514
set_color() (*matplotlib.collections.PatchCollection method*), 2536
set_color() (*matplotlib.collections.PathCollection method*), 2558
set_color() (*matplotlib.collections.PolyCollection method*), 2580
set_color() (*matplotlib.collections.PolyQuadMesh method*), 2603
set_color() (*matplotlib.collections.QuadMesh method*), 2628
set_color() (*matplotlib.collections.RegularPolyCollection method*), 2650
set_color() (*matplotlib.collections.StarPolygonCollection method*), 2672
set_color() (*matplotlib.collections.TriMesh method*), 2694
set_color() (*matplotlib.lines.Line2D method*), 3024
set_color() (*matplotlib.patches.Patch method*), 3165
set_color() (*matplotlib.spines.Spine method*), 3641
set_color() (*matplotlib.text.Text method*), 3671
set_color() (*mpl_toolkits.axisartist.axis_artist.Ticks method*), 4024
set_colors() (*matplotlib.collections.EventCollection method*), 2491
set_colors() (*matplotlib.collections.LineCollection method*), 2514
set_connectionstyle() (*matplotlib.patches.FancyArrowPatch method*), 3150
set_constrained_layout() (*matplotlib.figure.Figure method*), 2851
set_constrained_layout_pads() (*matplotlib.figure.Figure method*), 2851
set_context() (*matplotlib.backends.backend_cairo.RendererCairo method*), 2320
set_cursor() (*matplotlib.backend_bases.FigureCanvasBase method*), 2260
set_cursor() (*matplotlib.backends.backend_webagg_core.FigureCanvasWebAggCore method*), 2359
set_dash_capstyle() (*matplotlib.lines.Line2D method*), 3024
set_dash_joinstyle() (*matplotlib.lines.Line2D method*), 3024
set_dashes() (*matplotlib.backend_bases.GraphicsContextBase method*), 2266
set_dashes() (*matplotlib.backends.backend_cairo.GraphicsContextCairo method*), 2316
set_dashes() (*matplotlib.collections.AsteriskPolygonCollection method*), 2402
set_dashes() (*matplotlib.collections.CircleCollection method*), 2423
set_dashes() (*matplotlib.collections.Collection method*), 2447
set_dashes() (*matplotlib.collections.EllipseCollection method*), 2468
set_dashes() (*matplotlib.collections.EventCollection method*), 2492
set_dashes() (*matplotlib.collections.LineCollection method*), 2515
set_dashes() (*matplotlib.collections.PatchCollection method*), 2536
set_dashes() (*matplotlib.collections.PathCollection method*), 2559
set_dashes() (*matplotlib.collections.PolyCollection method*), 2580
set_dashes() (*matplotlib.collections.PolyQuadMesh method*), 2604
set_dashes() (*matplotlib.collections.QuadMesh method*), 2628
set_dashes() (*matplotlib.collections.RegularPolyCollection method*), 2650
set_dashes() (*matplotlib.collections.StarPolygonCollection method*), 2672
set_dashes() (*matplotlib.collections.TriMesh method*), 2695
set_dashes() (*matplotlib.lines.Line2D method*), 3024
set_data() (*matplotlib.image.FigureImage method*), 2974
set_data() (*matplotlib.image.NonUniformImage method*), 2977
set_data() (*matplotlib.image.PcolorImage method*), 2980
set_data() (*matplotlib.lines.Line2D method*), 3025
set_data() (*matplotlib.offsetbox.OffsetImage method*), 3085
set_data() (*matplotlib.patches.Arrow method*), 3107
set_data() (*matplotlib.patches.FancyArrow method*), 3143
set_data() (*matplotlib.patches.StepPatch method*), 3175
set_data() (*matplotlib.widgets.ToolHandles method*), 3800
set_data() (*matplotlib.widgets.ToolLineHandles method*), 3802
set_data_3d() (*mpl_toolkits.mplot3d.art3d.Line3D method*), 3882
set_data_interval() (*matplotlib.axis.Axis method*), 2237
set_default_alignment() (*mpl_toolkits.axisartist.axis_artist.AxisLabel method*), 4012
set_default_angle()

(*mpl_toolkits.axisartist.axis_artist.AxisLabel method*), 4012
set_default_handler_map() (*matplotlib.legend.Legend class method*), 3000
set_default_intervals() (*matplotlib.axis.Axis method*), 2247
set_default_locators_and_formatters() (*matplotlib.scale.AsinhScale method*), 3615
set_default_locators_and_formatters() (*matplotlib.scale.FuncScale method*), 3617
set_default_locators_and_formatters() (*matplotlib.scale.LinearScale method*), 3622
set_default_locators_and_formatters() (*matplotlib.scale.LogitScale method*), 3625
set_default_locators_and_formatters() (*matplotlib.scale.LogScale method*), 3623
set_default_locators_and_formatters() (*matplotlib.scale.ScaleBase method*), 3627
set_default_locators_and_formatters() (*matplotlib.scale.SymmetricalLogScale method*), 3628
set_default_weight() (*matplotlib.font_manager.FontManager method*), 2944
set_depthshade() (*mpl_toolkits.mplot3d.art3d.Patch3DCollection method*), 3890
set_depthshade() (*mpl_toolkits.mplot3d.art3d.Path3DCollection method*), 3892
set_dpi() (*matplotlib.figure.Figure method*), 2844
set_dpi() (*matplotlib.figure.SubFigure method*), 2903
set_draggable() (*matplotlib.legend.Legend method*), 3000
set_drawstyle() (*matplotlib.lines.Line2D method*), 3025
set_ds() (*matplotlib.lines.Line2D method*), 3025
set_ec() (*matplotlib.collections.AsteriskPolygonCollection method*), 2402
set_ec() (*matplotlib.collections.CircleCollection method*), 2423
set_ec() (*matplotlib.collections.Collection method*), 2447
set_ec() (*matplotlib.collections.EllipseCollection method*), 2469
set_ec() (*matplotlib.collections.EventCollection method*), 2492
set_ec() (*matplotlib.collections.LineCollection method*), 2515
set_ec() (*matplotlib.collections.PatchCollection method*), 2536
set_ec() (*matplotlib.collections.PathCollection method*), 2559
set_ec() (*matplotlib.collections.PolyCollection method*), 2581
set_ec() (*matplotlib.collections.PolyQuadMesh method*), 2604
set_ec() (*matplotlib.collections.QuadMesh method*), 2628
set_ec() (*matplotlib.collections.RegularPolyCollection method*), 2650
set_ec() (*matplotlib.collections.StarPolygonCollection method*), 2672
set_ec() (*matplotlib.collections.TriMesh method*), 2695
set_ec() (*matplotlib.patches.Patch method*), 3165
set_edgecolor() (*matplotlib.collections.AsteriskPolygonCollection method*), 2402
set_edgecolor() (*matplotlib.collections.CircleCollection method*), 2423
set_edgecolor() (*matplotlib.collections.Collection method*), 2447
set_edgecolor() (*matplotlib.collections.EllipseCollection method*), 2469
set_edgecolor() (*matplotlib.collections.EventCollection method*), 2492
set_edgecolor() (*matplotlib.collections.LineCollection method*), 2515
set_edgecolor() (*matplotlib.collections.PatchCollection method*), 2536
set_edgecolor() (*matplotlib.collections.PathCollection method*), 2559
set_edgecolor() (*matplotlib.collections.PolyCollection method*), 2581
set_edgecolor() (*matplotlib.collections.PolyQuadMesh method*), 2604
set_edgecolor() (*matplotlib.collections.QuadMesh method*), 2628
set_edgecolor() (*matplotlib.collections.RegularPolyCollection method*), 2650
set_edgecolor() (*matplotlib.collections.StarPolygonCollection method*), 2672
set_edgecolor() (*matplotlib.collections.TriMesh method*), 2695
set_edgecolor() (*matplotlib.figure.Figure method*), 2856
set_edgecolor() (*matplotlib.figure.FigureBase method*), 2928
set_edgecolor() (*matplotlib.figure.SubFigure method*), 2903
set_edgecolor() (*matplotlib.patches.Patch method*), 3165
set_edgecolor() (*mpl_toolkits.mplot3d.art3d.Poly3DCollection method*), 3898
set_edgecolors() (*matplotlib.collections.AsteriskPolygonCollection method*), 2402
set_edgecolors() (*matplotlib.collections.CircleCollection method*), 2424
set_edgecolors() (*matplotlib.collections.Collection method*), 2447
set_edgecolors() (*matplotlib.collections.EllipseCollection method*), 2469
set_edgecolors() (*matplotlib.collections.EventCollection method*), 2492
set_edgecolors() (*matplotlib.collections.LineCollection method*), 2515

set_edgecolors() (*matplotlib.collections.PatchCollection method*), 2537
set_edgecolors() (*matplotlib.collections.PathCollection method*), 2559
set_edgecolors() (*matplotlib.collections.PolyCollection method*), 2581
set_edgecolors() (*matplotlib.collections.PolyQuadMesh method*), 2604
set_edgecolors() (*matplotlib.collections.QuadMesh method*), 2628
set_edgecolors() (*matplotlib.collections.RegularPolyCollection method*), 2650
set_edgecolors() (*matplotlib.collections.StarPolygonCollection method*), 2672
set_edgecolors() (*matplotlib.collections.TriMesh method*), 2695
set_epoch() (*in module matplotlib.dates*), 2786
set_extent() (*matplotlib.image.AxesImage method*), 2970
set_extremes() (*matplotlib.colors.Colormap method*), 2733
set_extremes() (*mpl_toolkits.axisartist.grid_helper_curvelinear.FloatingAxisArtistHelper method*), 4050
set_facecolor() (*matplotlib.axes.Axes method*), 2116
set_facecolor() (*matplotlib.collections.AsteriskPolygonCollection method*), 2402
set_facecolor() (*matplotlib.collections.CircleCollection method*), 2424
set_facecolor() (*matplotlib.collections.Collection method*), 2447
set_facecolor() (*matplotlib.collections.EllipseCollection method*), 2469
set_facecolor() (*matplotlib.collections.EventCollection method*), 2492
set_facecolor() (*matplotlib.collections.LineCollection method*), 2515
set_facecolor() (*matplotlib.collections.PatchCollection method*), 2537
set_facecolor() (*matplotlib.collections.PathCollection method*), 2559
set_facecolor() (*matplotlib.collections.PolyCollection method*), 2581
set_facecolor() (*matplotlib.collections.PolyQuadMesh method*), 2604
set_facecolor() (*matplotlib.collections.QuadMesh method*), 2628
set_facecolor() (*matplotlib.collections.RegularPolyCollection method*), 2650
set_facecolor() (*matplotlib.collections.StarPolygonCollection method*), 2672
set_facecolor() (*matplotlib.collections.TriMesh method*), 2695
set_facecolor() (*matplotlib.figure.Figure method*), 2856
set_facecolor() (*matplotlib.figure.FigureBase method*), 2928
set_facecolor() (*matplotlib.figure.SubFigure method*), 2902
set_facecolor() (*matplotlib.patches.Patch method*), 3165
set_facecolor() (*mpl_toolkits.mplot3d.art3d.Poly3DCollection method*), 3898
set_facecolors() (*matplotlib.collections.AsteriskPolygonCollection method*), 2402
set_facecolors() (*matplotlib.collections.CircleCollection method*), 2424
set_facecolors() (*matplotlib.collections.Collection method*), 2447
set_facecolors() (*matplotlib.collections.EllipseCollection method*), 2469
set_facecolors() (*matplotlib.collections.EventCollection method*), 2492
set_facecolors() (*matplotlib.collections.LineCollection method*), 2515
set_facecolors() (*matplotlib.collections.PatchCollection method*), 2537
set_facecolors() (*matplotlib.collections.PathCollection method*), 2559
set_facecolors() (*matplotlib.collections.PolyCollection method*), 2581
set_facecolors() (*matplotlib.collections.PolyQuadMesh method*), 2604
set_facecolors() (*matplotlib.collections.QuadMesh method*), 2628
set_facecolors() (*matplotlib.collections.RegularPolyCollection method*), 2650
set_facecolors() (*matplotlib.collections.StarPolygonCollection method*), 2672
set_facecolors() (*matplotlib.collections.TriMesh method*), 2695
set_family() (*matplotlib.font_manager.FontProperties method*), 2946
set_family() (*matplotlib.text.Text method*), 3671
set_fc() (*matplotlib.collections.AsteriskPolygonCollection method*), 2402
set_fc() (*matplotlib.collections.CircleCollection method*), 2424
set_fc() (*matplotlib.collections.Collection method*), 2447
set_fc() (*matplotlib.collections.EllipseCollection method*), 2469
set_fc() (*matplotlib.collections.EventCollection method*), 2492
set_fc() (*matplotlib.collections.LineCollection method*), 2515
set_fc() (*matplotlib.collections.PatchCollection method*), 2537
set_fc() (*matplotlib.collections.PathCollection method*), 2559
set_fc() (*matplotlib.collections.PolyCollection method*),

2581
set_fc() (*matplotlib.collections.PolyQuadMesh method*), 2604
set_fc() (*matplotlib.collections.QuadMesh method*), 2628
set_fc() (*matplotlib.collections.RegularPolyCollection method*), 2650
set_fc() (*matplotlib.collections.StarPolygonCollection method*), 2672
set_fc() (*matplotlib.collections.TriMesh method*), 2695
set_fc() (*matplotlib.patches.Patch method*), 3165
set_figheight() (*matplotlib.figure.Figure method*), 2843
set_figure() (*matplotlib.artist.Artist method*), 1890
set_figure() (*matplotlib.backend_managers.ToolManager method*), 2286
set_figure() (*matplotlib.backend_tools.ToolBase method*), 2291
set_figure() (*matplotlib.backend_tools.ToolCursorPosition method*), 2292
set_figure() (*matplotlib.backend_tools.ToolSetCursor method*), 2297
set_figure() (*matplotlib.backend_tools.ToolToggleBase method*), 2298
set_figure() (*matplotlib.collections.AsteriskPolygonCollection method*), 2403
set_figure() (*matplotlib.collections.CircleCollection method*), 2424
set_figure() (*matplotlib.collections.Collection method*), 2447
set_figure() (*matplotlib.collections.EllipseCollection method*), 2469
set_figure() (*matplotlib.collections.EventCollection method*), 2492
set_figure() (*matplotlib.collections.LineCollection method*), 2515
set_figure() (*matplotlib.collections.PatchCollection method*), 2537
set_figure() (*matplotlib.collections.PathCollection method*), 2559
set_figure() (*matplotlib.collections.PolyCollection method*), 2581
set_figure() (*matplotlib.collections.PolyQuadMesh method*), 2604
set_figure() (*matplotlib.collections.QuadMesh method*), 2629
set_figure() (*matplotlib.collections.RegularPolyCollection method*), 2651
set_figure() (*matplotlib.collections.StarPolygonCollection method*), 2673
set_figure() (*matplotlib.collections.TriMesh method*), 2695
set_figure() (*matplotlib.offsetbox.AnnotationBbox method*), 3073
set_figure() (*matplotlib.offsetbox.OffsetBox method*), 3083
set_figure() (*matplotlib.quiver.QuiverKey method*), 3599
set_figure() (*matplotlib.table.Cell method*), 3648
set_figure() (*matplotlib.text.Annotation method*), 3684
set_figwidth() (*matplotlib.figure.Figure method*), 2844
set_file() (*matplotlib.font_manager.FontProperties method*), 2946
set_fill() (*matplotlib.patches.Patch method*), 3166
set_fillstyle() (*matplotlib.lines.Line2D method*), 3025
set_filternorm() (*matplotlib.image.NonUniformImage method*), 2977
set_filterrad() (*matplotlib.image.NonUniformImage method*), 2977
set_font() (*matplotlib.backends.backend_ps.RendererPS method*), 2346
set_font() (*matplotlib.text.Text method*), 3671
set_font_properties() (*matplotlib.text.Text method*), 3671
set_font_settings_for_testing() (*in module matplotlib.testing*), 3657
set_fontconfig_pattern() (*matplotlib.font_manager.FontProperties method*), 2946
set_fontfamily() (*matplotlib.text.Text method*), 3671
set_fontname() (*matplotlib.text.Text method*), 3672
set_fontproperties() (*matplotlib.text.Text method*), 3672
set_fontsize() (*matplotlib.offsetbox.AnnotationBbox method*), 3073
set_fontsize() (*matplotlib.table.Cell method*), 3649
set_fontsize() (*matplotlib.table.Table method*), 3654
set_fontsize() (*matplotlib.text.Text method*), 3672
set_fontstretch() (*matplotlib.text.Text method*), 3672
set_fontstyle() (*matplotlib.text.Text method*), 3673
set_fontvariant() (*matplotlib.text.Text method*), 3673
set_fontweight() (*matplotlib.text.Text method*), 3673
set_foreground() (*matplotlib.backend_bases.GraphicsContextBase method*), 2266
set_foreground() (*matplotlib.backends.backend_cairo.GraphicsContextCairo method*), 2317
set_forward_navigation_events() (*matplotlib.axes.Axes method*), 2199
set_frame_on() (*matplotlib.axes.Axes method*), 2112
set_frame_on() (*matplotlib.legend.Legend method*), 3001
set_frame_props() (*matplotlib.widgets.CheckButtons method*), 3777
set_frameon() (*matplotlib.figure.Figure method*), 2855
set_frameon() (*matplotlib.figure.FigureBase method*), 2929
set_frameon() (*matplotlib.figure.SubFigure method*), 2901
set_gamma() (*matplotlib.colors.LinearSegmentedColormap method*), 2736
set_gapcolor() (*matplotlib.collections.EventCollection method*), 2492
set_gapcolor() (*matplotlib.collections.LineCollection method*), 2515
set_gapcolor() (*matplotlib.lines.Line2D method*), 3026
set_gid() (*matplotlib.artist.Artist method*), 1894

set_gid() (*matplotlib.backend_bases.GraphicsContextBase method*), 2266
set_gid() (*matplotlib.collections.AsteriskPolygonCollection method*), 2403
set_gid() (*matplotlib.collections.CircleCollection method*), 2424
set_gid() (*matplotlib.collections.Collection method*), 2448
set_gid() (*matplotlib.collections.EllipseCollection method*), 2469
set_gid() (*matplotlib.collections.EventCollection method*), 2493
set_gid() (*matplotlib.collections.LineCollection method*), 2516
set_gid() (*matplotlib.collections.PatchCollection method*), 2537
set_gid() (*matplotlib.collections.PathCollection method*), 2560
set_gid() (*matplotlib.collections.PolyCollection method*), 2581
set_gid() (*matplotlib.collections.PolyQuadMesh method*), 2604
set_gid() (*matplotlib.collections.QuadMesh method*), 2629
set_gid() (*matplotlib.collections.RegularPolyCollection method*), 2651
set_gid() (*matplotlib.collections.StarPolygonCollection method*), 2673
set_gid() (*matplotlib.collections.TriMesh method*), 2696
set_grid_helper() (*mpl_toolkits.axisartist.axis_artist.GridlinesCollection method*), 4015
set_ha() (*matplotlib.text.Text method*), 3673
set_hatch() (*matplotlib.backend_bases.GraphicsContextBase method*), 2267
set_hatch() (*matplotlib.collections.AsteriskPolygonCollection method*), 2403
set_hatch() (*matplotlib.collections.CircleCollection method*), 2424
set_hatch() (*matplotlib.collections.Collection method*), 2448
set_hatch() (*matplotlib.collections.EllipseCollection method*), 2470
set_hatch() (*matplotlib.collections.EventCollection method*), 2493
set_hatch() (*matplotlib.collections.LineCollection method*), 2516
set_hatch() (*matplotlib.collections.PatchCollection method*), 2537
set_hatch() (*matplotlib.collections.PathCollection method*), 2560
set_hatch() (*matplotlib.collections.PolyCollection method*), 2582
set_hatch() (*matplotlib.collections.PolyQuadMesh method*), 2605
set_hatch() (*matplotlib.collections.QuadMesh method*), 2629
set_hatch() (*matplotlib.collections.RegularPolyCollection method*), 2651
set_hatch() (*matplotlib.collections.StarPolygonCollection method*), 2673
set_hatch() (*matplotlib.collections.TriMesh method*), 2696
set_hatch() (*matplotlib.patches.Patch method*), 3166
set_hatch_color() (*matplotlib.backend_bases.GraphicsContextBase method*), 2267
set_height() (*matplotlib.offsetbox.OffsetBox method*), 3083
set_height() (*matplotlib.patches.Ellipse method*), 3138
set_height() (*matplotlib.patches.FancyBboxPatch method*), 3157
set_height() (*matplotlib.patches.Rectangle method*), 3183
set_height_ratios() (*matplotlib.gridspec.GridSpecBase method*), 2963
set_heights() (*matplotlib.collections.EllipseCollection method*), 2470
set_history_buttons() (*matplotlib.backend_bases.NavigationToolbar2 method*), 2272
set_history_buttons() (*matplotlib.backends.backend_webagg_core.NavigationToolbar2WebAgg method*), 2361
set_horizontal() (*mpl_toolkits.axes_grid1.axes_divider.Divider method*), 3935
set_horizontalalignment() (*matplotlib.text.Text method*), 3674
set_image_mode() (*matplotlib.backends.backend_webagg_core.FigureCanvasWebAggCore method*), 2359
set_in_layout() (*matplotlib.artist.Artist method*), 1896
set_in_layout() (*matplotlib.collections.AsteriskPolygonCollection method*), 2403
set_in_layout() (*matplotlib.collections.CircleCollection method*), 2425
set_in_layout() (*matplotlib.collections.Collection method*), 2448
set_in_layout() (*matplotlib.collections.EllipseCollection method*), 2470
set_in_layout() (*matplotlib.collections.EventCollection method*), 2494
set_in_layout() (*matplotlib.collections.LineCollection method*), 2516
set_in_layout() (*matplotlib.collections.PatchCollection method*), 2538
set_in_layout() (*matplotlib.collections.PathCollection method*), 2560
set_in_layout() (*matplotlib.collections.PolyCollection method*), 2582
set_in_layout() (*matplotlib.collections.PolyQuadMesh method*), 2605
set_in_layout() (*matplotlib.collections.QuadMesh method*), 2629
set_in_layout() (*matplotlib.collections.RegularPolyCollection method*), 2651
set_in_layout()

(*matplotlib.collections.StarPolygonCollection method*), 2673
set_in_layout() (*matplotlib.collections.TriMesh method*), 2696
set_interpolation() (*matplotlib.image.NonUniformImage method*), 2977
set_inverted() (*matplotlib.axis.Axis method*), 2238
set_joinstyle() (*matplotlib.backend_bases.GraphicsContextBase method*), 2267
set_joinstyle() (*matplotlib.backends.backend_cairo.GraphicsContextCairo method*), 2317
set_joinstyle() (*matplotlib.collections.AsteriskPolygonCollection method*), 2404
set_joinstyle() (*matplotlib.collections.CircleCollection method*), 2425
set_joinstyle() (*matplotlib.collections.Collection method*), 2449
set_joinstyle() (*matplotlib.collections.EllipseCollection method*), 2470
set_joinstyle() (*matplotlib.collections.EventCollection method*), 2494
set_joinstyle() (*matplotlib.collections.LineCollection method*), 2517
set_joinstyle() (*matplotlib.collections.PatchCollection method*), 2538
set_joinstyle() (*matplotlib.collections.PathCollection method*), 2561
set_joinstyle() (*matplotlib.collections.PolyCollection method*), 2582
set_joinstyle() (*matplotlib.collections.PolyQuadMesh method*), 2605
set_joinstyle() (*matplotlib.collections.QuadMesh method*), 2630
set_joinstyle() (*matplotlib.collections.RegularPolyCollection method*), 2652
set_joinstyle() (*matplotlib.collections.StarPolygonCollection method*), 2674
set_joinstyle() (*matplotlib.collections.TriMesh method*), 2697
set_joinstyle() (*matplotlib.patches.Patch method*), 3166
set_label() (*matplotlib.artist.Artist method*), 1894
set_label() (*matplotlib.collections.AsteriskPolygonCollection method*), 2404
set_label() (*matplotlib.collections.CircleCollection method*), 2425
set_label() (*matplotlib.collections.Collection method*), 2449
set_label() (*matplotlib.collections.EllipseCollection method*), 2471
set_label() (*matplotlib.collections.EventCollection method*), 2494
set_label() (*matplotlib.collections.LineCollection method*), 2517
set_label() (*matplotlib.collections.PatchCollection method*), 2538
set_label() (*matplotlib.collections.PathCollection method*), 2561
set_label() (*matplotlib.collections.PolyCollection method*), 2582
set_label() (*matplotlib.collections.PolyQuadMesh method*), 2606
set_label() (*matplotlib.collections.QuadMesh method*), 2630
set_label() (*matplotlib.collections.RegularPolyCollection method*), 2652
set_label() (*matplotlib.collections.StarPolygonCollection method*), 2674
set_label() (*matplotlib.collections.TriMesh method*), 2697
set_label() (*matplotlib.colorbar.Colorbar method*), 2708
set_label() (*matplotlib.container.Container method*), 2754
set_label() (*mpl_toolkits.axes_grid1.mpl_axes.SimpleAxisArtist method*), 3984
set_label() (*mpl_toolkits.axisartist.axis_artist.AxisArtist method*), 4008
set_label1() (*matplotlib.axis.Tick method*), 2252
set_label2() (*matplotlib.axis.Tick method*), 2252
set_label_coords() (*matplotlib.axis.Axis method*), 2227
set_label_minor() (*matplotlib.ticker.LogFormatter method*), 3700
set_label_mode() (*mpl_toolkits.axes_grid1.axes_grid.Grid method*), 3943
set_label_position() (*matplotlib.axis.Axis method*), 2228
set_label_position() (*matplotlib.axis.XAxis method*), 2243
set_label_position() (*matplotlib.axis.YAxis method*), 2245
set_label_position() (*mpl_toolkits.mplot3d.axis3d.Axis method*), 3878
set_label_props() (*matplotlib.widgets.CheckButtons method*), 3777
set_label_props() (*matplotlib.widgets.RadioButtons method*), 3787
set_label_text() (*matplotlib.axis.Axis method*), 2228
set_latitude_grid() (*matplotlib.projections.geo.GeoAxes method*), 3569
set_layout_engine() (*matplotlib.figure.Figure method*), 2847
set_linecap() (*matplotlib.backends.backend_ps.RendererPS method*), 2346
set_linedash() (*matplotlib.backends.backend_ps.RendererPS method*), 2346
set_linejoin() (*matplotlib.backends.backend_ps.RendererPS method*), 2346
set_linelength() (*matplotlib.collections.EventCollection method*), 2494

set_lineoffset() (*matplotlib.collections.EventCollection method*), 2494
set_linespacing() (*matplotlib.text.Text method*), 3674
set_linestyle() (*matplotlib.collections.AsteriskPolygonCollection method*), 2404
set_linestyle() (*matplotlib.collections.CircleCollection method*), 2425
set_linestyle() (*matplotlib.collections.Collection method*), 2449
set_linestyle() (*matplotlib.collections.EllipseCollection method*), 2471
set_linestyle() (*matplotlib.collections.EventCollection method*), 2494
set_linestyle() (*matplotlib.collections.LineCollection method*), 2517
set_linestyle() (*matplotlib.collections.PatchCollection method*), 2538
set_linestyle() (*matplotlib.collections.PathCollection method*), 2561
set_linestyle() (*matplotlib.collections.PolyCollection method*), 2583
set_linestyle() (*matplotlib.collections.PolyQuadMesh method*), 2606
set_linestyle() (*matplotlib.collections.QuadMesh method*), 2630
set_linestyle() (*matplotlib.collections.RegularPolyCollection method*), 2652
set_linestyle() (*matplotlib.collections.StarPolygonCollection method*), 2674
set_linestyle() (*matplotlib.collections.TriMesh method*), 2697
set_linestyle() (*matplotlib.lines.Line2D method*), 3026
set_linestyle() (*matplotlib.patches.Patch method*), 3166
set_linestyles() (*matplotlib.collections.AsteriskPolygonCollection method*), 2404
set_linestyles() (*matplotlib.collections.CircleCollection method*), 2426
set_linestyles() (*matplotlib.collections.Collection method*), 2449
set_linestyles() (*matplotlib.collections.EllipseCollection method*), 2471
set_linestyles() (*matplotlib.collections.EventCollection method*), 2495
set_linestyles() (*matplotlib.collections.LineCollection method*), 2517
set_linestyles() (*matplotlib.collections.PatchCollection method*), 2539
set_linestyles() (*matplotlib.collections.PathCollection method*), 2561
set_linestyles() (*matplotlib.collections.PolyCollection method*), 2583
set_linestyles() (*matplotlib.collections.PolyQuadMesh method*), 2606
set_linestyles() (*matplotlib.collections.QuadMesh method*), 2630
set_linestyles() (*matplotlib.collections.RegularPolyCollection method*), 2652
set_linestyles() (*matplotlib.collections.StarPolygonCollection method*), 2674
set_linestyles() (*matplotlib.collections.TriMesh method*), 2697
set_linewidth() (*matplotlib.backend_bases.GraphicsContextBase method*), 2267
set_linewidth() (*matplotlib.backends.backend_cairo.GraphicsContextCairo method*), 2317
set_linewidth() (*matplotlib.backends.backend_ps.RendererPS method*), 2346
set_linewidth() (*matplotlib.collections.AsteriskPolygonCollection method*), 2404
set_linewidth() (*matplotlib.collections.CircleCollection method*), 2426
set_linewidth() (*matplotlib.collections.Collection method*), 2449
set_linewidth() (*matplotlib.collections.EllipseCollection method*), 2471
set_linewidth() (*matplotlib.collections.EventCollection method*), 2495
set_linewidth() (*matplotlib.collections.LineCollection method*), 2517
set_linewidth() (*matplotlib.collections.PatchCollection method*), 2539
set_linewidth() (*matplotlib.collections.PathCollection method*), 2561
set_linewidth() (*matplotlib.collections.PolyCollection method*), 2583
set_linewidth() (*matplotlib.collections.PolyQuadMesh method*), 2606
set_linewidth() (*matplotlib.collections.QuadMesh method*), 2630
set_linewidth() (*matplotlib.collections.RegularPolyCollection method*), 2652
set_linewidth() (*matplotlib.collections.StarPolygonCollection method*), 2674
set_linewidth() (*matplotlib.collections.TriMesh method*), 2697
set_linewidth() (*matplotlib.figure.Figure method*), 2856
set_linewidth() (*matplotlib.figure.FigureBase method*), 2929
set_linewidth() (*matplotlib.figure.SubFigure method*), 2902
set_linewidth() (*matplotlib.lines.Line2D method*), 3027
set_linewidth() (*matplotlib.patches.Patch method*), 3167
set_linewidth() (*mpl_toolkits.mplot3d.art3d.Path3DCollection*

*method*), 3892
set_linewidths() (*matplotlib.collections.AsteriskPolygonCollection method*), 2405
set_linewidths() (*matplotlib.collections.CircleCollection method*), 2426
set_linewidths() (*matplotlib.collections.Collection method*), 2450
set_linewidths() (*matplotlib.collections.EllipseCollection method*), 2471
set_linewidths() (*matplotlib.collections.EventCollection method*), 2495
set_linewidths() (*matplotlib.collections.LineCollection method*), 2518
set_linewidths() (*matplotlib.collections.PatchCollection method*), 2539
set_linewidths() (*matplotlib.collections.PathCollection method*), 2562
set_linewidths() (*matplotlib.collections.PolyCollection method*), 2583
set_linewidths() (*matplotlib.collections.PolyQuadMesh method*), 2606
set_linewidths() (*matplotlib.collections.QuadMesh method*), 2631
set_linewidths() (*matplotlib.collections.RegularPolyCollection method*), 2653
set_linewidths() (*matplotlib.collections.StarPolygonCollection method*), 2675
set_linewidths() (*matplotlib.collections.TriMesh method*), 2698
set_loc() (*matplotlib.legend.Legend method*), 3001
set_locator() (*mpl_toolkits.axes_grid1.axes_divider.Divider method*), 3935
set_locs() (*matplotlib.ticker.Formatter method*), 3696
set_locs() (*matplotlib.ticker.LogFormatter method*), 3700
set_locs() (*matplotlib.ticker.LogitFormatter method*), 3703
set_locs() (*matplotlib.ticker.ScalarFormatter method*), 3710
set_locs_angles() (*mpl_toolkits.axisartist.axis_artist.Ticks method*), 4025
set_locs_angles_labels() (*mpl_toolkits.axisartist.axis_artist.TickLabels method*), 4022
set_loglevel() (*in module matplotlib*), 1827
set_loglevel() (*in module matplotlib.pyplot*), 3522
set_longitude_grid() (*matplotlib.projections.geo.GeoAxes method*), 3570
set_longitude_grid_ends() (*matplotlib.projections.geo.GeoAxes method*), 3570
set_ls() (*matplotlib.collections.AsteriskPolygonCollection method*), 2405
set_ls() (*matplotlib.collections.CircleCollection method*), 2426
set_ls() (*matplotlib.collections.Collection method*), 2450
set_ls() (*matplotlib.collections.EllipseCollection method*), 2471
set_ls() (*matplotlib.collections.EventCollection method*), 2495
set_ls() (*matplotlib.collections.LineCollection method*), 2518
set_ls() (*matplotlib.collections.PatchCollection method*), 2539
set_ls() (*matplotlib.collections.PathCollection method*), 2562
set_ls() (*matplotlib.collections.PolyCollection method*), 2583
set_ls() (*matplotlib.collections.PolyQuadMesh method*), 2607
set_ls() (*matplotlib.collections.QuadMesh method*), 2631
set_ls() (*matplotlib.collections.RegularPolyCollection method*), 2653
set_ls() (*matplotlib.collections.StarPolygonCollection method*), 2675
set_ls() (*matplotlib.collections.TriMesh method*), 2698
set_ls() (*matplotlib.lines.Line2D method*), 3027
set_ls() (*matplotlib.patches.Patch method*), 3167
set_lw() (*matplotlib.collections.AsteriskPolygonCollection method*), 2405
set_lw() (*matplotlib.collections.CircleCollection method*), 2426
set_lw() (*matplotlib.collections.Collection method*), 2450
set_lw() (*matplotlib.collections.EllipseCollection method*), 2472
set_lw() (*matplotlib.collections.EventCollection method*), 2495
set_lw() (*matplotlib.collections.LineCollection method*), 2518
set_lw() (*matplotlib.collections.PatchCollection method*), 2539
set_lw() (*matplotlib.collections.PathCollection method*), 2562
set_lw() (*matplotlib.collections.PolyCollection method*), 2584
set_lw() (*matplotlib.collections.PolyQuadMesh method*), 2607
set_lw() (*matplotlib.collections.QuadMesh method*), 2631
set_lw() (*matplotlib.collections.RegularPolyCollection method*), 2653
set_lw() (*matplotlib.collections.StarPolygonCollection method*), 2675
set_lw() (*matplotlib.collections.TriMesh method*), 2698
set_lw() (*matplotlib.lines.Line2D method*), 3027
set_lw() (*matplotlib.patches.Patch method*), 3167
set_ma() (*matplotlib.text.Text method*), 3674
set_major_formatter() (*matplotlib.axis.Axis method*), 2223
set_major_locator() (*matplotlib.axis.Axis method*), 2225
set_marker() (*matplotlib.lines.Line2D method*), 3027
set_markeredgecolor() (*matplotlib.lines.Line2D method*), 3027

set_markeredgewidth() (*matplotlib.lines.Line2D method*), 3027
set_markerfacecolor() (*matplotlib.lines.Line2D method*), 3027
set_markerfacecoloralt() (*matplotlib.lines.Line2D method*), 3028
set_markersize() (*matplotlib.lines.Line2D method*), 3028
set_markevery() (*matplotlib.lines.Line2D method*), 3028
set_mask() (*matplotlib.tri.Triangulation method*), 3756
set_math_fontfamily() (*matplotlib.font_manager.FontProperties method*), 2946
set_math_fontfamily() (*matplotlib.text.Text method*), 3674
set_matrix() (*matplotlib.transforms.Affine2D method*), 3721
set_max() (*matplotlib.widgets.RangeSlider method*), 3789
set_mec() (*matplotlib.lines.Line2D method*), 3029
set_message() (*matplotlib.backend_bases.NavigationToolbar2 method*), 2272
set_message() (*matplotlib.backend_bases.ToolContainerBase method*), 2282
set_message() (*matplotlib.backends.backend_webagg_core.NavigationToolbar2WebAgg method*), 2361
set_mew() (*matplotlib.lines.Line2D method*), 3029
set_mfc() (*matplotlib.lines.Line2D method*), 3029
set_mfcalt() (*matplotlib.lines.Line2D method*), 3029
set_min() (*matplotlib.widgets.RangeSlider method*), 3790
set_minor_formatter() (*matplotlib.axis.Axis method*), 2226
set_minor_locator() (*matplotlib.axis.Axis method*), 2226
set_minor_number() (*matplotlib.ticker.LogitFormatter method*), 3703
set_minor_threshold() (*matplotlib.ticker.LogitFormatter method*), 3703
set_mouseover() (*matplotlib.artist.Artist method*), 1871
set_mouseover() (*matplotlib.collections.AsteriskPolygonCollection method*), 2405
set_mouseover() (*matplotlib.collections.CircleCollection method*), 2426
set_mouseover() (*matplotlib.collections.Collection method*), 2450
set_mouseover() (*matplotlib.collections.EllipseCollection method*), 2472
set_mouseover() (*matplotlib.collections.EventCollection method*), 2495
set_mouseover() (*matplotlib.collections.LineCollection method*), 2518
set_mouseover() (*matplotlib.collections.PatchCollection method*), 2539
set_mouseover() (*matplotlib.collections.PathCollection method*), 2562
set_mouseover() (*matplotlib.collections.PolyCollection method*), 2584
set_mouseover() (*matplotlib.collections.PolyQuadMesh method*), 2607
set_mouseover() (*matplotlib.collections.QuadMesh method*), 2631
set_mouseover() (*matplotlib.collections.RegularPolyCollection method*), 2653
set_mouseover() (*matplotlib.collections.StarPolygonCollection method*), 2675
set_mouseover() (*matplotlib.collections.TriMesh method*), 2698
set_ms() (*matplotlib.lines.Line2D method*), 3029
set_multialignment() (*matplotlib.text.Text method*), 3674
set_multilinebaseline() (*matplotlib.offsetbox.TextArea method*), 3091
set_mutation_aspect() (*matplotlib.patches.FancyArrowPatch method*), 3150
set_mutation_aspect() (*matplotlib.patches.FancyBboxPatch method*), 3157
set_mutation_scale() (*matplotlib.patches.FancyArrowPatch method*), 3151
set_mutation_scale() (*matplotlib.patches.FancyBboxPatch method*), 3157
set_name() (*matplotlib.font_manager.FontProperties method*), 2947
set_name() (*matplotlib.text.Text method*), 3675
set_navigate() (*matplotlib.axes.Axes method*), 2198
set_navigate_mode() (*matplotlib.axes.Axes method*), 2199
set_ncols() (*matplotlib.legend.Legend method*), 3002
set_norm() (*matplotlib.cm.ScalarMappable method*), 2387
set_norm() (*matplotlib.collections.AsteriskPolygonCollection method*), 2405
set_norm() (*matplotlib.collections.CircleCollection method*), 2427
set_norm() (*matplotlib.collections.Collection method*), 2450
set_norm() (*matplotlib.collections.EllipseCollection method*), 2472
set_norm() (*matplotlib.collections.EventCollection method*), 2495
set_norm() (*matplotlib.collections.LineCollection method*), 2518
set_norm() (*matplotlib.collections.PatchCollection method*), 2540
set_norm() (*matplotlib.collections.PathCollection method*), 2562
set_norm() (*matplotlib.collections.PolyCollection method*), 2584
set_norm() (*matplotlib.collections.PolyQuadMesh method*), 2607
set_norm() (*matplotlib.collections.QuadMesh method*),

2631
set_norm() (*matplotlib.collections.RegularPolyCollection method*), 2653
set_norm() (*matplotlib.collections.StarPolygonCollection method*), 2675
set_norm() (*matplotlib.collections.TriMesh method*), 2698
set_norm() (*matplotlib.image.NonUniformImage method*), 2978
set_offset() (*matplotlib.offsetbox.AuxTransformBox method*), 3075
set_offset() (*matplotlib.offsetbox.DrawingArea method*), 3078
set_offset() (*matplotlib.offsetbox.OffsetBox method*), 3084
set_offset() (*matplotlib.offsetbox.TextArea method*), 3091
set_offset_position() (*matplotlib.axis.YAxis method*), 2245
set_offset_string() (*matplotlib.ticker.FixedFormatter method*), 3695
set_offset_string() (*matplotlib.ticker.FuncFormatter method*), 3697
set_offset_transform() (*matplotlib.collections.AsteriskPolygonCollection method*), 2406
set_offset_transform() (*matplotlib.collections.CircleCollection method*), 2427
set_offset_transform() (*matplotlib.collections.Collection method*), 2451
set_offset_transform() (*matplotlib.collections.EllipseCollection method*), 2472
set_offset_transform() (*matplotlib.collections.EventCollection method*), 2496
set_offset_transform() (*matplotlib.collections.LineCollection method*), 2519
set_offset_transform() (*matplotlib.collections.PatchCollection method*), 2540
set_offset_transform() (*matplotlib.collections.PathCollection method*), 2563
set_offset_transform() (*matplotlib.collections.PolyCollection method*), 2584
set_offset_transform() (*matplotlib.collections.PolyQuadMesh method*), 2607
set_offset_transform() (*matplotlib.collections.QuadMesh method*), 2632
set_offset_transform() (*matplotlib.collections.RegularPolyCollection method*), 2654
set_offset_transform() (*matplotlib.collections.StarPolygonCollection method*), 2676
set_offset_transform() (*matplotlib.collections.TriMesh method*), 2699
set_offsets() (*matplotlib.collections.AsteriskPolygonCollection method*), 2406
set_offsets() (*matplotlib.collections.CircleCollection method*), 2427
set_offsets() (*matplotlib.collections.Collection method*), 2451
set_offsets() (*matplotlib.collections.EllipseCollection method*), 2472
set_offsets() (*matplotlib.collections.EventCollection method*), 2496
set_offsets() (*matplotlib.collections.LineCollection method*), 2519
set_offsets() (*matplotlib.collections.PatchCollection method*), 2540
set_offsets() (*matplotlib.collections.PathCollection method*), 2563
set_offsets() (*matplotlib.collections.PolyCollection method*), 2584
set_offsets() (*matplotlib.collections.PolyQuadMesh method*), 2607
set_offsets() (*matplotlib.collections.QuadMesh method*), 2632
set_offsets() (*matplotlib.collections.RegularPolyCollection method*), 2654
set_offsets() (*matplotlib.collections.StarPolygonCollection method*), 2676
set_offsets() (*matplotlib.collections.TriMesh method*), 2699
set_offsets() (*matplotlib.quiver.Barbs method*), 3605
set_one_half() (*matplotlib.ticker.LogitFormatter method*), 3703
set_orientation() (*matplotlib.collections.EventCollection method*), 2496
set_over() (*matplotlib.colors.Colormap method*), 2733
set_pad() (*matplotlib.axis.Tick method*), 2252
set_pad() (*mpl_toolkits.axisartist.axis_artist.AxisLabel method*), 4012
set_pane_color() (*mpl_toolkits.mplot3d.axis3d.Axis method*), 3878
set_params() (*matplotlib.ticker.AsinhLocator method*), 3692
set_params() (*matplotlib.ticker.FixedLocator method*), 3695
set_params() (*matplotlib.ticker.IndexLocator method*), 3697
set_params() (*matplotlib.ticker.LinearLocator method*), 3697
set_params() (*matplotlib.ticker.Locator method*), 3698
set_params() (*matplotlib.ticker.LogitLocator method*), 3704
set_params() (*matplotlib.ticker.LogLocator method*), 3702
set_params() (*matplotlib.ticker.MaxNLocator method*), 3706

set_params() (*matplotlib.ticker.MultipleLocator method*), 3707
set_params() (*matplotlib.ticker.SymmetricalLogLocator method*), 3714
set_params() (*mpl_toolkits.axisartist.angle_helper.LocatorBase method*), 3994
set_parse_math() (*matplotlib.text.Text method*), 3675
set_patch_arc() (*matplotlib.spines.Spine method*), 3641
set_patch_circle() (*matplotlib.spines.Spine method*), 3641
set_patch_line() (*matplotlib.spines.Spine method*), 3641
set_patchA() (*matplotlib.patches.FancyArrowPatch method*), 3151
set_patchB() (*matplotlib.patches.FancyArrowPatch method*), 3151
set_path() (*matplotlib.patches.PathPatch method*), 3172
set_path_effects() (*matplotlib.artist.Artist method*), 1888
set_path_effects() (*matplotlib.collections.AsteriskPolygonCollection method*), 2406
set_path_effects() (*matplotlib.collections.CircleCollection method*), 2427
set_path_effects() (*matplotlib.collections.Collection method*), 2451
set_path_effects() (*matplotlib.collections.EllipseCollection method*), 2473
set_path_effects() (*matplotlib.collections.EventCollection method*), 2496
set_path_effects() (*matplotlib.collections.LineCollection method*), 2519
set_path_effects() (*matplotlib.collections.PatchCollection method*), 2540
set_path_effects() (*matplotlib.collections.PathCollection method*), 2563
set_path_effects() (*matplotlib.collections.PolyCollection method*), 2585
set_path_effects() (*matplotlib.collections.PolyQuadMesh method*), 2608
set_path_effects() (*matplotlib.collections.QuadMesh method*), 2632
set_path_effects() (*matplotlib.collections.RegularPolyCollection method*), 2654
set_path_effects() (*matplotlib.collections.StarPolygonCollection method*), 2676
set_path_effects() (*matplotlib.collections.TriMesh method*), 2699
set_paths() (*matplotlib.collections.AsteriskPolygonCollection method*), 2406
set_paths() (*matplotlib.collections.CircleCollection method*), 2427
set_paths() (*matplotlib.collections.Collection method*), 2451
set_paths() (*matplotlib.collections.EllipseCollection method*), 2473
set_paths() (*matplotlib.collections.EventCollection method*), 2496
set_paths() (*matplotlib.collections.LineCollection method*), 2519
set_paths() (*matplotlib.collections.PatchCollection method*), 2540
set_paths() (*matplotlib.collections.PathCollection method*), 2563
set_paths() (*matplotlib.collections.PolyCollection method*), 2585
set_paths() (*matplotlib.collections.PolyQuadMesh method*), 2608
set_paths() (*matplotlib.collections.QuadMesh method*), 2632
set_paths() (*matplotlib.collections.RegularPolyCollection method*), 2654
set_paths() (*matplotlib.collections.StarPolygonCollection method*), 2676
set_paths() (*matplotlib.collections.TriMesh method*), 2699
set_picker() (*matplotlib.artist.Artist method*), 1873
set_picker() (*matplotlib.collections.AsteriskPolygonCollection method*), 2406
set_picker() (*matplotlib.collections.CircleCollection method*), 2428
set_picker() (*matplotlib.collections.Collection method*), 2451
set_picker() (*matplotlib.collections.EllipseCollection method*), 2473
set_picker() (*matplotlib.collections.EventCollection method*), 2496
set_picker() (*matplotlib.collections.LineCollection method*), 2519
set_picker() (*matplotlib.collections.PatchCollection method*), 2541
set_picker() (*matplotlib.collections.PathCollection method*), 2563
set_picker() (*matplotlib.collections.PolyCollection method*), 2585
set_picker() (*matplotlib.collections.PolyQuadMesh method*), 2608
set_picker() (*matplotlib.collections.QuadMesh method*), 2632
set_picker() (*matplotlib.collections.RegularPolyCollection method*), 2654
set_picker() (*matplotlib.collections.StarPolygonCollection method*), 2676
set_picker() (*matplotlib.collections.TriMesh method*), 2699

set_picker() (*matplotlib.lines.Line2D method*), 3029
set_pickradius() (*matplotlib.axis.Axis method*), 2240
set_pickradius() (*matplotlib.collections.AsteriskPolygonCollection method*), 2407
set_pickradius() (*matplotlib.collections.CircleCollection method*), 2428
set_pickradius() (*matplotlib.collections.Collection method*), 2452
set_pickradius() (*matplotlib.collections.EllipseCollection method*), 2473
set_pickradius() (*matplotlib.collections.EventCollection method*), 2497
set_pickradius() (*matplotlib.collections.LineCollection method*), 2520
set_pickradius() (*matplotlib.collections.PatchCollection method*), 2541
set_pickradius() (*matplotlib.collections.PathCollection method*), 2564
set_pickradius() (*matplotlib.collections.PolyCollection method*), 2586
set_pickradius() (*matplotlib.collections.PolyQuadMesh method*), 2609
set_pickradius() (*matplotlib.collections.QuadMesh method*), 2633
set_pickradius() (*matplotlib.collections.RegularPolyCollection method*), 2655
set_pickradius() (*matplotlib.collections.StarPolygonCollection method*), 2677
set_pickradius() (*matplotlib.collections.TriMesh method*), 2700
set_pickradius() (*matplotlib.lines.Line2D method*), 3029
set_points() (*matplotlib.transforms.Bbox method*), 3729
set_position() (*matplotlib.axes.Axes method*), 2195
set_position() (*matplotlib.spines.Spine method*), 3641
set_position() (*matplotlib.text.Text method*), 3675
set_position() (*mpl_toolkits.axes_grid1.axes_divider.Divider method*), 3935
set_position_3d() (*mpl_toolkits.mplot3d.art3d.Text3D method*), 3903
set_positions() (*matplotlib.collections.EventCollection method*), 2497
set_positions() (*matplotlib.patches.FancyArrowPatch method*), 3151
set_powerlimits() (*matplotlib.ticker.ScalarFormatter method*), 3710
set_proj_type() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3868
set_prop_cycle() (*matplotlib.axes.Axes method*), 2117
set_radii() (*matplotlib.patches.Annulus method*), 3100
set_radio_props() (*matplotlib.widgets.RadioButtons method*), 3787
set_radius() (*matplotlib.patches.Circle method*), 3124
set_radius() (*matplotlib.patches.Wedge method*), 3192
set_rasterization_zorder() (*matplotlib.axes.Axes method*), 2206
set_rasterized() (*matplotlib.artist.Artist method*), 1887
set_rasterized() (*matplotlib.collections.AsteriskPolygonCollection method*), 2407
set_rasterized() (*matplotlib.collections.CircleCollection method*), 2428
set_rasterized() (*matplotlib.collections.Collection method*), 2452
set_rasterized() (*matplotlib.collections.EllipseCollection method*), 2473
set_rasterized() (*matplotlib.collections.EventCollection method*), 2497
set_rasterized() (*matplotlib.collections.LineCollection method*), 2520
set_rasterized() (*matplotlib.collections.PatchCollection method*), 2541
set_rasterized() (*matplotlib.collections.PathCollection method*), 2564
set_rasterized() (*matplotlib.collections.PolyCollection method*), 2586
set_rasterized() (*matplotlib.collections.PolyQuadMesh method*), 2609
set_rasterized() (*matplotlib.collections.QuadMesh method*), 2633
set_rasterized() (*matplotlib.collections.RegularPolyCollection method*), 2655
set_rasterized() (*matplotlib.collections.StarPolygonCollection method*), 2677
set_rasterized() (*matplotlib.collections.TriMesh method*), 2700
set_remove_overlapping_locs() (*matplotlib.axis.Axis method*), 2227
set_reproducibility_for_testing() (*in module matplotlib.testing*), 3657
set_rgrids() (*matplotlib.projections.polar.PolarAxes method*), 3542
set_rlabel_position() (*matplotlib.projections.polar.PolarAxes method*), 3543
set_rlim() (*matplotlib.projections.polar.PolarAxes method*), 3543
set_rmax() (*matplotlib.projections.polar.PolarAxes method*), 3543
set_rmin() (*matplotlib.projections.polar.PolarAxes method*), 3543
set_rorigin() (*matplotlib.projections.polar.PolarAxes method*), 3544
set_rotate_label() (*mpl_toolkits.mplot3d.axis3d.Axis method*), 3878
set_rotation() (*matplotlib.text.Text method*), 3675
set_rotation_mode() (*matplotlib.text.Text method*), 3675

set_rscale() (*matplotlib.projections.polar.PolarAxes method*), 3544
set_rticks() (*matplotlib.projections.polar.PolarAxes method*), 3544
set_scale() (*matplotlib.backend_tools.ToolXScale method*), 2299
set_scale() (*matplotlib.backend_tools.ToolYScale method*), 2299
set_scientific() (*matplotlib.ticker.ScalarFormatter method*), 3711
set_segments() (*matplotlib.collections.EventCollection method*), 2497
set_segments() (*matplotlib.collections.LineCollection method*), 2520
set_segments() (*mpl_toolkits.mplot3d.art3d.Line3DCollection method*), 3885
set_semimajor() (*matplotlib.patches.Annulus method*), 3100
set_semiminor() (*matplotlib.patches.Annulus method*), 3100
set_size() (*matplotlib.font_manager.FontProperties method*), 2947
set_size() (*matplotlib.ft2font.FT2Font method*), 2956
set_size() (*matplotlib.text.Text method*), 3675
set_size() (*matplotlib.text.TextPath method*), 3686
set_size_inches() (*matplotlib.figure.Figure method*), 2842
set_sizes() (*matplotlib.collections.AsteriskPolygonCollection method*), 2407
set_sizes() (*matplotlib.collections.CircleCollection method*), 2429
set_sizes() (*matplotlib.collections.PathCollection method*), 2564
set_sizes() (*matplotlib.collections.PolyCollection method*), 2586
set_sizes() (*matplotlib.collections.PolyQuadMesh method*), 2609
set_sizes() (*matplotlib.collections.RegularPolyCollection method*), 2655
set_sizes() (*matplotlib.collections.StarPolygonCollection method*), 2677
set_sizes() (*mpl_toolkits.mplot3d.art3d.Path3DCollection method*), 3893
set_sketch_params() (*matplotlib.artist.Artist method*), 1886
set_sketch_params() (*matplotlib.backend_bases.GraphicsContextBase method*), 2267
set_sketch_params() (*matplotlib.collections.AsteriskPolygonCollection method*), 2407
set_sketch_params() (*matplotlib.collections.CircleCollection method*), 2429
set_sketch_params() (*matplotlib.collections.Collection method*), 2452
set_sketch_params() (*matplotlib.collections.EllipseCollection method*), 2474
set_sketch_params() (*matplotlib.collections.EventCollection method*), 2497
set_sketch_params() (*matplotlib.collections.LineCollection method*), 2520
set_sketch_params() (*matplotlib.collections.PatchCollection method*), 2542
set_sketch_params() (*matplotlib.collections.PathCollection method*), 2564
set_sketch_params() (*matplotlib.collections.PolyCollection method*), 2586
set_sketch_params() (*matplotlib.collections.PolyQuadMesh method*), 2609
set_sketch_params() (*matplotlib.collections.QuadMesh method*), 2633
set_sketch_params() (*matplotlib.collections.RegularPolyCollection method*), 2655
set_sketch_params() (*matplotlib.collections.StarPolygonCollection method*), 2677
set_sketch_params() (*matplotlib.collections.TriMesh method*), 2700
set_slant() (*matplotlib.font_manager.FontProperties method*), 2947
set_slope() (*matplotlib.lines.AxLine method*), 3036
set_snap() (*matplotlib.artist.Artist method*), 1883
set_snap() (*matplotlib.backend_bases.GraphicsContextBase method*), 2267
set_snap() (*matplotlib.collections.AsteriskPolygonCollection method*), 2408
set_snap() (*matplotlib.collections.CircleCollection method*), 2429
set_snap() (*matplotlib.collections.Collection method*), 2453
set_snap() (*matplotlib.collections.EllipseCollection method*), 2474
set_snap() (*matplotlib.collections.EventCollection method*), 2498
set_snap() (*matplotlib.collections.LineCollection method*), 2521
set_snap() (*matplotlib.collections.PatchCollection method*), 2542
set_snap() (*matplotlib.collections.PathCollection method*), 2565
set_snap() (*matplotlib.collections.PolyCollection method*), 2587
set_snap() (*matplotlib.collections.PolyQuadMesh method*), 2610
set_snap() (*matplotlib.collections.QuadMesh method*),

2634
set_snap() (*matplotlib.collections.RegularPolyCollection method*), 2656
set_snap() (*matplotlib.collections.StarPolygonCollection method*), 2678
set_snap() (*matplotlib.collections.TriMesh method*), 2701
set_solid_capstyle() (*matplotlib.lines.Line2D method*), 3030
set_solid_joinstyle() (*matplotlib.lines.Line2D method*), 3030
set_sort_zpos() (*mpl_toolkits.mplot3d.art3d.Line3DCollection method*), 3885
set_sort_zpos() (*mpl_toolkits.mplot3d.art3d.Patch3DCollection method*), 3890
set_sort_zpos() (*mpl_toolkits.mplot3d.art3d.Path3DCollection method*), 3893
set_sort_zpos() (*mpl_toolkits.mplot3d.art3d.Poly3DCollection method*), 3898
set_stretch() (*matplotlib.font_manager.FontProperties method*), 2947
set_stretch() (*matplotlib.text.Text method*), 3676
set_style() (*matplotlib.font_manager.FontProperties method*), 2947
set_style() (*matplotlib.text.Text method*), 3676
set_subplotspec() (*matplotlib.axes.Axes method*), 2194
set_subplotspec() (*mpl_toolkits.axes_grid1.axes_divider.SubplotDivider method*), 3937
set_text() (*matplotlib.ft2font.FT2Font method*), 2956
set_text() (*matplotlib.offsetbox.TextArea method*), 3092
set_text() (*matplotlib.text.Text method*), 3676
set_text_props() (*matplotlib.table.Cell method*), 3649
set_theta1() (*matplotlib.patches.Wedge method*), 3192
set_theta2() (*matplotlib.patches.Wedge method*), 3192
set_theta_direction() (*matplotlib.projections.polar.PolarAxes method*), 3544
set_theta_offset() (*matplotlib.projections.polar.PolarAxes method*), 3544
set_theta_zero_location() (*matplotlib.projections.polar.PolarAxes method*), 3544
set_thetagrids() (*matplotlib.projections.polar.PolarAxes method*), 3544
set_thetalim() (*matplotlib.projections.polar.PolarAxes method*), 3545
set_thetamax() (*matplotlib.projections.polar.PolarAxes method*), 3546
set_thetamin() (*matplotlib.projections.polar.PolarAxes method*), 3546
set_tick_out() (*mpl_toolkits.axisartist.axis_artist.Ticks method*), 4025
set_tick_params() (*matplotlib.axis.Axis method*), 2235
set_ticklabel_direction() (*mpl_toolkits.axisartist.axis_artist.AxisArtist method*), 4008
set_ticklabels() (*matplotlib.axis.Axis method*), 2248
set_ticklabels() (*matplotlib.colorbar.Colorbar method*), 2708
set_ticks() (*matplotlib.axis.Axis method*), 2247
set_ticks() (*matplotlib.colorbar.Colorbar method*), 2709
set_ticks_position() (*matplotlib.axis.XAxis method*), 2242
set_ticks_position() (*matplotlib.axis.YAxis method*), 2245
set_ticks_position() (*mpl_toolkits.mplot3d.axis3d.Axis method*), 3878
set_ticksize() (*mpl_toolkits.axisartist.axis_artist.Ticks method*), 4025
set_tight_layout() (*matplotlib.figure.Figure method*), 2850
set_title() (*matplotlib.axes.Axes method*), 2138
set_title() (*matplotlib.legend.Legend method*), 3002
set_title() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3851
set_top_view() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3868
set_transform() (*matplotlib.artist.Artist method*), 1892
set_transform() (*matplotlib.collections.AsteriskPolygonCollection method*), 2408
set_transform() (*matplotlib.collections.CircleCollection method*), 2430
set_transform() (*matplotlib.collections.Collection method*), 2453
set_transform() (*matplotlib.collections.EllipseCollection method*), 2475
set_transform() (*matplotlib.collections.EventCollection method*), 2498
set_transform() (*matplotlib.collections.LineCollection method*), 2521
set_transform() (*matplotlib.collections.PatchCollection method*), 2542
set_transform() (*matplotlib.collections.PathCollection method*), 2565
set_transform() (*matplotlib.collections.PolyCollection method*), 2587
set_transform() (*matplotlib.collections.PolyQuadMesh method*), 2610
set_transform() (*matplotlib.collections.QuadMesh method*), 2634
set_transform() (*matplotlib.collections.RegularPolyCollection method*), 2656
set_transform() (*matplotlib.collections.StarPolygonCollection method*), 2678
set_transform() (*matplotlib.collections.TriMesh method*), 2701
set_transform() (*matplotlib.lines.Line2D method*), 3030
set_transform() (*matplotlib.offsetbox.AuxTransformBox method*), 3075

set_transform() (*matplotlib.offsetbox.DrawingArea method*), 3078
set_transform() (*matplotlib.offsetbox.TextArea method*), 3092
set_transform() (*matplotlib.table.Cell method*), 3650
set_transform() (*mpl_toolkits.axisartist.grid_finder.GridFinder method*), 4047
set_transform_rotates_text() (*matplotlib.text.Text method*), 3676
set_transOffset() (*matplotlib.collections.AsteriskPolygonCollection method*), 2408
set_transOffset() (*matplotlib.collections.CircleCollection method*), 2430
set_transOffset() (*matplotlib.collections.Collection method*), 2453
set_transOffset() (*matplotlib.collections.EllipseCollection method*), 2475
set_transOffset() (*matplotlib.collections.EventCollection method*), 2498
set_transOffset() (*matplotlib.collections.LineCollection method*), 2521
set_transOffset() (*matplotlib.collections.PatchCollection method*), 2542
set_transOffset() (*matplotlib.collections.PathCollection method*), 2565
set_transOffset() (*matplotlib.collections.PolyCollection method*), 2587
set_transOffset() (*matplotlib.collections.PolyQuadMesh method*), 2610
set_transOffset() (*matplotlib.collections.QuadMesh method*), 2634
set_transOffset() (*matplotlib.collections.RegularPolyCollection method*), 2656
set_transOffset() (*matplotlib.collections.StarPolygonCollection method*), 2678
set_transOffset() (*matplotlib.collections.TriMesh method*), 2701
set_tzinfo() (*matplotlib.dates.DateFormatter method*), 2776
set_tzinfo() (*matplotlib.dates.DateLocator method*), 2777
set_under() (*matplotlib.colors.Colormap method*), 2733
set_unit() (*matplotlib.text.OffsetFrom method*), 3685
set_units() (*matplotlib.axis.Axis method*), 2241
set_url() (*matplotlib.artist.Artist method*), 1895
set_url() (*matplotlib.axis.Tick method*), 2253
set_url() (*matplotlib.backend_bases.GraphicsContextBase method*), 2267
set_url() (*matplotlib.collections.AsteriskPolygonCollection method*), 2408
set_url() (*matplotlib.collections.CircleCollection method*), 2430
set_url() (*matplotlib.collections.Collection method*), 2453
set_url() (*matplotlib.collections.EllipseCollection method*), 2475
set_url() (*matplotlib.collections.EventCollection method*), 2498
set_url() (*matplotlib.collections.LineCollection method*), 2521
set_url() (*matplotlib.collections.PatchCollection method*), 2543
set_url() (*matplotlib.collections.PathCollection method*), 2565
set_url() (*matplotlib.collections.PolyCollection method*), 2587
set_url() (*matplotlib.collections.PolyQuadMesh method*), 2610
set_url() (*matplotlib.collections.QuadMesh method*), 2634
set_url() (*matplotlib.collections.RegularPolyCollection method*), 2656
set_url() (*matplotlib.collections.StarPolygonCollection method*), 2678
set_url() (*matplotlib.collections.TriMesh method*), 2701
set_urls() (*matplotlib.collections.AsteriskPolygonCollection method*), 2409
set_urls() (*matplotlib.collections.CircleCollection method*), 2430
set_urls() (*matplotlib.collections.Collection method*), 2453
set_urls() (*matplotlib.collections.EllipseCollection method*), 2475
set_urls() (*matplotlib.collections.EventCollection method*), 2499
set_urls() (*matplotlib.collections.LineCollection method*), 2521
set_urls() (*matplotlib.collections.PatchCollection method*), 2543
set_urls() (*matplotlib.collections.PathCollection method*), 2566
set_urls() (*matplotlib.collections.PolyCollection method*), 2588
set_urls() (*matplotlib.collections.PolyQuadMesh method*), 2611
set_urls() (*matplotlib.collections.QuadMesh method*), 2634
set_urls() (*matplotlib.collections.RegularPolyCollection method*), 2657
set_urls() (*matplotlib.collections.StarPolygonCollection method*), 2679
set_urls() (*matplotlib.collections.TriMesh method*), 2701
set_useLocale() (*matplotlib.ticker.ScalarFormatter method*), 3711
set_useMathText() (*matplotlib.ticker.EngFormatter method*), 3695

set_useMathText() (*matplotlib.ticker.ScalarFormatter method*), 3711
set_useOffset() (*matplotlib.ticker.ScalarFormatter method*), 3712
set_usetex() (*matplotlib.text.Text method*), 3676
set_usetex() (*matplotlib.ticker.EngFormatter method*), 3695
set_UVC() (*matplotlib.quiver.Barbs method*), 3605
set_UVC() (*matplotlib.quiver.Quiver method*), 3596
set_va() (*matplotlib.text.Text method*), 3676
set_val() (*matplotlib.widgets.RangeSlider method*), 3790
set_val() (*matplotlib.widgets.Slider method*), 3795
set_val() (*matplotlib.widgets.TextBox method*), 3800
set_variant() (*matplotlib.font_manager.FontProperties method*), 2948
set_variant() (*matplotlib.text.Text method*), 3676
set_vertical() (*mpl_toolkits.axes_grid1.axes_divider.Divider method*), 3935
set_verticalalignment() (*matplotlib.text.Text method*), 3676
set_vertices_and_codes() (*matplotlib.hatch.HorizontalHatch method*), 2966
set_vertices_and_codes() (*matplotlib.hatch.NorthEastHatch method*), 2966
set_vertices_and_codes() (*matplotlib.hatch.Shapes method*), 2966
set_vertices_and_codes() (*matplotlib.hatch.SouthEastHatch method*), 2966
set_vertices_and_codes() (*matplotlib.hatch.VerticalHatch method*), 2967
set_verts() (*matplotlib.collections.EventCollection method*), 2499
set_verts() (*matplotlib.collections.LineCollection method*), 2522
set_verts() (*matplotlib.collections.PolyCollection method*), 2588
set_verts() (*matplotlib.collections.PolyQuadMesh method*), 2611
set_verts() (*mpl_toolkits.mplot3d.art3d.Poly3DCollection method*), 3898
set_verts_and_codes() (*matplotlib.collections.PolyCollection method*), 2588
set_verts_and_codes() (*matplotlib.collections.PolyQuadMesh method*), 2611
set_verts_and_codes() (*mpl_toolkits.mplot3d.art3d.Poly3DCollection method*), 3898
set_view_interval() (*matplotlib.axis.Axis method*), 2238
set_viewlim_mode() (*mpl_toolkits.axes_grid1.parasite_axes.ParasiteAxesBase method*), 3987
set_visible() (*matplotlib.artist.Artist method*), 1884
set_visible() (*matplotlib.collections.AsteriskPolygonCollection method*), 2409
set_visible() (*matplotlib.collections.CircleCollection method*), 2430
set_visible() (*matplotlib.collections.Collection method*), 2454
set_visible() (*matplotlib.collections.EllipseCollection method*), 2475
set_visible() (*matplotlib.collections.EventCollection method*), 2499
set_visible() (*matplotlib.collections.LineCollection method*), 2522
set_visible() (*matplotlib.collections.PatchCollection method*), 2543
set_visible() (*matplotlib.collections.PathCollection method*), 2566
set_visible() (*matplotlib.collections.PolyCollection method*), 2588
set_visible() (*matplotlib.collections.PolyQuadMesh method*), 2611
set_visible() (*matplotlib.collections.QuadMesh method*), 2635
set_visible() (*matplotlib.collections.RegularPolyCollection method*), 2657
set_visible() (*matplotlib.collections.StarPolygonCollection method*), 2679
set_visible() (*matplotlib.collections.TriMesh method*), 2702
set_visible() (*matplotlib.widgets.ToolHandles method*), 3800
set_visible() (*matplotlib.widgets.ToolLineHandles method*), 3802
set_visible() (*mpl_toolkits.axes_grid1.mpl_axes.SimpleAxisArtist method*), 3985
set_weight() (*matplotlib.font_manager.FontProperties method*), 2948
set_weight() (*matplotlib.text.Text method*), 3677
set_which() (*mpl_toolkits.axisartist.axis_artist.GridlinesCollection method*), 4015
set_width() (*matplotlib.offsetbox.OffsetBox method*), 3084
set_width() (*matplotlib.patches.Annulus method*), 3101
set_width() (*matplotlib.patches.Ellipse method*), 3139
set_width() (*matplotlib.patches.FancyBboxPatch method*), 3157
set_width() (*matplotlib.patches.Rectangle method*), 3183
set_width() (*matplotlib.patches.Wedge method*), 3192
set_width_ratios() (*matplotlib.gridspec.GridSpecBase method*), 2963
set_widths() (*matplotlib.collections.EllipseCollection method*), 2475
set_window_title() (*matplotlib.backend_bases.FigureManagerBase method*), 2263
set_window_title() (*matplotlib.backends.backend_webagg_core.FigureManagerWebAgg method*), 2360
set_wrap() (*matplotlib.text.Text method*), 3677

set_x() (*matplotlib.patches.FancyBboxPatch method*), 3157
set_x() (*matplotlib.patches.Rectangle method*), 3183
set_x() (*matplotlib.text.Text method*), 3677
set_xbound() (*matplotlib.axes.Axes method*), 2129
set_xbound() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3848
set_xdata() (*matplotlib.lines.Line2D method*), 3030
set_xlabel() (*matplotlib.axes.Axes method*), 2131
set_xlim() (*matplotlib.axes.Axes method*), 2120
set_xlim() (*matplotlib.projections.geo.GeoAxes method*), 3570
set_xlim() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3841
set_xmargin() (*matplotlib.axes.Axes method*), 2158
set_xscale() (*matplotlib.axes.Axes method*), 2153
set_xscale() (*matplotlib.projections.geo.GeoAxes method*), 3570
set_xscale() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3852
set_xticklabels() (*matplotlib.axes.Axes method*), 2171
set_xticks() (*matplotlib.axes.Axes method*), 2168
set_xy() (*matplotlib.patches.Polygon method*), 3178
set_xy() (*matplotlib.patches.Rectangle method*), 3183
set_xy1() (*matplotlib.lines.AxLine method*), 3037
set_xy2() (*matplotlib.lines.AxLine method*), 3037
set_y() (*matplotlib.patches.FancyBboxPatch method*), 3157
set_y() (*matplotlib.patches.Rectangle method*), 3183
set_y() (*matplotlib.text.Text method*), 3677
set_ybound() (*matplotlib.axes.Axes method*), 2129
set_ybound() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3849
set_ydata() (*matplotlib.lines.Line2D method*), 3031
set_ylabel() (*matplotlib.axes.Axes method*), 2134
set_ylim() (*matplotlib.axes.Axes method*), 2124
set_ylim() (*matplotlib.projections.geo.GeoAxes method*), 3570
set_ylim() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3843
set_ymargin() (*matplotlib.axes.Axes method*), 2158
set_yscale() (*matplotlib.axes.Axes method*), 2154
set_yscale() (*matplotlib.projections.geo.GeoAxes method*), 3570
set_yscale() (*matplotlib.projections.polar.PolarAxes method*), 3546
set_yscale() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3853
set_yticklabels() (*matplotlib.axes.Axes method*), 2176
set_yticks() (*matplotlib.axes.Axes method*), 2174
set_z() (*mpl_toolkits.mplot3d.art3d.Text3D method*), 3904
set_zbound() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3850
set_zlabel() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3851
set_zlim() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3845
set_zlim3d() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3870
set_zmargin() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3854
set_zoom() (*matplotlib.offsetbox.OffsetImage method*), 3086
set_zorder() (*matplotlib.artist.Artist method*), 1885
set_zorder() (*matplotlib.collections.AsteriskPolygonCollection method*), 2409
set_zorder() (*matplotlib.collections.CircleCollection method*), 2430
set_zorder() (*matplotlib.collections.Collection method*), 2454
set_zorder() (*matplotlib.collections.EllipseCollection method*), 2476
set_zorder() (*matplotlib.collections.EventCollection method*), 2499
set_zorder() (*matplotlib.collections.LineCollection method*), 2522
set_zorder() (*matplotlib.collections.PatchCollection method*), 2543
set_zorder() (*matplotlib.collections.PathCollection method*), 2566
set_zorder() (*matplotlib.collections.PolyCollection method*), 2588
set_zorder() (*matplotlib.collections.PolyQuadMesh method*), 2611
set_zorder() (*matplotlib.collections.QuadMesh method*), 2635
set_zorder() (*matplotlib.collections.RegularPolyCollection method*), 2657
set_zorder() (*matplotlib.collections.StarPolygonCollection method*), 2679
set_zorder() (*matplotlib.collections.TriMesh method*), 2702
set_zscale() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3853
set_zsort() (*mpl_toolkits.mplot3d.art3d.Poly3DCollection method*), 3899
set_zticklabels() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3861
set_zticks() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3859
setcolor_nonstroke (*matplotlib.backends.backend_pdf.Op attribute*), 2325
setcolor_stroke (*matplotlib.backends.backend_pdf.Op attribute*), 2325
setcolorspace_nonstroke (*matplotlib.backends.backend_pdf.Op attribute*), 2325
setcolorspace_stroke (*matplotlib.backends.backend_pdf.Op attribute*), 2325
setdash (*matplotlib.backends.backend_pdf.Op attribute*), 2325
setgray_nonstroke (*matplotlib.backends.backend_pdf.Op attribute*), 2325
setgray_stroke (*matplotlib.backends.backend_pdf.Op*

*attribute*), 2325
setgstate (*matplotlib.backends.backend_pdf.Op attribute*), 2325
setlinecap (*matplotlib.backends.backend_pdf.Op attribute*), 2325
setlinejoin (*matplotlib.backends.backend_pdf.Op attribute*), 2325
setlinewidth (*matplotlib.backends.backend_pdf.Op attribute*), 2325
setp() (*in module matplotlib.artist*), 1899
setp() (*in module matplotlib.pyplot*), 3523
setrgb_nonstroke (*matplotlib.backends.backend_pdf.Op attribute*), 2325
setrgb_stroke (*matplotlib.backends.backend_pdf.Op attribute*), 2325
setup() (*in module matplotlib.testing*), 3657
setup() (*matplotlib.animation.AbstractMovieWriter method*), 1860
setup() (*matplotlib.animation.FileMovieWriter method*), 1865
setup() (*matplotlib.animation.HTMLWriter method*), 1848
setup() (*matplotlib.animation.MovieWriter method*), 1863
setup() (*matplotlib.animation.PillowWriter method*), 1845
shade() (*matplotlib.colors.LightSource method*), 2744
shade_normals() (*matplotlib.colors.LightSource method*), 2745
shade_rgb() (*matplotlib.colors.LightSource method*), 2746
shading (*matplotlib.backends.backend_pdf.Op attribute*), 2326
Shadow (*class in matplotlib.patches*), 3187
Shapes (*class in matplotlib.hatch*), 2966
shareview() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3864
sharex() (*matplotlib.axes.Axes method*), 2191
sharey() (*matplotlib.axes.Axes method*), 2191
sharez() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3864
should_simplify (*matplotlib.path.Path property*), 3201
show (*matplotlib.backends.backend_pdf.Op attribute*), 2326
show() (*in module matplotlib.pyplot*), 3497
show() (*matplotlib.backend_bases.FigureManagerBase method*), 2263
show() (*matplotlib.backends.backend_nbagg.FigureManagerNbAgg method*), 2322
show() (*matplotlib.backends.backend_webagg_core.FigureCanvasWebAggCore method*), 2359
show() (*matplotlib.backends.backend_webagg_core.FigureManagerWebAgg method*), 2360
show() (*matplotlib.figure.Figure method*), 2862
ShowBase (*class in matplotlib.backend_bases*), 2279
showkern (*matplotlib.backends.backend_pdf.Op attribute*), 2326
shrunk() (*matplotlib.transforms.BboxBase method*), 3734
shrunk_to_aspect() (*matplotlib.transforms.BboxBase method*), 3734
silent_list (*class in matplotlib.cbook*), 2381
silverman_factor() (*matplotlib.mlab.GaussianKDE method*), 3048
simple_linear_interpolation() (*in module matplotlib.cbook*), 2381
SimpleAxisArtist (*class in mpl_toolkits.axes_grid1.mpl_axes*), 3983
SimpleChainedObjects (*class in mpl_toolkits.axes_grid1.mpl_axes*), 3985
SimpleLineShadow (*class in matplotlib.patheffects*), 3206
SimplePatchShadow (*class in matplotlib.patheffects*), 3207
simplify_threshold (*matplotlib.path.Path property*), 3201
single_shot (*matplotlib.backend_bases.TimerBase property*), 2280
size (*matplotlib.dviread.DviFont attribute*), 2789
size (*matplotlib.hatch.LargeCircles attribute*), 2966
size (*matplotlib.hatch.SmallCircles attribute*), 2966
size (*matplotlib.hatch.SmallFilledCircles attribute*), 2966
size (*matplotlib.hatch.Stars attribute*), 2967
size (*matplotlib.transforms.BboxBase property*), 3734
skew() (*matplotlib.transforms.Affine2D method*), 3721
skew_deg() (*matplotlib.transforms.Affine2D method*), 3721
Slider (*class in matplotlib.widgets*), 3793
SliderBase (*class in matplotlib.widgets*), 3795
SmallCircles (*class in matplotlib.hatch*), 2966
SmallFilledCircles (*class in matplotlib.hatch*), 2966
SouthEastHatch (*class in matplotlib.hatch*), 2966
SpanSelector (*class in matplotlib.widgets*), 3796
specgram() (*in module matplotlib.mlab*), 3061
specgram() (*in module matplotlib.pyplot*), 3335
specgram() (*matplotlib.axes.Axes method*), 1996
Spine (*class in matplotlib.spines*), 3637
Spines (*class in matplotlib.spines*), 3642
SpinesProxy (*class in matplotlib.spines*), 3642
split_bezier_intersecting_with_closedpath() (*in module matplotlib.bezier*), 2367
split_de_casteljau() (*in module matplotlib.bezier*), 2368
split_path_inout() (*in module matplotlib.bezier*), 2368
splitx() (*matplotlib.transforms.BboxBase method*), 3734
splity() (*matplotlib.transforms.BboxBase method*), 3735
spy() (*in module matplotlib.pyplot*), 3391
spy() (*matplotlib.axes.Axes method*), 2060
Stack (*class in matplotlib.cbook*), 2373
stackplot() (*in module matplotlib.pyplot*), 3294
stackplot() (*matplotlib.axes.Axes method*), 1956
start_event_loop() (*in module matplotlib.pyplot*), 3362
stairs() (*matplotlib.axes.Axes method*), 2028
stale (*matplotlib.artist.Artist property*), 1897
stale (*matplotlib.axes.Axes property*), 2196
stale (*matplotlib.collections.AsteriskPolygonCollection property*), 2409
stale (*matplotlib.collections.CircleCollection property*), 2431
stale (*matplotlib.collections.Collection property*), 2454
stale (*matplotlib.collections.EllipseCollection property*), 2476
stale (*matplotlib.collections.EventCollection property*), 2499
stale (*matplotlib.collections.LineCollection property*), 2522

stale (*matplotlib.collections.PatchCollection property*), 2543
stale (*matplotlib.collections.PathCollection property*), 2566
stale (*matplotlib.collections.PolyCollection property*), 2589
stale (*matplotlib.collections.PolyQuadMesh property*), 2612
stale (*matplotlib.collections.QuadMesh property*), 2635
stale (*matplotlib.collections.RegularPolyCollection property*), 2657
stale (*matplotlib.collections.StarPolygonCollection property*), 2679
stale (*matplotlib.collections.TriMesh property*), 2702
StarPolygonCollection (*class in matplotlib.collections*), 2658
Stars (*class in matplotlib.hatch*), 2967
start() (*matplotlib.backend_bases.TimerBase method*), 2280
start() (*matplotlib.backends.backend_svg.XMLWriter method*), 2356
start() (*matplotlib.backends.backend_webagg.WebAggApplication class method*), 2364
start_event_loop() (*matplotlib.backend_bases.FigureCanvasBase method*), 2261
start_filter() (*matplotlib.backend_bases.RendererBase method*), 2278
start_filter() (*matplotlib.backends.backend_agg.RendererAgg method*), 2314
start_main_loop() (*matplotlib.backend_bases.FigureManagerBase class method*), 2263
start_pan() (*matplotlib.axes.Axes method*), 2200
start_pan() (*matplotlib.projections.geo.GeoAxes method*), 3571
start_pan() (*matplotlib.projections.polar.PolarAxes method*), 3546
start_rasterizing() (*matplotlib.backend_bases.RendererBase method*), 2278
start_rasterizing() (*matplotlib.backends.backend_mixed.MixedModeRenderer method*), 2303
started (*matplotlib.backends.backend_webagg.WebAggApplication attribute*), 2364
stem() (*in module matplotlib.pyplot*), 3287
stem() (*matplotlib.axes.Axes method*), 1949
stem() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3834
stem3D() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3870
StemContainer (*class in matplotlib.container*), 2755
step() (*in module matplotlib.pyplot*), 3268
step() (*matplotlib.axes.Axes method*), 1930
StepPatch (*class in matplotlib.patches*), 3173
sticky_edges (*matplotlib.artist.Artist property*), 1896
sticky_edges (*matplotlib.collections.AsteriskPolygonCollection property*), 2409
sticky_edges (*matplotlib.collections.CircleCollection property*), 2431
sticky_edges (*matplotlib.collections.Collection property*), 2454
sticky_edges (*matplotlib.collections.EllipseCollection property*), 2476
sticky_edges (*matplotlib.collections.EventCollection property*), 2499
sticky_edges (*matplotlib.collections.LineCollection property*), 2522
sticky_edges (*matplotlib.collections.PatchCollection property*), 2543
sticky_edges (*matplotlib.collections.PathCollection property*), 2566
sticky_edges (*matplotlib.collections.PolyCollection property*), 2589
sticky_edges (*matplotlib.collections.PolyQuadMesh property*), 2612
sticky_edges (*matplotlib.collections.QuadMesh property*), 2635
sticky_edges (*matplotlib.collections.RegularPolyCollection property*), 2657
sticky_edges (*matplotlib.collections.StarPolygonCollection property*), 2679
sticky_edges (*matplotlib.collections.TriMesh property*), 2702
STOP (*matplotlib.path.Path attribute*), 3195
stop() (*matplotlib.backend_bases.TimerBase method*), 2280
stop_event_loop() (*matplotlib.backend_bases.FigureCanvasBase method*), 2261
stop_filter() (*matplotlib.backend_bases.RendererBase method*), 2278
stop_filter() (*matplotlib.backends.backend_agg.RendererAgg method*), 2314
stop_rasterizing() (*matplotlib.backend_bases.RendererBase method*), 2278
stop_rasterizing() (*matplotlib.backends.backend_mixed.MixedModeRenderer method*), 2304
stop_typing() (*matplotlib.widgets.TextBox method*), 3800
StrCategoryConverter (*class in matplotlib.category*), 2368
StrCategoryFormatter (*class in matplotlib.category*), 2369
StrCategoryLocator (*class in matplotlib.category*), 2369
Stream (*class in matplotlib.backends.backend_pdf*), 2334
streamplot() (*in module matplotlib.pyplot*), 3446
streamplot() (*matplotlib.axes.Axes method*), 2105
string_width_height() (*matplotlib._afm.AFM method*), 1831
strip_math() (*in module matplotlib.cbook*), 2382
StrMethodFormatter (*class in matplotlib.ticker*), 3713
Stroke (*class in matplotlib.patheffects*), 3207
stroke (*matplotlib.backends.backend_pdf.Op attribute*), 2326

stroke() (*matplotlib.backends.backend_pdf.GraphicsContextPdf method*), 2324
style_flags (*matplotlib.ft2font.FT2Font attribute*), 2956
style_name (*matplotlib.ft2font.FT2Font attribute*), 2956
SubFigure (*class in matplotlib.figure*), 2864
subfigures() (*matplotlib.figure.Figure method*), 2813
subfigures() (*matplotlib.figure.FigureBase method*), 2929
subfigures() (*matplotlib.figure.SubFigure method*), 2880
subgridspec() (*matplotlib.gridspec.SubplotSpec method*), 2961
Subplot (*in module mpl_toolkits.axisartist.axislines*), 4041
subplot() (*in module matplotlib.pyplot*), 3225
subplot2grid() (*in module matplotlib.pyplot*), 3230
subplot_mosaic() (*in module matplotlib.pyplot*), 3231
subplot_mosaic() (*matplotlib.figure.Figure method*), 2808
subplot_mosaic() (*matplotlib.figure.FigureBase method*), 2930
subplot_mosaic() (*matplotlib.figure.SubFigure method*), 2876
subplot_tool() (*in module matplotlib.pyplot*), 3475
SubplotDivider (*class in mpl_toolkits.axes_grid1.axes_divider*), 3937
SubplotHost (*in module mpl_toolkits.axes_grid1.parasite_axes*), 3988
SubplotParams (*class in matplotlib.gridspec*), 2965
subplots() (*in module matplotlib.pyplot*), 3234
subplots() (*matplotlib.figure.Figure method*), 2806
subplots() (*matplotlib.figure.FigureBase method*), 2932
subplots() (*matplotlib.figure.SubFigure method*), 2873
subplots() (*matplotlib.gridspec.GridSpecBase method*), 2963
subplots_adjust() (*in module matplotlib.pyplot*), 3474
subplots_adjust() (*matplotlib.figure.Figure method*), 2845
subplots_adjust() (*matplotlib.figure.FigureBase method*), 2934
SubplotSpec (*class in matplotlib.gridspec*), 2960
SubplotTool (*class in matplotlib.widgets*), 3798
SubplotZero (*in module mpl_toolkits.axisartist.axislines*), 4041
subprocess_run_for_testing() (*in module matplotlib.testing*), 3658
subprocess_run_helper() (*in module matplotlib.testing*), 3659
Substitution (*class in matplotlib._docstring*), 2787
supported_formats (*matplotlib.animation.FFMpegFileWriter attribute*), 1854
supported_formats (*matplotlib.animation.HTMLWriter attribute*), 1848
supported_formats (*matplotlib.animation.ImageMagickFileWriter attribute*), 1855
supported_formats (*matplotlib.animation.MovieWriter attribute*), 1863
supports_binary (*matplotlib.backends.backend_webagg.WebAggApplication.WebSocket attribute*), 2364
supports_blit (*matplotlib.backend_bases.FigureCanvasBase attribute*), 2261
supports_blit (*matplotlib.backends.backend_webagg_core.FigureCanvasWebAggCore attribute*), 2360
suppress_matplotlib_deprecation_warning() (*in module matplotlib._api.deprecation*), 3808
suptitle() (*in module matplotlib.pyplot*), 3469
suptitle() (*matplotlib.figure.Figure method*), 2832
suptitle() (*matplotlib.figure.FigureBase method*), 2935
suptitle() (*matplotlib.figure.SubFigure method*), 2893
supxlabel() (*matplotlib.figure.Figure method*), 2834
supxlabel() (*matplotlib.figure.FigureBase method*), 2936
supxlabel() (*matplotlib.figure.SubFigure method*), 2894
supylabel() (*matplotlib.figure.Figure method*), 2836
supylabel() (*matplotlib.figure.FigureBase method*), 2937
supylabel() (*matplotlib.figure.SubFigure method*), 2895
switch_backend() (*in module matplotlib.pyplot*), 3515
switch_backends() (*matplotlib.backend_bases.FigureCanvasBase method*), 2261
switch_orientation() (*matplotlib.collections.EventCollection method*), 2500
symbol (*matplotlib.ticker.PercentFormatter property*), 3709
SymLogNorm (*class in matplotlib.colors*), 2728
SymmetricalLogLocator (*class in matplotlib.ticker*), 3713
SymmetricalLogScale (*class in matplotlib.scale*), 3627
SymmetricalLogTransform (*class in matplotlib.scale*), 3628

## T

Table (*class in matplotlib.table*), 3650
table() (*in module matplotlib.pyplot*), 3416
table() (*in module matplotlib.table*), 3655
table() (*matplotlib.axes.Axes method*), 2082
TABLEAU_COLORS (*in module matplotlib.colors*), 2753
tcolors (*matplotlib.contour.ContourSet property*), 2764
texcache (*matplotlib.texmanager.TexManager property*), 3689
TexManager (*class in matplotlib.texmanager*), 3688
texname (*matplotlib.dviread.DviFont attribute*), 2789
texname (*matplotlib.dviread.PsFont attribute*), 2790
Text (*class in matplotlib.text*), 3662
text (*matplotlib.widgets.TextBox property*), 3800
text() (*in module matplotlib.pyplot*), 3411
text() (*matplotlib.axes.Axes method*), 2078
text() (*matplotlib.figure.Figure method*), 2829
text() (*matplotlib.figure.FigureBase method*), 2938
text() (*matplotlib.figure.SubFigure method*), 2891
text() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3835
text2D() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3836
Text3D (*class in mpl_toolkits.mplot3d.art3d*), 3899
text3D() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3871

text_2d_to_3d() (*in module mpl_toolkits.mplot3d.art3d*), 3907
TextArea (*class in matplotlib.offsetbox*), 3089
TextBox (*class in matplotlib.widgets*), 3798
textmatrix (*matplotlib.backends.backend_pdf.Op attribute*), 2326
TextPath (*class in matplotlib.text*), 3685
textpos (*matplotlib.backends.backend_pdf.Op attribute*), 2326
TextToPath (*class in matplotlib.text*), 3686
Tfm (*class in matplotlib.dviread*), 2790
ThetaAxis (*class in matplotlib.projections.polar*), 3552
ThetaFormatter (*class in matplotlib.projections.polar*), 3554
thetagrids() (*in module matplotlib.pyplot*), 3456
ThetaLocator (*class in matplotlib.projections.polar*), 3554
ThetaTick (*class in matplotlib.projections.polar*), 3555
thumbnail() (*in module matplotlib.image*), 2984
Tick (*class in matplotlib.axis*), 2250
tick_bottom() (*matplotlib.axis.XAxis method*), 2243
tick_left() (*matplotlib.axis.YAxis method*), 2245
tick_params() (*in module matplotlib.pyplot*), 3458
tick_params() (*matplotlib.axes.Axes method*), 2180
tick_params() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3859
tick_right() (*matplotlib.axis.YAxis method*), 2246
tick_top() (*matplotlib.axis.XAxis method*), 2244
tick_values() (*matplotlib.category.StrCategoryLocator method*), 2370
tick_values() (*matplotlib.dates.AutoDateLocator method*), 2774
tick_values() (*matplotlib.dates.MicrosecondLocator method*), 2778
tick_values() (*matplotlib.dates.RRuleLocator method*), 2780
tick_values() (*matplotlib.ticker.AsinhLocator method*), 3693
tick_values() (*matplotlib.ticker.AutoMinorLocator method*), 3693
tick_values() (*matplotlib.ticker.FixedLocator method*), 3695
tick_values() (*matplotlib.ticker.IndexLocator method*), 3697
tick_values() (*matplotlib.ticker.LinearLocator method*), 3698
tick_values() (*matplotlib.ticker.Locator method*), 3698
tick_values() (*matplotlib.ticker.LogitLocator method*), 3704
tick_values() (*matplotlib.ticker.LogLocator method*), 3702
tick_values() (*matplotlib.ticker.MaxNLocator method*), 3706
tick_values() (*matplotlib.ticker.MultipleLocator method*), 3707
tick_values() (*matplotlib.ticker.NullLocator method*), 3708
tick_values() (*matplotlib.ticker.SymmetricalLogLocator method*), 3714
TickedStroke (*class in matplotlib.patheffects*), 3207
Ticker (*class in matplotlib.axis*), 2221
TickHelper (*class in matplotlib.ticker*), 3714
ticklabel_format() (*in module matplotlib.pyplot*), 3460
ticklabel_format() (*matplotlib.axes.Axes method*), 2179
TickLabels (*class in mpl_toolkits.axisartist.axis_artist*), 4019
Ticks (*class in mpl_toolkits.axisartist.axis_artist*), 4022
tight_layout() (*in module matplotlib.pyplot*), 3476
tight_layout() (*matplotlib.figure.Figure method*), 2848
tight_layout() (*matplotlib.gridspec.GridSpec method*), 2958
TightLayoutEngine (*class in matplotlib.layout_engine*), 2988
TimedAnimation (*class in matplotlib.animation*), 1856
TimerAsyncio (*class in matplotlib.backends.backend_webagg_core*), 2361
TimerBase (*class in matplotlib.backend_bases*), 2279
TimerTornado (*class in matplotlib.backends.backend_webagg_core*), 2361
title() (*in module matplotlib.pyplot*), 3471
tlinewidths (*matplotlib.contour.ContourSet property*), 2764
to_filehandle() (*in module matplotlib.cbook*), 2382
to_hex() (*in module matplotlib.colors*), 2749
to_html5_video() (*matplotlib.animation.Animation method*), 1836
to_jshtml() (*matplotlib.animation.Animation method*), 1836
to_polygons() (*matplotlib.path.Path method*), 3201
to_rgb() (*in module matplotlib.colors*), 2750
to_rgba() (*in module matplotlib.colors*), 2750
to_rgba() (*matplotlib.cm.ScalarMappable method*), 2388
to_rgba() (*matplotlib.collections.AsteriskPolygonCollection method*), 2410
to_rgba() (*matplotlib.collections.CircleCollection method*), 2431
to_rgba() (*matplotlib.collections.Collection method*), 2455
to_rgba() (*matplotlib.collections.EllipseCollection method*), 2476
to_rgba() (*matplotlib.collections.EventCollection method*), 2500
to_rgba() (*matplotlib.collections.LineCollection method*), 2523
to_rgba() (*matplotlib.collections.PatchCollection method*), 2544
to_rgba() (*matplotlib.collections.PathCollection method*), 2567
to_rgba() (*matplotlib.collections.PolyCollection method*), 2589
to_rgba() (*matplotlib.collections.PolyQuadMesh method*), 2612
to_rgba() (*matplotlib.collections.QuadMesh method*), 2636
to_rgba() (*matplotlib.collections.RegularPolyCollection method*), 2658
to_rgba() (*matplotlib.collections.StarPolygonCollection method*), 2680
to_rgba() (*matplotlib.collections.TriMesh method*), 2703
to_rgba_array() (*in module matplotlib.colors*), 2751

to_values() (*matplotlib.transforms.Affine2DBase method*), 3722
toggle() (*mpl_toolkits.axes_grid1.mpl_axes.SimpleAxisArtist method*), 3985
toggle() (*mpl_toolkits.axisartist.axis_artist.AxisArtist method*), 4008
toggle_axisline() (*mpl_toolkits.axisartist.axislines.Axes method*), 4034
toggle_label() (*mpl_toolkits.axes_grid1.axes_grid.CbarAxesBase method*), 3940
toggle_toolitem() (*matplotlib.backend_bases.ToolContainerBase method*), 2282
toggled (*matplotlib.backend_tools.ToolToggleBase property*), 2298
too_close() (*matplotlib.contour.ContourLabeler method*), 2759
ToolBack (*class in matplotlib.backend_tools*), 2290
ToolbarCls (*matplotlib.backends.backend_nbagg.FigureManagerNbAgg attribute*), 2321
ToolbarCls (*matplotlib.backends.backend_webagg_core.FigureManagerWebAgg attribute*), 2360
ToolBase (*class in matplotlib.backend_tools*), 2290
ToolContainerBase (*class in matplotlib.backend_bases*), 2280
ToolCopyToClipboardBase (*class in matplotlib.backend_tools*), 2291
ToolCursorPosition (*class in matplotlib.backend_tools*), 2292
ToolEvent (*class in matplotlib.backend_managers*), 2284
ToolForward (*class in matplotlib.backend_tools*), 2292
ToolFullScreen (*class in matplotlib.backend_tools*), 2293
ToolGrid (*class in matplotlib.backend_tools*), 2293
ToolHandles (*class in matplotlib.widgets*), 3800
ToolHelpBase (*class in matplotlib.backend_tools*), 2294
ToolHome (*class in matplotlib.backend_tools*), 2294
toolitems (*matplotlib.backend_bases.NavigationToolbar2 attribute*), 2272
toolitems (*matplotlib.backends.backend_nbagg.NavigationIPy attribute*), 2322
toolitems (*matplotlib.backends.backend_webagg_core.NavigationToolbar2WebAgg attribute*), 2361
ToolLineHandles (*class in matplotlib.widgets*), 3800
ToolManager (*class in matplotlib.backend_managers*), 2284
toolmanager (*matplotlib.backend_tools.ToolBase property*), 2291
toolmanager_connect() (*matplotlib.backend_managers.ToolManager method*), 2286
toolmanager_disconnect() (*matplotlib.backend_managers.ToolManager method*), 2287
ToolManagerMessageEvent (*class in matplotlib.backend_managers*), 2288
ToolMinorGrid (*class in matplotlib.backend_tools*), 2295
ToolPan (*class in matplotlib.backend_tools*), 2295
ToolQuit (*class in matplotlib.backend_tools*), 2296
ToolQuitAll (*class in matplotlib.backend_tools*), 2296
tools (*matplotlib.backend_managers.ToolManager property*), 2287
ToolSetCursor (*class in matplotlib.backend_tools*), 2297
ToolToggleBase (*class in matplotlib.backend_tools*), 2297
ToolTriggerEvent (*class in matplotlib.backend_managers*), 2288
ToolViewsPositions (*class in matplotlib.backend_tools*), 2298
ToolXScale (*class in matplotlib.backend_tools*), 2299
ToolYScale (*class in matplotlib.backend_tools*), 2299
ToolZoom (*class in matplotlib.backend_tools*), 2299
tostring_argb() (*matplotlib.backends.backend_agg.FigureCanvasAgg method*), 2311
tostring_argb() (*matplotlib.backends.backend_agg.RendererAgg method*), 2315
tostring_rgb() (*matplotlib.backends.backend_agg.FigureCanvasAgg method*), 2311
tostring_rgb() (*matplotlib.backends.backend_agg.RendererAgg method*), 2315
Transform (*class in matplotlib.transforms*), 3745
transform() (*in module mpl_toolkits.mplot3d.proj3d*), 3909
transform() (*matplotlib._type1font.Type1Font method*), 3767
transform() (*matplotlib.transforms.AffineBase method*), 3723
transform() (*matplotlib.transforms.IdentityTransform method*), 3742
transform() (*matplotlib.transforms.Transform method*), 3747
transform_affine() (*matplotlib.transforms.Affine2DBase method*), 3722
transform_affine() (*matplotlib.transforms.AffineBase method*), 3724
transform_affine() (*matplotlib.transforms.CompositeGenericTransform method*), 3740
transform_affine() (*matplotlib.transforms.IdentityTransform method*), 3742
transform_affine() (*matplotlib.transforms.Transform method*), 3747
transform_angles() (*matplotlib.transforms.Transform method*), 3748
transform_bbox() (*matplotlib.transforms.Transform method*), 3748
transform_non_affine() (*matplotlib.projections.geo.AitoffAxes.AitoffTransform method*), 3559
transform_non_affine() (*mat-*

*plotlib.projections.geo.AitoffAxes.InvertedAitoffTransform method*), 3560
transform_non_affine() (*matplotlib.projections.geo.HammerAxes.HammerTransform method*), 3574
transform_non_affine() (*matplotlib.projections.geo.HammerAxes.InvertedHammerTransform method*), 3575
transform_non_affine() (*matplotlib.projections.geo.LambertAxes.InvertedLambertTransform method*), 3579
transform_non_affine() (*matplotlib.projections.geo.LambertAxes.LambertTransform method*), 3580
transform_non_affine() (*matplotlib.projections.geo.MollweideAxes.InvertedMollweideTransform method*), 3585
transform_non_affine() (*matplotlib.projections.geo.MollweideAxes.MollweideTransform method*), 3586
transform_non_affine() (*matplotlib.projections.polar.InvertedPolarTransform method*), 3528
transform_non_affine() (*matplotlib.projections.polar.PolarAxes.InvertedPolarTransform method*), 3532
transform_non_affine() (*matplotlib.projections.polar.PolarAxes.PolarTransform method*), 3534
transform_non_affine() (*matplotlib.projections.polar.PolarTransform method*), 3548
transform_non_affine() (*matplotlib.scale.AsinhTransform method*), 3616
transform_non_affine() (*matplotlib.scale.FuncTransform method*), 3618
transform_non_affine() (*matplotlib.scale.InvertedAsinhTransform method*), 3619
transform_non_affine() (*matplotlib.scale.InvertedLogTransform method*), 3620
transform_non_affine() (*matplotlib.scale.InvertedSymmetricalLogTransform method*), 3621
transform_non_affine() (*matplotlib.scale.LogisticTransform method*), 3624
transform_non_affine() (*matplotlib.scale.LogitTransform method*), 3626
transform_non_affine() (*matplotlib.scale.LogTransform method*), 3623
transform_non_affine() (*matplotlib.scale.SymmetricalLogTransform method*), 3628
transform_non_affine() (*matplotlib.transforms.AffineBase method*), 3724
transform_non_affine() (*matplotlib.transforms.BlendedGenericTransform method*), 3738
transform_non_affine() (*matplotlib.transforms.CompositeGenericTransform method*), 3741
transform_non_affine() (*matplotlib.transforms.IdentityTransform method*), 3743
transform_non_affine() (*matplotlib.transforms.Transform method*), 3749
transform_path() (*matplotlib.transforms.AffineBase method*), 3724
transform_path() (*matplotlib.transforms.IdentityTransform method*), 3743
transform_path() (*matplotlib.transforms.Transform method*), 3749
transform_path_affine() (*matplotlib.transforms.AffineBase method*), 3725
transform_path_affine() (*matplotlib.transforms.IdentityTransform method*), 3743
transform_path_affine() (*matplotlib.transforms.Transform method*), 3749
transform_path_non_affine() (*matplotlib.projections.polar.PolarAxes.PolarTransform method*), 3534
transform_path_non_affine() (*matplotlib.projections.polar.PolarTransform method*), 3548
transform_path_non_affine() (*matplotlib.transforms.AffineBase method*), 3725
transform_path_non_affine() (*matplotlib.transforms.CompositeGenericTransform method*), 3741
transform_path_non_affine() (*matplotlib.transforms.IdentityTransform method*), 3744
transform_path_non_affine() (*matplotlib.transforms.Transform method*), 3749
transform_point() (*matplotlib.transforms.Transform method*), 3749
transform_xy() (*mpl_toolkits.axisartist.grid_finder.GridFinder method*), 4047
transformed() (*matplotlib.markers.MarkerStyle method*), 3042
transformed() (*matplotlib.path.Path method*), 3202
transformed() (*matplotlib.transforms.BboxBase method*), 3735
TransformedBbox (*class in matplotlib.transforms*), 3751
TransformedPatchPath (*class in matplotlib.transforms*), 3752
TransformedPath (*class in matplotlib.transforms*), 3752
TransformNode (*class in matplotlib.transforms*), 3749
TransformWrapper (*class in matplotlib.transforms*), 3750
translate() (*matplotlib.transforms.Affine2D method*), 3721
translated() (*matplotlib.transforms.BboxBase method*), 3735
transmute() (*matplotlib.patches.ArrowStyle.Fancy*

*method*), 3116
transmute() (*matplotlib.patches.ArrowStyle.Simple method*), 3117
transmute() (*matplotlib.patches.ArrowStyle.Wedge method*), 3117
TrapezoidMapTriFinder (*class in matplotlib.tri*), 3758
TriAnalyzer (*class in matplotlib.tri*), 3764
Triangulation (*class in matplotlib.tri*), 3755
tricontour() (*in module matplotlib.pyplot*), 3399
tricontour() (*matplotlib.axes.Axes method*), 2065
tricontour() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3827
tricontourf() (*in module matplotlib.pyplot*), 3403
tricontourf() (*matplotlib.axes.Axes method*), 2069
tricontourf() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3828
TriContourSet (*class in matplotlib.tri*), 3756
TriFinder (*class in matplotlib.tri*), 3758
trigger() (*matplotlib.backend_tools.AxisScaleBase method*), 2288
trigger() (*matplotlib.backend_tools.RubberbandBase method*), 2289
trigger() (*matplotlib.backend_tools.ToolBase method*), 2291
trigger() (*matplotlib.backend_tools.ToolCopyToClipboardBase method*), 2292
trigger() (*matplotlib.backend_tools.ToolFullScreen method*), 2293
trigger() (*matplotlib.backend_tools.ToolGrid method*), 2293
trigger() (*matplotlib.backend_tools.ToolMinorGrid method*), 2295
trigger() (*matplotlib.backend_tools.ToolQuit method*), 2296
trigger() (*matplotlib.backend_tools.ToolQuitAll method*), 2296
trigger() (*matplotlib.backend_tools.ToolToggleBase method*), 2298
trigger() (*matplotlib.backend_tools.ViewsPositionsBase method*), 2300
trigger() (*matplotlib.backend_tools.ZoomPanBase method*), 2300
trigger_tool() (*matplotlib.backend_bases.ToolContainerBase method*), 2283
trigger_tool() (*matplotlib.backend_managers.ToolManager method*), 2287
TriInterpolator (*class in matplotlib.tri*), 3759
TriMesh (*class in matplotlib.collections*), 2680
tripcolor() (*in module matplotlib.pyplot*), 3398
tripcolor() (*matplotlib.axes.Axes method*), 2063
triplot() (*in module matplotlib.pyplot*), 3396
triplot() (*matplotlib.axes.Axes method*), 2064
TriRefiner (*class in matplotlib.tri*), 3762
ttfFontProperty() (*in module matplotlib.font_manager*), 2950
twin() (*mpl_toolkits.axes_grid1.parasite_axes.HostAxesBase method*), 3986
twinx() (*in module matplotlib.pyplot*), 3248
twinx() (*matplotlib.axes.Axes method*), 2189
twinx() (*mpl_toolkits.axes_grid1.parasite_axes.HostAxesBase method*), 3986
twiny() (*in module matplotlib.pyplot*), 3249
twiny() (*matplotlib.axes.Axes method*), 2190
twiny() (*mpl_toolkits.axes_grid1.parasite_axes.HostAxesBase method*), 3986
TwoSlopeNorm (*class in matplotlib.colors*), 2730
Type1Font (*class in matplotlib._type1font*), 3766

## U

underline_position (*matplotlib.ft2font.FT2Font attribute*), 2956
underline_thickness (*matplotlib.ft2font.FT2Font attribute*), 2956
UniformTriRefiner (*class in matplotlib.tri*), 3762
uninstall_repl_displayhook() (*in module matplotlib.pyplot*), 3515
union() (*matplotlib.transforms.BboxBase static method*), 3735
unit() (*matplotlib.transforms.Bbox static method*), 3729
unit_circle() (*matplotlib.path.Path class method*), 3202
unit_circle_righthalf() (*matplotlib.path.Path class method*), 3202
unit_rectangle() (*matplotlib.path.Path class method*), 3202
unit_regular_asterisk() (*matplotlib.path.Path class method*), 3202
unit_regular_polygon() (*matplotlib.path.Path class method*), 3202
unit_regular_star() (*matplotlib.path.Path class method*), 3202
UnitData (*class in matplotlib.category*), 2370
units_per_EM (*matplotlib.ft2font.FT2Font attribute*), 2956
unregister() (*matplotlib.cm.ColormapRegistry method*), 2385
unregister() (*matplotlib.colors.ColorSequenceRegistry method*), 2740
update() (*matplotlib._docstring.Substitution method*), 2787
update() (*matplotlib.artist.Artist method*), 1876
update() (*matplotlib.backend_bases.NavigationToolbar2 method*), 2272
update() (*matplotlib.category.UnitData method*), 2370
update() (*matplotlib.collections.AsteriskPolygonCollection method*), 2410
update() (*matplotlib.collections.CircleCollection method*), 2432
update() (*matplotlib.collections.Collection method*), 2455
update() (*matplotlib.collections.EllipseCollection method*), 2477
update() (*matplotlib.collections.EventCollection method*), 2500
update() (*matplotlib.collections.LineCollection method*), 2523

update() (*matplotlib.collections.PatchCollection method*), 2544
update() (*matplotlib.collections.PathCollection method*), 2567
update() (*matplotlib.collections.PolyCollection method*), 2589
update() (*matplotlib.collections.PolyQuadMesh method*), 2613
update() (*matplotlib.collections.QuadMesh method*), 2636
update() (*matplotlib.collections.RegularPolyCollection method*), 2658
update() (*matplotlib.collections.StarPolygonCollection method*), 2680
update() (*matplotlib.collections.TriMesh method*), 2703
update() (*matplotlib.gridspec.GridSpec method*), 2959
update() (*matplotlib.gridspec.SubplotParams method*), 2965
update() (*matplotlib.text.Text method*), 3677
update() (*mpl_toolkits.axisartist.grid_finder.GridFinder method*), 4047
update_bbox_position_size() (*matplotlib.text.Text method*), 3677
update_datalim() (*matplotlib.axes.Axes method*), 2128
update_datalim() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3873
update_default_handler_map() (*matplotlib.legend.Legend class method*), 3002
update_frame() (*matplotlib.offsetbox.AnchoredOffsetbox method*), 3068
update_frame() (*matplotlib.offsetbox.PaddedBox method*), 3089
update_from() (*matplotlib.artist.Artist method*), 1877
update_from() (*matplotlib.collections.AsteriskPolygonCollection method*), 2410
update_from() (*matplotlib.collections.CircleCollection method*), 2432
update_from() (*matplotlib.collections.Collection method*), 2455
update_from() (*matplotlib.collections.EllipseCollection method*), 2477
update_from() (*matplotlib.collections.EventCollection method*), 2500
update_from() (*matplotlib.collections.LineCollection method*), 2523
update_from() (*matplotlib.collections.PatchCollection method*), 2544
update_from() (*matplotlib.collections.PathCollection method*), 2567
update_from() (*matplotlib.collections.PolyCollection method*), 2590
update_from() (*matplotlib.collections.PolyQuadMesh method*), 2613
update_from() (*matplotlib.collections.QuadMesh method*), 2636
update_from() (*matplotlib.collections.RegularPolyCollection method*), 2658
update_from() (*matplotlib.collections.StarPolygonCollection method*), 2680
update_from() (*matplotlib.collections.TriMesh method*), 2703
update_from() (*matplotlib.lines.Line2D method*), 3031
update_from() (*matplotlib.patches.Patch method*), 3167
update_from() (*matplotlib.text.Text method*), 3678
update_from_data_x() (*matplotlib.transforms.Bbox method*), 3729
update_from_data_xy() (*matplotlib.transforms.Bbox method*), 3729
update_from_data_y() (*matplotlib.transforms.Bbox method*), 3730
update_from_first_child() (*in module matplotlib.legend_handler*), 3015
update_from_path() (*matplotlib.transforms.Bbox method*), 3730
update_grid_finder() (*mpl_toolkits.axisartist.grid_helper_curvelinear.GridHelperCurveLinear method*), 4051
update_home_views() (*matplotlib.backend_tools.ToolViewsPositions method*), 2298
update_keymap() (*matplotlib.backend_managers.ToolManager method*), 2287
update_lim() (*mpl_toolkits.axisartist.axislines.GridHelperBase method*), 4040
update_lim() (*mpl_toolkits.axisartist.floating_axes.FixedAxisArtistHelper method*), 4042
update_lim() (*mpl_toolkits.axisartist.grid_helper_curvelinear.FixedAxisArtistHelper method*), 4049
update_lim() (*mpl_toolkits.axisartist.grid_helper_curvelinear.FloatingAxisArtistHelper method*), 4050
update_normal() (*matplotlib.colorbar.Colorbar method*), 2709
update_offset() (*matplotlib.offsetbox.DraggableAnnotation method*), 3076
update_offset() (*matplotlib.offsetbox.DraggableBase method*), 3076
update_offset() (*matplotlib.offsetbox.DraggableOffsetBox method*), 3077
update_position() (*matplotlib.axis.Tick method*), 2253
update_position() (*matplotlib.projections.polar.RadialTick method*), 3552
update_position() (*matplotlib.projections.polar.ThetaTick method*), 3556
update_positions() (*matplotlib.offsetbox.AnnotationBbox method*), 3074
update_positions() (*matplotlib.text.Annotation method*), 3684
update_prop() (*matplotlib.legend_handler.HandlerBase*

*method*), 3004
update_prop() (*matplotlib.legend_handler.HandlerRegularPolyCollection method*), 3012
update_scalarmappable() (*matplotlib.collections.AsteriskPolygonCollection method*), 2410
update_scalarmappable() (*matplotlib.collections.CircleCollection method*), 2432
update_scalarmappable() (*matplotlib.collections.Collection method*), 2455
update_scalarmappable() (*matplotlib.collections.EllipseCollection method*), 2477
update_scalarmappable() (*matplotlib.collections.EventCollection method*), 2500
update_scalarmappable() (*matplotlib.collections.LineCollection method*), 2523
update_scalarmappable() (*matplotlib.collections.PatchCollection method*), 2544
update_scalarmappable() (*matplotlib.collections.PathCollection method*), 2567
update_scalarmappable() (*matplotlib.collections.PolyCollection method*), 2590
update_scalarmappable() (*matplotlib.collections.PolyQuadMesh method*), 2613
update_scalarmappable() (*matplotlib.collections.QuadMesh method*), 2636
update_scalarmappable() (*matplotlib.collections.RegularPolyCollection method*), 2658
update_scalarmappable() (*matplotlib.collections.StarPolygonCollection method*), 2680
update_scalarmappable() (*matplotlib.collections.TriMesh method*), 2703
update_ticks() (*matplotlib.colorbar.Colorbar method*), 2710
update_transform() (*mpl_toolkits.axisartist.grid_finder.GridFinder method*), 4047
update_units() (*matplotlib.axis.Axis method*), 2241
update_view() (*matplotlib.backend_tools.ToolViewsPositions method*), 2299
use() (*in module matplotlib*), 1812
use() (*in module matplotlib.style*), 3643
use_overline() (*matplotlib.ticker.LogitFormatter method*), 3704
use_sticky_edges (*matplotlib.axes.Axes property*), 2156
use_xobject (*matplotlib.backends.backend_pdf.Op attribute*), 2326
useLocale (*matplotlib.ticker.ScalarFormatter property*), 3712
useMathText (*matplotlib.ticker.EngFormatter property*), 3695
useMathText (*matplotlib.ticker.ScalarFormatter property*), 3712
useOffset (*matplotlib.ticker.ScalarFormatter property*), 3712
usetex (*matplotlib.ticker.EngFormatter property*), 3695

## V

v_interval (*mpl_toolkits.mplot3d.axis3d.Axis property*), 3879
validate_any() (*in module matplotlib.rcsetup*), 3607
validate_anylist() (*in module matplotlib.rcsetup*), 3607
validate_aspect() (*in module matplotlib.rcsetup*), 3607
validate_axisbelow() (*in module matplotlib.rcsetup*), 3607
validate_backend() (*in module matplotlib.rcsetup*), 3607
validate_bbox() (*in module matplotlib.rcsetup*), 3607
validate_bool() (*in module matplotlib.rcsetup*), 3607
validate_color() (*in module matplotlib.rcsetup*), 3607
validate_color_for_prop_cycle() (*in module matplotlib.rcsetup*), 3607
validate_color_or_auto() (*in module matplotlib.rcsetup*), 3607
validate_color_or_inherit() (*in module matplotlib.rcsetup*), 3607
validate_colorlist() (*in module matplotlib.rcsetup*), 3607
validate_cycler() (*in module matplotlib.rcsetup*), 3607
validate_dashlist() (*in module matplotlib.rcsetup*), 3607
validate_dpi() (*in module matplotlib.rcsetup*), 3607
validate_fillstylelist() (*in module matplotlib.rcsetup*), 3607
validate_float() (*in module matplotlib.rcsetup*), 3607
validate_float_or_None() (*in module matplotlib.rcsetup*), 3607
validate_floatlist() (*in module matplotlib.rcsetup*), 3608
validate_font_properties() (*in module matplotlib.rcsetup*), 3608
validate_fontsize() (*in module matplotlib.rcsetup*), 3608
validate_fontsize_None() (*in module matplotlib.rcsetup*), 3608
validate_fontsizelist() (*in module matplotlib.rcsetup*), 3608
validate_fontstretch() (*in module matplotlib.rcsetup*), 3608
validate_fonttype() (*in module matplotlib.rcsetup*), 3608
validate_fontweight() (*in module matplotlib.rcsetup*), 3608
validate_hatch() (*in module matplotlib.rcsetup*), 3608

validate_hatchlist() (*in module matplotlib.rcsetup*), 3608
validate_hist_bins() (*in module matplotlib.rcsetup*), 3608
validate_int() (*in module matplotlib.rcsetup*), 3608
validate_int_or_None() (*in module matplotlib.rcsetup*), 3608
validate_markevery() (*in module matplotlib.rcsetup*), 3608
validate_markeverylist() (*in module matplotlib.rcsetup*), 3608
validate_ps_distiller() (*in module matplotlib.rcsetup*), 3609
validate_sketch() (*in module matplotlib.rcsetup*), 3609
validate_string() (*in module matplotlib.rcsetup*), 3609
validate_string_or_None() (*in module matplotlib.rcsetup*), 3609
validate_stringlist() (*in module matplotlib.rcsetup*), 3609
validate_whiskers() (*in module matplotlib.rcsetup*), 3609
ValidateInStrings (*class in matplotlib.rcsetup*), 3605
valign (*matplotlib.quiver.QuiverKey attribute*), 3599
VBoxDivider (*class in mpl_toolkits.axes_grid1.axes_divider*), 3938
vcenter (*matplotlib.colors.CenteredNorm property*), 2722
vcenter (*matplotlib.colors.TwoSlopeNorm property*), 2731
VectorParse (*class in matplotlib.mathtext*), 3044
Verbatim (*class in matplotlib.backends.backend_pdf*), 2335
VertexSelector (*class in matplotlib.lines*), 3033
VerticalHatch (*class in matplotlib.hatch*), 2967
vertices (*matplotlib.path.Path property*), 3202
vertices (*matplotlib.text.TextPath property*), 3686
verts (*matplotlib.widgets.PolygonSelector property*), 3785
Vf (*class in matplotlib.dviread*), 2791
view_init() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3866
view_limits() (*matplotlib.projections.polar.PolarAxes.RadialLocator method*), 3535
view_limits() (*matplotlib.projections.polar.PolarAxes.ThetaLocator method*), 3535
view_limits() (*matplotlib.projections.polar.RadialLocator method*), 3551
view_limits() (*matplotlib.projections.polar.ThetaLocator method*), 3555
view_limits() (*matplotlib.ticker.LinearLocator method*), 3698
view_limits() (*matplotlib.ticker.Locator method*), 3699
view_limits() (*matplotlib.ticker.LogLocator method*), 3702
view_limits() (*matplotlib.ticker.MaxNLocator method*), 3706
view_limits() (*matplotlib.ticker.MultipleLocator method*), 3707
view_limits() (*matplotlib.ticker.SymmetricalLogLocator method*), 3714
view_transformation() (*in module mpl_toolkits.mplot3d.proj3d*), 3909
viewlim_to_dt() (*matplotlib.dates.DateLocator method*), 2777
ViewsPositionsBase (*class in matplotlib.backend_tools*), 2300
violin() (*matplotlib.axes.Axes method*), 2014
violin_stats() (*in module matplotlib.cbook*), 2382
violinplot() (*in module matplotlib.pyplot*), 3348
violinplot() (*matplotlib.axes.Axes method*), 2009
visible_edges (*matplotlib.table.Cell property*), 3650
vlines() (*in module matplotlib.pyplot*), 3298
vlines() (*matplotlib.axes.Axes method*), 1960
vmax (*matplotlib.colors.CenteredNorm property*), 2722
vmax (*matplotlib.colors.Normalize property*), 2716
vmin (*matplotlib.colors.CenteredNorm property*), 2722
vmin (*matplotlib.colors.Normalize property*), 2716
voxels() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3830
VPacker (*class in matplotlib.offsetbox*), 3092

## W

WAIT (*matplotlib.backend_tools.Cursors attribute*), 2289
waitforbuttonpress() (*in module matplotlib.pyplot*), 3525
waitforbuttonpress() (*matplotlib.figure.Figure method*), 2854
warn_deprecated() (*in module matplotlib._api.deprecation*), 3808
warn_external() (*in module matplotlib._api*), 3806
WebAggApplication (*class in matplotlib.backends.backend_webagg*), 2362
WebAggApplication.AllFiguresPage (*class in matplotlib.backends.backend_webagg*), 2362
WebAggApplication.Download (*class in matplotlib.backends.backend_webagg*), 2363
WebAggApplication.FavIcon (*class in matplotlib.backends.backend_webagg*), 2363
WebAggApplication.MplJs (*class in matplotlib.backends.backend_webagg*), 2363
WebAggApplication.SingleFigurePage (*class in matplotlib.backends.backend_webagg*), 2363
WebAggApplication.WebSocket (*class in matplotlib.backends.backend_webagg*), 2363
Wedge (*class in matplotlib.patches*), 3190
wedge() (*matplotlib.path.Path class method*), 3202
WeekdayLocator (*class in matplotlib.dates*), 2780
weeks (*matplotlib.dates.relativedelta property*), 2785
Widget (*class in matplotlib.widgets*), 3802
width (*matplotlib._afm.CharMetrics attribute*), 1831
width (*matplotlib.dviread.Tfm attribute*), 2791
width (*matplotlib.mathtext.RasterParse attribute*), 3044
width (*matplotlib.mathtext.VectorParse attribute*), 3045
width (*matplotlib.patches.Annulus property*), 3101
width (*matplotlib.patches.Ellipse property*), 3139
width (*matplotlib.transforms.BboxBase property*), 3735
widths (*matplotlib.dviread.DviFont attribute*), 2789
win32FontDirectory() (*in module matplotlib.font_manager*), 2951

window_hanning() (*in module matplotlib.mlab*), 3063
window_none() (*in module matplotlib.mlab*), 3063
with_extremes() (*matplotlib.colors.Colormap method*), 2733
withSimplePatchShadow (*class in matplotlib.patheffects*), 3208
withStroke (*class in matplotlib.patheffects*), 3209
withTickedStroke (*class in matplotlib.patheffects*), 3210
world_transformation() (*in module mpl_toolkits.mplot3d.proj3d*), 3910
write() (*matplotlib.backends.backend_pdf.PdfFile method*), 2328
write() (*matplotlib.backends.backend_pdf.Reference method*), 2330
write() (*matplotlib.backends.backend_pdf.Stream method*), 2335
writeExtGSTates() (*matplotlib.backends.backend_pdf.PdfFile method*), 2328
writeFonts() (*matplotlib.backends.backend_pdf.PdfFile method*), 2328
writeGouraudTriangles() (*matplotlib.backends.backend_pdf.PdfFile method*), 2328
writeHatches() (*matplotlib.backends.backend_pdf.PdfFile method*), 2328
writeImages() (*matplotlib.backends.backend_pdf.PdfFile method*), 2328
writeInfoDict() (*matplotlib.backends.backend_pdf.PdfFile method*), 2328
writeMarkers() (*matplotlib.backends.backend_pdf.PdfFile method*), 2328
writeObject() (*matplotlib.backends.backend_pdf.PdfFile method*), 2328
writePath() (*matplotlib.backends.backend_pdf.PdfFile method*), 2328
writePathCollectionTemplates() (*matplotlib.backends.backend_pdf.PdfFile method*), 2328
writeTrailer() (*matplotlib.backends.backend_pdf.PdfFile method*), 2328
writeXref() (*matplotlib.backends.backend_pdf.PdfFile method*), 2328

## X

x (*matplotlib.widgets.ToolHandles property*), 3800
x0 (*matplotlib.transforms.Bbox property*), 3731
x0 (*matplotlib.transforms.BboxBase property*), 3735
x1 (*matplotlib.transforms.Bbox property*), 3731
x1 (*matplotlib.transforms.BboxBase property*), 3735
XAxis (*class in matplotlib.axis*), 2220
xaxis_date() (*matplotlib.axes.Axes method*), 2174
xaxis_inverted() (*matplotlib.axes.Axes method*), 2120
xaxis_inverted() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3847
xcorr() (*in module matplotlib.pyplot*), 3338
xcorr() (*matplotlib.axes.Axes method*), 1999
xkcd() (*in module matplotlib.pyplot*), 3525
XKCD_COLORS (*in module matplotlib.colors*), 2753
xlabel() (*in module matplotlib.pyplot*), 3461
xlim() (*in module matplotlib.pyplot*), 3462
xmax (*matplotlib.transforms.BboxBase property*), 3735
xmin (*matplotlib.transforms.BboxBase property*), 3735
XMLWriter (*class in matplotlib.backends.backend_svg*), 2355
xpdf_distill() (*in module matplotlib.backends.backend_ps*), 2346
xscale() (*in module matplotlib.pyplot*), 3463
XTick (*class in matplotlib.axis*), 2250
xticks() (*in module matplotlib.pyplot*), 3464
xy (*matplotlib.patches.Polygon property*), 3179
xy (*matplotlib.patches.Rectangle property*), 3183
xyann (*matplotlib.offsetbox.AnnotationBbox property*), 3074
xyann (*matplotlib.text.Annotation property*), 3684
xycoords (*matplotlib.text.Annotation property*), 3685

## Y

y (*matplotlib.widgets.ToolHandles property*), 3800
y0 (*matplotlib.transforms.Bbox property*), 3731
y0 (*matplotlib.transforms.BboxBase property*), 3735
y1 (*matplotlib.transforms.Bbox property*), 3731
y1 (*matplotlib.transforms.BboxBase property*), 3735
YAxis (*class in matplotlib.axis*), 2221
yaxis_date() (*matplotlib.axes.Axes method*), 2178
yaxis_inverted() (*matplotlib.axes.Axes method*), 2120
yaxis_inverted() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3847
YearLocator (*class in matplotlib.dates*), 2781
ylabel() (*in module matplotlib.pyplot*), 3465
ylim() (*in module matplotlib.pyplot*), 3466
ymax (*matplotlib.transforms.BboxBase property*), 3735
ymin (*matplotlib.transforms.BboxBase property*), 3735
yscale() (*in module matplotlib.pyplot*), 3467
YTick (*class in matplotlib.axis*), 2250
yticks() (*in module matplotlib.pyplot*), 3468

## Z

zaxis_date() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3863
zaxis_inverted() (*mpl_toolkits.mplot3d.axes3d.Axes3D method*), 3848
zoom() (*matplotlib.backend_bases.NavigationToolbar2 method*), 2272
zoom() (*matplotlib.backends.backend_webagg_core.NavigationToolbar2WebAgg method*), 2361
zoomed_inset_axes() (*in module mpl_toolkits.axes_grid1.inset_locator*), 3974
ZoomPanBase (*class in matplotlib.backend_tools*), 2300
zorder (*matplotlib.artist.Artist attribute*), 1885
zorder (*matplotlib.axes.Axes attribute*), 2213
zorder (*matplotlib.collections.AsteriskPolygonCollection attribute*), 2410

zorder (*matplotlib.collections.CircleCollection attribute*), 2432
zorder (*matplotlib.collections.Collection attribute*), 2455
zorder (*matplotlib.collections.EllipseCollection attribute*), 2477
zorder (*matplotlib.collections.EventCollection attribute*), 2501
zorder (*matplotlib.collections.LineCollection attribute*), 2523
zorder (*matplotlib.collections.PatchCollection attribute*), 2545
zorder (*matplotlib.collections.PathCollection attribute*), 2567
zorder (*matplotlib.collections.PolyCollection attribute*), 2590
zorder (*matplotlib.collections.PolyQuadMesh attribute*), 2613
zorder (*matplotlib.collections.QuadMesh attribute*), 2636
zorder (*matplotlib.collections.RegularPolyCollection attribute*), 2658
zorder (*matplotlib.collections.StarPolygonCollection attribute*), 2680
zorder (*matplotlib.collections.TriMesh attribute*), 2703
zorder (*matplotlib.image.FigureImage attribute*), 2975
zorder (*matplotlib.legend.Legend attribute*), 3002
zorder (*matplotlib.lines.Line2D attribute*), 3031
zorder (*matplotlib.offsetbox.AnchoredOffsetbox attribute*), 3068
zorder (*matplotlib.offsetbox.AnnotationBbox attribute*), 3074
zorder (*matplotlib.patches.Patch attribute*), 3167
zorder (*matplotlib.text.Text attribute*), 3678
zorder (*mpl_toolkits.axisartist.axis_artist.AxisArtist attribute*), 4008